Managing the Quality of Software Product Line Architectures through Reusable Model Transformations *

Amogh Kavimandan †, Aniruddha Gokhale, Gabor Karsai
Dept. of EECS, Vanderbilt University
Nashville, TN 37235, USA
Contact: a.gokhale@vanderbilt.edu

Jeff Gray
Dept of CS, Univ of Alabama
Tuscaloosa, AL 35487, USA
gray@cs.ua.edu

ABSTRACT

In model-driven engineering of applications, the quality of the software architecture is realized and preserved in the successive stages of its lifecycle through model transformations. However, limited support for reuse in contemporary model transformation techniques forces developers of product line architectures to reinvent transformation rules for every variant of the product line, which can adversely impact developer productivity and in turn degrade the quality of the resulting software architecture for the variant. To overcome these challenges, this paper presents the MTS (Model-transformation Templatization and Specialization) generative transformation process, which promotes reuse in model transformations through parameterization and specialization of transformation rules. MTS defines two higher order transformations to capture the variability in transformation rules and to specialize them across product variants. The core idea behind MTS is realized within a graphical model transformation tool in a way that is minimally intrusive to the underlying tool's implementation. The paper uses two product line case studies to evaluate MTS in terms of reduction in efforts to define model transformation rules as new variants are added to the product line, and the overhead in executing the higher order transformations. These metrics provide an indirect measure of how potential degradation in the quality of software architectures of product lines caused due to lack of reuse can be alleviated by MTS.

Categories and Subject Descriptors

D:Software [**2:Software Engineering**]: 2:Design Tools and Techniques

General Terms

Design, Algorithms, Measurement

*Research supported by NSF CAREER #CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

†Author has since graduated

QoSA '11 Boulder, CO, USA

Keywords

model transformations, reuse, software quality

1. INTRODUCTION

Model transformation is a key element of the Model-Driven Engineering (MDE) paradigm [18]. Model transformations are used to define progressive refinements of application models from abstract, high-level views into low-level, detailed views that are used by the execution platform for different purposes, such as application configuration, deployment, and code synthesis. They are also used in transforming models to representations suitable for analysis tools that check various properties, such as correctness or deadlock free behavior. Each of these outcomes represents a different dimension of application software architecture quality.

Despite their importance, however, contemporary model transformation tools and techniques [4, 8, 12, 20] have limited support for reuse, which becomes problematic for software product lines [3] since it forces developers to reinvent the entire set of transformation rules, and repeat the entire transformation process despite significant commonalities among product variants of the product line. The result is a set of negative outcomes: increased developer effort, loss of productivity, less reuse, limitations on evolution of the product line, and increased cost of maintenance. All these outcomes may degrade the quality of software architectures for the product line.

Overcoming these problems requires reuse capabilities in model transformations. Recent efforts [5, 21, 23] have applied model transformations to product lines. Yet, the following questions remain to be resolved so that the quality of software architectures for product lines can be managed effectively over their development and maintenance lifecycle.

(a) ***Invariants*****:** How can the commonalities in the transformation process be factored out such that they can be reused by the entire product line?

(b) ***Variability*****:** How can the variabilities in the model transformation rules be decoupled from each other while maximizing the flexibility of the transformation process?

(c) ***Extensibility*****:** How can the model transformation process for a product line be extended with new variants, with minimally invasive changes to the existing transformation rules?

(d) ***Minimal Intrusiveness*****:** How can contemporary transformation tools be enhanced to provide first-class support for reuse of transformations rules with minimal changes, if any, to their design and implementation so that these tools remain backward compatible yet be able to support reuse in product lines?

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 factor out variabilities in the transformation rules for individual product variants and form parameterized transformation rules that are decoupled from the transformation rules that represent the commonalities of the product line. A sequence of two higher order transformations (HOT) [22] [1] subsequently are used in the stepwise refinement [2] of the base rules using the parameterized rules to generate an entire set of transformation rules for individual product variants. In our prior work, we presented preliminary ideas on reusable model transformations [9] and demonstrated its use in the configuration of end point devices used by insurance agents in enterprise appliacations [10]. The aim of this paper is to describe the scientific principles and a process behind realizing reuse capabilities in model transformations, and demonstrating how to realize them concretely in a contemporary model transformation tool.

The rest of the paper is organized as follows: Section 2 describes the MTS solution; Section 3 illustrates a concrete realization of MTS within a contemporary model transformation tool and application to a case study; Section 4 evaluates the merits of the MTS process; Section 5 compares our work with the existing literature; and Section 6 provides concluding remarks.

2. DESIGNING REUSE CAPABILITIES FOR MODEL TRANSFORMATIONS

This paper focuses on the model transformations that are carried out using model transformation tools, such as ATL [7] and GReAT [8]. These tools conform to a transformation process shown in Figure 1, 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 target modeling language corresponding to the same or 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 target language.

