Open MDA Using Transformational Patterns

Mika Siikarla, Kai Koskimies, and Tarja Systä

Tampere University of Technology, Institute of Software Systems,
P.O.Box 553, FI-33101 Tampere, Finland
{mika.siikarla, kai.koskimies, tarja.systa}@tut.fi
http://practise.cs.tut.fi

Abstract. No generally accepted understanding on the characteristics of MDA transformation mechanisms exists. Various approaches to support such transformations have been proposed. In this paper, we discuss general requirements for MDA transformation mechanisms. We claim that, above all else, transformation mechanisms should be open, i.e. clear, transparent and user-guided. We propose a new concept, a transformational pattern, as a basis of an MDA transformation mechanism. We exploit existing tool support for this concept and show a small example of how it can be applied. Finally, we analyse the ability of the proposed technique to fill the requirements.

1 Introduction

A clearly identified long-term trend in software engineering is the introduction of higher and higher abstractions from which actual implementations are derived. OMG's Model-Driven Architecture (MDA) initiative [1] is a recent manifestation of this trend. A key idea in MDA is that system development should be based on high-level, platform independent models (PIM) from which lower level platform-specific models (PSM) and eventually implementations are derived with the support of transformation tools.

Although the vision behind MDA is generally accepted, the required tool technology is just taking its first steps. Some early tool support exists (e.g., ArcStyler [2]), but the underlying concepts and paradigms of the tools are far from well understood, if even existing.

Obviously, there are many ways to specify and execute transformations from one model to another. A straightforward approach to specify the transformations in an executable form would be a script language with access to a model repository and appropriate navigation and query capabilities. Then, transformations could be realized simply as scripts.

The real challenge of MDA transformation tool support is not in devising the computational vehicle, but rather in the collaboration of the designer and the tool. A simple black-box approach (e.g. a Python script) would hide the relationship between the source and the target model from the designer, making it very difficult to work with the result. If the path from a platform independent model to executable implementation were completely automated, this would not be a problem, but we argue that this is an unrealistic idea, at least in the near future. Typically, the designer has to examine the result, understand it, and apply further transformations or modifications on some parts of the result. Thus, we propose that an MDA transformation tool should be open in the sense that it allows the designer to be involved in the transformation process.

In this paper we will first discuss the required properties of an MDA transformation mechanism in more detail. As a potential approach to satisfy these requirements, we introduce the concept of a transformational pattern, which we believe can serve as the basis of open MDA transformations. This concept is an application of a generic pattern facility originally developed for supporting framework specialization [3]. We demonstrate the use of this technique by showing how a J2EE model can be generated from a platform-independent UML model for a Web Services application. Based on this example, we briefly analyse the extent to which this approach meets the requirements. Finally, we discuss related work and the future directions of our work.

2 MDA Transformation Mechanics

We see the primary role of a transformation as documenting the relations between different models of the same system. With this added information expressed in computer readable form, the models can be kept synchronized and a change in one model does not render all other models obsolete. This is absolutely vital for MDA. The description of these relations, i.e. the *record of transformation* according to [4], contains unique information about the system, and should therefore be considered a model itself.

Another important, although secondary, role is to support the designer in deriving one model from others, by alleviating the burden of at least the repetitious and trivial tasks. In some very specialized cases, such as a specific product-line, it might be possible to achieve fully automated transformations. However, it seems overly optimistic to expect fire-and-forget solutions for all possible situations any time soon. The intermediate, or derived, models do therefore contain more information than just what is derived. They have value as original artefacts and should not be considered as mere documentation.

In our view, transformation definitions are software artefacts. They are subject to evolution the same way design models or program files are. We expect, for example, that a set of model transformations can be given for a product-line platform to be used for the derivation of the designs for individual software products. Such a set of transformations is an integral part of the product-line and goes through changes and versions together with the other assets belonging to the product-line. It is likely, in fact, that the transformation mechanisms themselves need maintenance and evolve as the subject system does.

We raise *openness* as the most important property that is required of an MDA transformation mechanism. The designer should participate in the transformation process, guiding it with her decisions, rather than receive the results of a black-box operation as an outsider. The mechanism itself should be transparent, allowing the designer to follow how the models are being manipulated.

The meaning of every step in the transformation process should be clear. When using a clear and transparent machinery, the designer is better equipped to make decisions affecting the transformation. She can be trusted to make an educated choice between possible courses of action even during the transformation.

We argue that the open approach is safer, allowing the designer to understand the resulting model and modify it, if needed. In the black-box case modification of the result is risky, because the designer does not understand the purpose of different parts of the result, and therefore cannot judge the relationship elements in the target and source models. Note, that a part not dependent on the source model can be altered without compromising the relationships between the models. In the absence of fully automated transformations, it would be unreasonable to completely forbid modifications of the resulting model.

