Higher-Order Rewriting of Model-to-Text Templates for Integrating Domain-specific Modeling Languages

Bernhard Hoisl[1,2], Stefan Sobernig[1], and Mark Strembeck[1,2]
[1]*Institute for Information Systems and New Media, WU Vienna, Austria*
[2]*Secure Business Austria Research (SBA Research), Austria*
{*firstname.lastname*}*@wu.ac.at*

Keywords: Domain-specific modeling language, M2T, code generation, higher-order transformation, Eclipse Modeling Framework, Epsilon

Abstract: Domain-specific modeling languages (DSMLs) are commonly used in model-driven development projects. In this context, model-to-text (M2T) transformation templates generate source code from DSML models. When integrating two (or more) DSMLs, the reuse of such templates for the composed DSML would yield a number of benefits, such as, a reduced testing and maintenance effort. However, in order to reuse the original templates for an integrated DSML, potential syntactical mismatches between the templates and the integrated metamodel must be solved. This paper proposes a technology-independent approach to template rewriting based on higher-order model transformations to address such mismatches in an automated manner. By considering M2T generator templates as first-class models and by reusing transformation traces, our approach enables syntactical template rewriting. To demonstrate the feasibility of this rewriting technique, we built a prototype for Eclipse EMF and Epsilon.

1 INTRODUCTION

A domain-specific modeling language (DSML) provides modeling abstractions and notations to describe the concepts and activities in a business domain (e.g., health care or banking) or a technical domain (e.g., access control or workflow specification). DSMLs commonly focus on narrow domain fragments and system concerns only, such as schedule management for power suppliers or security properties of business process data (see, e.g., Spinellis 2001).

The benefits of DSMLs include reduced development times for DSML-based software products, an improved time-to-market, as well as reductions in development and delivery costs (e.g., for developer or customer trainings; see Bettin 2002). However, the development of a DSML and corresponding tool support most often requires substantial efforts that add to the overall costs of the underlying software development project (see, e.g., Krueger 1992; White et al. 2009). Thus, benefits of a domain-specific development approach only realize over time.

As a result, the costs of DSML development are strong drivers for reusing DSMLs as design artifacts, both during the life cycle of a single software product and for multiple software products (see, e.g., Krueger 1992; White et al. 2009). For a single software product, the development of a tailored DSML can be justified if the underlying software product is subject to frequent modifications or if the respective project demands for multiple and rapidly available prototypes. While a DSML would generate significantly more benefits if it was used in the development of different software products, this reuse is often barred by the narrow domains covered by DSMLs. In this situation, one option is to start from a joint metamodel and to refine this metamodel, the corresponding structural and behavioral semantics, as well as the DSML notation to cover an extended domain.

To develop a software product using two or more pre-existing DSMLs, with each DSML defining a subsystem of the product, integrating the corresponding DSMLs into a new consolidated DSML is an important design option (see, e.g., viewpoints in Vallecillo, 2010). Consider, for example, modeling the billing domain in a power supply company which covers company-specific accounting and branch-specific schedule management. Provided that compatible DSMLs for both tasks (i.e., accounting and schedule management) are available (e.g., based on the same metamodeling infrastructure), their integration is a viable strategy (e.g., via product line

techniques; White et al. 2009). Similarly, in security-critical domains integrating security-related DSMLs can support the systematic composition of different security concerns (see, e.g., Hoisl et al. 2012).

In addition to reusing the domain-specific language models through metamodel composition (see, e.g., Kalnina et al. 2010; Object Management Group 2008), the model-to-text (M2T) transformations available for the source DSMLs could be applied to models of the new DSML. This way, generator artifacts (such as platform code, configuration specifications, and deployment descriptors) could be reused for the new DSML.

From a DSML integration perspective, however, there are major barriers to reusing model transformations (Wimmer et al., 2012). One barrier for M2T generator templates, in particular, is the conformance relation between M2T transformations and a given metamodel. Consider the template sketch in Figure 1. The transformation template which accompanies class `B` contains a variable assignment expression, with the right-hand side calling on `propertyB` of an instance of `B` (assuming that `y` stores an instance of `B`). This way, the template is confined to a metamodel containing a corresponding class `B`.

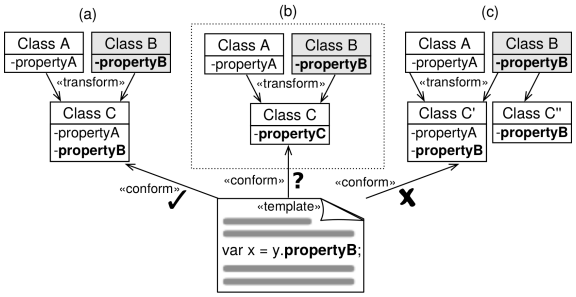

Figure 1: Syntactical metamodel conformance.

This conformance relation is affected by the metamodel composition applied for DSML integration. In the model composition scenarios (`a`) and (`b`) in Figure 1, class `B` is composed into new classes, using different composition operations. In scenario (`a`) the conformance relation is preserved by the composition operation (e.g., a full-property-preserving composition; Herrmann et al. 2007). Hence, the template still applies and can be reused over instances of class `C`. In scenario (`b`), however, the naming difference between `propertyB` and `propertyC`, while structurally equivalent, breaks the conformance relation. The template cannot be applied to instances of class `C`. Refactoring a template clone would be required, for example.

Existing reuse techniques for M2T generator templates fall short on addressing such syntactical mismatches. For example, language-level reuse techniques for M2T transformation languages only provide reuse capabilities at the block level (e.g. for entire template or function blocks) rather than at the expression level. Furthermore, most M2T transformation languages lack the ability to use generic transformations to abstract from different, but structurally compatible metamodels. Generic transformation techniques, as available for some model-to-model (M2M) transformation languages, require an upfront definition in terms of parametric templates and explicit bindings (see, e.g., Cuadrado et al. 2011). However, the demand for such an upfront definition also adds to the development overhead of a DSML. Adapter models mimic the metamodels of the source DSMLs. This, however, impedes the definition of additional M2T transformations that are specific to the integrated DSML. Yet, additional transformations are required to glue the different artifacts that are generated for the integrated DSML.

In this paper, we suggest an approach to overcome the above limitations of M2T transformation templates for DSML integration. Our *transformation rewriting technique* allows for the automated modification of transformation templates to fix important syntactical mismatches between templates and the composed DSML (Wimmer et al., 2010). In scenario (`b`) from Figure 1, for example, the naming difference would be tackled by rewriting the template expression based on tracing data of the class composition, using the new property name `propertyC`.