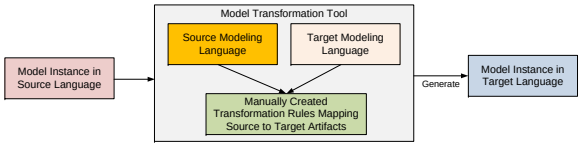

Figure 1: Model Transformation Process

2.1 Problem Statement and Solution Overview

Due to limited support for reuse in contemporary model transformation tools, developers of software product lines are forced to reinvent transformation rules for every variant as illustrated in Figure 2, which may adversely impact the quality of software architectures.

To overcome these problems, this paper presents the design principles behind the MTS (Model-transformation Templatization and Specialization) process. The core idea behind MTS is shown in Figure 3 and explained in the rest of this section. MTS realizes reusable model transformations in graphical model transformation frameworks using the following four-step generative process:

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

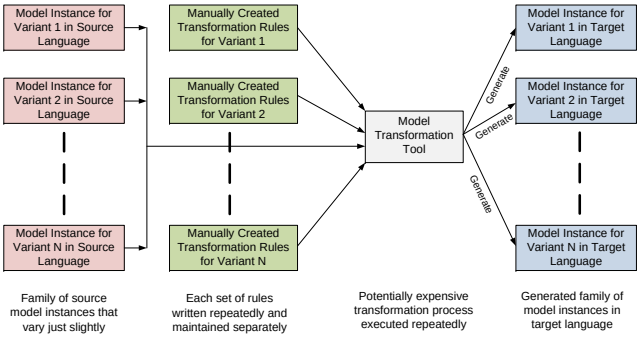

Figure 2: Reinventing Transformation Rules

1. Decoupling the variabilities from commonalities: In Step 1 of Figure 3, developers capture the variabilities in transformation rules in terms of a simple *constraint notation specification* (see Section 2.2). This step decouples the transformation rules for the commonalities from those for the variabilities.

2. Generating variability metamodel: In this step, developers use a higher order transformation (HOT) (*i.e.*, transformations that work on meta metamodels to translate source metamodel(s) to target metamodel(s)) defined in MTS to generate the variability metamodel (VMM) for their application family (see Section 2.3). A VMM modularizes the variability in transformation rules by parameterizing the rules and representing them in the form of a metamodel, which is the level of abstraction understood by the underlying model transformation tool.

3. Synthesizing specialization repository: Next, developers create VMM models (a manual step), where each VMM model corresponds to an instantiation of the variability captured in Step 1 for individual family members. A collection of all the VMM models is termed as a specialization repository for that family (see Section 2.4).

4. Specializing the transformation instances: Finally, as shown in Step 4, developers use another HOT defined in MTS to create a set of transformation rules for the desired product variant (see Section 2.5). This step is based on the principles of stepwise refinement [2] of features, which is used in product line development.

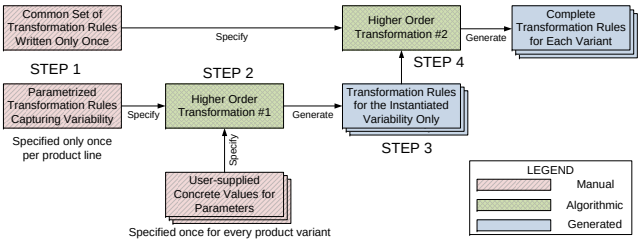

Figure 3: MTS Approach to Reusable Model Transformations

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

2.2 Step I: Decoupling Commonalities and Variabilities in Transformation Rules

Mixins and mixin layers [19], which are realized using parametric polymorphism such as C++ templates, have been used in large-scale refinements for product line development. Our solution in

MTS is based on a similar philosophy; specifically, we seek a generative solution to promote reuse of transformation rules through parameterized transformation rules and their specialization. The basic idea is that all the commonalities, which constitute the invariants of an application family, are separated from the variabilities, and maintained as family *instance-independent* transformation rules that are specified in the normal way using the underlying transformation tool, while the variabilities are captured as parameterized transformation rules discussed below.

Since contemporary model transformation tools often do not provide first-class support to separate the commonalities in transformation rules from the variabilities, a number of questions arise. How is this separation achieved? What types of variabilities exist? How are the variabilities to be captured without unduly impacting the basic design and architecture of the model transformation tool (which is a key objective otherwise existing projects will no longer be supported by the tool)?

A close scrutiny of model transformation rules for different product variants of an application family (such as our case studies) illustrates that at a minimum, variability in transformation rules 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 parameterized transformation approach:

(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.

Note that 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. The two types of variabilities we address in this paper can be denoted as simple implication relations that can be characterized by one of the types of associations between source ($s \in S$) and target ($t \in T$) objects shown in Table 1, where $P1$ and $P2$ denote patterns of source and target objects.

Table 1: Associations between 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 $t = \{P2(t_1, t_2, ..., t_n) \mid t_{i=1..n} \in T\}$.

