Model-to-Model Transformation Authoring: An MDA Approach

Maged E. Elaasar¹, Jeffrey M. Fischer²

¹ IBM Ottawa Software Lab
770 Palladium Drive, Kanata, ON, K2V 1C8, Canada
melaasar@ca.ibm.com

² IBM Beaverton Software Lab
15350 SW Koll Parkway, Beaverton, OR, 97006, USA
fischer@us.ibm.com

Abstract. Model-driven architecture (MDA) can dramatically increase the efficiency of software projects and the quality of produced software. Model-to-model transformation technologies play a key role in automating MDA as they help move data efficiently between different modeling languages, concerns and levels of abstraction. Technologies in this area have been subject of intense research and development efforts in recent years. One problem with research proposals is their lack of reference implementations to assess their effectiveness. On the other hand, industrial tools are still immature and generally lack theoretical foundation. This paper presents the Model Transformation Authoring Framework (MTAF), a new pragmatic approach to implementing transformations between MOF-based modeling languages including UML profiles. MTAF adopts an MDA approach to transformations by first specifying them using a high level declarative visual mapping language and then automatically generating efficient and extensible imperative implementations. MTAF is fully implemented and provides tools to automate its authoring process.

1 Introduction

MDA [1] is an approach to software development advocated by the Object Management Group (OMG). It provides a set of guidelines for structuring specifications in the form of models. The approach suggests describing a system's specifications using a platform independent model (PIM). A PIM is usually specified in a language defined using the Meta Object Facility (MOF) [2], a standard by the OMG for describing modeling languages. A prominent example of such a language is the Unified Modeling Language (UML) [3], which is well adopted by the software engineering community. Alternatives to UML also exist and are collectively referred to as Domain Specific Languages (DSL) [4], as they are more specialized and target certain domains. Once a system has been specified using a PIM, a platform is then chosen to realize it using specific implementation technologies, producing what is referred to as a platform specific model (PSM). A PSM can be specified using several domain specific languages or using a general purpose language like Java. The process of going
from a PIM to a PSM is called model-to-model transformation and can usually be automated.

Model transformation technologies play a key role in MDA automation. Technologies that help transform a model from one language to another, or from one level of abstraction to the next, are in demand by any organization looking to adopt MDA. This application of model transformations is referred to as forward engineering [5]. Other applications include reverse engineering of high level models from lower level models [1], creating query-based views of a system [6] and performing model refactoring [7]. A recent survey of transformation technologies is given in [8].

Model transformation technologies have been a subject of intense research in recent years as a result of the OMG’s call for standardization. This has resulted in many proposals and has eventually led to the adoption of the Query-View-Transformation (QVT) specifications [9]. However, a major problem with most research proposals is their lack of a reference implementation to properly assess their effectiveness. On the other hand, industrial tools for model-to-model transformations are starting to emerge but are still immature and generally lack theoretical foundation. More experience with practical applications of these tools is needed to assess their effectiveness [8].

Unfortunately, model-to-model transformations are inherently complex to author. The complexity derives from two sources: obviously the transformation problem domain but also the transformation solution domain. The larger and more complicated the modeling languages, the more effort it takes to author their transformations. However, the technology used in realizing a transformation can also have a profound effect on the authoring effort in terms of time and resources required.

The contribution of this paper is a new MDA approach to authoring model-to-model transformations that addresses the requirements of simplicity of specification, ease of adoption, extensibility and ease of integration with existing tools. The approach is implemented in a framework called the Model Transformation Authoring Framework (MTAF). A unique feature of the framework is that it applies, itself, the MDA principle of starting with a PIM then transforming it to a PSM. In this case, the PIM is an instance of a dedicated MOF-based DSL for specifying declarative mappings between other MOF-based modeling languages. The PSM is a best-practice imperative transformation implementation that is expressed using a well-established object-oriented transformation framework, which has been commercialized for a number of years now within IBM’s Rational Software Architect (RSA) [10] tool. MTAF is able to transform the declarative PIM to the imperative PSM using code generation.

MTAF’s adoption of MDA allows its transformations to enjoy the advantages of two worlds: a high level declarative mapping language that is simple to specify and has a visual notation and a lower level imperative implementation that uses a proven extensible design pattern in a popular object-oriented programming language. MTAF also has a standard implementation that is available in the new RSA 7.0 [10] tool. The implementation leverages open standards and open source technologies that are used by an increasing number of tool vendors, facilitating its adoption.

In addition to supporting the mapping of MOF-based modeling languages, MTAF also supports, in a uniform way, the mapping to and from UML profiles [3]. A profile is UML’s own extensibility mechanism and generally represents an alternative to implementing a full fledged DSL. There exist a large number of defined UML pro-
files as the technology has been around much longer than MOF-based DSLs have. A model-to-model transformation approach that ignores profiles is considered lacking.

In addition, MTAF provides transform authors with prepackaged solutions for recurring transformation design problems. These problems include traceability between source and target elements, incremental updating of target model, and the ability to mix imperative with declarative transformation logic. Having efficient and transparent solutions to these problems greatly simplifies the solution domain for transformation authors and allows them to concentrate on their problem domain.

The remainder of this paper is organized as follows: An overview of the used technologies by MTAF is given in section 2. Section 3 presents a quick overview of MTAF and its design choices. The details of MTAF, illustrated with a working example, are then given in section 4. In Section 5, more details of the working example are illustrated using the framework’s tooling. After that, a review of related works is shown in section 6. Section 7 concludes the paper and outlines future directions.

2 Used Technologies

As previously mentioned, MTAF references some open standards and leverages in its implementation both open source projects and other projects that are provided in the IBM suite of software tools. This section provides an overview of those used technologies and explains how MTAF relates to them.

2.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [5] is an open-source project that provides a model-driven development (MDD) solution in the Eclipse platform [11]. EMF provides the Ecore metamodel, a platform implementation of the Essential MOF (EMOF) specification [2]. Ecore is used to define metamodels for DSLs, which makes it a meta-metamodel. The MTAF framework supports the mapping of Ecore-based DSLs (section 4.3). A simplified subset of the Ecore metamodel is shown in Figure 1.

