Applying MTBE Manually: a Method and an Example

Fábio Levy Siqueira[1, 2] and Paulo Sérgio Muniz Silva[2]

[1] Programa de Educação Continuada da Poli-USP, São Paulo, Brazil
[2] Escola Politécnica da Universidade de São Paulo, São Paulo, Brazil
fabio@levysiqueira.com.br, paulo.muniz@poli.usp.br

Abstract. The idea of model transformation by example (MTBE) is to use an example of the transformation to derive transformation rules. Although some works propose automatic and semi-automatic approaches, there are some limitations to use them in practice. As a result, depending on the model transformation it may be necessary to apply an MTBE approach manually. Therefore, this work presents a manual MTBE method. It also describes an example of its use in a stakeholder requirements to software requirements transformation, which was the motivation to create the method.

Keywords: method, manual, transformation, MTBE, stakeholder requirements.

1 Introduction

Model transformation by example (MTBE) approaches[1] simplify the definition of model transformations by allowing the transformation designer (modeler) to work with the concrete syntax of the metamodel [3]. Ideally, the designer does not need to understand details of the metamodel abstract syntax or the transformation language. The transformation rules are generated automatically or semi-automatically by a rule inference mechanism which analyzes the operations executed or a mapping.

However, there are some limitations to use MTBE approaches in practice. There are few tools [10, 12] that implement the proposed approaches. Moreover, these tools have limitations and they do not generate transformations in some transformation languages[2], such as QVT [4]. As a consequence, the designer may need to apply a MTBE approach manually if he wants to create a transformation based on an example of the source and target models. However, as these approaches were created to be executed automatically, apply them manually may be difficult. For instance, in the MTBE approach proposed by Varró [11], to create and analyze contexts manually (*n*-context, generally) in a real example can be complex and tiresome.

---

[1] This work uses the term "model transformation by example" as a general concept, including model transformation by demonstration.

[2] Ideally, the transformation language generated would not be important, as the designer describes the transformation using only the concrete syntax. However, usually the designer needs to manipulate the generated code to correct or optimize the transformation.

Furthermore, some MTBE approaches have theoretical limitations that can be problematic for specific transformations. A common limitation is the existence of *n:m* mappings, i.e., a mapping with more than one element as source and with more than one element as target. Yet, these mappings are necessary when there is more than one source and/or target model, or in complex transformations. Even if this mapping is discussed by Strommer, Murzek, and Wimmer [9], it is not proposed an algorithm to implement it. Another limitation is the attribute operations allowed. Some works discuss attribute operations [9, 10], but it is not possible to have complex operations such as combining several attributes in the source model to generate an attribute in the target model. Finally, some types of mappings may be difficult to describe in the concrete syntax, such as mapping implicit elements [9].

When there is no adequate MTBE tool and yet using the by-example philosophy may help finding transformation rules – for instance, in complex transformation where the rules are not evident –, a manual approach can be useful. To help applying MTBE manually, this work proposes a method. It uses source and target model examples to generate manually and iteratively the transformation rules. As it does not pose any constraints in the models used, it can be used in horizontal or vertical, and endogenous or exogenous transformations. Furthermore, it can be used in transformations where the source model and/or the target model are composed of more than one model.

This work is organized as follows: In Section 2 it is presented the proposed method. In Section 3 it is discussed the related works. Next, in Section 4 it is presented an example of how the method can be used. The example describes a transformation from stakeholder requirements to software requirements, which was the motivation to create the method. In Section 5 it is discussed some issues and limitations of the method. Finally, in Section 6 it is presented the conclusion and future works.