In MTS our goal was to define a simple approach to encode these associations. To that end we defined a special but simple syntax called the constraint specification notation that captures the relations outlined above for the variabilities. Recall that supporting any additional syntax and semantics must be minimally intrusive to the underlying transformation tool in which it is going to be realized. One promising implementation choice was to use the mechanism of tunneling, which has been used successfully in the networking area to retrofit existing infrastructure with new ideas. Hence, we propose the use of *comment blocks* supported by the underlying tool to encode the metadata. Such techniques have been used to great advantage by document generators, such as Javadoc.[2]

Figure 4 illustrates the syntax, which codifies the implication relations described in Table 1. The `Quantitative` block captures all the attributes while the `Structural` block captures all the model elements that vary between family members. In the `Structural` block shown, there is an association defined between source model object `I1`, and target model objects `O1` and `O2` which implies that the composition of `O1` and `O2` is dependent on `I1`. The presence of `O7` in the `Structural` block implies that the object is created in the target model irrespective of the presence (or absence) of any specific source model object(s).

```
1   Structural {                Quantitative {
2     I1::O1;O2                   I2:A1::O3:A1,O3:A2,O3:A3
3     O7                          I3:A3::O5:A6
4   ...}                          O6:A7
5                               ...}
```

Figure 4: Syntax of Constraint Specification Notation to Capture Variability.

The `Quantitative` block captures many-to-many and one-to-one relations between source and target elements. For example, the values of attributes `A1`, `A2`, and `A3` of model object `O3` are dependent on that of the `A1` attribute of object `I2`, as shown in Line 2 of `Quantitative` block. The specification `O6:A7` states that the attribute `A7` is directly mapped, *i.e.*, it is hard-coded and is assigned statically in the model transformation.

2.3 Step II: Generating the Variability Metamodel from Constraint Specifications

Although the constraint specifications discussed in Step I capture the variability in the transformations, the notation used (*e.g.*, comment block) is oblivious to the model transformation tool and cannot be leveraged unless this information is presented as a first class entity to the transformation tool. In other words, this information must be made available to the underlying model transformation tools at the same level of abstraction as the transformation rules supported by the tool. Transforming the constraint specifications, which themselves are transformation rules using a special syntax that transform objects of the source metamodel to objects of the target metamodel, into a metamodel that is used by the transformation tool to encode transformation rules is a form of a higher order transformation (HOT). Therefore, in MTS we define a HOT to transform the constraint specifications into what we call a *Variability Meta Model* (VMM). A VMM essentially modularizes the variabilities as parameterized transformation rules and decouples them from the model transformation rules already specified for the commonalities.

Algorithm 1 depicts the HOT for generating the VMM. The basic idea behind the algorithm is as follows: Recall from Section 2.2 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.

[2]To use Javadoc, documentation is inserted as comments in source code using special annotations. See http://java.sun.com/j2se/javadoc for more details.

Algorithm 1: HOT: Constraint Specifications to VMM.

```
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
17                     composeVariabilityAssociation(V); /* creates a
                         connection between source and target objects created
                         earlier                                                  */
18                 end
19             if cnb.quantitativeVariabilities() ≠ ∅ then
20                 /* Similarly, create model objects for quantitative variabilities. */
21             createContainingObject(V); /* name of the containing object is a
                 combination of rule name, and constraint block name, each of which
                 must be unique                                                   */
22         end
23         CNB ← ∅; /* constraint blocks from previous loop are deleted, s.t. those from
             the next rule can be read                                           */
24     end
25 end
```

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 the modeling tool, *e.g.*, GME upon which GReAT is based. Line 11 reads the source patterns that correspond to every structural variability in the parameterized 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 21.

2.4 Step III: Synthesizing a Specialization Repository

In the next step transformation developers use the generated VMM to manually create VMM model instances, where each VMM model corresponds to variabilities in a product variant. This manual step is in the same spirit as any other model creation process where a model instance is manually produced using the metamodel of its modeling language. The difference here is that the VMM is a metamodel that focuses only on the variabilities of a product variant transforming it from its source to target modeling languages. The variabilities of every product variant is modeled in this step using their corresponding generated VMM to give rise to a collection of VMM model instances which we call a specialization repository. This step is akin to the process of providing functors in C++ template programming (*e.g.*, consider a parameterized C++ sort function, where programmers are required to supply explicit instantiation of the $\leq$ operator for all the user-defined types for which the sort function is specialized).

2.5 Step IV: Specializing the Transformation Instances

In product line development, artifacts such as code are often synthesized using stepwise refinement of base features incrementally with additional features representing the variability. In similar manner, in MTS we generate the complete set of transformation rules for individual product variants using the base rules that capture the commonalities, the parameterized rules that capture variabilities, and the specializations available in the repository.

Notice that until this step we have the transformation rules for the commonalities, the parameterized rules for the variabilities, and VMM instances that supply specializations for the parameterized types. However, these are all determined in isolation. We do not yet have a complete set of transformation rules for every variant as should be the case if a developer manually created the rules, as shown in Figure 2. The basic question we answer is how can model transformation tools support such a stepwise refinement process to generate the entire set of transformations for every variant?