![Figure 1 A simplified subset of the Ecore metamodel](image)

An Ecore-based metamodel is organized in a nested collection of named EPackage, which own a collection of named EClassifiers. Two types of classifiers exist: a primitive EDataType and a complex EClass. Classes own a collection of named EStructuralFeatures that have multiplicities. Two types of features exist: a primitive EAttribute and a complex (containment or not) EReference.
2.2 UML2

UML2 [12] is an EMF-based implementation of the UML 2.x metamodel [2] for the Eclipse platform. UML is a popular general-purpose modeling language that can be customized through its own extensibility mechanism, called profiles. A profile is a lightweight mechanism to create a DSL as it is defined using a UML2 instance model. This can be contrasted to the heavyweight approach of defining a DSL using Ecore. The MTAF framework supports the mapping of UML2 profiles (section 4.4). A simplified sample of the UML2 metamodel related to profiles is shown in Figure 2.

![UML2 Metamodel Diagram](image)

**Figure 2** A simplified subset of the UML2 metamodel

Being a special type of Package, a Profile owns a collection of Classes in addition to a collection of Stereotypes. A stereotype extends one or more UML types (metaclasses) by defining extra attributes for them. The attributes are virtually added to the extended UML type when the stereotype is applied to an instance of that type.

2.3 RSx Transformation Service

RSx [13] is a family of software development tools from IBM Rational. It provides a framework called the RSx Transformation Service (RSxTS) that enables the definition of transformations between arbitrary domains. The framework provides a well-established abstract API that has been used for years to implement various transformations. MTAF provides a realization of that API (section 4.2) to support model-to-model transformations. An overview of the original pattern is shown in Figure 3.

![RSx Transformation Service API Diagram](image)

**Figure 3** A simplified subset of RSx Transformation Service API

An RSx transformation is organized into a nested collection of executable AbstractTransformElements. Three types of elements exist: AbstractRule, which provides some transformation logic, AbstractContentExtractor, which extracts source contents for further processing with its transform, and Transform.
which is a composite container of other transform elements. A transform element can have an optional accept condition that determines whether the element is allowed to execute or not. An extractor can also have a filter condition to screen content that is allowed to be processed by its transform. A transformation gets registered with the Transformation Service using AbstractTransformationProvider, which returns the root container of all transform elements.

Transform elements are executed with an instance of TransformContext, which contains a source, a target, a target container, in addition to a property map for sharing data across the various elements. A context instance is passed to a transform and is shared across the composed transform elements. When an extractor delegates to its transform, it creates a new context instance for every extracted content and sets that content as a new source and its target as a new target container.

2.4 IBM Common Mapping Framework

The Common Mapping Framework (CMF) [10] is an IBM technology that provides a mapping specification solution between arbitrary domains. CMF provides a dedicated Ecore-based Mapping DSL for specifying mappings between domains and an extensible graphical editor for it. The MTAF framework refines CMF by enabling Ecore-based DSLs as mapping domains and by supporting code generation of RSx transformations from mapping models between these domains (section 4.3). A simplified subset of the CMF mapping DSL is shown in Figure 4.

![Figure 4 A simplified subset of CMF Mapping DSL](image)

A mapping model is organized into a collection of nested Mappings. The top most container mapping is a MappingRoot, which specifies a namespace and designates a collection of input and output mapped domains. The root contains a collection of top-level named MappingDeclarations, each designating its own input and output types from the mapped domains. In every declaration, there exists a collection of individual mappings designating specific input and output features from the mapped types. In addition, a mapping can have refinements, which are extra semantics. Two types of refinements exist: a SubmapRefinement, which has a reference to a MappingDeclaration, essentially providing support for mapping composition, and a CodeRefinement, which has user-provided source code in some language. The code can either specify a condition as in ConditionRefinement or be general-purpose as in CustomRefinement.
2.5 RSx Compare and Merge Framework

The Compare and Merge Framework (C&M) [14] is another technology available in the RSx family of software development tools. The framework provides the ability to compare several Ecore-based DSL instance models and to merge them into one instance model. The framework provides generic support that can be customized for any particular DSL through an extensibility API. The compare feature of C&M calculates deltas, such as Add/Delete/Change/Move, between two models. Elements in those models are identified with a configurable matching strategy. One such strategy identifies objects with their unique global IDs. This strategy is often used to compare models that are derived from a common ancestor, since similar elements in both models would have the same ID. Another strategy identifies objects using their fully qualified containment path from the root. This last strategy is called the Fuse strategy and is usually used then the compared models do not share a common ancestor model.

The merge feature of the C&M merges the contents of input models into one consolidated output model. The merge runs in one of two modes: visual or silent. In the visual mode, C&M allows a user to visualize, inspect and select the deltas to apply. In the silent mode, the selection is made by a configurable application strategy. The MTAF framework uses C&M for implementing its target model update strategy (the merge strategy).

2.6 Java Emitter Templates

Java Emitter Templates (JET) [15] is an open-source Eclipse project that is used to implement code generators. The project allows the definition of textual templates that can be instantiated using data from an Ecore-based DSL instance model. When generating Java source code, JET uses a well-established text merge technology called JMerge. This technology allows generated Java code to have a special ‘@generated’ tag that is added to the comments of Java elements like classes, attributes and operations. The tag informs the code generator that the Java element is under its control, allowing it to be updated based on changes to the input model. On the other hand, when these tags are removed or the postfix ‘NOT’ is added to it, the code generator preserves the corresponding java elements on regeneration. The MTAF framework uses JET to code generate RSx transformations from its Ecore-based mapping models. The generated transformations are Java based and hence enjoy the advantages of the JMerge strategy. In essence, the transformation code becomes extensible as manual changes are preserved during code regeneration. This allows transformation authors to either change or augment the code with extra transformation logic that is not specifiable using the provided Mapping DSL.

3 Overview of Contribution

The contribution presented in this paper is MTAF, a new pragmatic approach to implementing model-to-model transformations. A high level overview of MTAF is giv-
en in section 3.1, while the various strategic choices that are made in its design are outlined in section 3.2. Further details about the framework are given in section 4.