In some cases, where no single transformation process can be found for a category of systems, it is still possible to find transformation principles that apply to each of the systems. E.g. software products developed from the same product-line, or systems belonging to a particular application domain, might form such categories. The transformation mechanism should support *customisable* transformations that contain the common principles and provide variation points for customisation. Unlike with direct editing of the result, some customisation needs are foreseen and built into the transformation.

It is possible that some part in the target model resulting from applying a transformation is not considered acceptable for the particular application. Instead of trying to guess what changes in the source would lead to the desired result, it should be possible to change the result directly and produce a source model corresponding to the modified result. In cases where the source or target metamodel or the transformation itself loses information, bi-directionality cannot be fully achieved. However, the transformation mechanisms themselves should be *unbiased* as far as the direction of the transformation is concerned, and not force or encourage the transformation definitions to be unidirectional.

If a modification breaks several transformation rules and there are several ways to repair them, user-assisted repairing might be preferable to automatic repairing actions. In both cases the elements impacted by the modification should be traceable. In order to fix a problem, or to correct a mistake, it might be desirable to reverse the application of a transformation, effectively undoing it. This might prove to be challenging in practise. *Traceability* and *reversibility* are examples where knowledge of the relations between models are needed. This implies, that applying a transformation leaves a persistent record of transformation.

It should be possible to carry out the transformation one step at a time, rather than as a batch. *Incremental* transformation process contributes to the fine-grained management of the transformation, with a number of benefits. First, it contributes to openness, supporting understanding in general: the process can be better followed when divided into small pieces. Second, it allows for fine-grained backtracking: if the process appears to be going in a wrong direction, individual steps can be undone without losing the results produced so far. This is useful for steps with variation points. Third, incremental processing supports fine-grained customisability: variation points can be attached to individual steps rather than to the entire process. In this way variation points can be shown only when really needed: if a variation becomes obsolete because of earlier choices, the variation point need not be presented at all. Fourth, partial transformation processes are supported, where sensible (but incomplete) target models are produced on the basis of incomplete source models. This allows for partial evaluation of the transformation when developing or maintaining the transformation itself.

A transformation process can consist of several single, well-focused transformation steps. Therefore, mechanisms to compose configurations of individual transformation operations are needed. In these configurations, dependences and constraints between the individual operations should be supported, yielding to a need for an actual transformation language. Such *combinability* enables and promotes transformation reuse. There is also a need to relate different models together in one transformation. For instance, to form a platform-specific model, information from a platform-independent model as well as from specific platform deployment and description models might be needed. Combining information from different source models in a "concern-oriented" way would help the user to better understand and manage the dependences among the models.

Since transformation definitions are software artefacts, they need to be maintained throughout their life cycle. Therefore, transformations should be *maintainable*, implying that a transformation is specified in a manner that allows easy replacement of its parts. Customisability and combinability promote reuse and therefore improve maintainability. Many properties, especially openness, make transformations easier to understand, which helps in maintenance.

Documentation of the transformations is needed, so they can be understood and applied. Documentation is also needed for maintenance. Therefore transformations need to have an *illustrative presentation*, e.g. a visual notation that can be understood by the different parties involved. Since people comprehend examples better than rules or algorithms, it would be beneficial if examples could be constructed out of definitions and vice versa. Some visual presentation is needed for examples, too, and for visualizing mappings between models.

3 Transformational Patterns

In this work a *pattern* is an organized collection of software elements capturing any concern that is relevant for some stakeholder of the system. To be able to define a pattern independently of any particular system, a pattern is defined in terms of element *roles* rather than concrete elements; a *pattern instance* is tied to a particular context by binding its *role instances* to concrete elements. The relationship between a pattern and its pattern instance can be viewed as that of a model and its instance. Roles can then be seen as classes and role instances as objects. In this paper, we simply use the term pattern when referring to pattern instance, or role when referring to a role instance. The full term is only used when there is a risk of misunderstanding.

A role has a *role type*, which determines the kind of system elements that can be bound to the role's instances; the set of all valid role types is called the *domain* of the pattern. For example, if the domain is UML, the role types are the element types (metaclasses) of UML: there are class roles, operation roles, association roles etc. In the following we assume that the domain is UML.

Each role may have a set of *constraints*. Constraints are conditions that must be satisfied by the model element bound to a role's instance. For example, a constraint of a class role *C* may require that only a class stereotyped as ≪Persistent≫ can be bound to an instance of *C*. Constraints may also refer to other roles, e.g. a constraint on association role *A* may require that an association bound to *A's* instance must appear between classes bound to (instances of) certain class roles *C1* and *C2*. Note that such a constraint implies (perhaps indirect) relationships between roles *A* and *C1* as well as *A* and *C2*. Continuing with the pattern-model analogy, these relationships can be though of as associations.