To address this question requires two mechanisms: (1) a way to combine the transformation rules representing commonalities with the parameterized rules, and (b) specializing the parameterized rules to provide concrete rules. Requirement (1) is a simple form of additive refinement; requirement (2) calls for an ability to weave in the specializations into the parameterized rules to create concrete instantiations of the parameterized rules. MTS supports these requirements through a second HOT that (1) reads the VMM model instance for a product variant, and (2) inserts temporary objects at specialization points, which are akin to joinpoints [11][3], in the parameterized transformation rules to enable the weaving (or refinement) of the rules with the instantiated variability of a product variant (corresponding to the current VMM model). Thereafter, a combination of the commonalities and the specialized rules together form the complete set of transformation rules for the variant.

Algorithm 2 defines our second HOT. Lines 3–5 create a new model transformation instance R' from the input parameterized transformation R, read the containing model objects in VMM V, and for every model object *ob* search the corresponding rule in the transformation R'.[4] This rule essentially represents the location of the variabilities contained in *ob* which were specified using the constraint specification notation described in Section 2.2. 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 to represent the specialization point where weaving will take place. 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

[3]Joinpoints are places in the control flow of a program where a crosscutting concern must be woven in.

[4]For creating product variant instances, it is not necessary to create a new instance R', but is done only for Algorithm 2 to avoid modification of the original parameterized transformation R.

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*.