### 3.1 High Level Overview

The MTAF framework provides the ability to author transformations between input and output Ecore-based DSLs (section 2.1). It also supports transformations involving UML2 profiles (section 2.2). The process of authoring and running such a transformation using MTAF is shown in Figure 5. The first step in the authoring process is to map the input and the output DSLs using a declarative mapping DSL provided by MTAF. The mapping DSL is a refinement to the one provided by CMF (section 2.4) and is detailed in section 4.3. The result of the first step is a mapping model that specifies how the input DSLs relate to the output DSLs. MTAF also extends the mapping graphical editor in CMF to simplify the specification of the mapping model.

Figure 5 MTAF Transform Authoring and Running Processes

The second step in the authoring process is to generate an imperative transformation implementation. MTAF provides a realization to the abstract transformation Java API provided by RxTS (section 2.3). The realization, detailed in section 4.2, provides an imperative implementation to the declarative specification in mapping models. The mapping between the specification and the corresponding implementation is specified by MTAF using JET templates (section 2.6). MTAF configures JET to use the JMerge strategy for code regeneration. This gives the ability to change the generated transformation or add to it without losing those changes on regeneration; a big advantage when that logic is not expressible in the declarative specification. Another advantage is the ability to debug the transformation using the implementation language debugger in addition to the high level trace functionality instrumented by RSxTS.

Once the transformation code is ready, it can be deployed at runtime. The first step of running a transformation is to configure it with various options including the input models and the post processing actions. The input models have to obviously be instances of the input DSLs. The second step actually invokes the transformation on the input models to produce output models that are instances of the output DSLs. MTAF generates new output models every time a transformation is run (a design choice to be discussed in section 3.2). The third step determines what to do with the output models, which depends on the configured post processing actions. One such action merges the output models with user specified target models. MTAF implements a merge strategy using the C&M framework (section 2.5), as detailed in section 4.2. Another action is to chain the output models as input models to another model-to-model tran-
formation. The last action provides the output models to JET to instantiate text templates.

3.2 Strategic Design Choices

All transformation approaches face design decisions and have to make choices. The MTAF framework is no different, as it makes some strategic design choices that are outlined in this section. The presentation of these choices is structured following a feature-model for transformation approaches that has been recently proposed [8]. The feature model specifies several design decisions for a transformation approach. The choices made by MTAF are discussed below:

**Specification.** The MTAF framework supports a hybrid declarative and imperative specification style. The declarative specification style is provided through a mapping DSL (section 4.3) that is used to specify mapping relationships between input and output domains. The resulting mapping model accepts as domains both Ecore-based DSLs and UML2 profiles (section 4.4). The imperative specification style is provided by first generating an equivalent imperative implementation (section 4.2) and second allowing the implementation to be augmented with additional imperative logic.

**Transformation Rules.** The declarative mapping rules provided by MTAF mainly have one input and one output domains. Although having more domains could make the mapping DSL more expressive, this choice seems non-limiting in most cases. One exception to be noted here is the CustomMapping, which is used to specify a user-defined imperative rule. This mapping is allowed to have any number input and output domains as the user is responsible for using them. There are three levels of nesting for mapping rules in MTAF specifying increasing levels of details. The top most level specifies mappings between the metamodels themselves, the second level specifies mappings between individual types in these metamodels and the third level specifies mappings between individual features in those types. MTAF also provides detailed semantics for every mapping including refinements like conditions, filters and user-defined code. The supported languages for query code are Java and (Object Constraint Language (OCL) [16], while the supported language for imperative code is only Java.

**Rule Organization.** MTAF allows physical organization of mapping rules in one or more mapping models. MTAF also currently allows one sort of reuse mechanism among mapping rules, which is aggregation. A second-level mapping between some types is allowed to delegate to another second-level mapping to map some features of those types. Recursion can be implemented when a mapping delegates to itself. Other forms of mapping reuse are not currently supported including generalization, refinement, genericity (templates) and grouping.

**Rule Application Control.** MTAF employs a depth first application strategy for mapping rules that is deterministic, once a starting point has been determined. Two reasons exist for this deterministic behavior: 1) mappings are nested in ordered collections and 2) there is no current support for mapping inheritance. The starting point for
the traversal strategy is determined based on the input model’s root element. Second-level mappings are traversed in sequence and the first one that accepts the root element as input becomes the starting point. Once there, the strategy moves on to its nested mappings in sequence and follows references to other second-level mappings in a depth first fashion. The lack of support of mapping inheritance means that the strategy always follows a concrete referenced mapping and not possibly one of its derivatives, like in other approaches (e.g. MTF [17]) where inheritance is supported.

**Source-Target Relationship.** MTAF supports the specification of mapping rules between input and output DSLs, which might be the same. Moreover, MTF does not support the specification of graph rewriting rules that are often specified on one DSL. This means that every time an MTAF transformation is run on input models it produces new output models even though the DSLs for both sides are the same. However, in-place transformations can be simulated if the transformation’s post processing action is configured to be the merge action. In this case, the transformation is configured with the input models as targets to merge into. This results in the transformation effectively merging the output models into the input models, simulating local changes.

**Incrementality.** MTAF does not currently support source incrementality, which is the ability to minimize the amount of input model to transform based on changes to that model. However, it supports target incrementality (or reapply), which is the ability to update target models based on changes to source models. The framework supports this functionality when the merge action is configured as a post processing action for a given transformation. Using the C&M framework, the merge action is able to detect change deltas between models at both sides, although it is not able to recognize which side has caused the changes. The reason for that inability is the lack of mechanism for marking manual changes to the target models (a topic of future research). The current support gives users a chance to review these deltas visually and decide for themselves whether to apply each of them or not.

**Directionality.** MTAF mapping rules are typically directional, specifying how their input domains map to their output domains usually with code refinements (e.g. conditions, filters, extractors...etc). However, simple rules that do not have code refinements are reversible. Therefore, mapping models that are made up only of simple mappings could possibly generate reversible transformations. The uncertainty is coming from the fact that different inputs may lead to the same output making the behavior non-deterministic in the reverse direction.