2 Manual Method

We propose an iterative method in which the designer manually creates new rules by comparing the target model example (the expected result) to the output of a transformation of the source model example. This output is obtained by applying a set of existing transformation rules, which may be initially empty. Therefore, the method has three inputs: a target model, a source model, and a set of initial rules. The steps are the following:

1. Apply the existing transformation rules on the source model example and obtain a target model.
2. Compare each element of the obtained target model to the elements of the target model example (expected result). If the element analyzed is semantically different from the one in the expected result (i.e., it represents a different information) or if there is no element in the obtained model corresponding to an element in the expected result, execute:

(a) If the element is different, search in the source model example for the element that has the necessary information and that seems to be the source of it.
   (i) Identify the transformation rule that generated this element in the obtained target model.
   (ii) Analyze whether the transformation rule should be specialized, an alternative for it should be created, the rule should be improved or revised, or whether is not possible to improve the rule.
   (iii) If a transformation rule requires improving or revising, first verify whether the change is relevant to all cases of this rule. If so, add the elements, the new information, or revise the transformation rule; if not, the rule must be specialized.
   (iv) If the transformation rule requires specialization, identify the condition to specialize it, taking into account the source model example. Then define the new rule, considering the source model example and the expected element, based on the target model example.
   (v) If an alternative is to be created, first create an initial transformation rule. Second, create a more general rule that is valid for the existing rule and the initial new rule considering the source model example. Then, represent the alternatives as items in this general rule. If the rule already has alternatives, just add the new alternative.
   (vi) If it is not possible to improve the transformation rule, continue with step 2.

(b) If there is no element in the obtained model, i.e., if there is no related rule, create a new rule.
   (i) Search in the source model for elements that have the information needed for the rule.
   (ii) Analyze the condition necessary for the rule to be valid.
   (iii) Create the rule.

3. If any change has been made in the rule set, re-execute the method from step 1.

The main idea of the method is to improve the set of rules, either by adding new elements, specializing, revising, representing alternatives, or proposing new rules. To obtain these rules there are several subjective decisions, which depend on the designer's experience. For example, in step 2, there is a semantic analysis; in steps 2.a.iii, 2.a.iv, 2.a.v, and 2.b.ii the designer must find relationships between elements of different models; and in steps 2.a.iii, 2.a.iv, 2.a.v, and 2.b.iii, new rules must be written. As other MTBE approaches, the generated rules depend both on the system in which the method was applied, and on the quality of the example used as input.

3 Related Work

The method is based on the philosophy of by-example approaches [3, 11, 12], using examples to create the transformation rules. The main difference between this approach to other MTBE approaches, such as the proposals of Varró [11] and Wimmer et al. [12], – besides from being manual – is that the transformation rules are generated by comparing the target model example to the model generated by applying

the existing transformation rules in the source model example. Based on the differences between these two models, the designer generates the transformation rules. Therefore, there is not a formal mapping between the models. On the other hand, in the approaches proposed by Varró [11] and Wimmer et al. [12], the rules are generated based on a mapping of the source model example to the target model example, which is created by the designer. The choice for comparing the models instead of using a mapping was made considering complex transformations. In these situations, the mapping is not clear, and its definition is heuristic. Therefore, it is possible that a new transformation rule conflicts with an existing rule. To prevent these problems, the transformation is re-executed on each iteration. Moreover, in complex transformations it is common that several objects are involved in a transformation, and a single object may be used by several rules. This makes the mapping even harder, and it seems easier to create the transformation rules by analyzing one mapping per iteration.

The method has similarities with the application of the concept of test-driven development (TDD) to generate model transformations [1]. The source and target model examples can be seen as a set of test cases, while each method iteration can be seen as a TDD process cycle. Yet, the main differences of this approach are that we do not propose creating formal test cases and we use an example as the transformation specification. Furthermore, we propose guidelines to create the transformation rules.

4 Example

To exemplify the proposed method, we describe its use to produce requirement refinement rules. In fact, the proposed method was created to solve this problem.

One of the key responsibilities of Requirements Engineering is to transform stakeholder requirements into system and software requirements. Stakeholder requirements represent "the intended interaction the system will have with its operational environment" [2, p.19], considering the needs of users, operators, and other stakeholders. Because these requirements may not be sufficiently detailed to be used during the development activities, they must be analyzed and transformed into system requirements, which represents a technical specification for the system [2]. Some system requirements may be further detailed in order to be allocated into a software element of the system, resulting in software requirements.

