Evolving Legacy Model Transformations to Aggregate Non Functional Requirements of the Domain

Joseba A. Agirre, Goiuria Sagardui and Leire Etxeberria
Department of Computing, Mondragon University, Loramendi, Mondragon, Spain

Keywords: Model Driven Development, Model Transformation, Model Transformation by Example, Model Transformation by Demonstration, Non Functional Requirements, Design Patterns.

Abstract: The use of Model Driven Development (MDD) is increasing in industry. When a Non Functional Requirement (NFR) not considered in the development must be added metamodels, models and also transformations are affected. Tasks for defining and maintaining model transformation rules can be complex in MDD. Model Transformation By Example (MTBE) approaches have been proposed to ease the development of transformation rules. In this paper an approach based on MTBE to derive the adaptation operations that must be implemented in a legacy model transformation when a NFR appears is presented. The approach derives semi-automatically the model transformations using execution traceability data and models differences. An example where access control property is integrated on a MDD system is introduced to demonstrate the usefulness of the tool to evolve model transformations.

1 INTRODUCTION

Software is increasingly becoming an integral part of electronic-end-customer products. When developing embedded systems the requirements to fulfill are not only defined in terms of the functional aspects of the system, but also on different design requirements, such as size, power consumption, response time, security or reliability, usually called Non-Functional-Requirements (NFR). NFRs are critical in the development of embedded systems. Designing an embedded real-time system is a complex process, which involves modeling, verification, validation of functional and non functional requirements. The combination of model driven software development (MDD) (Völter et al., 2013) and software architecture concepts is considered especially advantageous for developing complex systems, such as embedded systems (Bunse et al., 2009).

The Model Driven development (MDD) paradigm raises the abstraction level of system specifications and increases automation in system development. MDD uses models as the primary artifact of the software production process, and development steps consist of the application of transformation steps over these models. On MDD, a model transformation is specified through a set of transformation rules, usually using transformation languages such as ATL (Jouault et al., 2008), QVT (OMG, 2011) or EPSILON (Kolovos et al., 2008). There are two kinds of model transformations: endogenous and exogenous. Endogenous transformations are transformations between models expressed with the same meta-model. Exogenous transformations are transformations between models expressed using different meta-models. Tasks for defining, specifying and maintaining transformation rules are usually complex and critical in MDD.

Most current MDD approaches focus on system functional requirements, and do not integrate NFRs into the MDD process. When a NFR not considered in the development must be added to the software not only the final source code is affected, also the metamodels, transformation rules and models must be evolved. Effectively integrating NFRs into the MDD production process requires several activities: (I) metamodels must be extended with the NFR (II) transformation rules must be adapted and (III) input models must be refined. The aim of our work is to ease the adaptation process of the transformation rules once metamodels have been extended to contemplate the NFR in the design models. In this paper, we present an approach and a tool to develop and evolve semi-automatically ATL transformation rules. To derive the adaptation operations that must be implemented in a legacy model transformation when a NFR appears is presented is not an easy task.

A. Agirre J., Sagardui G. and Etxeberria L..
Evolving Legacy Model Transformations to Aggregate Non Functional Requirements of the Domain.
DOI: 10.5220/0005227304370448
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 437-448
ISBN: 978-989-758-083-3

The proposed approach is based on Model Transformation By Example (MTBE) (Varró, 2006) concept. By-example approaches define transformations using examples models. Examples are easier to write than a transformation rule. In MTBE starting from pairs of example input/output models the transformation rules are derived. The approach, based on MTBE, is focused on generating semi-automatically transformation rules from pairs of example input/output models and transformation rules execution traces (see figure 1). The model transformation developer applies a demonstration based approach to specify the desired transformation of non-functional properties and semi-automatically the model transformation is adapted. Once the model transformation has been adapted to integrate the NFR the MDD system can be applied to any application design that integrates the NFR specification. This paper provides the following contributions to the study of non-functional system property in the maintenance of model transformations:

- A metamodel for expressing adaptation operations for transformation rules
- An MTBE approach and a tool to evolve ATL transformations. The tool is called Transevol.
- An example of the integration of security properties on a MDD system evolution tasks. Concretely an exogenous model transformation is adapted to integrate the security property.

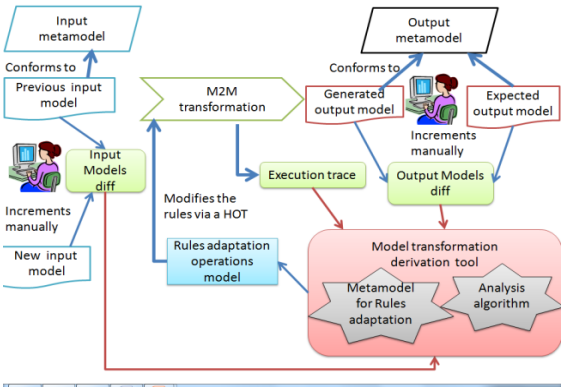


Figure 1: Approach for automatic model transformation analysis to derive adaptation operations.