**Tracing.** The mapping DSL used by MTAF does not require the specification of traceability information between the input and output DSLs. Traceability information is needed by other approaches (e.g. QVT [9] and MTF [17]) to properly update target models by matching existing target elements to source elements. MTAF uses a different strategy for tracing. The strategy is implemented as a matching strategy that is configured in the merge post processing action and used by C&M framework (section 2.5). One strategy is defined for each DSL, which means the effort to define traceability information is done once, and not in every mapping like the other approaches; a
big advantage to the MTAF approach. A typically used strategy for a DSL is the *Fuse* strategy as the two merged models, the newly created output model and the target model, do not share a common ancestor.

4  Model Transformation Authoring Framework

The Model Transformation Authoring Framework (MTAF) is a new approach to authoring model-to-model transformations. A high level overview of the framework is given in section 3.1 and a review of its strategic design choices is given in section 3.2. In this section, the detailed design of MTAF is discussed in light of a working example, introduced in section 4.1. In particular, the framework’s imperative transformation API is given in section 4.2 and its declarative mapping DSL is presented in section 4.3 along with how the DSL maps to the API. Finally, section 4.4 discusses the UML2 profile mapping support.

4.1  Working Example

A working example is briefly introduced in this section and is illustrated further in section 5. The example demonstrates a typical transformation between a high level DSL, used to capture a design, and a low level DSL used for code generation. The example is an authoring of a transformation between two DSLs. An input DSL, called *BeanDesign*, is defined using the *UML2* metamodel [3] in addition to the *Bean* profile. The profile has two stereotypes: *BeanProject*, which applies to a *Package* and *Bean*, which applies to a *Class*. An output DSL, called *BeanCodeGenerator*, has its own Ecore-based metamodel. Figure 6 shows the input DSL on the left, as the relevant subset of UML2 augmented by the *Bean* profile, and shows the output DSL on the right.

![Diagram](image_url)

*Figure 6 The example input (left) and output (right) DSL metamodels*
4.2 Imperative Transformation API

The imperative transformation API provided by MTAF is a realization of the abstract one provided by RSxTS (section 2.3). More specifically, MTAF defines a collection of concrete rules, extractors and transforms that help in implementing a model-to-model transformation, as shown right section of Figure 7.

![Diagram](image)

Figure 7 A map from the mapping DSL (left) to the transformation API (right)

**Transforms.** The following extend Transform from RSxTS:

- **RootTransformation.** The top level or root transform that is provided for using a transformation provider. This transformation is configured with two sets of transformation items. The first set has an **InitializeRule**, which is used to initialize the transformation’s global variables (discussed later on), followed by a **MainTransform**, which is the entry point to the transformation traversal strategy, then finally a **FinalizeRule** that performs post-traversal processing. After executing this set, a transformation would have its output models ready for consumption by the second set, which has the post processing rules. That set can have one or more of the following: a **MergeRule** that merges the output models to some configured target models, a **JETRule** that passes the output models to a JET transform, and a **ChainRule** that chains the output models as input to another transformation.

- **MainTransform.** This transform is the entry point to the transformation’s traversal strategy. It owns all the transformation’s **MapTransforms**, which implement the various mappings between the input and output types, and acts as a switch statement for them. In other words, it iterates over them and delegates to the first nested transform that accepts the transformation’s source element as input.
MapTransform. This transform implements a mapping between a type from the input DSL and a type from the output DSL, where the types are represented by their corresponding EClasses. The transform owns a CreateRule, which is used to create an instance of the output EClass, as a first rule to ensure that the output instance is available as target to subsequently owned transform elements. Those elements can include instances of MoveRule, CustomRule, SubmapExtractor and CustomExtractor that implement the transform’s mapping rules between its input and output types.

Extractors. The following extend AbstractContentExtractor from RSxTS:

SubmapExtractor. This extractor represents a mapping between an input EReference and an output EReference. It is implemented to extract the contents of the input EReference from the owning transform’s source element. Those contents are then used as inputs to another MapTransform delegated to by this extractor. If the output EReference represents containment, the delegate transform is invoked with the contents immediately to create corresponding output elements. On the other hand, if it represents non-containment, the invocation is postponed to the FinalizeRule, as the intent is to set references to output elements, which might not have been created yet, in the output EReference of the owning transform’s target container. Furthermore, in addition to the ‘accept’ Condition and the ‘filter’ Condition, this extractor can be configured with a ‘choose’ Condition when the output EReference is non-containment. This helps set only output references that satisfies the condition. In addition, the extractor can also be configured with a CustomExtractor, providing user-defined logic for contents extraction. The logic is encoded as Java or OCL code that returns a collection.

Rules. The following extend AbstractRule from RSxTS:

CreateRule. This rule creates an instance of an output EClass and inserts it in an output EReference in the owning transform’s target container. It also stores the created instance as the owning transform’s new target. An ‘accept’ Condition is set on the rule to verify that a transformation’s source element conforms to the input EClass (type conformance).

MoveRule. This rule represents a mapping between an input EAttribute and an output EAttribute. It is implemented to copy the contents of the input EAttribute of the transform’s source element to sets it to the output EAttribute of the transform’s target element. Generally, the types of both attributes have to be compatible (e.g. both are boolean). An exception is when the output type is a String and the input is a type that can be serialized to a String.

CustomRule. This rule extends the MoveRule and differs from it by allowing user-provided logic for setting data to the transform’s target. The logic is encoded in Java.
**InitializeRule.** This rule is used to initialize the transform global variables defined as properties in the transform context. Two variables are defined: a ‘containment’ map from input to output elements and a ‘reference’ list of deferred non-containment references to be handled in **FinalizeRule.**

**FinalizeRule.** This rule is used to process the ‘reference’ list of deferred non-containment references, collected from *SubmapExtractors* throughout the transformation. For every such deferred reference, the rule looks up the output element that corresponds to each input element using the ‘containment’ map and then sets a reference to that element in the corresponding target container. Input elements that do not have corresponding output elements are reported. This situation can occur when those input elements have not been transformed, usually a developer’s mistake that can be corrected by making sure that those elements get covered by containment reference mappings in the scope of the transformation.