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
```

3. A CONCRETE IMPLEMENTATION AND APPLICATION OF MTS

This section illustrates how the MTS principles are implemented in a contemporary model transformation tool called Graph Rewriting And Transformation (GReAT) [8]. GReAT is developed using the Generic Modeling Environment (GME) [14], 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.

Transformation rules are defined using the GReAT visual language. First, the source and target metamodels of the domain-specific modeling languages (DSMLs) for the transformation tool chain are defined. Next, the transformation developers use the GReAT transformation language to define transformation rules in terms of patterns[5] of source and target modeling objects. Subsequently, a source model instance is provided to the GReAT framework. Finally, developers execute the GReAT engine (called the GR-engine) that translates the source model using rules specified in the second step above into the target model.

3.1 Case Study: Quality Determined by QoS Configuration Mapping

Our case study involves an exogenous transformation which translates domain-specified quality of service (QoS) requirements[6] into the underlying middleware platform-specific QoS configuration options. Figure 5 shows the UML representation of both the source and the target metamodels used in the QoS configuration case study. As shown, the source 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.

The target metamodel defines a language to represent real-time CORBA [15] middleware configurations and defines the following

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

[6]These could be considered policies but we choose to refer to them as requirements.

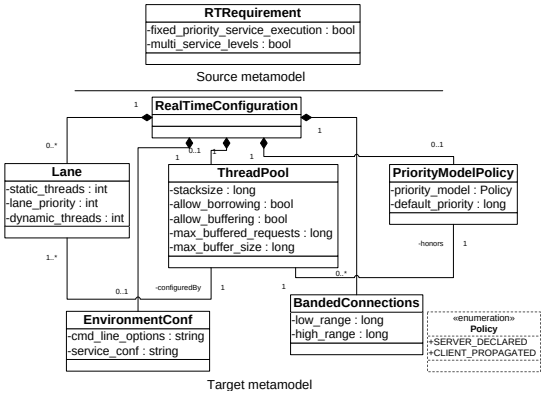

Figure 5: UML Metamodels for Middleware QoS Configuration.

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 determine the quality of the software architecture. Such transformations should easily be 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.

Figure 6 illustrates a sample transformation project for our QoS configuration case study.[7] The entire transformation is composed of a sequence of transformation rule blocks, which themselves can be nested. At the lowest level, a rule block comprises a pattern that describes how one or more elements from the source metamodel are 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 cannot be captured in visual form can be embedded as C++ code in an `AttributeMapping` block as shown in the figure.

[7]The purpose of the screenshot is simply to illustrate the user view of GReAT. Details inside the models and transformation rules are not important.

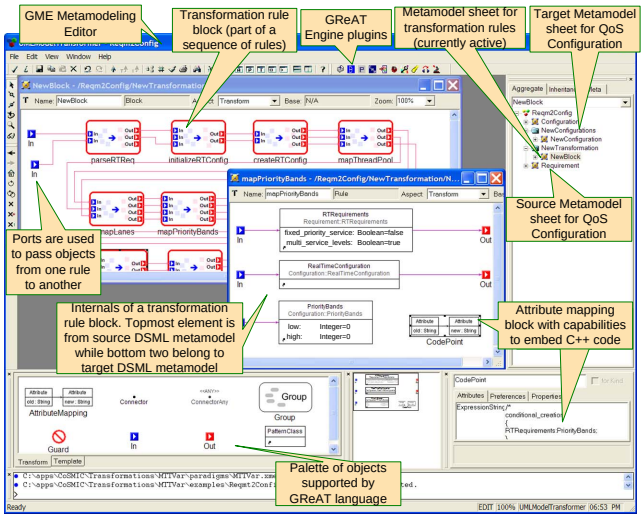

Figure 6: Model Transformation for QoS Mapping in GReAT (callouts depict various features in GReAT)

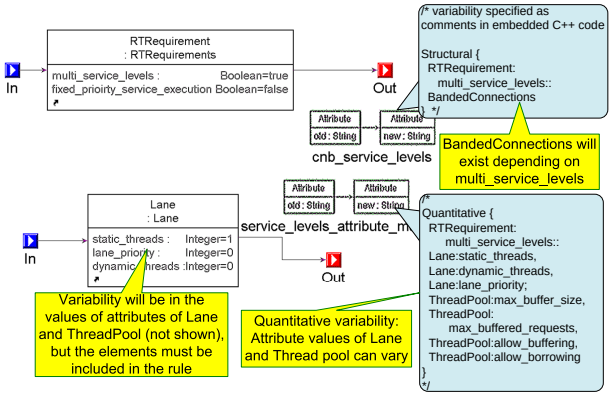

Figure 7: Parametrized Transformation Rule in QoS Configuration Case Study.

3.2 Impediments to Transformation Reuse in GReAT

For our case study, a number of transformations are feasible depending on the variability in the QoS requirements. Using GReAT, a complex set of transformation rules (*e.g.*, sequence of rule blocks) must be developed for each such variation. 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 reuse capabilities in GReAT, the sequence of rule blocks shown in Figure 6 will have to be created for each variant and follow all the transformation steps imposed by the tool. For example, execution of GReAT's GR-engine execution on a model instance, such as a model of QoS configuration in the source modeling language, involves the following steps: (1) execute the *master interpreter* to generate 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 to obtain the transformed model instance in the target language. The consequence of having to repeat the transformations for every variant is that the master interpreter must be executed every time any change is made to any rule. This will require recompilation of the generated source code and executing it on the source models. These limitations make a compelling case for reusable model transformations.

3.3 Realizing MTS in GReAT

We now show how the four steps of the MTS process are realized in GReAT. We use our QoS configuration case study as a guiding example to describe the implementation.

3.3.1 MTS Step I: Decoupling Commonality from Variability

In GReAT we supported the constraint specification notation by exploiting the C++ code embedding feature (see *Attribute Mapping block* feature in Figure 6). In particular, we use C++ comments within the embedded code as a means to capture variability. Because these are comments, it has no impact on the execution of the GR-engine; in other words, our approach satisfies the requirement of remaining minimally intrusive.

Figure 7 shows an excerpt of the parameterized 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`. What this specification implies is that a `BandedConnection` is to be introduced in the final transformation rules for this variant only if the `RTRequirement` includes `multi_service_levels`. The `Lane` and `Threadpool` (not shown) are included in the parameterized rule because they demonstrate variability in the values as discussed below. Otherwise, they would have been part of the transformation rules that reflect the commonalities.

Since the attribute values for `Lane` and `Threadpool` may vary, they are captured separately in the form of a *Quantitative* block notation, which indicates what all attributes and their values can vary. Since the constraint specification is simply a parameterization step, the concrete values for these attributes are not mentioned in this step. Instead, they are supplied at the time of specialization as shown in Section 2.4.

3.3.2 MTS Step II: Generating Variability Metamodels

