"Weaving" MTL Model Transformations

Raul Silaghi, Frédéric Fondement, Alfred Strohmeier

Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne EPFL, Switzerland

E-mail: {Raul.Silaghi, Frederic.Fondement, Alfred.Strohmeier}@epfl.ch

Abstract. Model transformations are the core of the MDA-approach to software development. As specified by the OMG, model transformations should act on any kind of model of any kind of metamodel, which implies the possible "reflective" use of model transformations, i.e., model transformations acting on model transformations. However, this would still require transformation developers to be familiar with the metamodel of the transformation language itself, which is not always the case. In order to overcome such a frustrating impediment for the MTL language, inspired by AOP approaches, we have designed and implemented an MTL *weaver* that modifies MTL transformations according to some *weaving behavior*, which is specified as special MTL transformations, called *MTL-aspects*, using an AOP-like extension to the MTL language. Both the weaver and the language extension are presented in this paper, and an example is used to show how transformation developers can take advantage of the proposed language extension constructs in order to write "reflective" model transformations in MTL without requiring any previous knowledge of the MTL metamodel itself.

Keywords. Model-Driven Architecture, MDA, Model Transformations, Aspect-Oriented Programming, AOP.

1 Introduction

To escape from the proliferation of middleware infrastructures and to avoid drowning in their implementation complexities, models are proposed as a far more accessible and easier means for developers to build, extend, and evaluate applications than working directly at code level. The Model Driven Architecture (MDA) [1][2], an Object Management Group (OMG) [3] initiative, promotes the separation of concerns between two modeling dimensions: one focusing on the business functionality (resulting in *Platform Independent Models – PIMs*), and another one focusing on the implementation of that functionality on a specific middleware platform (resulting in *Platform Specific Models – PSMs*). Since in the context of this paper we consider the *middleware* to be our MDA platform, further on we will directly refer to the middleware instead of the general concept of (MDA) platform.

Besides the obvious importance of PIMs and PSMs in MDA, *model transformations* are undoubtedly the key technology in the realization of the MDA vision. Among other usages, model transformations are the ones responsible for refining PIMs into PSMs (or abstracting PSMs into PIMs) and mapping PSMs to concrete middleware-based implementations, providing thus an elegant approach to adapt PIMs to the peculiarities of the new middleware infrastructures that do not cease to appear.

Unfortunately, there is not yet a standard language for defining model transformations. To fill this gap, OMG has issued a Request for Proposal called MOF 2.0 Query/ Views/Transformations RFP [4], which has been answered by eight different initial submissions, five revised submissions, and finally two "joint" revised submissions.

A clear requirement in OMG's RFP was (and still is) that model transformations should be able to act on *any kind of model of any kind of metamodel*. Since model transformations are at the same time models compliant with the metamodel of the transformation language, model transformations should be able to *transform* other model transformations independently of their metamodels. As a consequence, all currently existing model transformation languages (to our knowledge) implement such a "reflective" behavior. However, the "reflective" use of model transformations is not trivial.

Typically, writing model transformations for driving the development process of domain-specific applications requires the transformation developer to be familiar with the metamodel of that specific domain and with the syntax of the model transformation language used – and no more than that. As a consequence, many transformation developers are not at all familiar with the metamodel of the transformation language itself, and thus they are not capable of writing "reflective" model transformations, i.e., model transformations that transform already existing model transformations.

In order to overcome this frustrating impediment for the INRIA Model Transformation Language (MTL) [5], we present in this paper a solution inspired by Aspect-Oriented Programming (AOP) [6] approaches. We have designed and implemented an MTL *weaver* that modifies MTL transformations according to some *weaving behavior* that is specified as a special kind of MTL transformations, called *MTL-aspects*. The MTL transformation produced by the MTL weaver can be immediately used for refining application models.

As in the case of AspectJ [7][8], which is an aspect-oriented extension to Java, the syntax defining the weaving behavior in MTL-aspects is a small AOP-like extension to the MTL language itself. In this way, relying on a few high-level AOP-like but MTL-based constructs for defining the weaving behavior, average MTL transformation developers should not have any problems using this MTL extension straightforwardly for defining their "reflective" model transformations.

The rest of the paper is structured as follows: Section 2 provides the motivation of this work by discussing concrete examples where such a weaving functionality is useful; Section 3 gives a concise overview of the MTL model transformation language; Section 4 introduces the MTL weaver, describes the AOP-like extension to MTL for defining the weaving behavior in MTL-aspects, and presents an example showing both the input and the output of a concrete weaving; Section 5 draws some conclusions and presents future work directions.

2 Motivation

In the context of our global research interests, we present in this section how currently applied MTL transformations benefit from the weaving support provided by the MTL weaver, promoting the separation of concerns paradigm even at level of model transformations.

Separation of concerns [9] and modularization are fundamental techniques of software engineering. Decomposing software into smaller, more manageable and compre-

hensible parts, each of which encapsulating and addressing a particular area of interest, called a *concern*, is a well-proven method towards developing applications that are easy to configure, adapt, or extend according to changes in the requirements specification.