A completely bound pattern instance therefore poses constraints on the elements bound to the role instances. The constraints from the pattern have in effect been joined with constraints in the domain model. For example, one of the UML well-formedness rule states that an attribute of a class may not have the same name as the class. The first pattern in the example above states that each class bound to (an instance of) the role *C* must be stereotyped ≪Persistent≫. Every element in the UML model must now fulfil both these constraints (although, the first still only applies to classes, and the second only to appropriately bound classes). If a model modification violates a constraint, the model must be fixed by adding, removing, or modifying elements until the constraint holds again.

In addition to constraints, *default values* can be specified for a role. For example, a class role might be given `"Breakfast"` as the default name, and `false` as the default value for the property *isAbstract*. If a role with default values needs to be bound, a new element can be generated and bound to the role. The default values do not need to be constants, and they can refer to other roles. For instance, the default name for a class role *KeyClass* might be defined as the name of the class role *Class* appended with `"Key"`. In order to make use of the default value, *Class* must of course be bound.

Exploiting the default values as a generative mechanism, a pattern can be used in any context where a collection of elements needs to be generated based on well-defined relationships between existing and the generated elements. This is the situation when a PSM is generated based on a PIM according to certain well-defined transformation rules. In this context a pattern implements a transformation rule or a set of transformation rules. We call such patterns *transformational patterns*. Assuming tool-assisted binding and element generation, patterns offer an attractive approach to realize MDA transformations.

A transformational pattern spans multiple domains. For example, consider a transformation from UML to EJB. The set of valid elements for binding contains all the elements from the UML and EJB models. The domain of the pattern contains the metaclasses from the UML metamodel and the metaclasses from the EJB metamodel. Role types include class role (UML), association role (UML), data schema role (EJB), component role (EJB), etc. From the pattern's point of

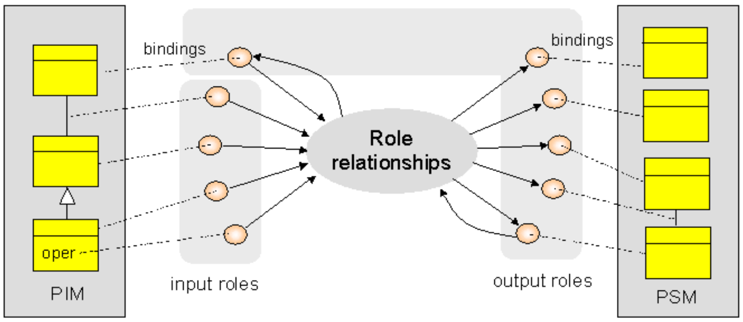


Fig. 1. A transformational pattern as a function

view, there is a single model, comprised of the UML model and the EJB model, side-by-side, but separate.

The purpose of the transformational pattern is to modify (a fragment of) the combined model so that the constraints for the pattern's roles hold for the model elements. Of course, the constraints only need to hold for elements bound to roles in the pattern. If the target model is empty, and all elements in the source model are bound, the constraints can be satisfied by creating new elements in the target model. If the default value cannot be computed, or does not exist, the constraints cannot be satisfied automatically. In a case where there are bound elements in both the source and the target model, constraint violating elements must be modified, and new elements provided for unbound roles.

To put it in a bit more formal way, a transformational pattern can be seen as a function that for a pattern instance computes the default values for unbound role instances (*output roles*) from the values of the bound role instances (*input roles*). One new element is created for each output role and the element is bound to the role. It is important to note, that the division to input and output roles does not necessarily reflect the division to source and target elements. It is very much possible to compute some source *and* target elements based on a set of existing source and target elements. Figure 1 illustrates a transformational pattern function.

The specifications of default values of roles are called *element templates*. For role r, this specification is denoted with function $Elem_r(Bound(r_1), \ldots, Bound(r_k))$, where $r_1, \ldots, r_k$ are the roles referenced in the element template specification. $Bound(r_i)$ represents the element e_i bound to role r_i. The function yields a new concrete element, assuming that the roles $r_1, \ldots, r_k$ have been bound. $Elem_o$ needs to be evaluated for each output role o. This is possible, if default values are specified and the references between the output roles imply a partial order. For any sequence $o_1, \ldots, o_n$, where o_i never depends on o_j when $i < j$, the elements for output roles can be computed with

$$Bound(o_1) = Elem_{o_1}(Bound(r_{1,1}), \ldots, Bound(r_{1,k_1})) = Elem_{o_1}(e_1, \ldots, e_{k_1})$$
$$\ldots$$
$$Bound(o_n) = Elem_{o_n}(Bound(r_{n,1}), ..., Bound(r_{n,k_n})) \;.$$

4 Tool Environment: MADE

MADE [5] is an integrated collection of tools for pattern-driven UML modelling. Rational Rose [6], a UML modelling tool, is one of the key components, enabling visualization and manual modification of models. We have used MADE as a prototype tool environment for transformational patterns (explained in Sect. 3). Although MADE has not been designed with transformations in mind, the underlying pattern concept is sufficiently generic to provide the required mechanisms and user interface for applying transformational patterns in the UML domain. MADE supports the specification of patterns, and the interactive binding of the roles of a pattern to UML model elements residing in Rose.