Our current prototype supports three higher-order rewriting operations (retyping, association retargeting, and property renaming). Note, however, that our approach allows for arbitrary rewriting operations. Semantic heterogeneity in metamodel-model relations (see, e.g., Wimmer et al. 2010) and types of syntactical heterogeneity between source and target metamodels which cannot be resolved in an automated manner are currently not addressed by our approach. This is, for instance, the case for m:n source/-target cardinality in scenario (`c`) in Figure 1. Our rewriting technique is applicable to M2T transformation languages which support a subset of the meta object facility (MOF) M2T transformation language (OMG MOFM2T; Object Management Group 2008), basic model transformation tracing, and higher-order transformations (HOTs; Tisi et al. 2009). All implementation artifacts are available from `http://nm.wu.ac.at/modsec`.

In Section 2, we give an overview on DSML integration, M2T transformations, and generator templates. Subsequently, we introduce our generic template rewriting approach in Section 3. In Section 4, we describe how our approach is then mapped to the

EMF and Epsilon infrastructures. A proof-of-concept implementation for the Epsilon Generation Language (EGL) is introduced. Section 5 gives an example of how our prototype environment can be applied. We discuss limitations as well as the pros and cons of our approach when compared to alternative techniques in Section 6. Section 7 provides an overview of related work and Section 8 concludes the paper.

2 TRANSFORMATIONS AND TRACES FOR DSML INTEGRATION

In general, the process of integrating two or more *source* DSMLs involves four major activities which may be repeated a number of times to derive an integrated *target* DSML (see Hoisl et al. 2012). The *language model composition* activity uses the language models (e.g., the MOF or Ecore metamodels including corresponding metamodel-level constraints) of the source DSMLs as input for the definition of a target metamodel. An important output of this activity is a composition specification that includes, for instance, correspondence rules and/or M2M transformations. In the *behavior composition* activity the behavioral semantics attached to the source metamodels are composed to match the target metamodel. Depending on the behavior definition corresponding composition operations are applied (such as M2M transformations or code-to-code transformations). The *concrete syntax composition* activity integrates the concrete syntaxes (e.g., textual, tree-based, tabular, or diagrammatic) of the source DSMLs. Finally, the *platform integration composition* activity integrates the software platforms of the source DSMLs. In particular, this activity integrates artifacts such as M2M and M2T transformations or model interpreters of the source DSMLs. The output of this activity is a set of platform integration specifications which conform to the target metamodel. Sometimes this composition step requires the generation of glue artifacts to realize a system-level composition (e.g., via pipelining, language extension, or front-end integration; Spinellis 2001).

The artifacts defined in the language model composition activity serve as the input for platform integration. In the remainder of this paper, we focus on M2T generator templates, M2M transformations, and transformation traces. Below, we discuss each of these artifact types in more detail.

M2T generator templates as models. Platform integration as described above (see Hoisl et al. 2012) includes the generation of platform-specific artifacts (such as, platform-specific source code or platform configuration and deployment documents; see Figure 2). The generation of these artifacts can be supported via M2T generators which receive a transformation definition and a set of source models as the input to produce a transformed representation of these models. For the remainder of this paper, we especially focus on template-based M2T generators and the corresponding generator templates (see, e.g., Czarnecki and Helsen 2006). In principle, a generator template consists of two kinds of code. On the one hand, there is template code to access and to select source model data by quantifying over the model structure that is specified in a metamodel (see also Figure 2). On the other hand, a template contains code to expand and to wrap the selected model data into string fragments.

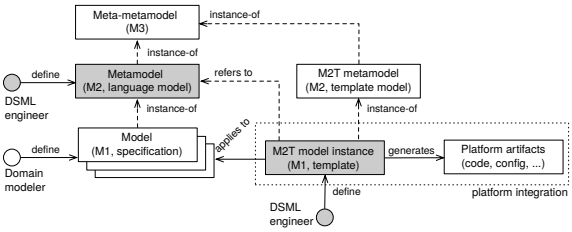

Figure 2: M2T template models.

Template-based M2T transformations are a widely supported platform integration technique in contemporary MDD tool chains, and a variety of template language implementations exist (such as, Eclipse Xpand, Xtend2, EGL, JET, or Acceleo; see also Czarnecki and Helsen 2006; Rose et al. 2012). For each of these languages, M2T generators and generator templates can be implemented in different ways. For example, one option is to use specification documents with a textual abstract syntax and a tree-based intermediate representation which is interpreted by the generator. In an alternative approach, we can use a DSL for M2T transformations that is embedded in a general-purpose scripting engine to realize generator templates via scripts. Such "generator template scripts" are then evaluated by the language interpreter (see, e.g., Zdun 2010).

To abstract from such implementation details and to benefit from generator templates as first-class modeling elements, our approach focuses on the model representations of generator templates. In other words, we consider generator templates as instances of a conceptual M2T template metamodel (see Figure 2). Rather than including all features that are available in different template languages, our approach requires only a generic subset of the features defined through the MOFM2T specification (Object Manage-

ment Group, 2008). In this way, the approach is portable to contemporary M2T template languages.

M2M transformations. In our approach we consider M2M transformations at the M1 and M2 modeling levels (see Figure 3). First, we compose the core language models of two (or more) source metamodels into a target metamodel. This composition is achieved via transformations that refer to the corresponding metamodel structures (M3). Second, M1 models of respective generator templates (see also Figure 2) are transformed into new M1 template models to adapt them to metamodel changes that result from the DSML integration. These M2M transformations are higher-order model transformations (HOTs): Transformations receiving input/output models which are themselves model representations of transformations; probably even expressed in the same transformation language (Tisi et al., 2010).

Note that the programming model that is used by a certain M2M transformation language (e.g. relational, operational mappings, hybrid) is transparent to our approach (see Czarnecki and Helsen 2006). Nevertheless, to demonstrate our approach we use hybrid transformation rules as supported by, for instance, the Atlas Transformation Language (ATL) and the Epsilon Transformation Language (ETL) in our examples.

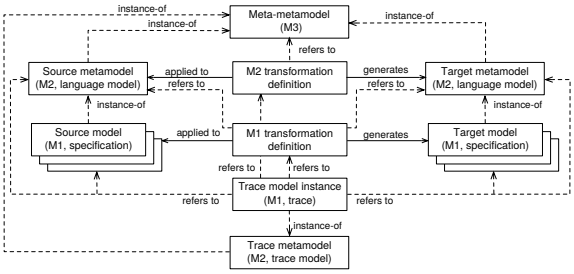

Figure 3: M2M transformations and traces.

Transformation traces. Even though our approach is generic and does not require a specific variant of M2M transformations, we assume that a transformation history is available. This history includes transformation traces that document the M2M transformations between language models (the M2 level in Figure 3). In particular, each transformation trace establishes a persistent link between a source and a target model element which are connected via a model transformation operation (merging, extension, renaming). Moreover, each transformation trace refers to a corresponding transformation rule. In order to introspect on these traces (for example, to identify the kind of transformation operation performed), transformation traces must be represented as first-class models. In other words, each trace must be an instance of a dedicated trace metamodel that complies with the transformed models at the M3 level (see Figure 3). Note, however, that we neither define restrictions on the time the traces are recorded (allowing, for example, partial evaluation of transformation rules, and runtime tracing) nor do we require a specific tracing engine: Built-in tracing, traces generated by transformation rules, as well as internal or external trace stores are supported.