Recall that the constraint specification notation used to capture the variability is not understandable by the GReAT tool. A conversion process is therefore needed to transform the specification into a first class entity supported by GReAT, which essentially happens to be a model or metamodel in GME. We applied Algorithm 1 to the parameterized model transformation of our QoS configuration case study to automatically generate a VMM. Figure 8 shows a screenshot in GReAT/GME of an excerpt of the generated VMM.

In this figure, `SourcePattern` and `TargetPattern` denote, respectively, the source and target language patterns. The variabilities are modeled as pairs of `SourcePattern` and `TargetPattern`, and annotated by whether they are `Structural` or `Quantitative` using Boolean attributes. The figure corresponds to the `Quantitative` variability rule of Figure 7 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

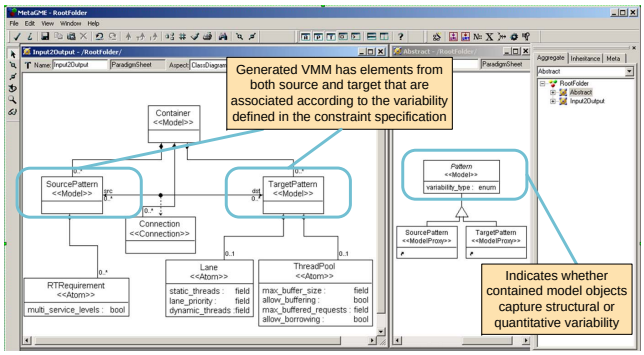

Figure 8: Excerpt of Generated VMM for QoS Configuration Study.

VMM in a similar manner using Algorithm 1. In effect, what we have achieved is a representation of the parameterized transformation in a form that the model transformation tool can understand.

3.3.3 MTS Step III: Synthesizing the Repository

In our QoS configuration case study, the specialization repository contains a distinct VMM model for every configuration. The aggregate of all the VMM models collectively contain variabilities of all the family members. Developers will create VMM models for all known variants of the product line. Figure 9 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 VMM model instances rather than being encoded in terms of transformation rules, it is considerably easier to modify these values without affecting the transformation rules. This step enables creating as many model instances as the structural and quantitative variabilities permit without having to modify the rules. The significant benefit for transformation tools like GReAT is that the transformation logic need not be re-compiled and linked into a new executable since the rules themselves do not change – only the model instances supplied change.

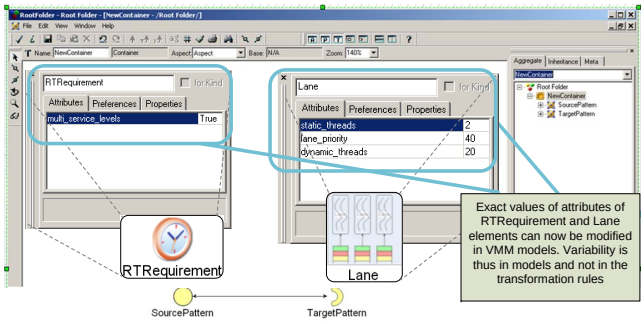

Figure 9: A VMM Model Instance for a Variant of QoS Configuration Case Study.

3.3.4 MTS Step IV: Specializing the Parametrized Rules

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, as identified earlier in Section 2.2, there is a quantitative variability involving these two elements. The same variability is also given in Figure 10 for reference. The attributes in `tempObject` are assigned values from 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. 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 instance.

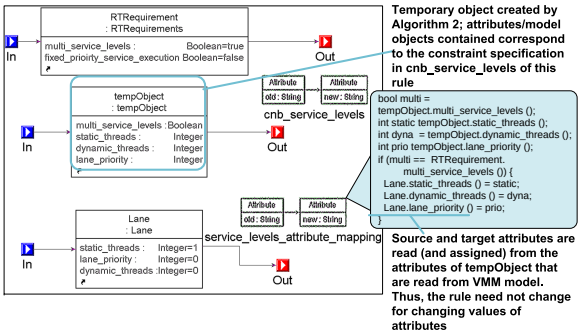

Figure 10: Specialization of a QoS configuration rule using MTS.

4. EVALUATING MERITS OF 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, which is an indirect measure of the software quality because there exists a potential for degradation in the quality due to the tedium involved in reinventing the steps without support for reuse. Second, since MTS provides higher order transformations, we also discuss the overhead incurred by the higher order transformations. Our second evaluation is important from the perspective of acceptance as a tool in production environments.

Our evaluations are based on rules for two case studies implemented in GReAT. One of the case studies involved the QoS configuration mapping described in Section 3.1. The second case study [10] involves the creation of dialogs for a set of communication endpoints from *workflow* decision points in an insurance company. Since the employees in an insurance company may potentially be using several endpoints (*i.e.*, communication devices), an important consideration in delivering information content from the workflows to the employees is the customization of communication dialogs for individual endpoints, which is accomplished using model transformations.

The prototype implementation of MTS is part of the CoSMIC[8] 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. All the overhead measurement experiments were run on a Windows XP SP2 workstation with 2.66 GHz Intel Xeon dual processor and 2 GB physical memory.

4.1 Reduction in Development Effort using MTS

Recall from Section 2 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

[8]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/.

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.

Without MTS, Steps 1 and 2 above have to be re-executed each time a new type of family instance (*e.g.*, addition of a new communication endpoint in insurance enterprise case study) has to be supported by the model transformations. Additionally, even for a single family instance, modifying a particular mapping (*e.g.*, changing the values of `Lane` attributes, for a particular `multi_service_levels` value in QoS configuration case study), requires the re-execution of the first two steps above. In contrast, in MTS the first two steps have to be executed only once when the model transformation is being executed for the first time. Since all the instance-specific customizations/changes are done in the corresponding VMM model instance, the developers only need to execute Step 3 after each change to produce the output of the transformation (*i.e.*, a new family instance). For our two case studies, as shown in Table 2, using MTS leads to savings of up to 90% in the time taken for a single transformation run, over non-parameterized approaches. We evaluate these qualities of MTS in the context of two situations common in managing quality product line software architectures.

Table 2: Time Taken in Executing the Two Model Transformations in GReAT.

Case Study	Master Interpreter	GR-engine	
		Compile & execute	Execute
Insurance Enterprise	16 sec.	64 sec.	8 sec.
QoS Configuration	16 sec.	104 sec.	12 sec.