A key functionality of the environment is, that it transforms a (possibly partially bound) pattern into a task list. A task is generated for each unbound role, but only if all the other roles it depends on are already bound. The designer completes such a task by providing an element to bind to the role. Either the designer points out an existing model element or she asks the tool to generate a new element based on the default values for that role. She has full access to the UML modelling tool, Rose, and can manually create an element, e.g. a class, and then point that out to complete a task. The pattern specification can be associated with informal instructions for binding the roles, which are shown to the user when the corresponding task is to be performed.

MADE checks that role constraints are satisfied by bound elements. In the case of constraint violations, new corrective tasks are created. In many cases the tool can provide an option to correct the model automatically. Because binding information is preserved even after applying the pattern has been completed, any constraint violating changes to the model can be detected. For example, free model editing actions in Rose can cause corrective tasks. Persistent bindings also save from having to re-apply patterns when the model is changed.

The tool also maintains a list of pattern instances. Patterns with constraint violations or unbound roles are indicated with a red marker. When the designer selects a pattern, only tasks related to that pattern are displayed. This helps the user keep focused and not get distracted by concerns irrelevant to her goal. For the same reason, tasks that can not be performed at the moment are not shown.

For use with transformational patterns, the central functionality of the tool environment is the incremental, task-driven binding process, combined with the generation of default elements. This allows for stepwise performing of a transformation, keeping the designer aware and in control of each step. The designer can customize the transformation process by following different task paths. Further, a pattern stores the information about the transformation, so that it can be later retrieved and used for various purposes (e.g. tracing, comprehension, visualization). Some parts of the transformation can be easily redone later, as long as the constraints defined by the pattern still hold after the changes.

The MADE environment is still in the prototype stage, and there are shortcomings in some areas. For example, pattern combining is still under development, and currently only allows static combining. A groups of patterns can be composed and then applied instead of a single pattern. However, patterns cannot be added to a group dynamically, for example, based on properties of the model or user decisions. Also, the tool provides no real visual notation for pattern definitions or for (partially or completely) bound pattern instances. It is possible to highlight elements bound to a specified pattern instance, but that does not show which element corresponds to which role in the pattern.

MADE has not been designed explicitly for transformational patterns. Relations between pattern roles are modelled as dependencies instead of associations, and are thus directed. This is not a problem with design patterns, but it does make transformational patterns unidirectional in practice. The lack of associations also makes it impossible to navigate from a role directly to a related role with OCL. Navigation is performed by referring to the role by its name.

5 Applying Transformational Patterns in MDA: An EJB Example

The example is a transformation of a UML model (PIM) (Fig. 2) to an EJB model (PSM). Starting with a set of informal, natural language transformation rules we form transformational patterns, which are entered into the MADE tool. The patterns are then applied to the source model to produce the target model.

The UML model and the set of transformation rules were adapted from an example by Kleppe et al. [7]. The model describes a small business, Rosa's Breakfast Service, and consists of 7 classes and 5 associations. The structure of the PIM is presented in the class diagram in Fig. 2, but most attributes have been omitted to keep the diagram small. The transformation rules (in Fig. 3) have been re-worded, but should still express the idea of the original ones. Although the example is rather small, it does require roughly 40 separate invocations of the rules listed. It is therefore suitable for demonstrating our approach.

Some of the rules refer to a *root class*. In this context it means the root of the hierarchy implied by composite-associations between classes. For example, in Fig. 2, the root class of *Breakfast* is *BreakfastOrder*, and the root class of *Customer* is *Customer* itself. An EJB data schema (or an EJB component) *cor-*

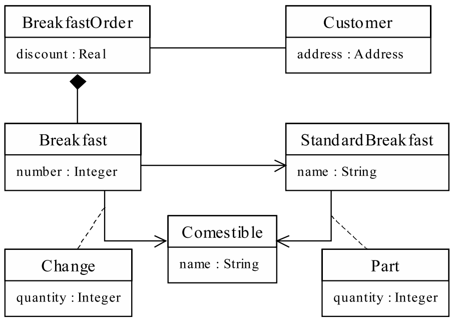


Fig. 2. PIM of Rosa's Breakfast Service (adapted from [7, Fig. 4-2, p.48])

1. (a) For each PIM class, an EJB key class is generated.
   (b) For each PIM association class, an EJB key class is generated.
2. For each root class, an EJB component and an EJB schema are generated.
3. For each PIM class, an EJB data class residing in the EJB data schema corresponding to the PIM class is generated.
4. Each PIM association is transformed into an EJB association.
5. For each PIM association class, two EJB associations and a data class are generated.
6. Each attribute of a PIM class is transformed into an EJB attribute of a data class.
7. Each PIM operation is transformed into an EJB operation of the EJB component corresponding to the PIM class of the PIM operation.