**MergeRule.** This rule is one of the post processing options for a transformation. The rule merges the transformation’s resulting output models into configured target models using the C&M framework (section 2.5). The intent is to synchronize the target models based on changes happening in the input models (as represented in the new output models). The matching strategy to use in the merge is the *Fuse* strategy by default, but it can be changed to one more suitable to the output DSL through a C&M extension point. The rule uses C&M to detect deltas between the two sets of models, selects the deltas to apply, and then applies them to the target models. Three merge modes are supported: *Visual* (allows users to visually inspect and select deltas to apply), *Silent* (selects all deltas by default), and *Automatic* (behaves like Silent if all deltas are ‘Add’ ones, otherwise behaves like Visual). The ability to visually inspect deltas is needed in case some deltas result from the target models being manually modified or modified by other tools since the last time the transformation has run. In which case, the user needs to decide whether to keep those changes or not.

**JETRule.** This rule is another post processing option for a transformation. It acts as a proxy to a JET transform (section 2.6) by providing the output models of a transformation as inputs to the JET transform. The rule is configured with the id of the JET transform. Having this rule as a post processing option allows the creation of a model to model transformation chain.

**ChainRule.** This rule is another post processing option for a transformation. It acts as a proxy to another model-to-model transformation (section 2.6) by providing the output models of a transformation as inputs to the delegate transformation. Having this rule as a post processing option allows the creation of a model transformation chain.

### 4.3 Declarative Mapping DSL

The declarative mapping DSL provided by MTAF is a refinement to the one provided by CMF (section 2.4). More specifically, MTAF defines the mapping domains to be
Ecore-based and adds new semantic refinements that are needed to implement model-to-model transformations. MTAF also provides a mapping from this refined DSL to the imperative transformation API defined in the previous section. This mapping, shown in Figure 7, is specified as a JET template (section 2.6) that is used to generate transformation implementation from mapping models. Using JET with its JMerge strategy allows the generated code to be augmented with more imperative transformation logic that cannot be expressed with the mapping model. In this subsection, the refined DSL and how it corresponds to the transformation API is presented.

**MappingRoot.** The top most level of nesting in a mapping model and contains a collection of MappingDeclarations. MTAF defines the input and output domains of a MappingRoot to be the root EPackages of the mapped DSLs. Upon code generation, a MappingRoot is mapped to a RootTransformation that owns an InitializeRule, a FinalizeRule and optionally one or more of MergeRule, JETRule and ChainRule. In the working example, a MappingRoot is created with the namespace ‘BeanDesignToBeanGeneratorModel’, where the input domain is the UML2 EPackage (the Bean profile is another input domain that is discussed in section 4.4) and the output domain is the BeanCodeGenerator EPackage.

**MappingDeclaration.** The second level of nesting in a mapping model and contains a collection of inner mappings. MTAF allows one input and one output domains for a MappingDeclaration and defines them to be EClasses from the mapped EPackages. Upon code generation, a MappingDeclaration is mapped to a MapTransform that owns a CreateRule, which in turn owns an ‘accept’ Condition. In the working example, several declarations are defined including: ‘PackageToProject’ from Package of UML2 (the beanProject Stereotype is another input domain that is discussed in section 4.4) to Project of BeanCodeGenerator, ‘ClassToBean’ from Class of UML2 (the bean Stereotype is another input domain that is discussed in section 4.4) to Bean of BeanCodeGenerator and ‘PropertyToAttribute’ from Property of UML2 to Attribute of BeanCodeGenerator.

**MoveMapping.** A simple mapping that is created within a MappingDeclaration. MTAF allows one input and one output domains for a MoveMapping and defines them to be compatible (same or assignable) EAttributes from the mapped EClasses. In addition, a MoveMapping is allowed to have an ‘accept’ ConditionRefinement, specified as a boolean expression in Java or OCL. The context of the expression is an instance of the owning declaration’s input type. Upon code generation, a MoveMapping is mapped to a MoveRule. If it has an ‘accept’ ConditionRefinement, then an ‘accept’ Condition is added to the MoveRule. In the working example, a MoveMapping is defined in the ‘ClassToBean’ MappingDeclaration from the name EAttribute of Class to the name EAttribute of Bean.

**CustomMapping.** A simple mapping that is created within a MappingDeclaration. MTAF allows any number of input and output domains for a CustomMapping and defines them to be EStructuralFeature from the mapped EClasses, or the EClasses themselves. A CustomMapping has a CustomRefinement, specified
with an imperative expression in Java. The context of the expression is both an instance of the owning declaration’s input type and an instance of its output type. Upon code generation, a **CustomMapping** is mapped to a **CustomRule**. In the working example, a **CustomMapping** is defined in the ‘**PropertyToAttribute**’ MappingDeclaration from the type **EReference of Property** to the type **EAttribute of Attribute**. The custom code is specified in Java as follows: ‘**Attribute_tgt.setType(Property_src.getType().getName());**’, where the ‘**Attribute_src**’ and ‘**Property_tgt**’ are the variable names of the two context instances.

**SubmapMapping**. A simple mapping that is created within a **MappingDeclaration**. **MTAF** allows one input and one output domains for a **SubmapMapping** and defines them to be **EReferences** from the mapped **EClasses**. In addition, a **SubmapMapping** has a **SubmapRefinement** with a reference to another **MappingDeclaration** (can recursively reference its owning **MappingDeclaration**). The mapping’s input type must be ‘coercible’ (same, sub or super type) to the input type of the referenced declaration. Also, the referenced declaration’s output type must be ‘assignable’ (same or sub type) to the output type of the mapping. In addition, a **SubmapMapping** can have an optional ‘accept’ **ConditionRefinement** to determine whether the mapping is allowed to run. Three additional optional refinements are introduced by **MTAF**: an ‘inputFilter’ **ConditionRefinement** to be checked on every element in the input **EReferences** to determine whether it gets processed, an ‘outputFilter’ **ConditionRefinement** to be checked on every candidate referenced output element to determine whether it gets set in the output **EReferences**, and an ‘extractor’ **CustomRefinement** to change the collection of candidate input elements to be processed.