3 TEMPLATE SYNTAX REWRITING

Before introducing our template rewriting approach, we must first review the types of potential mismatch between different DSML metamodels in more detail. We consider MOF-compliant metamodels. Regarding M2T template language concepts, we use the corresponding MOFM2T terminology for explanatory purposes (Object Management Group, 2008).

Figure 4 provides a sketch of DSML `A` and DSML `B` being composed into DSML `C` using an M2M transformation definition. The mismatch problem can then be restated as question: *How can we make the generator templates `A` and `B` apply to instances of the composed metamodel `C` rather than to instances of metamodels `A` and `B`, respectively?* To answer this question, we must establish some background: First, it is important to identify the types of structural differences encountered during metamodel composition. More specifically, we must collect the details about the structural differences and make them accessible to the template transformation. Then, the elements of a generator template which are affected by these structural differences must be highlighted. Finally, corresponding transformations of the generator templates must be defined. The objective of template rewriting is the transformation of the source templates (`A`, `B`) into derivatives (`A'`, `B'`) which refer to metamodel `C` directly (see Figure 4).

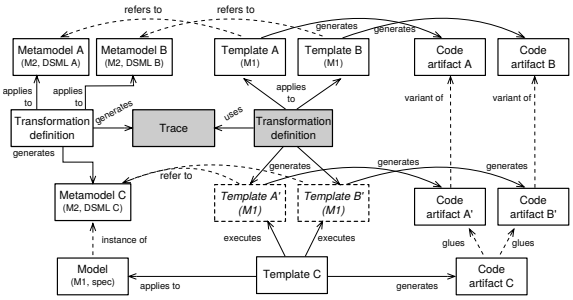

Figure 4: M2T template rewriting for DSML integration.

Metamodel composition. When composing the source metamodels (`A` and `B` in Figure 4), the DSML

engineer can choose from a variety of model composition operations when defining the transformation, only limited by the M2M transformation language's capacity (Vallecillo, 2010). These include *model mergers* using merge operators with different precedence and conflict resolution schemes, operating at different granularity levels (package, metaclass). In a *model extension*, subsets of either language model enter the new metamodel as disjoint sets to complement each other. *Model refinements* realize is-kind-of dependencies between all or selected elements of the source metamodels. *Model interfacing* involves introducing model elements specific to the new metamodel as a structural glue between elements merged from the source metamodels. A *hybrid composition* can involve any combination of the above operations.

In our approach, we restrict the composition operations so that structural semantics of the source metamodels are preserved. Thus, it is assured that the code artifacts generated based on the composed metamodel are semantically equivalent to the code artifacts generated with M2T templates of the individual source metamodels. The source-target cardinality can be either *1:1* or *n:1*: Either the element (class, attribute etc.) is preserved in the target model or a set of elements is merged into one new element.

Our template rewriting approach considers the effects of metamodel composition as changes between two model states across predefined model correspondences, rather than as a sequence of transformation operations. State-based differentiation refers to computing the changes between the source and target models after the completed composition, by contrasting the source and target metamodels at the M2 level. Listing 1 presents two examples of state-based differentiation to identify element name changes, defined as OCL expressions over two couples of source and target elements of two MOF-compliant metamodels. The two introspection definitions `isRenamedClass` and `isRenamedProperty` reflect important cases for M2T template rewriting, to be reconsidered later in this section.

Listing 1: State-based differentiation.

```
1  def: isRenamedClass(source : NamedElement,
2                      target : NamedElement) : Boolean
3      = source.oclIsKindOf(Class) and
4        target.oclIsKindOf(Class) and
5        target.name != source.name
6
7  def: isRenamedProperty(source : NamedElement,
8                         target : NamedElement) : Boolean
9      = source.oclIsKindOf(Property) and
10       target.oclIsKindOf(Property) and
11       target.name != source.name
```

A state-based technique has a number of advantages. To begin with, it is minimalistic and implementable for several M2M transformation engines. In addition, it turns our template rewriting approach agnostic about the actual composition operations used.

Trace models. To make state differences between source and target metamodels computable, the correspondences between metamodel elements established during the metamodel composition must be preserved at metamodel composition time (Paige et al., 2011). Alternative tracing techniques (implicit tracing, model annotation; see Drivalos et al. 2008) are not suitable for state-based differentiation. The tracing scheme can be achieved by storing the transformed element couples as instances of a simplistic trace model (see Figure 5). Instances of `Trace` refer to a `source` metamodel element and a `target` metamodel element which is paired during metamodel composition. In addition, a reference to the `Transformation` is stored along with the element couple. The `Trace` concept provides the context for the introspection expressions in Listing 1. This tracing scheme is suitable for detecting unsupported heterogeneity situations in the context of model composition (i.e., composition operations violating defined structural semantic preservation conditions, such as, m:n cardinality of source/target element mappings). Traces can be used to interrupt the rewriting process and to aid the debugging of allowed composition operations (Amar et al., 2008).

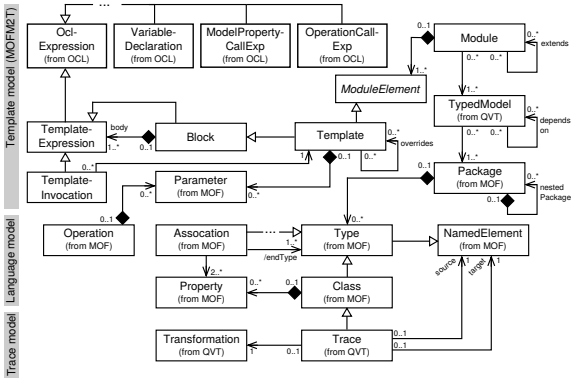

Figure 5: Excerpts from MOF and MOFM2T.

Template models. Based on the transformation data gained from introspecting the transformation traces, the templates are to be adapted (e.g., a template `A'` is to be derived from template `A` in Figure 4). To represent M2T generator templates as first-class models, we consider them instances of a subset of the MOFM2T reference language (see Figure 5).

In MOFM2T, transformations are structured in `Modules` (e.g., for namespace or public/private part definitions) which can contain a number of `Templates` (see Figure 5). A `Template` as a specialized `Block` contains string production expressions

(TemplateExpressions) with placeholders for data to be extracted from models. Templates can be invoked with Parameters and can return Parameters. They realize function-style language elements (Wimmer et al., 2012). A Template can override other templates, with the overriding template being invoked in place of the shadowed template. A TemplateExpression, among others, represents expressions for calls to model elements, for declaring and managing variables, for declaring control statements, and for expanding to strings. A TemplateInvocation specifies an invocation of a Template. As TemplateInvocation is a specializing classifier of TemplateExpression, Templates can be invoked from within other Templates. A TypedModel specifies an input model (a Package) to be referenced and accessed throughout the TemplateExpressions. The Types contained by the Package represent the domain of model element types available to the Templates. These type references are primary rewriting targets.