Fig. 3. Informal transformation rules (adapted from [7, p.58])

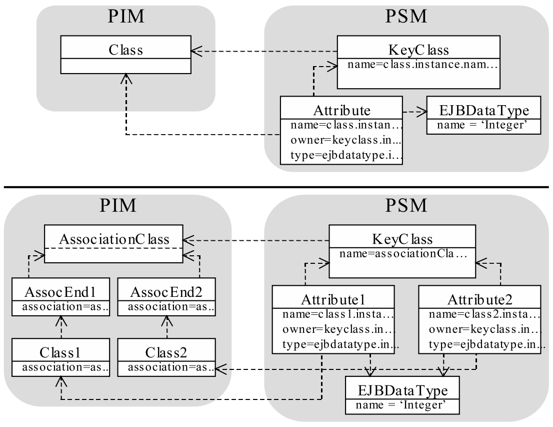
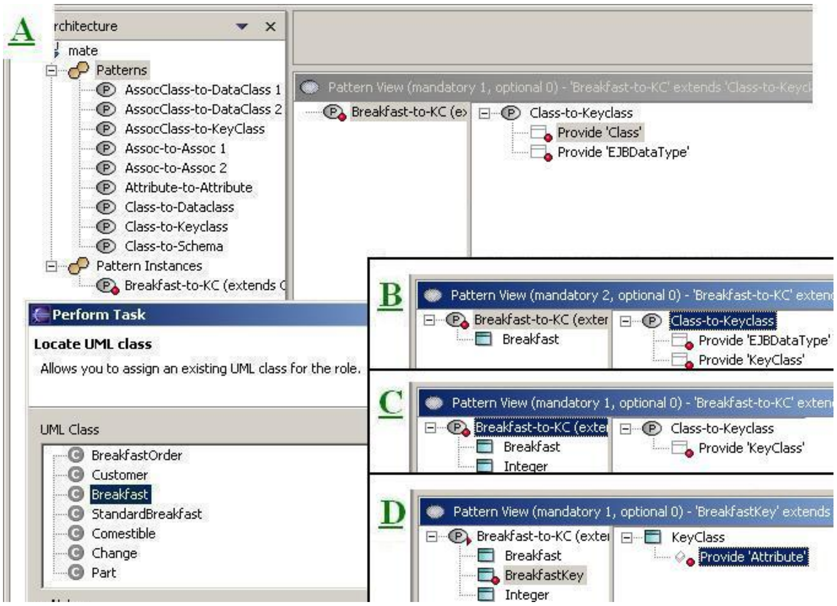


Fig. 4. Two of the transformational patterns corresponding to the informal rules

responding to a PIM class is the schema (component) that was generated by rule 2 based on the root class of the PIM class.

Each informal rule is modelled as a single transformational pattern, except for rules 1, 4, and 5, which have two alternative patterns. The rules could have, of course, been modelled in many different ways, resulting in a different set of patterns. Fig. 4 shows two of the patterns, and the rest are omitted for brevity. The top one corresponds to rule 1a, and describes the relationship between a UML class and an EJB key class. The bottom one corresponds to rule 1b. Both patterns have already been converted to a form required by the MADE tool. I.e., associations have been replaced by dependencies and OCL-constraints refer to pattern roles directly by their names instead of navigating along associations.

Each class (rectangle) in the picture represents a role and a dependence (arrow) between roles means that one role refers to the other in a constraint or a default value specification. The smaller of the two patterns in Fig. 4 contains four roles; *Class* (UML class role), *KeyClass* (EJB key class role), *Attribute* (EJB attribute role), and *EJBDataType* (EJB datatype role).


A

rchitecture

mate

Patterns

AssocClass-to-DataClass 1

AssocClass-to-DataClass 2

AssocClass-to-KeyClass

Assoc-to-Assoc 1

Assoc-to-Assoc 2

Attribute-to-Attribute

Class-to-Dataclass

Class-to-Keyclass

Class-to-Schema

Pattern Instances

Breakfast-to-KC (extends C

Pattern View (mandatory 1, optional 0) - 'Breakfast-to-KC' extends 'Class-to-Keycl

Breakfast-to-KC (e>

Class-to-Keyclass

Provide 'Class'

Provide 'EJBDataType'

Perform Task

Locate UML class

Allows you to assign an existing UML class for the role.