In the following sections we detail the solution which guides the evolution of model transformations to integrate non functional requirements. A case study to illustrate the approach is presented in section 2. Section 3 describes the developed MTBE approach for the model transformation development. The fourth resumes the results of applying the approach on the selected case study. Then in section 5 the conclusions and future work are resumed. Finally in section 6 a brief description of the related work is presented.

2 THE CASE STUDY

This section presents the example used to illustrate the integration of non functional properties in the model transformation of a MDD system using the Transevol tool. The selected MDD system transforms UML components models into models that represent C code of applications. The new non functional requirement to integrate in the MDD system is a role based access control (RBAC). A UML profile is used to specify RBAC properties. The protection proxy pattern (Buschmann et al., 1996) is used to implement the RBAC pattern at the code level. The M2M transformation rules must be adapted to transform the RBAC design data into ANSI-C implementation of the protection proxy pattern. The approach deducts automatically the adaptations operation that must be implemented on the transformation rules using a pair of example input/output models demonstrating the new transformation requirement. The case study is explained in more detail in next subsections.

2.1 The Model Driven System

The MDD system (Agirre et al., 2012) generates ANSI-C code from component-based SW architectures, designed in UML in two steps. As in Model Driven Architecture (MDA) platform independent models (PIM) are transformed into platform specific models (PSM), and finally the PSM is transformed in code. UML is used as the component metamodel for the design. The UML designs are transformed to intermediate models representing ANSI-C code through a model to model (M2M) transformation. SIMPLEC (Agirre et al., 2010) meta-model is used to represent a subset of ANSI-C. The exogenous M2M transformation is implemented in ATL. Once the SIMPLEC models are obtained, a Model to Text (M2T) transformation is applied to SIMPLEC models to generate ANSI-C code. XPAND2 based templates are used to generate the output source code. Figure 2 resumes the MDD code generation system.

The M2M transformation is composed by 8 ATL modules with 40 matched transformation rules, 33 lazy transformation rules and 44 helper functions. The M2T transformation has 31 templates to generate the ANSI-C code from SIMPLE-C models.

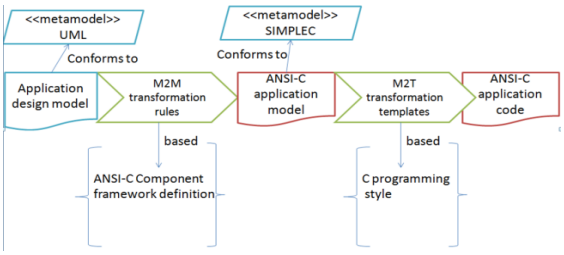


Figure 2: UML to C MDD code generation system.

2.2 The Example Design Model

The Example design model is a simple folder manager application (see figure 3), composed by two components: the folder manager and the client. The folder manager offers services to list and browse entire folders or directories structure, create folders or remove folders. The SimpleC example model that represents the ANSI-C code of the application is automatically generated applying the actual M2M transformation to the design model. The ANSI-C code is obtained applying the M2T transformation.

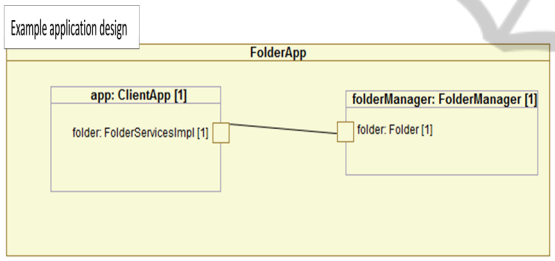


Figure 3: Composite structure diagram of the example application model.

2.3 Specifying the New Non Functional Requirement

At one point, it was required to add access control capabilities to components interfaces. Originally the MDD system did not offer access control capabilities neither at the design model not at the generated code. To specify the access control security properties the methodology presented in (Bouaziz et al., 2011) was selected. This methodology is based on the use of UML component metamodel to capture the domain concepts and security patterns to encode solutions to security problems. In the case study an UML profile associated with RBAC security pattern is used to specify the access control requirement. In figure 4 the UML profile to express RBAC pattern is presented.