Middleware is an essential element in large distributed systems such as those that support enterprise applications, requiring multiple heterogeneous components to interoperate. Moreover, middleware, like software in general, is subject to concerns. Several concern-dimensions about middleware can be grouped into a category called Middleware Services, as the middleware addresses specific concerns of a system, such as distribution, concurrency, security, or transactions. An extended list of categories that group several middleware-specific concern-dimensions can be found in [10].

In order to address such middleware services in an MDA fashion and following the separation of concerns principles, we defined the Enterprise Fondue software development method [11]. In the context of Enterprise Fondue we defined several MDA-oriented UML profiles that address middleware-specific concerns at different levels of abstraction. MTL transformations are used to incrementally refine existing design models (within the same or between different MDA-levels) along middleware-specific concern-dimensions and according to the UML profiles defined. A complete example of applying the Enterprise Fondue method for addressing the distribution concern in the concrete case of the CORBA [12] technology was presented in [13]. The *UML-D Profiles* proposed in [13] address the distribution concern at three different MDA-levels of abstraction: *platform-independent* level (the `DistributionProfile`), *abstract realization* level (the `AbstractDistributionRealizationProfile`), and *concrete realization* level (the `CORBADistributionRealizationProfile`).

Based on the support provided by the MTL weaver, we refactored the MTL transformation that refined application designs in the context of the Enterprise Fondue method along the distribution concern-dimension and according to the `DistributionProfile`. Out of one big model transformation that performed the entire refinement, we have now one standard MTL transformation that performs the `copy` of an input model to an output model, both models being compliant with the same UML metamodel, and a very small MTL-aspect that defines the weaving behavior according to the `DistributionProfile` that has to be applied. Both the `MTL-Copy` transformation and the `MTL1-D-Aspect` are now fully separated as they should be, since they address totally different concerns. Figure 1 a sketches the refinement process in the presence of the `MTL1-D-Aspect`, or more general in the presence of MTL-aspects. Its name, `MTL1-D-Aspect`, was chosen in accordance with the `MTL1-D` transformation defined in [13] for refining along the distribution concern-dimension. The `MTL-Distribution-Copy` transformation is the result produced by the weaver when modifying the `MTL-Copy` transformation according to the weaving directives defined in the `MTL1-D-Aspect`.

A more complex example is shown in Figure 1 b, where the metamodel of the input and output models changes. In this example we move from a UML model to a Java model ready to be mapped to concrete Java implementation. Considering as input the output model of the previous refinement process, we refine this time along the RMI-technology [14] and Java-language concern-dimensions as defined in the context of the Enterprise Fondue method. While the `MTL-UML2Java` deals with transforming any UML model to its correspondent Java model (relying on their respective metamodels),

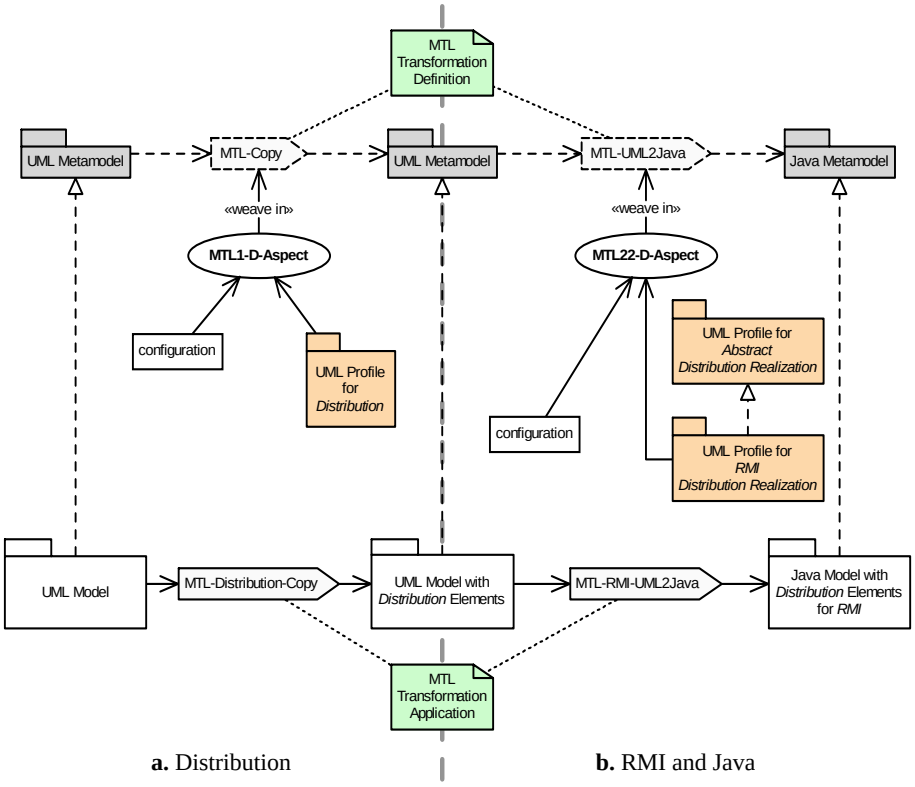

Fig. 1. Refining along the Distribution, RMI-Technology, and Java-Language Concern-Dimensions

the `MTL22-D-Aspect` addresses how distribution specific elements in the UML model are transformed into their Java model counterparts when employing RMI as their implementation technology. For instance, interfaces marked as «`Distributed`» in the UML model will extend `java.rmi.Remote` in the Java model; similarly, the class of the object marked as «`Servant`» will extend `java.rmi.UnicastRemoteObject` in the Java model, and so on. Once again, the name, `MTL22-D-Aspect`, was chosen in accordance with the `MTL22-D` transformation defined in [13] even though we considered this time another technology, i.e., we have chosen RMI instead of CORBA. The `MTL-RMI-UML2Java` transformation is the result produced by the weaver when modifying the `MTL-UML2Java` transformation according to the weaving directives defined in the `MTL22-D-Aspect`.