Template model transformations. Metamodel changes as identified by state-based differentiation (isRenamedClass and isRenamedProperty in Listing 1) affect the M2T generator templates syntactically in two ways. First, block expressions (variants of OclExpressions) maintain references to Types (as can be learned from Figure 5). This is the case for Parameters of Templates and for type-aware template expressions, in particular type annotations in VariableDeclarations and type annotations for parameters of operation calls (OperationCallExp). Second, navigating OclExpressions, such as, ModelPropertyCallExp may refer to renamed metamodel elements (e.g., Property). From these syntactical dependencies, three rewriting requirements follow for pairs of source and target metamodel elements, which are represented by a set of Trace instances (see Figure 5):

Retyping: References to a Type named after a renamed source Class must be replaced by the target Class name.

Association retargeting: When an Association between two Classes is redirected and receives another target (i.e., a Property owned by another Class), the corresponding endTypes must be modified. Thus, the return Type of the expression must be adapted and set to the new endType. Note that the name of a corresponding navigation reference (e.g., a property call to retrieve an element) in navigating OclExpressions is not affected.

Property renaming: A renamed Property (isRenamedProperty) affects navigating OclExpressions as the navigation path in these expressions changes. To rewrite these navigation paths according to the renaming, the property attributes of these OclExpressions are adjusted.

All three rewriting operations may occur repeatedly for identical pairs of transformed source and target metamodel element types, depending on the number of state-based differences computed from the set of Traces. Corresponding transformations must be defined in an M2M transformation definition which operates on the source templates (A, B) based on a set of Traces to produce syntactically adapted template model instances (A', B'; see Figure 4).

4 EPSILON/EMF PROTOTYPE

Based on our notion of template rewriting (see Section 3), we introduce a prototypical realization of this rewriting technique for the EMF and the Epsilon family of model transformation languages. In this technology projection, MOF-compliant DSML models are approximated by Ecore metamodels. To perform language model composition, as described in Section 3, we use transformation definitions expressed in Epsilon task languages. M2T generator templates are represented by EGL transformations. To obtain model representations of EGL templates, we map the respective MOFM2T concepts to their corresponding language concepts in the Epsilon language family. The Epsilon language runtime provides a built-in tracing facility for capturing transformation correspondences between Ecore metamodels as required for our rewriting approach (Kolovos et al., 2012).

Ecore metamodel composition. In the EMF/Epsilon toolkit, metamodel composition is divided into three tasks: (1) matching, (2) copying, and (3) merging metamodels. The first task (matching) is performed via the Epsilon Comparison Language (ECL) and has the source metamodels, as well as comparison rules provided by the DSML engineer as inputs. During copying, the unmodified metamodel elements (i.e., which do not match any comparison rule) are copied into the target metamodel of the composed DSML. This can be achieved using the ETL. The third step in an Epsilon-specific DSML composition applies merge rules defined by the DSML engineer on element triples of the source metamodels and the target metamodel. The output of this 3-pass transformation is a composed DSML metamodel and transformation traces, to be used for state-based differentiation (see Section 3).

The introspection operations isRenamedClass and isRenamedProperty translate into the Epsilon Object Language (EOL) equivalents for Ecore metamodels in Listing 2, to compute the state-based differ-

ences between the source and the target metamodels.

Listing 2: State-based differentiation for Ecore metamodels.

```
operation isRenamedClass(source : Ecore!EObject,
                         target : Ecore!EObject) : Boolean {
        return  source.isKindOf(Ecore!EClass) and
                target.isKindOf(Ecore!EClass) and
                target.name <> source.name;
        }

operation isRenamedProperty(source : Ecore!EObject,
                            target : Ecore!EObject) : Boolean {
        return  source.isKindOf(Ecore!EReference) and
                target.isKindOf(Ecore!EReference) and
                target.name <> source.name;
        }
```

Minimal Ecore trace model. To provide a model representation of transformation traces (as sketched in Figure 5 in Section 3), we realized a custom trace metamodel for our prototype. Listing 3 gives the metamodel definition. `CompositionLink` is the corresponding concept of `Trace` in Figure 5.

Listing 3: Trace metamodel in EMFatic textual syntax.

```
@namespace(uri="CompositionTrace", prefix="CompositionTrace")
package CompositionTrace;

class CompositionLink {
   ref EObject source;
   ref EObject target;
}
```

During the metamodel composition step (see above), transformation correspondences obtained from the Epsilon tracing machinery are stored as `CompositionLink` instances. Each `CompositionLink` stores a pair of source and target `EObjects` extracted from the Epsilon tracing sources (ECL match, ETL transformation, and EML merge traces). In this custom metamodel, references to the respective merge and transformation rules are omitted for brevity.

Ecore metamodel for EGL templates. EGL templates are natively processed by an ANTLR-based parser and transformed into an AST structure. Currently, Epsilon neither provides EMF metamodel representations of its language family, nor the tooling to perform round-tripping between ASTs and any model representation. Therefore, we extended an early prototype for EOL model representations (Wei, 2012) to cover EGL language concepts. We extracted the language abstractions from Kolovos et al. (2012) and by screening the Epsilon code base for missing details.

Figure 6 presents an excerpt of the extended Ecore metamodel. In this technology projection, the M2T template model concepts introduced in Figure 5 must be mapped to their Epsilon correspondences in Figure 6. Some Epsilon concepts such as `Module` and `Block` are directly compliant with MOFM2T. Nevertheless, some EGL concepts deviate from the MOFM2T metamodel structure: For example, Epsilon distinguishes between `Statements` and `Expressions`. In contrast, the MOFM2T specification summarizes text production rules of `Templates` as specialized expressions (`TemplateExpression`; Object Management Group 2008). Generally speaking, the complete EOL/EGL metamodel exceeds the domain coverage of the MOFM2T metamodel because Epsilon provides an integrated collection of several task-specific languages. For our prototype, however, only a subset of the Epsilon metamodel was relevant (see Figure 6). The resulting concept mapping used for our prototype is shown in Table 1.

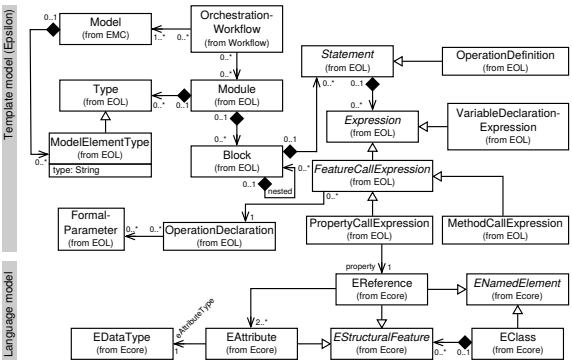

Figure 6: Excerpt from the Ecore metamodel for the Epsilon language family.

Table 1: Mappings between MOFM2T and Epsilon.