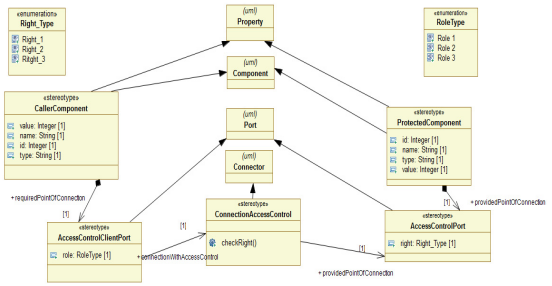


Figure 4: UML profile for access control (RBAC).

2.4 Specifying the New Model Transformation

Once the UML extension for access control is selected the example design model and the automatically generated SimpleC model are incremented manually to demonstrate the new model transformation. Figure 5 represent the new design model for the folder manager application with RBAC properties and also the desired SimpleC output model with the protection proxy pattern implemented in the protected component. The general specification of the new transformation requirement can be resumed as:

- **For each** <<accessControlPort>> in a protected component instance, create a module where the Protection Proxy pattern will be implemented
  1. Create the module where the proxy pattern will be implemented
  2. Create *actualClientRole* field.
  3. Create the *setRole* method.
  4. Create the wrapper functions that call the original interface methods.
  5. Add the reference to the header that defines the *Role* structure.
  6. Add the reference to the header that defines the original interface methods.
- **For each** <<accessControlConnection>> create
  1. The *checkRights* method in a module inside the protected component instance directory.
  2. The *Role* structure in a module inside the protected component instance package
  3. Create *Roles* and *Right* enumerations

Once, we have an example, transformation rules and their integration in the legacy model transformation must be implemented.. This task requires a hard and error-prone navigation through

the legacy model transformation. To ease this step and reduce the transformation rules implementation time the Transevol tool deducts automatically the new rules implementation and integration in the legacy model transformation, in this case the UML to SimpleC transformation.

3 THE TRANSEVOL TOOL

Transevol tool is used to derive adaptation operations that must be implemented in a legacy model transformation to fulfill a new transformation requirement. Starting from pairs of example input/output models the tool induces a number of adaptations in the model transformation. The transformation rules analysis process consists of the following phases (see figure 1):

1. Adapt manually a previous input model to add the new requirement and obtain the differential model between the both models (For example, the addition of the RBAC stereotypes to UML components to express the required access control property).
2. Adapt manually a previous output model to add the requirement and obtain the differential model between both models (For example, adding SimpleC elements that represents the implementation of the protection proxy pattern.).
3. Obtain traceability between the previous design model, the generated output model and the transformation rules.
4. Deduct the adaptations to be made in the transformation rules to fulfill the new transformation requirement using Transevol.
5. Execute a Higher Order Transformation (HOT) to semi-automatically adapt the transformation rules.
6. Manually finish the transformation rules implementation (The helper functions algorithms).
7. Validate the transformation implementation using the manually generated input and output model.

The transformation rules analysis tool, Transevol, relates EMFDiff (Toulmé., 2006) differences types of the output models with adaptation operations to apply on the model transformation. The tool implements an algorithm that derive adaptation operations from the difference model between a model generated by the M2M transformation (GOm, Generated output model) and an expected output model (EOm).). This difference model is called Output models differential (ΔOm = EOm – GOm) and is generated using EMFCompare (Brun and Pierantonio , 2008) and conforms to EMFDiff metamodel. The EMFDiff metamodel types used to analyze the model transformation are: addition of an element (*ModelElementChangeLeft*), removal of an element (*ModelElementChangeRight*), change of an element container (*MoveModelElement*), addition of an attribute (*AttributeChangeLeftTarget*), addition of a reference (*ReferenceChangeLeftTarget*), modification of a reference (*UpdateReference*) and modification of an attribute(*UpdateAttribute*). The EMFDiff differences offer basically the data of the new element, the deleted or updated element, the element affected by the change and the container of the new element.

3.1 Specifying Adaptation Operation for the Transformation Rules

Transevol uses a metamodel called MMRuleAdaptation to express the required adaptation operations for the transformation rules. The transformation rules are subject to the following refinement modifications: *addRule*, *splitRule*, *deleteRule*, *deleteOutputPatternElement*, *deleteBinding*, *addInputPatternElement* *addOutputPatternElement*, *addBinding*, *moveOutputPatternElement*, *moveBinding*, *updateBinding*, *UpdateFilter* and *UpdateSource*.

After the analysis, the tool generates a model expressing the adaptation operations. Any modification operation is defined as an *AdaptationTarget*. Each *AdaptationTarget* has a set of adaptation operations. Each adaptation operation requires different information to specify the Modification, see table 1. The metamodel uses ATL metamodel elements to express the data related with each modification operation. Table 1 collects the data required to express each adaptation operation.