As can be seen in Figure 1, the support provided by the MTL weaver has enabled us to modularize the different concerns in stand-alone units of encapsulation represented by MTL-aspects. In this way, we give transformation developers not only the possibility, but also the means to rely on the well-proven power of separation of concerns even at model transformation level. Moreover, the size of such MTL-aspects is very much reduced, compared to their corresponding implementation in the initial MTL transformations, since they rely on the MTL weaver which is now the one carrying all

the burden of the weaving. The example presented in Figure 1 a is reconsidered further on in section 4.2 where we discuss in more details its complete implementation.

Besides encapsulating middleware-specific concerns into MTL-aspects as presented in this section, the number of possible usages of such MTL-aspects is unlimited since the support provided by the MTL language enables us to implement almost anything in the MTL weaver, and thus, the expressiveness power that could be provided to transformation developers through the MTL extension syntax may be very broad, covering all possible and impossible needs that developers may think of.

3 The Model Transformation Language (MTL)

This section provides a concise overview of the MTL transformation language focusing mainly on the concepts that are relevant in the context of this paper. Readers that are familiar with the MTL language may skip this section and jump directly to section 4 which presents the MTL weaver.

Many different solutions have been proposed for model transformation languages, and therefore it is a hard task to merge all ideas into one future standard. Unfortunately, standards of the future are not solutions to problems of today. The idea of the INRIA Model Transformation Language (MTL) [5] is to provide *all* model transformation facilities, including the possibility to transform MTL transformations. This makes it possible for the future QVT language standard to be mapped to an MTL transformation by means of an MTL transformation. This *pivot* approach has already been validated. The MTL itself is developed according to a bootstrapped approach: a simple language, called BasicMTL [15], provides the most important facilities, such as classes or attributes, and new facilities are added by extending the abstract syntax and by making a transformation from the extended to the initial syntax, always relying in this way on the small "kernel" of BasicMTL. As an example, the associations between classes have been added following such an approach. Moreover, the plan is to transform, or in other words, to compile the Atlas Transformation Language [16] into an MTL transformation. As a conclusion, MTL aims more at *motorizing* model transformations than proposing a new standard.

As suggested just before, MTL is an object-oriented imperative language for model transformations. Therefore, MTL transformations are defined as programs in terms of classes, methods, attributes, etc. In order not to confuse these MTL constructs with the ones that the manipulated model may contain, we will further on refer to them as MTL classes, MTL methods, MTL attributes, and so on. A special entry point, the `main` method, has to be defined for each MTL transformation. Pieces of MTL transformations are organized in MTL *libraries*, each library being in addition responsible for holding models. Each such model can either be a collection of instances of MTL classes from an MTL library, or a collection of model elements inside a repository.

MTL is a compiled language, Figure 2 presenting the compilation process. In order to compile an MTL transformation `T` described in an `mtl` file, the first step is to parse it. A parser (①) reads the transformation as text and transforms it into an internal model that is compliant with the abstract syntax of MTL [15]. In the next step, a type checker (②) refines this model by adding information about types. For instance, in order to deal with polymorphism, it is the type checker that will perform the analysis of MTL meth-

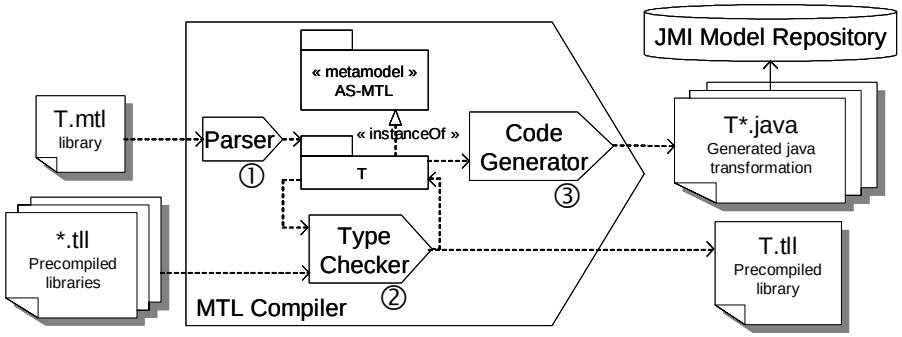

Fig. 2. The MTL Compilation Process

ods in order to reference, for each of them, other MTL methods that they are overriding. If necessary, the types used by the transformation `T` might need to be referred from already compiled MTL libraries. For example, the MTL standard library, which defines the MTL predefined types and operations, is typically used by all MTL transformations, and thus, it participates in such library-usage dependencies. In order for the MTL transformation `T` to be reused by other MTL transformations, its internal model, decorated with type information, is stored in a binary file (`T.tll`). In the end, a code generation step is performed (③). Java source files that implement the behavior described by the internal (refined) model of the MTL transformation `T` are generated, and they will make use of the model repositories on which the implemented transformation was defined to act. We used two `*` signs in Figure 2 in order to show that many precompiled libraries (`*.tll`) may be needed, on one hand, and several Java source files (`*.java`) may be generated, on the other hand, for one MTL transformation. If transformation `T` relies on other libraries, the generated Java source files for `T` will require the Java source files resulted from the compilation of those libraries.