MOFM2T	Epsilon
`Association`	`EReference`
`Block`	`Block`
`Class`	`EClass`
`ModelPropertyCallExpr`	`PropertyCallExpression`
`Module`	`Module`
`NamedElement`	`ENamedElement`
`OperationCallExpr`	`MethodCallExpression`
`Parameter`	`FormalParameter`
`Property`	`EAttribute`
`Template`	`OperationDefinition`
`TemplateExpression`	`Statement`, `Expression`
`TemplateInvocation`	`FeatureCallExpression`
`Type`	`Type`, `EDataType`
`TypedModel`	`Model`, `ModelElementType`
`VariableDeclaration`	`VariableDeclaration-Expression`

ETL transformations on EGL template models. Finally, assuming the availability of model representations of source EGL templates and of transformation traces (`CompositionLink` instances), equivalents to the template-to-template transformations as defined in Section 3 must be defined based on the mappings shown in Table 1. Note that these transformations apply to any occurrence of EGL templates to be adapted. The definitions below are generic in this sense and must be bound to the concrete templates un-

der transformation. These M2M transformations are expressed in ETL.

Retyping, *association retargeting*: In Epsilon, the type of a model element acting as an input to a transformation is represented by the `ModelElementType` metaclass. To change the type reference owned by a template element (e.g., an expression) or to retarget an association (i.e., an `EReference` in Ecore), the respective `type` attribute of the corresponding `ModelElementType` element must be changed to match the name of the target metamodel metaclass (see Figure 6). The corresponding ETL rule is depicted in lines 10–20 of Listing 4.

Property renaming: An `EReference` in Ecore represents a navigation axis from one `EClass` to another by pairing the opposite metaclasses. Therefore, to reflect a renamed `EReference` in the template model, the `property` attribute of a `PropertyCallExpression` element must be adapted (see Figure 6). The ETL rule for this transformation is defined in lines 22–32 of Listing 4.

Listing 4: EGL snippet creating ETL rewrite rules.

```
1 for (l in links) {
2   var src : Ecore!EObject; var trgt : Ecore!EObject;
3   src = l.source;
4   trgt = l.target;
5   if (src.name <> trgt.name) {
6     etl = etl + TemplateFactory.prepare(renameElement(src, trgt)).
            process();
7   }
8 }
9
10 @template
11 operation renameElement(src : Ecore!EClass, trgt : Ecore!EClass) { %]
12   rule retype[%=src.name%]2[%=trgt.name%]
13     transform s : egl_in!ModelElementType
14     to t : egl_out!ModelElementType
15     extends Type
16     {
17       guard : s.type == "[%=src.name%]"
18       t.type = "[%=trgt.name%]";
19     }
20 [% }
21
22 @template
23 operation renameElement(src : Ecore!EReference, trgt : Ecore!EReference
        ) { %]
24   rule rename[%=src.name%]2[%=trgt.name%]
25     transform s : egl_in!PropertyCallExpression
26     to t : egl_out!PropertyCallExpression
27     extends FeatureCallExpression
28     {
29       guard : s.property == "[%=src.name%]"
30       t.property = "[%=trgt.name%]";
31     }
32 [% }
```

As mentioned above, the two M2M transformation rules must be instantiated for a concrete set of EGL template models (e.g., to reflect the concrete element names). As shown in Listing 4, the ETL rules are themselves generated by instantiating an M2T EGL template for a given set of transformation traces. For demonstration purposes, the top-level EGL script processes the available traces retrieved from the preceeding metamodel composition in lines 1–8, to dispatch to the expanded ETL transformation rules for each pair of source and target elements with name mismatches (see line 5 in Listing 4).

Epsilon composition and rewriting procedure. A process flow view of the overall composition and transformation steps is presented in Figure 7. This Eclipse-specific process flow realizes the abstracted scheme shown in Figure 4. Two activities must be performed as the prerequisites for applying the actual rewriting to the M2T EGL templates: 1) The DSML metamodel composition in three Epsilon-specific steps (matching, copying, merging) and 2) the transformation of M2T EGL templates into their model representations—that is, the instances of the metamodel depicted in Figure 6. The trace model generated during the metamodel composition and the instantiated ETL rewrite rules enter the actual template-to-template transformation along with the EGL template models. The rewritten EGL template models are finally serialized into EGL script representations to be applied to the composed metamodel at the end of this process (this last step is not shown in Figure 7).

This process flow can be automated in Epsilon by providing a specific build script which turns the flow into a sequence of Epsilon-specific Apache Ant tasks (Kolovos et al., 2008). Such an `OrchestrationWorkflow` defines the sequence of tasks, such as, `Model` loading or `Module` invocation (see Figure 6). Alternatively, such a process flow can be realized by instrumenting the Epsilon and EMF APIs in a piece of Java glue code.

5 AN INTEGRATION SCENARIO

In this section, we describe a composition scenario of two DSMLs to exemplify the integration process. We run through the whole process of applying one higher-order rewrite rule[1]. The first DSML models system audits (referred to as DSML `A`, hereafter) by providing abstractions for audit events and audit rules. The second DSML (DSML `B`, hereafter) allows for modeling generic state machines. The scenario integrates the two DSMLs into a composed DSML `C` capable of modeling a reactive distributed system with auditing support. Both DSMLs provide M2T generator templates written in EGL to generate Java code. The objective is to reuse these EGL templates for models of DSML `C` through syntax rewriting. For this scenario, we explain the application of a particular higher-order rewriting rule to a template specific to DSML `A`.

[1] All software artifacts as well as the complete example can be obtained from `http://nm.wu.ac.at/modsec`.

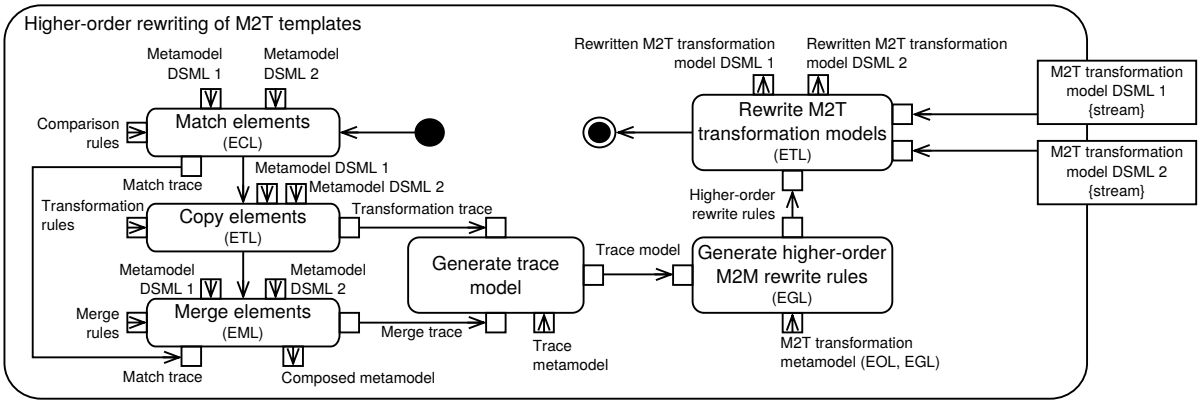

Figure 7: Process of higher-order rewriting of M2T templates.