UML Class

BreakfastOrder

Customer

Breakfast

StandardBreakfast

Comestible

Change

Part

B

Pattern View (mandatory 2, optional 0) - 'Breakfast-to-KC' exten

Breakfast-to-KC (exter

Breakfast

Class-to-Keyclass

Provide 'EJBDataType'

Provide 'KeyClass'

C

Pattern View (mandatory 1, optional 0) - 'Breakfast-to-KC' exten

Breakfast-to-KC (exter

Breakfast

Integer

Class-to-Keyclass

Provide 'KeyClass'

D

Pattern View (mandatory 1, optional 0) - 'BreakfastKey' extends

Breakfast-to-KC (exter

Breakfast

BreakfastKey

Integer

KeyClass

Provide 'Attribute'

Fig. 5. Four screenshots from applying the pattern Class-to-Keyclass

KeyClass refers to *Class*, and *Attribute* refers to every other role. Constraints are shown inside the class symbols, one OCL constraint per line. For example, *Attribute* has three constraints; the first one constrains the name, the second one requires *Attribute* to be contained within *KeyClass*, and the third one states that *Attribute's* type must be *EJBDataType*. The constraints have been cut off at the edge of the class. For example, the complete constraint for *KeyClass* is `name = class.instance.name +'Key'`. The default values for attributes have been omitted, since in this case they would look exactly like the constraints.

To apply the patterns, the designer opens the source model in Rose and starts MADE. She can then select, e.g. the Class-to-KeyClass -pattern (Fig. 4, on top) and begin applying it. No roles are bound yet, and only two of the pattern's roles, *Class* and *EJBDataType*, do not depend on other roles. Therefore there will be two visible tasks: `Provide 'Class'` and `Provide 'EJBDataType'`. The designer can now select the task for *Class* and choose to locate an existing element.

Figure 5a contains a screenshot of MADE at this moment. The top left corner shows the pattern selection, as well as a list of pattern instances. The next pane to the right contains a view of the active pattern instance. This pane is empty, because there are no elements bound to the pattern instance's roles. The pane in the top right corner shows current tasks. A dialogue for selecting a UML class to be bound to the *Class* role is in the lower left corner.

The designer chooses to apply the transformation on the class *Breakfast*. The class *Breakfast* is bound to the role *Class*, satisfying the task `Provide 'Class'`, which disappears. A new task appears for *KeyClass*, because it only refers to *Class*, which is now bound. *Breakfast* appears in the list of bound elements, signifying that it is bound to a class role. Figure 5b shows the list of bound elements and unfinished tasks as they would appear.

Let us assume that the target model has already been populated with the basic data types, such as string, integer, etc. The designer can select the task for *EJBDataType* and choose to locate an existing element. The constraint (`name = 'Integer'`) is used to locate the class. The task is now completed, and disappears (Fig. 5c). *Integer* is added to the list of bound elements. No new tasks appear.

The designer highlights the remaining task and chooses to automatically complete it. Default value has been specified for role *KeyClass* and the tool creates a new class with the name *BreakfastKey* in Rose. The generated element is bound to the role, and added to the list of bound elements. The completed task disappears and a new one appears for the role *Attribute* (Fig. 5d).

The element for the attribute role, too, can be generated, and the designer chooses to have MADE do that. A new attribute is created for the class *BreakfastKey*, the name of the attribute is set to `"BreakfastID"`, and its type is set to be the class *Integer*. The task is completed, *BreakfastID* is added to the list of bound elements, and the task disappears. The pattern has now been applied successfully for *Breakfast*. For this PIM, the pattern would be applied 4 more times, once for each regular class.

For the two association classes, the pattern in the lower half of Fig. 4 is used. After choosing to apply the pattern, tasks appear for *AssociationClass* and *EJBDataType*. The latter can be automatically bound to the correct basic type (*Integer*). The designer has to select the association class manually, and she picks *Change*. Three new tasks appear, one for each of *KeyClass*, *AssocEnd1*, and *AssocEnd2*. The designer lets the tool create an element for *KeyClass*. She completes the active tasks by manually binding each association end.

The two new tasks (for *Class1* and *Class2*) could be fulfilled automatically, since an association end can only be connected to a single class. However, the automatic locating of elements in MADE works based on the value of the name field only. The designer has to choose those manually, too. The elements for *Attribute1* and *Attribute2* can be generated automatically, and that is what she decides to do. Applying the pattern is finished.