The entire compilation process relies on the model of the MTL transformation `T` itself, which complies with the well-defined MTL metamodel. Therefore, steps ①, ②, and ③ can be viewed as special transformations acting on the MTL model of the transformation `T` itself. Besides these three steps, it is at this MTL model level of the MTL transformations that new special transformations may be defined in order to change the very behavior of those MTL transformations. Following this idea, our MTL weaver is indeed implemented as such a special transformation, acting on the MTL models of the MTL transformations and transforming them according to the weaving behavior defined in MTL-aspects, as we will see in section 4.

4 The MTL Weaver

Reusability has always been an important concern in the software development industry due to its potential to reduce the cost of software development. During the last decade, different levels of reuse have been proliferated, such as functions, procedures, classes, components, aspects, or even entire models. But how can we achieve the reuse of model transformations? How to adapt existing model transformations that already accomplish most of our needs?

The reuse of MTL transformations is currently promoted at the level of MTL libraries, which are some kind of light model transformation components. In this section, we present some implementation details and the provided facilities of an aspect-oriented support that allows transformation developers to reuse existing MTL transformations and to easily adapt them in order to address new needs, or concerns, that the application under development has to incorporate. The main concepts of the MTL weaver are introduced along with the AOP-like extension to MTL for defining the weaving behavior in MTL-aspects. We also present an example showing both the input and the output of a concrete weaving.

The standard MTL language already provides support for transformation developers to define MTL transformations that *transform* other MTL transformations. However, writing such "reflective" MTL transformations still requires transformation developers to be familiar with the metamodel of the MTL language itself, a requirement that significantly reduces the number of such developers. In order to overcome this impediment for the MTL language, we propose a solution inspired by AOP approaches. We have designed and implemented an MTL *weaver* that modifies MTL transformations according to some *weaving behavior* that is specified in terms of *weaving directives* modularized in special stand-alone MTL transformation encapsulation units, called *MTL-aspects*. As in the case of AspectJ, which is an aspect-oriented extension to Java, the syntax defining the weaving behavior in MTL-aspects is a small AOP-like extension to the MTL language itself. In this way, relying on a few high-level AOP-like but MTL-based constructs for defining the weaving behavior, average MTL transformation developers should not have any problems using this MTL extension straightforwardly for defining their "reflective" model transformations.

The place of the MTL weaver in the MTL compilation process and the evolution of the MTL weaving process are presented in Figure 3, where the MTL transformation `T` is refined according to the weaving directives defined in the MTL-aspect `A`. The weaving process is very similar to the compilation process presented in Figure 2. First, both `T` and `A` are parsed (①) in order to transform the two text files into internal MTL models compliant with the MTL metamodel. The important change comes next, when the *MTL Weaver* (②) reads the two internal models of `T` and `A`, and produces a new model instance (of the MTL metamodel) for the new MTL transformation `T+A`, which represents the result of modifying `T` according to the weaving directives defined in `A`. Even though it is not explicitly shown in Figure 3, the MTL weaver itself is implemented as an MTL transformation as well. Once this weaving step is finished, the normal compilation pro-

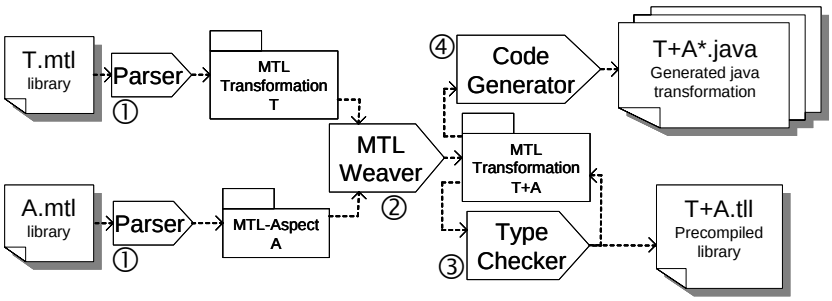

Fig. 3. The MTL Weaving Process

cess can continue with the type checking step (③), which produces a reusable precompiled MTL library, and the code generation step (④), which produces Java source files. Please notice that the weaving process results in a completely new MTL transformation, without making any changes to the original MTL transformation `T`. In this way, both transformations can independently be reused later on in order to transform application models. Moreover, the MTL-aspect `A` may be reused as well for refining other MTL transformations according to the same weaving directives.

4.1 MTL-Based Syntax for Describing the Weaving Behavior

There are two major requirements that an MTL-aspect must fulfill. First, it must clearly identify *where* the modifications have to be performed, and second, it must clearly define *what* are those modifications. In AOP terminology, a *join point* is a well-defined point in the execution of a program where additional functionality may be "injected". To identify such points in our weaving process, a *pattern matching* mechanism is used with respect to the names of the MTL libraries, MTL classes, MTL methods, etc. Both requirements can be expressed using the MTL syntax as well, relying on small extensions that are detailed in this section.