In DSML A for system audits, an AuditRule subscribes to a Signal type and, when an AuditEvent is triggered, checks the corresponding Conditions against the published Signal occurrence (see Figure 8). If all Conditions evaluate to true, a notification action will be executed to perform audit-related tasks, such as generating an entry in an audit trail or notifying the system administrator (not displayed in Figure 8; for details see Hoisl and Strembeck 2012).

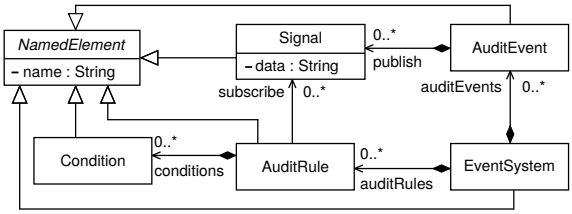

Figure 8: DSML A—auditing in event-based systems.

For DSML B, we have chosen a state/transition pattern (see Figure 9) for its communicability in an example. In a state machine, Transitions model the change from one State to another. A Transition is triggered by an Event.

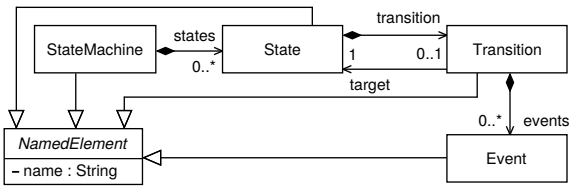

Figure 9: DSML B—a state/transition behavioral system.

DSML metamodel composition. In this step, we merge the AuditEvent element from DSML A and the Event element from DSML B into a unified AuditElement' element of the composed DSML C (see Figure 10). Thereby, we connect both DSMLs structurally by merging these two core concepts into one concept of DSML C. Otherwise, the metamodel composition preserves all structural semantics present in the source DSMLs (inheritance, attributes, references).

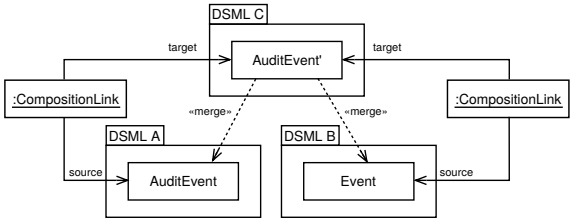

Figure 10: DSML composition via element merge.

This concept merge is defined by the ECL comparison rule shown in Listing 5. Therein, a match is defined iff the corresponding metamodel elements of the two DSMLs are named *AuditEvent* and *Event*, respectively (line 5)[2].

Listing 5: ECL comparison rule for AuditEvent'.

```
1 rule AuditEventandEvent2AuditEvent'
2   match l : EventSystem!EClass
3   with r : StateMachine!EClass {
4   compare :
5     l.name = 'AuditEvent' and r.name = 'Event'
6 }
```

For all elements missed by the rule in Listing 5, a direct copy operation into the target metamodel is defined via an ETL transformation (not shown). All elements matching the above ECL rule are processed by the merge operation in Listing 6. Therein, a new element name is constructed (line 5) and all properties, references, and inheritance relations (lines 6–7) from both the DSML A and the DSML B metamodels are transferred into the newly created element in the target metamodel. This preserves the n:1 source/target cardinality (see Section 3).

[2]Please note that we show only relevant code parts in the example listings (excerpts).

Listing 6: EML merge rule for `AuditEvent'`.

```
1 rule MergeAuditEvent
2   merge l : EventSystem!EClass
3   with r : StateMachine!EClass
4   into t : EventSystemStateMachine!EClass {
5       t.name = l.name + "'";
6       t.eStructuralFeatures ::= l.eStructuralFeatures + r.
            eStructuralFeatures;
7       t.eSuperTypes ::= l.eSuperTypes + r.eSuperTypes;
8 }
```

Ecore-based trace model. The merge and the transformation yield an instance of the trace metamodel (see Listing 3, Section 4). In Figure 10, the two resulting instances of `CompositionLink` are illustrated, recording pairs of transformation sources and transformation targets: (`AuditEvent`, `AuditEvent'`) and (`Event`, `AuditEvent'`).

Ecore-based template model. Listing 7 shows an example code snippet of an EGL template. For now, only line 1 is of interest: A loop is defined iterating over all `AuditEvents` in an `EventSystem`. The return type of the reference `EventSystem.auditEvents` is defined as `AuditEvent`. In the composed DSML `C`, the corresponding concept is `AuditEvent'`. To reuse this snippet for DSML `C`, the type annotation of the iterator variable `ae` must be modified to `AuditEvent'`.

Listing 7: EGL code snippet with typed iterator.

```
1 [% for (ae : AuditEvent in EventSystem.auditEvents) {
2   for (signal in ae.publish) {
3     out.println('private Signal ' + signal.name + ';');
4   }
5 } %]
```

For applying syntactical rewrite rules, the EGL template (Listing 7) needs to be transformed into its model representation. Listing 8 shows the corresponding instance model representation of line 1 of Listing 7 (simplified).

Listing 8: EGL model representation.

```
1 <statements xsi:type="dom:ForStatement">
2  <iterator name="ae">
3   <type xsi:type="dom:ModelElementType" type="AuditEvent"/>
4  </iterator>
5  <iterated xsi:type="dom:PropertyCallExpression" property="auditEvents
       ">
6   <target xsi:type="dom:NameExpression" name="EventSystem"/>
7  </iterated>
```

EGL template model transformation. The abstracted higher-order rewrite rules documented in Listing 4, Section 4, must be instantiated using the trace model shown in Figure 10. The ETL rewrite rule generated by this template instantiation for the DSML `A` element `AuditEvent` is reproduced in Listing 9. All other rewrite rules are omitted due to space limitations. The rule in Listing 9 resets the `type` properties of `ModeElementType` instances, which equal to `AuditEvent`, to the value `AuditEvent'`.

Listing 9: ETL higher-order rewrite rule.

```
1 rule renameAuditEvent2AuditEvent'
2   transform s : egl_in!ModelElementType
3   to t : egl_out!ModelElementType
4   extends Type {
5     guard : s.type == "AuditEvent"
6     t.type = "AuditEvent'";
7 }
```

Applying this rule to the EGL model as shown in Listing 8 results in an EGL model which is corrected for the changed type name. Line 3 of Listing 10 shows that the type of the iterator variable named `ae` was effectively changed to `AuditEvent'`.