The semantics of those refinements are given in the following table:

<table>
<thead>
<tr>
<th>Refinement</th>
<th>Language</th>
<th>Expression Type</th>
<th>Context Variable : Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept Condition</td>
<td>Java, OCL</td>
<td>boolean</td>
<td>&lt;Type&gt;_src : Input Type</td>
</tr>
<tr>
<td>Input Filter</td>
<td>Java, OCL</td>
<td>boolean</td>
<td>&lt;feature&gt;_src : Input Feature’s Type</td>
</tr>
<tr>
<td>Output Filter</td>
<td>Java, OCL</td>
<td>boolean</td>
<td>&lt;feature&gt;_tgt : Output Feature’s Type</td>
</tr>
<tr>
<td>Custom Extractor</td>
<td>Java</td>
<td>collection</td>
<td>&lt;Type&gt;_src : Input Type</td>
</tr>
</tbody>
</table>

Upon code generation, a **SubmapMapping** is mapped to a **SubmapExtractor**. If it has an ‘accept’ **ConditionRefinement**, then an ‘accept’ **Condition** is added to the **SubmapMapping**. If it has an ‘inputFilter’ **ConditionRefinement**, then a ‘filter’ **Condition** is added to the **SubmapMapping**. If it has an ‘outputFilter’ **ConditionRefinement**, then a ‘choose’ **Condition** is added to the **SubmapMapping**. Finally, if it has an ‘extractor’ **CustomRefinement**, then a **CustomExtractor** is added to the **SubmapMapping**. In the working example, a **SubmapMapping** is defined in the ‘**PackageToPackage**’ **MappingDeclaration** from the **packageElement EReference of Package** to the bean **EReference of Package referencing the **ClassToBean**’ **MappingDeclaration**. An ‘inputFilter’ refinement is added that **SubmapMapping**, with the following boolean OCL expression: ‘**packageElement_src.visibility = Visibility.Public**, where **packageElement_src** is context’s name.
4.4 UML2 Profile Mapping

In addition to mapping Ecore-based DSLs, the MTAF framework also supports the mapping of UML2 profiles. One or more UML2 Profiles are allowed to be input or output domains to a MappingRoot. However, since a profile is an extension to the UML2 metamodel, that metamodel has to also be specified as a domain in the same side as the Profile. In the working example, the Bean Profile is added as an input domain along side the UML2 metamodel.

Profiles have a collection of Classes and Stereotypes, which are allowed to be input and output domains of MappingDeclarations. While a profile Class can be the only input or output domain of a MappingDeclaration, a profile Stereotype can be specified as a domain only if an EClass from UML2, representing a type that is extended by this Stereotype, is also specified as a domain in the same side. In the working example, the beanProject Stereotype is specified as an input domain in the ‘PackageToProject’ MappingDeclaration, along side the Package EClass. Also, the bean Stereotype is specified as an input domain in the ‘ClassToBean’ MappingDeclaration, along side the Class EClass.

Profile Classes and Stereotypes have attributes of type Property, which are allowed to be input and output domains for the various kind of mappings nested in a MappingDeclaration in exactly the same way as EStructuralFeatures. In the working example, the ‘PackageToProject’ MappingDeclaration has a Move-Mapping from the basePackage Property of the beanProject Stereotype to the basePackage EAttribute of the Project EClass of BeanCodeGenerator.

In order to complete the UML profile support, MTAF also provides profile specific extensions to the transformation API presented in section 4.2. For example, a special subtype of the ‘accept’ Condition for MapTransforms is provided to additionally check for applied stereotypes on UML2 elements if the input domains of the transform include stereotypes. Also, a special subtype of CreateRule is provided to apply stereotypes to the created UML2 elements if the output domains include stereotypes. Finally, special subtypes of MoveRule and SubmapExtractor are provided to access the stereotype input and output attributes using UML2 specific API.

5 Illustrated Case Study

The case study presented in this section is an elaboration on the working example introduced in section 4.1 using the tooling provided by MTAF to automate the transformation authoring process (section 3.1). The transformation from the BeanDesign DSL to the BeanCodeGenerator DSL is a realization of one link in a chain of model transformations used in a hypothetical project adopting an MDA process, as shown in Figure 8. The BeanDesign model is a PSM that is created by another model-to-model transformation from a higher level PIM. Also, the BeanCodeGenerator model is used as input to a model-to-text transformation used to generate Java code.
5.1 Defining The Mapping Model

The first step in the authoring process of the BeanDesign to BeanCodeGenerator transformation is to specify a mapping between the two DSLs. MTAF provides an Eclipse-based mapping editor that is an extension of the one provided in CMF (section 2.4). The editor allows for choosing the input and output DSLs as domains of a MappingRoot. Then, it allows for graphically specifying MappingDeclarations between those DSLs. In the case study, the input DSLs are UML2 and the Bean profile, while the output DSL is BeanCodeGenerator. Five declarations are identified:

ModelToRoot. This mapping, shown in Figure 9, is from Model EClass of UML2 to Root EClass of BeanCodeGenerator. The input’s name is mapped to the output’s name using a CustomMapping with the custom Java code: 'Root_tgt.setName("Beans "+Model_src.getName());'. Also, the input’s packagedElement is mapped to the output’s project using a SubmapMapping with reference to the ‘PackageToProject’ declaration.

PackageToProject. This mapping, shown in Figure 10, is from Package EClass of UML2, with the beanProject Stereotype, to Project EClass of BeanCodeGenerator. The input’s name is mapped to the output’s name using a MoveMapping. The input’s name and packagedElement are mapped to the output’s basePackage using a CustomMapping with the custom Java code: 'Project_tgt.setName(Utils.getStereotypeValue(Package_src,"beanProject")+Package_src.getName());'. Finally, the input’s packagedElement is mapped to the output’s package using a SubmapMapping with reference to the ‘PackageToPackage’ declaration.
Figure 10 The PackageToProject Mapping Declaration

**PackageToPackage.** This mapping, shown in Figure 11, is from `Package EClass` of `UML2` to `Package EClass` of `BeanCodeGenerator`. The input’s name is mapped to the output’s name using a `MoveMapping`. The input’s `packageElement` is mapped to the output’s `package` using a `SubmapMapping` with reference to the ‘ClassToBean’ declaration. Also, the input’s `packageElement` is mapped to the output’s `export` using a `SubmapMapping` with reference to the ‘ClassToBean’ declaration. However, this last mapping has the following refinements: an ‘accept’ `Condition` with the OCL expression: `'Package_src.visibility = Visibility.Public'`, an ‘inputFilter’ with the OCL expression: `'packageElement_src.visibility = Visibility.Public'`, and an ‘outputFilter’ with the Java code: `return !export_tgt.getAttribute().isEmpty();`.