One of the extension mechanisms proposed by the MTL language is the tagging facility. *Tags* are key/value pairs associated either with an MTL library, an MTL class, or an MTL method. Since tags are part of the MTL metamodel, once they are analyzed by the MTL parser, they populate the internal MTL model representing the MTL transformation. This makes it possible for the MTL weaver presented in Figure 3 ② to access these tags and to use them for very different purposes. Since MTL-aspects only rely on the *tag* extension mechanism to define additional weaving directives, it is possible to use the same parser for reading both MTL-aspects and MTL transformations, as shown in Figure 3 ①.

In order to give an example of an MTL-aspect that could play the role of `A` in Figure 3, we show in Figure 4 some snippets of the `MTL1-D-Aspect`. For the sake of readability, we will further on refer to as *input library* the MTL library taken as input for the weaving process, i.e., the library that plays the role of `T` in Figure 3, and its elements *input classes*, *input methods*, etc. The MTL library produced as a result of the weaving process, `T+A` in Figure 3, will further on be referred to as *output library*, and its elements *output classes*, *output methods*, etc.

Each line in Figure 4 may be considered as a weaving directive for the MTL weaver. For instance, the first line defines the name of the input library that the `MTL1-D-Aspect` will have to be weaved in, i.e., `Copy`. In order not to alter the `Copy` input library during the weaving process and to avoid name clashes between input and output libraries, the name of the output library has to be provided. This can be achieved by defining a tag on the MTL library of the MTL-aspect. We have named this tag `rename`, and its value represents the name of the MTL library produced as a result of the weaving process, e.g., `Distribution` in this particular case.

By default, elements of the input library will be simply reproduced in the output library. However, this simple reproduction can be tuned by the rest of the MTL-aspect. For instance, in Figure 4 ①, the MTL class `Copier` is defined. This weaving directive indicates to the MTL weaver that if a class with the same name exists in the input library, then the reproduced class in the output library contains both the members in the

```
library Copy;
tag rename := specialtag [Distribution];

① class Copier {
    servantIterfaceName : Standard::String;

    initDI(sin : Standard::String) : Copier {
      self.servantIterfaceName := sin;
      return self;
    }
  }

② class [{Copier$}] {
    [{^getTarget(.*)}](theSource : Standard::ModelElement)
    tag merge := specialtag [Append];
    tag refactorParameters := booleantag true; {
      theSource.toOut();
    }
  }
```

Fig. 4. Snippets of the `MTL1-D-Aspect`

input class and the ones defined in the MTL-aspect class. This process is called *class merge*. On the other hand, if this class does not exist in the input library, then it will simply be added to the output library exactly as it is defined in the MTL-aspect, i.e., it will include all member definitions defined by the MTL-aspect, e.g., the `servantIterfaceName` MTL attribute and the `initDI` MTL method.

A *conflict* may appear during a class merge if some members in the matching input classes and in the MTL-aspect class have the same name. If the member in the MTL-aspect is an attribute, it will be added as it is, without worrying whether the name of the attribute already exists in the input MTL library, since the rest of the compilation process will detect such a duplicate attribute, if any, and an error will be thrown. For methods, the detected conflict is registered to be solved later.

MTL-aspect developers may refer many MTL classes or MTL methods in a single pattern by relying on "wildcard" facilities, such as "_", which matches any name, or the more sophisticated regular expressions delimited by curly brackets. For instance, in Figure 4 ②, the class named `{Copier$}`, matches all input classes whose name ends (denoted by `$`) with "Copier", and its method `{^getTarget(.*)}` matches all input methods, defined on the matched input classes, whose name starts (denoted by `^`) with "getTarget". As a rule, MTL-aspect developers should not abuse of such constructs in order to add new classes or methods to the output library.

The class merge process, as it is implemented in the MTL weaver, is shown in Figure 5. The `libClass` represents the input class, and the `behaviorClass` represents the MTL-aspect class. Please note that the name of the `behaviorClass` matches the name of the `libClass` as a precondition for the `mergeClass` method.

A method conflict may be solved according to some predefined rules. We have identified three kinds of possible rules that prescribe the MTL weaver how to manage the instructions defined by the conflicting method of the MTL-aspect:

- run MTL-aspect instructions at the very beginning of the output method,
- run MTL-aspect instructions just before returning from the output method, or
- replace input instructions with MTL-aspect instructions in the output method.

```
mergeClass(libClass : BasicMtlASTView::UserClass;
           behaviorClass : BasicMtlASTView::UserClass) {
  lo : Standard::Set;
  // adding attributes
  if (isNull(behaviorClass.definedAttributes).not()) {
    foreach (at : BasicMtlASTView::Attribute)in (behaviorClass.definedAttributes) {
      libClass.appendDefinedAttributes(at);
    }
  }
  // merging operations
  foreach (bo : BasicMtlASTView::Operation) in (behaviorClass.definedMethods) {
    lo := matchingOperations(libClass, bo);
    if (lo.size().[=](0)) { // to be added
      if (self.canAdd(bo)) {
        libClass.appendDefinedMethods(bo);
      } else {
       bo.name.concat(' seems to be a pattern; no correspondance found.').toOut();
       'ignoring addition to class '.concat(libClass.name).toOut();
      }
    } else { // conflict, to be treated later
      self.operationConflicts := operationConflicts.including(
                              new OperationConflict().init(libClass, lo, bo));
    }
  }
}
```