Traditionally the transformation from stakeholder requirements into system and software requirements is executed manually by a requirements engineer. In another work [7], we propose a semi-automatic transformation of stakeholder requirements into software requirements. This transformation considers only functional requirements, using an enterprise model as source and a use case model as target. The main benefit of this approach is avoiding mistakes during the requirement analysis, as the requirements engineer would make fewer decisions. Furthermore, it facilitates managing changes in requirements, which is essential in dynamic environments.

To implement this transformation, some transformation rules were obtained by analyzing related works [7]. However, as these works propose a transformation

between requirements notations – and not between different levels of requirements –, using these rules typically generate a software requirements model that only organize differently the information available in the stakeholder requirements model. To obtain more adequate rules, we applied the proposed manual MTBE method.

4.1 Metamodels and initial rules

The stakeholder requirements model used as input for the transformation is composed by an as-is and a to-be models, both conforming to a single enterprise metamodel. Therefore, the transformation has two source models. The enterprise metamodel is organized in five views: organization, process, layout, motivation, and document. In Figure 1a, it is presented a MOF class diagram representing part of the abstract syntax for the process, document, and organization views, organized in packages. A detailed discussion of this metamodel is presented in another work [8].

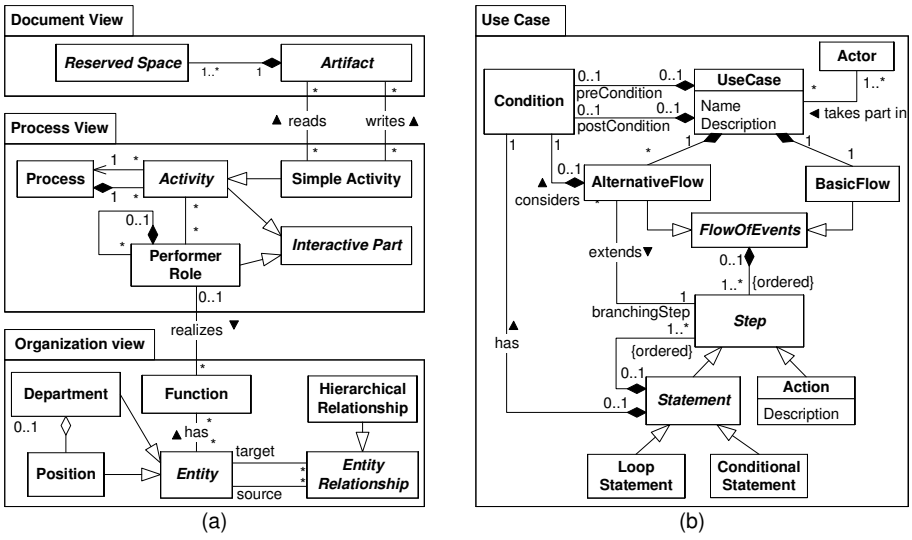

Fig. 1. Part of the (a) enterprise metamodel and the (b) use case metamodel.

Table 1. Four of the eleven transformation rules found in the analysis of related works.

BR1. A *PerformerRole* executing *SimpleActivities* that exchange (sends or receives) a *MessageFlow* with a *SimpleActivity*, executed by a *PerformerRole* carried out by the System, is an Actor.
BR2. A *Function* executed by a *PerformerRole* who is an Actor is a UseCase. This Actor participates in the UseCase.
BR3. A *SimpleActivity* that exchanges a *MessageFlow* with the *PerformerRole* carried out by the System will be a Step of the UseCase. If several *SimpleActivities* interact with the System, then the first *PerformerRole* who interacts with it defines the UseCase. The *Activities* from other *PerformerRoles* will be also a Step, but they do not define a UseCase.
BR4. A *SimpleActivity* internal to a *PerformerRole* carried out by the System, which exchanges *MessageFlow* with a *SimpleActivity* indentified as a Step (in BR3) will be a Step of the same UseCase.

The as-is and to-be models are transformed into a textual use case model, which is used as a software requirements model. The use case metamodel was created based on a survey of the most frequent elements proposed by existing metamodels and templates [6]. Part of the metamodel abstract syntax is presented in Figure 1b.