Case 1: Newly added variant is subsumed by existing constraint specifications: The existing constraint specification for the product line may be sufficient to capture all the variabilities of a newly introduced product variant. Thus, the developers can create a new variant simply by re-executing the same model transformation with the VMM model instance of the variant as one of the inputs to the transformation. Note that the first two steps of GReAT 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 product variant are not completely captured by existing constraint specification for the product variant, MTS requires enhancements to the constraint specification itself. Such a change necessitates executing the first two steps above but only once to generate a new VMM which can be used to model variabilities in the new variant. Note that despite this change, the VMM model instances corresponding to the existing variants will still be valid provided the changes in constraint specification (because of a new product variant) are orthogonal to the existing commonalities and variabilities.

4.2 Performance Overhead of using MTS

The rationale behind these experiments is to quantify the overhead placed by the higher order transformations in Algorithms 1 and 2 when the number of structural and quantitative variabilities are increased. The performance overhead was calculated in terms of the time taken by each of these algorithms when used in the context of each of the two case studies.

In all we identified (a maximum of) fifteen variabilities for insurance enterprise case study, and eleven variabilities for the QoS configuration case study. The performance overhead was measured by increasing the variabilities in each case study from a minimum value of two to the maximum values above. The size of both the metamodels is given in Table 3. Table 4 shows the distribution of variabilities across the quantitative and structural dimensions for these case studies.

Table 3: The Size of the Metamodels

Metamodel	# of modeling elmts.	# of attribs.	# of conns.
Insurance Enterprise			
SRC/TRGT	8	14	0
QoS Configuration			
SRC	3	2	2
TRGT	8	14	4

Table 4: Distribution of Variabilities

Data Point	Insurance Enterprise		QoS Configuration	
	Quantitative	Structural	Quantitative	Structural
1	2	0	2	0
2	2	2	4	0
3	3	4	5	0
4	3	6	5	2
5	5	6	6	3
6	6	7	8	3
7	6	9	n.a.	n.a.

Figures 11 and 12 show the overhead involved in using MTS to generate VMM (MTS Step 2), and specialize the transformation (MTS Step 4), for the insurance and QoS configuration case studies, respectively. In general, the algorithms take slightly more time for QoS configuration than the insurance enterprise for the same number of variabilities, which is attributed to the larger combined size of the source and target metamodels of the former.

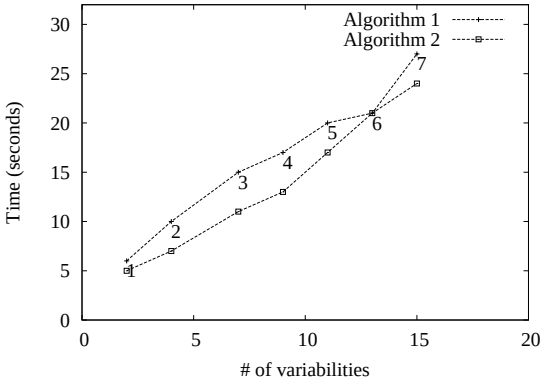

Figure 11: Insurance enterprise case study

For a variation of {Q=4, S=9} in the insurance enterprise case study, where Q and S denote the total variation in quantitative and

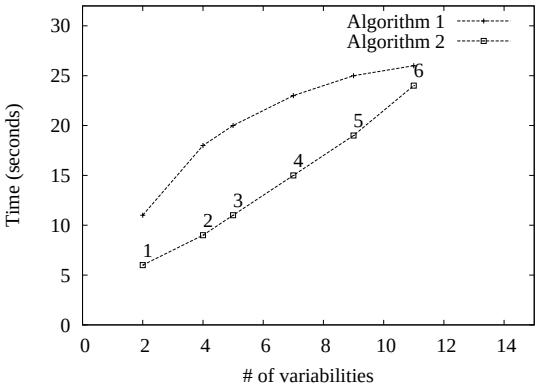

Figure 12: QoS configuration case study

structural variabilities, respectively, the time complexity of Algorithm 1 increased by 350% from an initial value of 6 seconds, while that of Algorithm 2 increased by 380% from an initial value of 5 seconds. For the QoS configuration case study, with a total variation of {Q=6, S=3}, the increase was ∼136% and ∼300%, for Algorithms 1 and 2, respectively.

Thus, if the new family variant is already subsumed by the notation as discussed in case 1 in Section 4.1, the developers incur an additional overhead in using MTS only for the first time when each of these algorithms have to be applied (*i.e.*, once for generating VMM, and once for creating temporary objects in the model transformation). Thus, the cost of using MTS is amortized over the total number of transformation runs during the development cycle of that application family. Since application development often undergoes multiple iterations of improvement, we believe the use of MTS is beneficial to the developers.

For the remaining cases, if the variabilities of the new variant are not captured by the existing specification, the two steps listed in Section 4.1 have to be executed once after modifications in the specification have been made according to the variabilities of the new variant.

5. RELATED WORK

Existing model transformation tools [4, 7, 17] 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 [5]. 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 [5] 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).

The Model-Driven Architecture (MDA) development process is centered around defining application platform-independent models and applying (typed, and attribute augmented) transformations to these models to obtain application platform-specific models. In the context of MDA, requirements and challenges in generating specialized transformations from generic transformations are discussed in [13].

Reflective Model-Driven Engineering [7] proposes a two-dimensional MDA process by expressing model transformations in a tool- or platform-independent way and transforming expressions into actual tool- or platform-specific model transformation expressions. There is return on investment (ROI) associated with developing and maintaining mappings from platform-independent transformations to platform-specific transformations in terms of reuse, composition, customization, and maintenance. Although reflective MDE focuses on having durable transformation expressions that naturally facilitate technological evolution and development of tool-agnostic transformation projects, the mappings still have to be evolved with a change in platform-specific technologies. In contrast, the MTS ideas are implemented directly in the underlying tool.

Asset variation points discussed in [16] deal with expressing variability in models of product lines [3]. 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. In recent work [1], authors discuss an approach to model refinement for product lines by adding model deltas through endogenous transformations. MTS has similar goals but focuses on refinements of model transformations.