Fig. 5. MTL Weaver Snippets for Class Merge (`mergeClass`)

It is the responsibility of the MTL-aspect developer to indicate which alternative s/he desires to be chosen for a given method conflict. For this purpose, we defined the `merge` tag that has to be added on each conflicting method in the MTL-aspect. The three possible values corresponding to the previously described rules are `Prepend`, `Append`, and `Replace` respectively. If a conflict cannot be solved, the weaving process ends in failure.

The instructions in the MTL-aspect method may need to refer to some parameters of the matched input methods. The presence of the boolean tag `refactorParameters` set to `true` makes the parameters of the input methods accessible inside the MTL-aspect according to the names provided in the MTL-aspect method. Moreover, this tag makes the method matching take care of the number of parameters in the input methods rather than just matching the names of the methods.

As an example, Figure 4 ② states that for all input methods whose names start with "getTarget" inside classes whose names end with "Copier", the first parameter, named in the MTL-aspect `theSource`, must be sent to the console by means of the MTL predefined operation `toOut`. This output must be performed before returning from the modified MTL methods, as stated by the value `Append` of the `merge` tag defined for the MTL-aspect method.

As a summary, the list of possible tags that may appear in the definition of an MTL-aspect is provided in Table 1. The first column gives the name of the tag as it must appear in the MTL-aspect. The second column indicates on which MTL element this tag may be defined. The third column indicates whether the presence of the tag is mandatory or optional; default values are indicated for optional tags. The fourth column gives a brief description of the semantics of the possible associated values.

Table 1. Predefined MTL-Aspect Tags

Tag Name	Base MTL Element	Presence	Description
`rename`	Library	mandatory	The name of the output library.
`merge`	Method	mandatory if conflict	`Prepend` to add instructions at the very beginning of the method. `Append` to add instructions just before returning from the method. `Replace` to replace initial instructions with MTL-aspect instructions.
`refactorPa-rameters`	Method	optional; default value is `false`	Indicates if the number of parameters has to be considered in the pattern matching, and if parameters have to be intercepted for further use inside MTL-aspect instructions.

As we showed on some concrete examples, the MTL-aspect developer does not need to have a deep knowledge of the MTL metamodel and its semantics in order to transform an MTL transformation. All s/he needs to know is the MTL syntax and some predefined tags. Moreover, with the current implementation of the MTL weaver, an MTL-aspect is about 10 times smaller (in lines of code) and about 50 times faster to develop than a standard MTL transformation that would achieve the same weaving behavior on another MTL transformation.

Please notice, however, that the MTL weaver and the aspect-oriented support provided are relatively young, still undergoing refinement and improvement as we move along. New constructs will be added in order to address MTL-aspect developer needs and to facilitate as much as possible the development of "reflective" MTL transformations. For instance, it would be very helpful to have a pattern matching for instructions or expressions, e.g., matching all *calls* to a given method. The pattern we adopted for extending the MTL language with AOP-like constructs will remain nevertheless the same, i.e., extending the language by providing new tags that change the semantics of their base element, just like UML profiles extend the UML.

4.2 Running Example

In this part, we consider the weaving of the `MTL1-D-Aspect` in the simple MTL `Copy` transformation in order to modify its behavior and make a system distributed by applying the stereotypes defined in the `DistributionProfile` [13] according to some configuration information. Since the goal is to illustrate the most important principles of the weaving process, we focus on very small parts of the example.

The input MTL `Copy` transformation is specialized in copying an input UML 1.4 model to an output UML 1.4 model. Snippets of the transformation are presented in Figure 6. The transformation is located in the MTL library `Copy`, having two variables, `in` and `out`, for referring to the input, and output models respectively. One of the MTL

classes of this library is `Copier`, which defines the `getTarget` method. This method takes as parameter a UML element `srcElt` from the `in` model, and retrieves and returns the corresponding UML element inside the `out` model. Another MTL class, extending `Copier`, is `UML14CreatorCopier`, which defines the `getTargetClass` method. This method takes a UML class `src` in the `in` model as parameter, and is responsible for creating and returning a UML class in the `out` model.

```
library Copy;
model in  : RepositoryModel;    // should be a UML1.4 MetaModel
model out : RepositoryModel;    // should be a UML1.4 MetaModel
class Copier {
  getTarget(srcElt : in::Core::Element) : out::Core::Element {
    r : out::Core::Element;
    ... // compute r
    return r;
  }
}
class UML14CreatorCopier extends Copier {
  getTargetClass(src : in::Core::Class) : out::Core::Class {
    r : out::Core::Class;
    r := new out::Core::Class();
    trace(src, r);
    return r;
  }
}
```

Fig. 6. Snippets of the `Copy` Input Library

We present now two of the modifications that have to be performed in order for the MTL `Copy` transformation to make a system distributed. The first one is to make an interface remotely available, but before doing this we still need to identify the right interface. The solution we considered is to add an attribute, `servantIterfaceName`, to the MTL `Copier` class as a placeholder for the name of the interface to be distributed. This attribute is transmitted to the MTL `Copier` class by means of the new method `initDI` defined in the `MTL1-D-Aspect`. The second modification is to display on the console UML elements from the `in` model for which a correspondence in the `out` model has been requested. A thorough analysis of the complete MTL `Copy` transformation would clarify that such correspondences are only requested when invoking methods whose names start with “getTarget”, and which belong to a class whose name ends with “Copier”. These modifications are prescribed in the `MTL1-D-Aspect` that was partly presented in Figure 4, where part ① corresponded to the first modification, and part ② to the second one.