Eleven transformation rules were found by analyzing related works [7]. In Table 1 it is presented four of them. They are called "BR", referring to "basic rules". Terms in italics represent meta-classes of the enterprise metamodel, and underlined the meta-classes of the use case metamodel. These rules are described in natural language, although they were implemented using Operational QVT [4].

4.2 Context

To generate stakeholder to software requirements transformation rules, we applied the method in a fictitious library system. The system's goal is to control the book lending procedure. Considering this context, it was created a source model example, which contains two models: the library as-is and the library to-be. Figure 2 describe the processes as-is and to-be pertaining to book registration, using BPMN [5] as concrete syntax. Part of another view, the as-is and to-be document view, is also presented in Figure 2. The *Artifacts* are represented using the same concrete syntax of an artifact in the BPMN diagram, and the *ReservedSpaces* is represented as rectangles inside it.

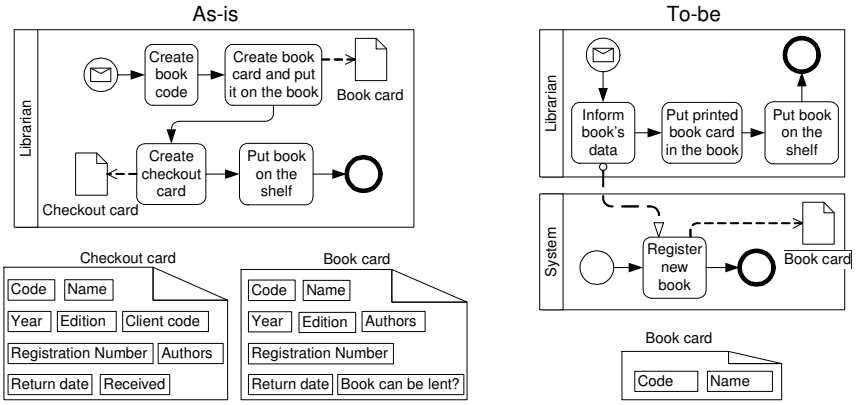

Fig. 2. Process view and document view pertaining to book registration process.

Seven use cases were created for this example as the target model example: book registration, book loan, book return, payment of fines, customer management, and book management. In Table 2 it is presented the use case for book registration.

4.3 Applying the method

To exemplify the method, just the book registration use case will be considered. The first step of the method is to apply the existing transformation rules to the source model example. Therefore, the transformation rules found in the analysis of existing

studies were applied. The use case obtained is presented in Table 3, referenced as the "first execution", indicating in parenthesis the rules that generated each element.

Table 2. The target model example.

Actor	Librarian
Precondition	None
Basic Flow	1. The system requests the following information: whether the book can be lent, name, authors, year, edition, and number of the book. 2. The librarian provides the information. 3. The system prints a book card with the following information: name and code.
Alternative Flow	Step3: The librarian does not provide some information. 3. The system presents an error message and the use case ends.
Post-condition	None

Table 3. First and second executions of the transformation rules.

	First Execution	**Second Execution**
Actor	Librarian (BR1)	Librarian (BR1)
Precondition	None	None
Basic Flow	1. Inform book's data (BR3) 2. Register new book (BR4)	1. The system requests the following information: (RR1) • Book can be lent?, name, authors, year, edition, and number of the book. 2. The Librarian Inform book's data. (RR1) 3. Register new book (BR4)
Alternative Flow	None	None
Post-condition	None	None

The second step of the method is to compare the elements of the target model (obtained use case) to the elements of the expected result (use case example, in Table 2). In this case, the Actor was correctly identified, but the Steps of the BasicFlow are different. The Step 1 of the use case is not detailed enough to be seen as a software requirement (it is not clear what "book's data" means). In Step 2, the book card should be printed (in addition to registering the new book). In other words, both Steps 1 and 2 in the obtained use case are semantically different from the ones in the expected result. Furthermore, the AlternativeFlow was not identified in the obtained use case.