Listing 10: Rewritten EGL model representation.

```
1 <statements xsi:type="dom:ForStatement">
2  <iterator name="ae">
3   <type xsi:type="dom:ModelElementType" type="AuditEvent'"/>
4  </iterator>
5  <iterated xsi:type="dom:PropertyCallExpression" property="auditEvents
       ">
6   <target xsi:type="dom:NameExpression" name="EventSystem"/>
7  </iterated>
```

To be able to execute the rewritten EGL template, in a last step, the EGL model representation in Listing 10 is serialized back into EGL template code (see Listing 11). Line 1 shows the changed type of the loop iterator named `ae`. This type conforms to the composed DSML metamodel `C` (see Figure 10). The rewritten EGL code template can be executed over models of DSML `C`.

Listing 11: EGL snippet with changed iterator type.

```
1 [% for (ae : AuditEvent' in EventSystem.auditEvents) {
2   for (signal in ae.publish) {
3     out.println('private Signal ' + signal.name + ';');
4   }
5 } %]
```

6 DISCUSSION

Our approach to rewriting M2T generator templates syntactically is motivated by examining barriers to reusing DSMLs, in general, and to reusing DSML-based M2T transformations for platform integration, in particular. An important barrier results from M2T transformation languages lacking the capacity of abstracting from certain structural conditions of a concrete metamodel (Wimmer et al., 2012).

While variants of template genericity (Cuadrado et al., 2011; Varró and Pataricza, 2004) help decouple from early bound references to concrete model element types, naming differences affecting navigational axes are not addressed, for example. Therefore, our approach can complement M2T template genericity. Given that generic transformations can also be implemented using HOTs on M2T templates, there is even a shared implementation vehicle.

In addition, our approach contributes to capturing M2T transformation logic independently from a particular transformation language or engine. This platform abstraction (Wimmer et al., 2012) contributes to the reusability of M2T transformations, as they can be migrated to another language environment. By providing a precise definition of our approach in terms of the MOFM2T specification (see Sections 3 and 4), we establish such an M2T platform abstraction.

Another barrier to M2T transformation reuse is the lack of contextual information about the conditions and requirements of reuse (Wimmer et al., 2012). While not fully elaborated in this paper, we enumerate working assumptions on the structural semantics of metamodel transformations (e.g., cardinality classes supported) in Section 3. These working assumptions can be formalized into executable pre- and post-conditions (e.g., OCL expressions) stored at the model-level. The conditions can then be evaluated based on the transformation traces generated during metamodel composition to establish whether the rewriting transformations are applicable.

One critique of using HOTs (Tisi et al., 2009) is that they expose the engineer to the internals of the transformation language (Tisi et al., 2010) and thus hinder reuse. In the case of M2T transformation models, this model complexity bears the risk of derailing the widely opaque text production expressions so that the platform artifacts are emitted malformed. In our approach, the M2T generator templates are represented by comparatively small metamodel domains (i.e., subsets of MOFM2T and the corresponding EGL mapping). On top, the HOTs remain completely hidden from the DSML engineer because they are themselves generated by template instantiation on the tracing data (see Section 3). This is a compromise balancing between automation and a limited support for metamodel heterogeneity.

The degree of DSML and M2T transformation reuse is directly related to the relative effort caused by the generative environment (transformation adjustments, manual configuration, automation of the generation tasks). To improve the reuse degree, this extra effort must be minimal. In our approach, most of the artifacts are only specified or generated once upon composing a DSML (e.g., the rewrite rules). Only when the source DSMLs are modified, the composing transformation definition must be updated. M2T transformations specific to the integrated DSML can vary independently from the generated M2T transformations. This allows for generating different kinds of patch code (pipelining or language extension; see Spinellis 2001).

7 RELATED WORK

The approach presented in this paper relates to existing work in two areas. First, we distinguish between three relevant language- and model-level **reuse techniques for generator templates**.

Higher-order transformation (HOT): Our syntax rewriting approach takes two M2T transformation models (EGL templates) as input and produces two modified M2T transformations (EGL templates) by applying an M2M transformation (via ETL) over these two transformation models. Hence, we apply HOTs for *transformation modifications* (Tisi et al., 2009). In recent years, a variety of alternative HOT application scenarios have attracted attention, including transformation analysis, transformation generation, and transformation composition (Tisi et al., 2009). In addition, language-level support for HOTs has been improved (Oldevik and Haugen, 2007; Tisi et al., 2010). However, related work has concentrated on specific transformation platforms (ATL; Tisi et al. 2010), rather than on HOTs in a technology-independent manner.

Generic templates: Generic templates abstract from the underlying metamodel and contain transformation rules which refer to abstracted metamodel types in terms of type variables (Cuadrado et al., 2011; Varró and Pataricza, 2004). The type variables are then late-bound to specific model element types at transformation runtime. This form of type parametrization must be employed by the DSML engineer right from the beginning to construct the generator templates in a reusable manner. Some approaches also require the explicit definition of bindings for type variables and structural adapters against a concrete metamodel to cope with types of structural heterogeneity in metamodels. This certainly adds to the initial effort of constructing the supporting transformations for a DSML. Our approach differs as we do not base the rewriting of templates on placeholder variables or adapters, but we rather extract the changes from a trace model. This offers the benefit of automation and of unanticipated reuse of generator templates. At the same time, our approach is limited in its expressiveness to handle metamodel heterogeneity (see Section 3). Because both approaches use HOTs as implementation vehicle, they can complement each other. Transformation genericity has not been documented for M2T generator templates so far.

Adapter models: To establish metamodel conformance for generator templates, another strategy is the use of intermediate models which adapt model accesses by the generator template to match the original metamodel structure. To mimic the original meta-

model, an adapter model consists of relational expressions which bind to the transformed metamodel and return model values according to the declared correspondences (Morin et al., 2009). Adapter models provide for unanticipated template reuse, however, as for M2T transformations the generated platform artifacts would not reflect the derived or newly introduced domain concepts. This increases the cognitive distance for the integrating DSML engineer. The generation of glue code using M2T transformations is restricted because concept correspondences between the integrated DSML and the source DSMLs can not be leveraged in code generation.

The second related area is the **encoding of tracing data** to be used in model transformations.

Modeled traces: Traces captured along the transformation process can be stored in a *separate* trace model or can be *attached* to the source/target model (e.g., via model annotations; Amar et al. 2008; Paige et al. 2011). The complexity of traces depends on their scope (e.g., only selected or all rules) and the tracing data needed. For our approach, it is sufficient to store trace links between source and target elements. Besides, the trace metamodel can be defined for more general or very specific purposes, such as our template rewriting scenario (Drivalos et al., 2008).

Delta models: Delta models are generated by comparing the input and output models of an M2M transformation (ex-post). The creation of delta models (or difference models) is comparable to diff tools for text artifacts. In contrast to model traces, delta models are an indirect method. Traces are not directly captured at transformation time, the actual transformation correspondences at the element level cannot be reconstructed. This black-box encoding of tracing data (Diskin et al., 2011) is not suitable for a rewriting approach which requires exact knowledge of source and target correspondences.