The result of weaving the `MTL1-D-Aspect` in the MTL `Copy` transformation is shown in Figure 7. Even though we have clearly stated in section 4 that the results of the MTL weaving process are just MTL binaries and Java source files, Figure 7 represents what a pretty printer would produce for the MTL binary. Changes introduced by the MTL-aspect are highlighted by change bars. Since the output MTL library is different from the original MTL `Copy` library, a renaming has occurred according to the `rename` tag that was specified on the library definition inside the `MTL1-D-Aspect`, as shown in Figure 4.

Part ① of the `MTL1-D-Aspect` in Figure 4 states that an MTL class named `Copier` must appear with a `servantIterfaceName` attribute and an `initDI` operation in the

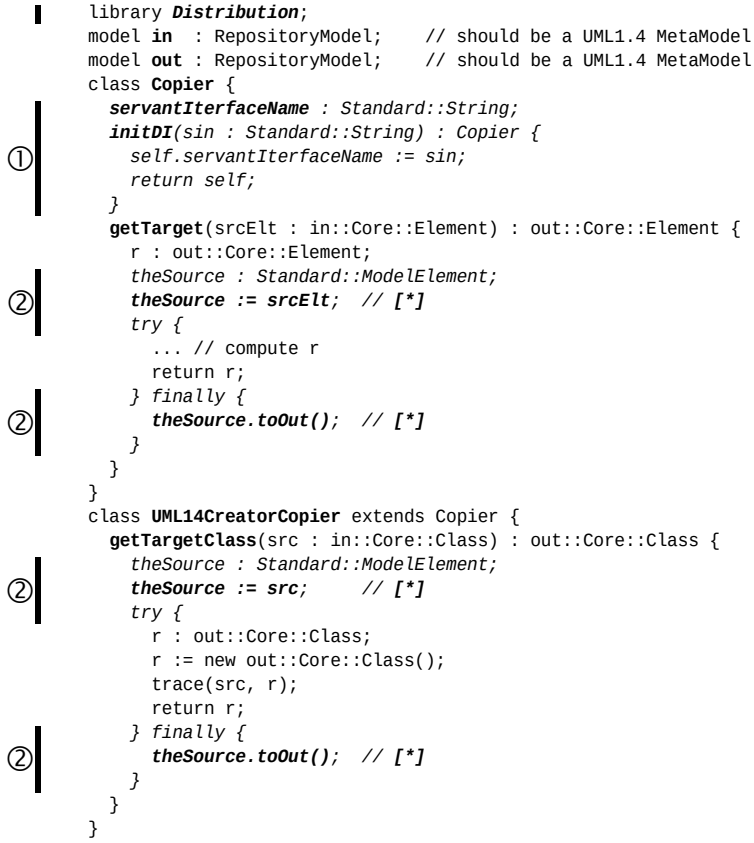

Fig. 7. Snippets of the `Distribution` Output Library

output library. Even though such an MTL `Copier` class already exists in the input library, no name conflicts have been found, and therefore member definitions from both the MTL-aspect and the input class are directly added to the MTL `Copier` output class, as shown by Figure 7 ①.

The MTL-aspect method defined in part ② of the `MTL1-D-Aspect` in Figure 4 matches the input methods `Copier::getTarget` and `UML14CreatorCopier::getTargetClass`. Please note that the presence of the `refactorParameters` tag set to `true` in the MTL-aspect has made the method matching check that only one parameter is defined for these input methods, parameter that will further on be used as the variable `theSource` inside the body of the MTL-aspect method. The tag `merge` set to `Append` defined on the MTL-aspect method indicates how possible conflicts should be solved. Since conflicts have indeed been found, the instructions defined in the MTL-aspect have to be inserted in the output class just before returning from the corresponding reproductions of the input methods in the output class, as part of the output library. To achieve this, we rely on the MTL `try-catch-finally` statement: instructions of the input method are reproduced in the `try` part, and instructions from the MTL-aspect method are reproduced in the `finally` part, as shown in Figure 7 ②. In this way, we enforce that instructions from the MTL-aspect method are executed just before return-

ing from the output method, wherever an MTL `return` instruction may appear in the input method. The `true` value for the `refactorParameters` tag also instructs the MTL weaver to produce new variables in the output methods according to the parameters defined in the MTL-aspect method that are supposed to match parameters from the input methods. These new variables represent placeholders for the values of the parameters of the input methods that were intercepted by the corresponding MTL-aspect method. Applying this rule for the two input methods matching the MTL-aspect method `{^getTarget(.*)}`, new `theSource` variables will be added in the corresponding output methods for storing the very input parameters that were previously matched (see Figure 7 [*]).