Because there are elements in the obtained use case that are semantically different from the use case example, step 2.a of the method should be executed. Starting with Step 1 of the obtained use case, the rule that generated it should be identified (2.a.i). In this case, it was the rule BR3. Then it must be analyzed whether the rule should be specialized, whether an alternative must be created, or whether the existing rule requires improvement (2.a.ii). In the example, the problem with the generated Step is that it is imprecise, so the solution is either to specialize BR3 or add an element to it. After the execution of step 2.a.iii, it seems that the difference between the use cases is due to the *Artifacts*: their *ReservedSpaces* should be considered in the Step. Because not all *Activities* write *Artifacts*, a change in this rule would not be relevant to all cases. Then, step 2.a.iv must be executed and a condition for the rule specialization should be found. In this case, it seems that the book data comes either from the "book card" or the "checkout card" in the as-is model. Both contain the information needed, but since the expected use case goal is to register a book (and not something related to

a checkout), the most appropriate *Artifact* seems to be the "book card", as it represents information about the book. Considering this *Artifact* and the as-is and to-be processes, the condition for the extension seems to be related to the fact that the *Activity* that creates the "book card" is substituted in the to-be model by an *Activity* that exchanges messages with the system ("Inform book's data"). Therefore, a possible condition is: "there is an *Activity* in the as-is model that writes in an *Artifact* and is substituted by an *Activity* that exchange messages to the System *PerfomerRole*". However, this condition has a problem: it is also true for the *Activity* that creates the "checkout card", as the *Activities* that create the "book card" and the "checkout card" are substituted by the same *Activity* in the to-be process. Worse, the information used to select the "book card" *Artifact* instead of the "checkout card" is not represented directly in the model: it comes from the semantics of the expected use case and the process to-be. Therefore, a solution is to ask the Requirements Engineer what *Artifacts* should be considered – and what *ReservedSpaces* should be used. As a result, the condition can be changed to "there are *Activities* in the as-is model that write in *Artifacts* and are substituted by an *Activity* that exchange messages to the System *PerfomerRole*". The rule should generate two Steps if this condition is true (below, a term between < and > is a piece of meta-information):

- "The system requests the following information", with the *ReservedSpaces* of the *Artifacts* as the information to be filled. The user needs to choose the necessary information from the available *Artifacts*.
- "The <Actor's name>" + the *Activity* executed by the Actor in the to-be model (inform book data, in this example).

This rule will be called RR1 (from refinement rule). Due to space limitations, its code in Operational QVT will not be presented. Since a rule was created, step 3 indicates that the method should be re-executed, returning to step 1. The result of applying once again the transformation rules is presented in Table 3 (second execution). Although only two Steps have changed, it is possible to see that the generated use case is more similar to the use case example.

Another rule should be created if the method is executed until this use case is semantically equivalent to the expected use case. This rule is called RR2, and it prints an *Artifact*. Furthermore, RR1 should be modified to generate an AlternativeFlow. Due to space limitations, these rules and the other rules obtained by executing the method in all use cases of the example will not be presented[3].

5 Discussion

The proposed method obtains transformation rules based on the concept of model transformation by example. Differently from other MTBE approaches, the rules are generated manually. Therefore, the designer must be proficient in a model

[3] The transformation rules generated are implemented by a tool, which is available at http://www.levysiqueira.com.br/projects/emucase/.

transformation language. As some rules may be complex, it may be difficult for him to specify a rule. Even though the main problem of the method is the fact it is manually executed, it also results in some benefits: more than one target and source models can be used (in the example, two source models were used); the transformation rule may include information not present in the source metamodels (in the example, text was added to the attributes); and there are no limitations on the rule complexity (in the example, it was possible to create a rule that asks the user for information).

Another issue of the method is applying the rules created. As several rules may be created when analyzing an example, it may be difficult for the designer to know which rule is being applied and exactly its results. To alleviate this issue, we recommend writing the rules directly in a model transformation language and applying them automatically. Yet, it is possible that a new rule have an impact on existing rules. Although the step 3 of the method is defined to avoid this problem, a new rule may affect other rules in a situation that is not taken into account in the example in which the method is being applied.