8 CONCLUSION

In this paper, we present an approach to rewriting M2T generator templates syntactically for reusing them in DSML integration. By considering M2T generator templates as first-class models and reusing M2M transformation traces, we developed a rewriting approach based on higher-order model transformations (HOTs). This approach is independent from a concrete transformation platform and the documentation in terms of the MOFM2T specification facilitates uptake in MOFM2T-compliant transformation languages. To demonstrate the feasibility of this rewriting technique, we provide a prototype implementation and a DSML integration example based on the Eclipse EMF project and the Epsilon language family. As a side product, we so contributed to constructing a metamodel for the M2T-specific parts of the Epsilon language infrastructure.

In our future work, we will extend our prototype to support a wider range of metamodel-level composition operations (e.g., extends, alternatives). Moreover, we will evaluate the applicability of our ideas to other transformation languages with HOT support (e.g., ATL).

ACKNOWLEDGEMENTS

This work has partly been funded by the Austrian Research Promotion Agency (FFG) of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) through the Competence Centers for Excellent Technologies (COMET K1) initiative and the FIT-IT program.

The authors would like to thank William Wei and Dimitris Kolovos for their valuable advice on various Epsilon issues.

References

Amar, B., Leblanc, H., and Coulette, B. (2008). A Traceability Engine Dedicated to Model Transformation for Software Engineering. In *Proc. of the 4th ECMDA Traceability Workshop*, volume WP09-09 of *CTIT Workshop Proceedings*, pages 7–16. Centre for Telematics and Information Technology (CTIT), University of Twente.

Bettin, J. (2002). Measuring the Potential of Domain-Specific Modelling Techniques. In *Proc. of the 2nd Domain-Specific Modelling Languages Workshop*, pages 39–44. Helsinki School of Economics.

Cuadrado, J. S., Guerra, E., and de Lara, J. (2011). Generic Model Transformations: "Write Once, Reuse Everywhere". In *Theory and Practice of Model Transformations*, volume 6707 of *LNCS*, pages 62–77. Springer.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646.

Diskin, Z., Xiong, Y., and Czarnecki, K. (2011). From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology*, 10:6:1–25.

Drivalos, N., Paige, R. F., Fernandes, K. J., and Kolovos, D. S. (2008). Towards Rigorously Defined Model-to-Model Traceability. In *Proc. of the 4th ECMDA Traceability Workshop*, volume WP09-09 of *CTIT Workshop Proceedings*, pages 17–26. Centre for Telemat-

ics and Information Technology (CTIT), University of Twente.

Herrmann, C., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S. (2007). An Algebraic View on the Semantics of Model Composition. In *Proc. of the 3rd European Conference on Model Driven Architecture—Foundations and Applications*, volume 4530 of *LNCS*, pages 99–113. Springer.

Hoisl, B. and Strembeck, M. (2012). A UML Extension for the Model-driven Specification of Audit Rules. In *Proc. of the 2nd International Workshop on Information Systems Security Engineering*, volume 112 of *LNBIP*, pages 16–30. Springer.

Hoisl, B., Strembeck, M., and Sobernig, S. (2012). Towards a Systematic Integration of MOF/UML-based Domain-specific Modeling Languages. In *Proc. of the 16th IASTED International Conference on Software Engineering and Applications*, pages 337–344. ACTA Press.

Kalnina, E., Kalnins, A., Celms, E., and Sostaks, A. (2010). Graphical Template Language for Transformation Synthesis. In *Software Language Engineering*, volume 5969 of *LNCS*, pages 244–253. Springer.

Kolovos, D., Rose, L., Paige, R., and García-Domínguez, A. (2012). The Epsilon Book. Available at: `http://www.eclipse.org/epsilon/doc/book/`. Last accessed: 30.11.2012.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2008). A Framework for Composing Modular and Interoperable Model Management Tasks. In *Proc. of the 1st ECMFA Workshop on Model Driven Tool and Process Integration*, pages 79–90. Frauenhofer IRB.

Krueger, C. W. (1992). Software Reuse. *ACM Computing Surveys*, 24(2):131–183.

Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., and Jézéquel, J.-M. (2009). Weaving Variability into Domain Metamodels. In *Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 690–705. Springer.

Object Management Group (2008). MOF Model To Text Transformation Language. Available at: `http://www.omg.org/spec/MOFM2T`. Version 1.0, formal/2008-01-16. Last accessed: 30.11.2012.

Oldevik, J. and Haugen, Ø. (2007). Higher-Order Transformations for Product Lines. In *Proc. of the 11th International Software Product Line Conference*, pages 243–254. IEEE Computer Society.

Paige, R., Drivalos, N., Kolovos, D., Fernandes, K., Power, C., Olsen, G., and Zschaler, S. (2011). Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. *Software and Systems Modeling*, 10:469–487.

Rose, L. M., Matragkas, N., Kolovos, D. S., and Paige, R. F. (2012). A Feature Model for Model-to-Text Transformation Languages. In *Proc. of the 2012 ICSE Workshop on Modeling in Software Engineering*, pages 57–63. IEEE Computer Society.

Spinellis, D. (2001). Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99.

Tisi, M., Cabot, J., and Jouault, F. (2010). Improving Higher-Order Transformations Support in ATL. In *Proc. of the 3rd International Conference on Theory and Practice of Model Transformations*, volume 6142 of *LNCS*, pages 215–229. Springer.

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. (2009). On the Use of Higher-Order Model Transformations. In *Proc. of the 5th European Conference on Model Driven Architecture—Foundations and Applications*, volume 5562 of *LNCS*, pages 18–33. Springer.

Vallecillo, A. (2010). On the Combination of Domain Specific Modeling Languages. In *Proc. of the 6th European Conference on Modelling Foundations and Applications*, volume 6138 of *LNCS*, pages 305–320. Springer.

Varró, D. and Pataricza, A. (2004). Generic and Meta-transformations for Model Transformation Engineering. In *Proc. of the 7th International UML Conference Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 290–304. Springer.

Wei, W. (2012). EpsilonLabs: Epsilon Static Analysis. Available at: `http://code.google.com/p/epsilonlabs/wiki/EpsilonStaticAnalysis`. Last accessed: 30.11.2012.

White, J., Hill, J. H., Gray, J., Tambe, S., Gokhale, A. S., and Schmidt, D. C. (2009). Improving Domain-Specific Language Reuse with Software Product Line Techniques. *IEEE Software*, 26(4):47–53.

Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W. (2010). Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In *Proc. of the 1st International Workshop on Model-Driven Interoperability*, pages 32–41. ACM.

Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W. (2012). Fact or Fiction—Reuse in Rule-Based Model-to-Model Transformation Languages. In *Proc. of the 2nd International Conference on Model Transformations*, volume 7307 of *LNCS*, pages 280–295. Springer.

Zdun, U. (2010). A DSL Toolkit for Deferring Architectural Decisions in DSL-based Software Design. *Information & Software Technology*, 52(7):733–748.