Figure 11 The PackageToPackage Mapping Declaration

**ClassToBean.** This mapping, shown in Figure 12, is from `Class EClass` of `UML2`, with the bean Stereotype, to `Bean EClass` of `BeanCodeGenerator`. The input’s name is mapped to the output’s name using a `MoveMapping`. The input’s `ownedAttribute` is mapped to the output’s `attribute` using a `SubmapMapping` with reference to the ‘PropertyToAttribute’ declaration. The last mapping has a custom ‘extractor’ with Java code: `return Util.getAllAttributes(Class_src);`.

Figure 12 The ClassToBean Mapping Declaration

**PropertyToAttribute.** This mapping, shown in Figure 13, is from `Property EClass` of `UML2` to `Attribute EClass` of `BeanCodeGenerator`. The input’s name is mapped to the output’s name using a `MoveMapping`. The input `EClass` is mapped to the output’s `kind` using a `CustomMapping` with the custom Java code: `Attribute_tgt.setKind (Property_src.getUpper()>1||Property_src.getUpper()==-1)?"LIST":"FIELD";`. The input type is mapped to the output’s type.
using a CustomMapping with the custom Java code: `Attribute_tgt.setType(Property_src.getType().getName());`.

Figure 13 The PropertyToAttribute MappingDeclaration

5.2 Generating The Transformation Implementation

The automation provided in MTAF enables incremental adding of mapping declarations and generating implementations for them. The implementations are in the form of Java source code that uses the MTAF provided transformation API. Every mapping declaration corresponds to a Java class named after the declaration. For the case study, a subset of the generated transformation is shown in Figure 14 (left). An extract of the generated code for the ‘PropertyToAttributeTransform’ is shown in Figure 14 (right).

Figure 14 An example of generated transformation (left) and sample of code (right)

5.3 Customizing The Generated Transformation

One of the distinguishing features of the MTAF authoring tool is the ease with which the generated transformations can be further customized and extended. Regions of
the generated source are marked with @generated tags (section 2.6). When these tags are present the code generator will recognize that it emitted the source for that region previously and so can freely discard it and emit new code as dictated by changes to the mapping model. When the tags are removed or a ‘NOT’ is added to them, the code generator preserves the section of code under control. An example is in Figure 15.

```java
/** @generated NOT */

protected void addGeneratedTransformElements(Registry registry) {
    add(getNameToName_Rule());
    add(getKind_Rule());
    add(getTypeToType_Rule());
    add(new DecorateAttributeRule()); // added rule
}
```

Figure 15 A ‘customized’ generated Java method

5.4 Deploying The Generated Transformation

The MTAF tooling provides the infrastructure required to deploy the new generated transformation as a plug-in in the Eclipse-based RSx family of modeling tools [10]. In addition to the generated transformation implementation, MTAF also generates code to register the new transformation with the RSx Transformation Service (section 2.3). At runtime, the transformation can be configured with various options including the input models and the post processing actions. If the action is to merge into target models, those models are also specified along with the merge mode: silent, visual or automatic. Figure 16 shows an example transformation configuration for the case study, where a UML2 model with the Bean profile applied (input.uml) is transformed and merged ‘visually’ to a BeanCodeGenerator target model (Output.bean).

![Figure 16 An example transformation configuration](image)

5.5 Running The Generated Transformation

MTAF allows for running a transformation using its configuration. The transformation takes the input models, transforms them and then post processes them as con-
figured. In the case study, the input model, shown in Figure 17 (left), is transformed into the output model, shown in Figure 17 (right) in UML notation.

The effect of the specified transformation logic can easily be seen when inspecting the output model. For example, the model’s Root gets the prefix ‘Bean’ appended to its name as dictated by the name-to-name CustomMapping in the ModelToRoot declaration. Also, all Attributes have their kind property set to ‘FIELD’ and are related to their owning Bean with a composition. Exceptions to this are the ones between Customer and each of Address and Purchase Order, where the kind is set to ‘LIST’ and are related to Bean by a reference association. This logic is specified in the Property-to-kind CustomMapping in the PropteryToAttribut declaration.

It is often the case that the intent of a transformation execution is to update an existing model rather than to create a new one. A typical use-case is when an input model changes and the transformation is rerun to update an existing target model. Similarly, generated models can also be manually edited and/or modified with other tools. In this case, when the transformation is rerun the user would want to decide which changes should be preserved and which get overridden. In the case study, the transformation is rerun after a change is done to the input model. More specifically, an existing attribute, address, was renamed to billing address and a new attribute, shipping address, was added. As the merge model as configured as visual, this causes the visual merge diagram to show up highlighting the detected deltas as in Figure 18.
6 Related Works

There has been a tremendous interest in model-to-model transformation technologies over the past few years; in particular since the OMG announced its call for standardization that has resulted in the adoption of the QVT specifications. Since then, a large number of approaches have been proposed and several were either prototyped or commercialized. However, there has not been much experience with applying most of those approaches in practice. A good categorization of those approaches, outlined in this section, is given in more details in [8]. Other surveys are given in [18] [19].

One category of approaches to model transformations encompasses pure imperative ones, where an object-oriented framework in some programming language (e.g. Java) offers basic infrastructure to help organize the transformation. That infrastructure consists mainly of APIs and a design pattern with several abstract classes to be implemented by concrete extenders. However, those extenders are left to implement some major strategies like traceability and application control on their own. An example of such an approach is the Transformation Service framework provided in the RSx family of modeling products [13]. The transformation imperative logic in RSx is done using the Java Metadata Interface (JMI) generated by the EMF framework for DSLs.