6 Evaluation

The first thing to note about the example is how much user interaction it requires. Each transformational pattern must be applied manually, and the user must initiate each generate or locate operation, even if the tool can complete it autonomously. With a source model of 41 elements (classes, attributes, etc.), as in the example, there will be 41 instances of transformational patterns applied. With the exact rules used here, this translates to 163 manual selections and 93 automatically located or generated elements. This observation, although correct, is in many ways misleading.

The numbers are in no way absolute, because they are highly dependent on how the rules are modelled. Regardless of the exact numbers, the burden on the user is far too high. However, the low level of autonomy is due to the tool, not the approach, and the user interaction could be greatly reduced with simple measures. In fact, because this particular set of rules is unambiguous, the user's participation could be limited to simply initiating the transformation. It should be noted, that we do not consider full automation as an important goal.

MADE can be instructed to automatically bind every role as soon as the roles it depends on are bound. If this option were extended to take into account more than just the name field, the user would be relieved of many monotonous tasks. If, in addition, each transformational pattern were applied automatically to each configuration of elements that satisfies the pattern's structure and other constraints, all but 14 cases of user choices could be eliminated. In those cases, a single pattern by itself does not have enough information of the transformation as a whole to locate or generate the necessary elements. In order to make the correct choices, the tool must have some idea of the way the individual patterns overlap and interact. We believe this could be achieved by expanding the experimental pattern composition functionality to allow dynamic composition.

Applying transformational patterns, even with the most basic tool support, fulfills many of the requirements discussed in Sect. 2. The approach is transparent, and does not hide transformation mechanics. The designer has full command of the process, and can change any details right down to the level of individual bindings. Even when the tool is improved to better facilitate automatic steps, they will only be engaged at the user's discretion, not the tool's. This all helps the designer to understand each step of the process, which leads to openness.

Customisability is limited to user's choices and relies on her decisions. A task can be defined as optional, and if chosen, can reveal an otherwise inaccessible path of tasks. It might also be possible to use dynamic composition of patterns to introduce more elaborate variation points. Customisability is one of the areas where better mechanisms and further research is needed.

Patterns, as implemented on MADE, are biased towards a direction, because dependencies are directed. But patterns as described in Sect. 3 use associations to express relations between roles. Forcing the user to choose in which role a symmetric constraint is placed does tilt the balance in favour of one direction over the other. Using associations, the other role(s), too, could have such a symmetric constraint. Constraints could even copied and added automatically for simple constraints, such as equality, that are easily recognised as symmetric.

MADE stores information about bindings, which makes the record of transformation persistent. Even after modifications to the models, the record can still be used as a starting point for synchronizing the models. None of the user decisions are lost, although some might have become irrelevant. Reversibility and traceability can thus be achieved. On the other hand, MADE lacks facilities, illustrative or not, to visualise these mappings. Elements that are bound to a particular pattern instance can be highlighted in Rational Rose, but there is no indication of which role an element is bound to. So, it is possible to find out information about the relations between elements of different models, but there is no easy way to study it in the tool.

Using patterns for transformations enables performing the transformation in small, incremental steps. The problems with the current machinery are in combining these steps into transformations and, further combining transformations together into bigger transformations. This has a negative effect on reusability and maintainability. Lack of visualisation also makes documenting more difficult.

To recap; our approach, as it is now, provides open, incremental, traceable, reversible, and unbiased transformations, but has problems when it comes to visualising, customising and combining transformations. The current tool environment works for evaluating the approach, but is not mature enough for real transformations. It burdens the user with some tasks it should carry out automatically and supports only unidirectional transformations. Improving facilities, both with the approach and the tool, to properly address the challenges is vital.

7 Related Work

Work by Hausmann et al. on visualizing model mappings [8] is partly driven by goals similar to ours. In their work, mappings between model elements are thought of as relations in the mathematical sense. Model mappings are expressed with extended UML class and object diagrams. The importance of comprehensible transformations and rules is one of the main issues raised and discussed. Being based on relations, the approach encourages bi-directional transformations.

QVT-Partners' response [9] to the Query / Views / Transformations (QVT) request for proposals is one of the most detailed and finished work. It describes a language for transformations and a textual and a visual representation for it. Transformations are divided into relations and mappings. Relations are biirectional, but can only be used for checking whether the source and target model are properly synchronized. Mappings are used for performing a transformation, but are restricted to one direction.

Many approaches at MDA transformations are based on graph grammars. Such approaches tend to produce strictly unidirectional transformations due to the clear separation to left hand side (LHS) and right hand side (RHS) in individual rules. Also, definitions are often given only in a textual form. For example, GREAT [10] is a graph rewrite system for transformations on UML models, where LHS is defined with a textual language. RHS is defined as Java code, which manipulates the model through an API. Such transformations are, of course, unidirectional.