An aspect-oriented approach to managing transformation variability is discussed in [23] that relies on capturing variability in terms of models and code generators. Another approach is model weaving [6], 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) that seamlessly integrates with an existing model transformation tool to support reusable model transformations for product lines. This approach is important to manage and improve the quality of product line software architectures at different stages of the development lifecycle where MDE techniques are used. MTS defines parameterized transformations to factor out the variabilities from commonalities in the transformation rules, and uses the notion of a generated variability metamodel to capture the variabilities in the transformation process across variants of a product line. MTS defines two higher order transformations to specialize the transformations for different variants. Currently, MTS is implemented for the GReAT model transformation tool and our ongoing work is investigating its feasibility for the ATL tool.

The results of evaluating MTS for our two cases studies indicate that developer efforts are minimized when new variants are added to the product line, which otherwise require transformation rules to be reinvented in traditional approaches. Although our case studies are small, we believe the results are a good indicator of the savings in general. MTS is available in open source as part of the CoSMIC MDE tool suite from `www.dre.vanderbilt.edu/cosmic`.

Discussion and Lessons Learned

The goals of MTS center around providing a capability that will significantly reduce the need to repeat and codify entire transfor-

mation rules for each variant of a product line. These benefits are more evident for transformation tools that employ visual languages to encode the transformation rules. By no means does MTS eliminate the need to encode the actual transformation rules; it simply lessens the need to redo them for every variant. We surmise that the benefits accrued using MTS will thus provide an indirect measure of the quality of software architecture. For example, MTS will be beneficial if model-driven engineering is heavily used in the development and maintenance of software architectures. The generative power of the MTS process derived through the higher order transformations enables a seamless composition of the rules representing the commonalities with the variabilities representing individual variants to result in a complete set of transformation rules. Our future work will investigate more user studies to understand the benefits. We are also working to showcase the MTS capabilities in ATL, which is a widely used model transformation tool.

7. REFERENCES

[1] M. Azanza, D. Batory, O. Díaz, and S. Trujillo. Domain-Specific Composition of Model Deltas. In L. Tratt and M. Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin / Heidelberg, 2010.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, USA, 2002.

[4] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In 17^{th} *IEEE International Conference on Automated Software Engineering*, pages 267–270, Edinburgh, UK, Sept. 2002. IEEE.

[5] J. S. Cuadrado and J. G. Molina. 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*, pages 168–182, Zurich, Switzerland, July 2008. Springer.

[6] J. Gray, T. Bapty, and S. Neema. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, pages 87–93, October 2001.

[7] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming, Special Second Issue on Experimental Software and Toolkits (EST)*, 72(1-2):31–39, 2008.

[8] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. 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*, pages 19–27, Huntsville, AL, USA, Apr. 2003. IEEE.

[9] A. Kavimandan and A. Gokhale. A Parameterized Model Transformations Approach for Automating Middleware QoS Configurations in Distributed Real-time and Embedded Systems. In *Proceedings of ASE Workshop on Automating Service Quality, (WRASQ 2007)*, Atlanta, GA, Nov. 2007.

[10] A. Kavimandan, R. Klemm, and A. Gokhale. Automated Context-sensitive Dialog Synthesis for Enterprise Workflows using Templatized Model Transformations. In *Proceedings of the 12th International Conference on Enterprise Computing (EDOC '08)*, pages 159–168, Munchen, Germany, Sept. 2008. IEEE.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[12] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon Transformation Language. In 1^{st} *International Conference on Theory and Practice of Model Transformations (ICMT 2008)*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60, Zurich, Switzerland, July 2008. Springer.

[13] J. Kovse. Generic Model-to-Model Transformations in MDA: Why and How? In 1^{st} *OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Nov. 2002.

[14] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[15] Object Management Group. *Real-time CORBA Specification*, 1.2 edition, Jan. 2005.

[16] S. Salicki and N. Farcet. 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*, pages 304–318, Bilbao, Spain, Oct. 2001. Springer.

[17] A. Schürr, A. J. Winter, and A. Zündorf. PROGRES: Language and Environment. In H. Ehrig, G. Engels, H. Kreowski, and G. Rozenberg, editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, pages 487–550. World Scientific Publishing Company, 1999.

[18] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[19] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[20] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *International Workshop on Application of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, pages 446–453, Charlottesville, VA, USA, Sept. 2003.

[21] F. Thomas, J. Delatour, F. Terrier, and S. Gérard. 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)*, pages 211–218, Orlando, FL, USA, May 2008.

[22] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In R. Paige, A. Hartman, and A. Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin / Heidelberg, 2009.

[23] M. Voelter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th Annual Software Product Line Conference (SPLC)*, pages 233–242, Kyoto, Japan, Sept. 2007.