5 Conclusions and Future Work

All model transformation languages that we know of provide transformation developers the facility to define "reflective" model transformations, i.e., model transformations that transform other model transformations. However, writing such model transformations is generally beyond the ability of many transformation developers since it requires the developer to be familiar with the metamodel of the transformation language itself. In order to overcome this frustrating impediment for the INRIA MTL transformation language, we presented in this paper an MTL *weaver* that modifies MTL transformations according to some *weaving behavior* that is specified as a special kind of MTL transformations, called *MTL-aspects*. Inspired from the AOP world in general, and from AspectJ in particular, the syntax defining the weaving behavior in MTL-aspects is a small AOP-like extension to the concrete syntax of the MTL language itself. In this way, relying on a few high-level AOP-like but MTL-based constructs for defining the weaving behavior, average MTL transformation developers should not have any problems using this MTL extension straightforwardly in order to define their "reflective" model transformations.

The support provided by the MTL weaver through the MTL extension syntax was illustrated on a concrete example, namely modularizing the distribution concern in stand-alone units of encapsulation represented by MTL-aspects. We have shown in this way that transformation developers are given not only the possibility, but also the means to rely on the well-proven power of separation of concerns even at model transformation level.

Even though this entire research was carried out on the INRIA MTL transformation language, most of the concepts presented in this paper are MTL independent and could easily be applied at the future QVT specification level by providing higher level constructs for specifying the weaving behavior. For example, we can very well imagine the `MTL1-D-Aspect` be written at the QVT specification level, and then automatically refine it for the MTL language when applying it in the context of MTL-based projects. Although the constructs introduced in this paper are very suitable for imperative model transformation languages (e.g., "`before method return`" or "`after call`"), we believe that similar counterparts may be identified in declarative model transformation languages as well (e.g., "`after rule match`"), and thus a common ground could be found at the QVT specification level.

References

[1] Object Management Group, Inc.: *Model Driven Architecture*. http://www.omg.org/mda/, April 2004.

[2] Miller, J.; Mukerji, J.: *Model Driven Architecture (MDA)*. Object Management Group, Draft Specification ormsc/2001-07-01, July 2001.

[3] Object Management Group, Inc., http://www.omg.org/, April 2004.

[4] Object Management Group, Inc.: *MOF 2.0 Query/Views/Transformations RFP*. http://www.omg.org/cgi-bin/doc?ad/02-04-10, 2002.

[5] French National Institute for Research in Computer Science and Control (INRIA): *Model Transformation Language (MTL)*. http://modelware.inria.fr/, April 2004.

[6] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: *Aspect-Oriented Programming*. Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP, Jyväskylä, Finland, June 9-13, 1997. LNCS Vol. **1241**, Springer-Verlag, 1997, pp. 220 ─ 242.

[7] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: *An Overview of AspectJ*. Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP, Budapest, Hungary, June 18-22, 2001. LNCS Vol. **2072**, Springer-Verlag, 2001, pp. 327 ─ 353.

[8] Eclipse Project: *AspectJ*. http://www.eclipse.org/aspectj/, April 2004.

[9] Parnas, D. L.: *On the Criteria to be used in Decomposing Systems into Modules*. Communications of the ACM, **15**(12), December 1972, pp. 1053 ─ 1058.

[10] Rouvellou, I.; Sutton, S. M. Jr.; Tai, S.: *Multidimensional Separation of Concerns in Middleware*. Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering, held at the International Conference on Software Engineering, ICSE, Limerick, Ireland, June 4-11, 2000. http://www.research.ibm.com/hyperspace/workshops/icse2000/.

[11] Silaghi, R.; Strohmeier, A.: *Integrating CBSE, SoC, MDA, and AOP in a Software Development Method*. Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Brisbane, Queensland, Australia, September 16-19, 2003. IEEE Computer Society, 2003, pp. 136 ─ 146. Also available as Technical Report, N° IC/2003/57, Swiss Federal Institute of Technology in Lausanne, Switzerland, September 2003.

[12] Object Management Group, Inc.: *The Common Object Request Broker: Architecture and Specification*, v3.0, July 2002.

[13] Silaghi, R.; Fondement, F.; Strohmeier, A.: *Towards an MDA-Oriented UML Profile for Distribution*. Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Monterey, CA, USA, September 20-24, 2004. IEEE Computer Society, 2004 (to appear). Also available as Technical Report, N° IC/2004/49, Swiss Federal Institute of Technology in Lausanne, Switzerland, May 2004.

[14] Sun Microsystems, Inc.: *Java Remote Method Invocation Specification*. Revision 1.7, Java 2 SDK, Standard Edition, v1.3.0, December 1999. http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html.

[15] Vojtisek, D.: *BasicMTL Realization Guide*. Inside the Carroll Research Program and part of the MOTOR project, Technical Report, February 2004. http://modelware.inria.fr/article.php3?id_article=45, April 2004.

[16] Bézivin, J.; Dupé, G.; Jouault, F.; Pitette, G.; Rougui, J. E.: *First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery*. Second International Workshop on Generative Techniques in the Context of MDA, held at the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, Anaheim, CA, USA, October 26-30, 2003.