Other issues and limitations of this method are similar to other MTBE approaches. This method also depends on the designer, who must be able to do a semantic analysis and find the relationship between models. However, as the designer also creates the transformation rules, this dependency is greater than the dependency in other MTBE approaches. Another issue is the adequacy and correctness of the example used. Ideally, the example must be a prototypical situation, generating transformation rules that are not specific to the example. In complex mappings, it may be necessary to apply the method in different models to improve the set of rules.

6 Conclusions

The present work proposes a manual method to apply the philosophy of model transformation by example (MTBE). This method was created considering complex transformations, involving several source and target models and with no evident transformation mapping. Therefore, its main contribution is providing a support to generate transformation models. Moreover, it describes a different MTBE approach and presents an example of its application.

One of the main characteristics of existing MTBE approaches is being automatic or semi-automatic. However, the fact that the proposed method is manual does not make it "non-MTBE". The method uses the essence of MTBE, using an example to generate the transformation. It was created considering a real problem of generating transformation rules for a stakeholder requirements to software requirements transformation – which is presented in this work as an example of the proposed method. In complex transformations, such as this one, it is easier to propose transformation rules by analyzing an example. Therefore, the method is a pragmatic answer to the difficulties of applying MTBE approaches in practice. It illustrates that the philosophy of MTBE can be applied even when automatic or semi-automatic approaches cannot be used.

As future works, the method can be improved by describing more precisely how to write a transformation rule and how to implement it in Operational QVT (or other transformation language). This information can be used to develop a tool that helps creating rules, thus assisting the designer decision process and preventing incorrect decisions. Regarding the impact analysis, a future work is to define a test suite for the rules, which would uncover problems that a new rule might cause. This suite could be also implemented in a tool.

Acknowledgements. This work was partly supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grant 2007/58489-4.

References

1. Giner, P., Pelechano, V.: Test-Driven Development of Model Transformations. In: 12th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'09). LNCS, vol. 5795, pp. 748–752. Springer (2009)
2. ISO: Systems and software engineering - Life cycle processes - Requirements engineering. ISO/IEC/IEEE 29148 (2011)
3. Kappel, G., Langer, P., Retzschitzegger, W., Schwinger, W., Wimmer, M.: Model Transformation By-Example: A Survey of the First Wave. Conceptual Modelling and Its Theoretical Foundations. LNCS, vol. 7260, pp. 197-215. Springer (2012)
4. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.1, formal/2011-01-01 (2011)
5. OMG: Business Process Model and Notation (BPMN). Version 2.0, formal/2011-01-03 (2011)
6. Siqueira, F.L., Muniz Silva, P.S.: An Essential Textual Use Case Meta-model Based on an Analysis of Existing Proposals. In: 14th Workshop on Requirements Engineering (WER'11), pp. 419-430 (2011)
7. Siqueira, F.L., Muniz Silva, P.S.: Transforming an Enterprise Model into a Use Case Model Using Existing Heuristics. In: 1st Model Driven Engineering Workshop (MoDRE'11), pp. 21- 30 (2011)
8. Siqueira, F.L., Muniz Silva, P.S.: Using an Enterprise Model as a Requirements Model in Process Automation Systems: A Proposal. In: 8th Int. Conf. on Information Systems and Technology Management (CONTECSI'11), pp. 3064-3088 (2011)
9. Strommer, M., Murzek, M., Wimmer, M.: Applying Model Transformation By-Example on Business Process Modeling Languages. In: 26th Int. Conf. on Conceptual Modeling (ER'07). LNCS, vol. 4802, pp. 116-125. Springer (2007)
10. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. In: 12th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'09). LNCS, vol. 5795, pp. 712–726. Springer (2009)
11. Varró, D.: Model Transformation by Example. In: 9th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'06). LNCS, vol. 4199, pp. 410–424. Springer (2006)
12. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation Generation By-Example. In: 40th Hawaiian Int. Conf. on Systems Science (HICSS'07). IEEE Computer Society (2007)