Another category of imperative approach uses higher level dedicated model manipulating languages (e.g. an imperative extension of OCL). A typical approach in this category allows a mapping rule to be specified as an operation on a model element. An operation declares its input type and output type and then the body has imperative logic that populates the attributes of the output type. An example of an approach in this category is QVT Operational language [9] [?] <put more here>.

A different category of approaches have the transformation specifications guided with the structure of the target DSL. Approaches in this category have two phases: one for creating the structure (containment hierarchy) of the target model and another
for initializing the attributes and connecting the cross-references. Transform authors do not need to explicitly specify scheduling or application strategies as they can automatically by inferred. One approach in this category is the Interactive Objects and Project Technology [21], which structures the transformation rules into a containment hierarchy based on the one implied by the target domain. There is one rule for every target element type and the application strategy creates the target model top down by traversing down the rule hierarchy. Another example is OptimalJ [20], where transformations are structured using source-to-target copy methods whose parameters’ types determine the input domains and whose return type determines the output domain. The framework is able to determine the based on method signatures the complete application strategy.

Another interesting category is template-based. Approaches in this category specify model transformations using output model templates with variable sections that contain logic to derive their values from input models. A common way to define those variable sections is through annotations expressed using the syntax of the target modeling language. For example, the Ecore language [5] allows for annotating models with user-defined annotations. The derivation logic within annotations can be expressed in any query-based language like OCL. An example of approaches in this category is given in [22], where a UML model can be templatized with specific stereotypes applied to UML elements. These stereotypes contain conditions, iterations and other expressions used when instantiating the template.

Furthermore, relational approaches represent another category, where the transformation logic is expressed declaratively using mathematical relationships. The main idea is to declaratively specify mappings between input and output domains using constraints expressed in some sort of mathematical logic. An inference engine is then provided to determine a strategy for applying these rules through a process of constraint solving. In contrast to imperative approaches, relational approaches are side-effect free and can usually support reversible transformations. Examples in this category include MTF [17] and QVT Relations [9].

Another main category of transformations use graph rewriting rules in their specifications. Structured models are treated as typed, attributed and labeled graphs of elements. Transformation rules in this category have two sides: a left hand side (LHS) that expresses a pattern to be matched on the input model, and a right hand side (RHS) with another pattern to replace that of the LHS. Patterns of both sides are usually specified in the concrete syntax of the modeling languages at each side; an obvious advantage to authors, who are usually familiar with those languages and not their abstract syntax (something often required in other approaches). Examples of approaches in this category include Fujaba [23] and Attributed Graph Grammar [24].

Hybrid approaches also exist. Those are approaches where multiple strategies are used. For example QVT has three different languages: Relational, Operational and Core. Another approach is ATL [25], where rules can be fully declarative, fully imperative or hybrid. The MTAF framework, presented in this paper, is definitely a hybrid approach as it uses a declarative style in a high level mapping model and an imperative style in a lower level transformation implementation. MTAF also provides a translation from the high level to the low level in an MDA style. A discussion of the strategic design choices of MTAF is given in section 3.2, using a feature model that is proposed in [8] to help compare MTAF to other approaches.
7 Conclusion and Future Works

Model-to-model transformation is a complex but integral process in MDA. The complexity derives from two sources: the complexity inherent in the transformation problem space and the complexity added by the transformation solution space. Technologies that help automate model-to-model transformations are needed for any project seriously adopting MDA. In this paper, a new pragmatic approach to model-to-model transformations has been presented. The approach is implemented in a framework called MTAF, which follows an MDA style for authoring transformations. MTAF allows transformations to be specified declaratively using a high level mapping DSL and then get converted to a lower level imperative implementation that can be augmented separately. This combination of styles caters to the needs of both simple and advanced transformations. The framework also provides strategies to address design problems common in this space. Furthermore, MTAF is able to author transformations for both MOF-based DSLs and UML profiles in the same fashion. The framework also provides tooling to improve authors’ productivity by automating all of its processes, including a graphical editor to specify mapping models, a code generator from the models to the imperative implementation and a wizard to configure and deploy transformations. The MTAF framework is available in the RSA 7.0 tool [10].

There are plenty of ways to take the MTAF framework forward including enriching the mapping semantics, enhancing the integration with other MDA processes and simplifying the authoring scenarios. More concretely, extensions to the mapping DSL, to improve the structure of mappings and promote reuse, can increase the efficiency of the transformation authoring process. For example, adding inheritance and refinement relationships to mapping declarations will likely yield the same efficiencies that have been seen in other languages. Also providing support for abstract declarations will leverage this even further. The incorporation of template or generic mappings is another promising area for investigation. Additionally, the ability to specify various rule multiplicities (e.g. 1-to-n and n-to-n) can simplify the mapping specifications. Other extensions to the mapping DSL can also improve productivity. For example, common usage scenarios that require the development of custom mappings could be replaced with declarative ones (e.g. concatenate and other common string operations, select and other common collection operations, and transformations calls and chains).

Other ways to improve MTAF is to fully support reversible mappings. Because of their declarative nature many mappings will be immediately reversible but others will not. For example, mappings are not symmetric with respect to conditions and filters, i.e., different conditions and filters might need to be specified for each direction. Alternatively, mapping rules can specify other as their reverse.

Another possibly is to improve MTAF’s target incrementality by supporting the ability to mark changes in target models so that they get preserved when the transformation is reapplied. Source incrementality is another possibility, which is about minimizing the amount of input model to transform based on changes to that model.

One more interesting point of future research is the ability to discover candidate mapping relationships between input and output DSLs. This can jump start the author-
ing process by suggesting possible mapping declarations and possible feature level mappings in the mapping editor.

Another promising direction is the ability to support visual debugging of mapping specifications. Declarative mappings, as opposed to imperative ones, can be hard to debug. Therefore, having a visual debugger that can animate the traversal strategy of mappings, while giving the ability to stop (have a break point) at a particular one, can have a dramatic effect on the efficiency of the authoring process.

References

[21] OMG. Interactive Objects and Project Technology. MOF Query/Views/Transformations, OMG Document ad/03-08-11, ad/03-08-12, and ad/03-08-13 (revised submission, 2003).