The theories behind VMT [11] and BOTL [12], too, are based on graphs. Both use attributed labelled graphs and offer a graphical notation for describing a source (LHS) and target template (RHS) for transformation rules. In VMT transformations can only be performed on UML models and in one direction. BOTL transformations can be bi-directional and can handle arbitrary metamodels. The expressive power of VMT's visual notation is enhanced with OCL.

Some approaches use XSLT to process models in XMI form. Due to the nature of XMI these approaches are rarely limited to a single metamodel. XSLT-based methods are often textual, but UMLX [13] is a graphical transformation language. The metamodels (called "schemas" in UMLX) is given using a subset of the UML class diagram notation. Transformations are defined with an extended class diagram notation, and are translated into XSLT form using the information about the structure of the metamodel. The XSLT is then executed on the input, which is provided in XMI form. Transformations are unidirectional.

A textual transformation language, based on rules, is described in [7]. The language is intended as a means to illustrate the sample transformations, and not as a real transformation language. Transformations are composed of small rules and can be declared as either unidirectional or bi-directional. OCL has an important role in the language.

The focus on most papers is on explaining the mechanics of the language or approach. Characteristics such as openness, customisability, maintainability, and how illustrative are the notations used, fall out of scope. It is therefore difficult to determine how much emphasis is placed on these aspects, which in our view are of great importance.

Similarly to our approach, Catalysis [14] makes use of role-based patterns (so-called "frameworks") for describing abstract collaborations of model elements. A major difference is that Catalysis emphasizes specifications of the semantics of the collaboration while we have a more pragmatic view emphasizing task-driven model (or code) generation based on the default value specifications.

8 Concluding Remarks and Future Work

In this paper we first listed and discussed what we believe to be key requirements for MDA transformations: openness, customisability, combinability, traceability, and maintainability. The new approach at MDA transformations, transformational patterns, was described and explained. A more generic pattern tool, MADE, was presented briefly. An example of performing a simple transformation using transformational patterns and the tool was presented. The approach was evaluated in light of its applicability in the example and its compliance with the key MDA requirements. Last, other groups' work on visualising and defining model transformations and mappings was discussed.

The example was very limited, but it did indicate some strengths and weaknesses of transformational patterns. We wish to pursue several related issues further. Visualisation of patterns, as well as the reverse, discovering patterns from examples are important for usability. Defining and utilizing variation points is another area of interest for us. The rule composition mechanism needs more flexibility. We are looking into implicit and explicit rule scheduling, as well as some hybrid solutions. Also, the tool support must be elevated. We are currently working on supporting arbitrary MOF-based metamodels in processing and visualisation. We hope to carry out a more realistic case study, where neither the transformation specification nor the set of models is unrealistically simple.

References

1. OMG: Model driven architecture (MDA) (2001) On-line at http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf.
2. Interactive Objects Software: Arcstyler tool homepage (2004) on-line at http://www.arcstyler.com/.
3. Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A., Viljamaa, J.: Generating application development environments for java frameworks. In: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering, Springer-Verlag (2001) 163–176
4. OMG: MDA guide version 1.0.1 (2003) On-line at http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.
5. Hammouda, I., Pussinen, M., Katara, M., Mikkonen, T.: Uml-based approach for documenting and specializing frameworks using patterns and concern architectures. In: The 4th AOSD Modeling With UML Workshop. (2003)
6. Rational: Rational Rose home page (2004) on-line at http://www.rational.com/.
7. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
8. Hausmann, J., Kent, S.: Visualizing model mappings in UML. In: Proceedings of the ACM Symposium on Software Visualization. (2003)
9. QVT-Partners: Revised submission for MOF 2.0 Query / Views / Transformations RFP (2003) On-line at http://www.omg.org/cgi-bin/apps/doc?ad/03-08-08.pdf.
10. Christoph, A.: Graph rewrite systems for software design transformations. In: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, Springer-Verlag (2003) 76–86
11. Sendall, S., Perrouin, G., Guelfi, N., Biberstein, O.: Supporting model-to-model transformations: The VMT approach. In Rensink, A., ed.: CTIT Technical Report TR-CTIT-03-27, Enschede, The Netherlands, University of Twente (2003) 61–72
12. Braun, P., Marschall, F.: BOTL - the bidirectional object oriented transformation language. Technical Report TUM-I0307, Technische Universität München (2003)
13. Willink, E.D.: UMLX: A graphical transformation language for MDA. In Rensink, A., ed.: CTIT Technical Report TR-CTIT-03-27, Enschede, The Netherlands, University of Twente (2003) 13–24
14. Catalysis: Catalysis home page (2005) on-line at http://www.catalysis.org/.