3.2 Relationship between EMFDiff Differences and Adaptation Operations

The tool relates EMFDiff differences types of the output models with adaptation operations. Table 2 resumes the relation between EMFDiff types and adaptation operations.

3.3 Model Transformation Analysis

Using only the output models differences the adaptations operations cannot be expressed correctly, (for example the affected rule in a binding could not be obtained). More information is required to deduct correctly the adaptation operations.

Table 1: MMAdapatationRule metamodel's adaptation operations.

Adaptation operation	Required Data
Add Rule	newRule*: ATL!Rule* relatedBinding: *MMRuleAdaptation!AddBinding*
Add MatchedRule (extends addRule)	newRule*: ATL!Rule* relatedBinding: *MMRuleAdaptation!AddBinding*
Add LazyRule (extends addRule)	newRule*: ATL!Rule* relatedBinding : *MMRuleAdaptation!AddBinding*
Split Rule	affectedRule: *ATL!Rule* newRule: *MMRuleAdaptation!AddRule*
Add Binding (extends BindingOperation)	affectedRule: *ATL!Rule* newBinding : *ATL!Binding*
Remove Binding (extends BindingOperation)	affectedRule: *ATL!Rule* affectedBinding: *ATL!Binding*
Update Binding (extends BindingOperation)	affectedRule: *ATL!Rule* affectedBinding: *ATL!Binding* newValue:OCL!OclExpression
Move Binding (extends BindingOperation)	affectedRule: *ATL!Rule* toRule: *ATL!Rule* binding: *ATL!Binding*
Add filter to input pattern	newFilter: *OCL!OclExpression* affectedRule: *ATL!Rule*
Add input pattern element	affectedRule: *ATL!Rule* *newInput:ATL!InputPatternElement*
Add output pattern element	affectedRule: *ATL!Rule* *outputPattern: ATL!OutputPatternElement*
Delete out pattern element	affectedRule: *ATL!Rule* *outputPattern: ATL!OutputPatternElement*

To derive the modification operations also the input models differential (Δ**Im**) must be used. This way the input elements can be related with the ouput models elements. The input and output metamodels class coverage (Fleurey et al., 2009) is required for the analysis. The metamodel class coverage represents each meta-class is instantiated at least once. The tool uses the differential of the input metamodel class coverage (Δ**Imc**) due to ΔIm. And also uses the output metamodel class coverage differential (Δ**Omc**) due to Δ**Om**. The execution traceability data (ETr) is fundamental for the automatic deduction of the transformation rules modifications. Combining the ΔOm, ΔIm, ETr, ΔImc and ΔOmc the tool obtains the modifications that must be done to adapt the transformation rules for the new transformation requirement.

To specify a model transformation example ΔOm, ΔIm, ETr, ΔImc and ΔOmc models are required. Each difference element between the ouput models is related with an adaptation operation depending on the difference type. The analysis algorithm fits correctly to scenarios that have ΔImc= 0 or ΔImc=1 and ΔOmc>0. When ΔImc is higher than one we recommend to divide the examples in a set of ΔImc=1 examples.

The tool first takes a difference element of the ΔOm and decides which kind of difference is:

1. Addition of output model elements
2. Removal of an output model element
3. Change of an element container
4. Addition and modification of attributes
5. Addition and modification of references

Table 2: Relationship between EMFDiff metamodel types and adaptation operations for model transformations.

EMFDiff difference type	EMFDiff type description	Adaptation operations
ModelElement ChangeLeft	Addition of an element	Add matched rule and add binding
		Add lazy rule and add binding
ModelElement ChangeRight	Removal of an element	Add filter
		Remove rule
MoveModelElement	Change of container	Split rule and modify binding
		Move binding
ReferenceChange LeftTarget	Addition of a reference	Add binding
UpdateReference	Update of a reference value	Update binding
		Add input pattern
AttributeChange LeftTarget	Addition of an attribute value	AddBinding
UpdateAttribute	Modification of an attribute value	Add binding
		Update binding
		Add input pattern

Once the type of the difference is decided the tool must induce the modification that must be applied to the model transformation. Depending on the scenario of the model transformation the adaptation operation for an output EMFDiff difference type may be slightly different. To select the scenario the tool uses the ΔOm, ΔIm, ΔImc and ΔOmc models data.

3.3.1 One-to-One Mapping Scenario

The conditions to detect a one-to-one mapping scenario are: (I) The number of element addition (*ModelElementChangeLeft)* in the ΔIm and the ΔOm must be the same (II) the metamodel class coverage increment for the input and output metamodel must be 1. This scenario requires a new matched rule. The adaptation operation of adding a new matched rule is compound by a new rule and a binding. The data required to define the new matched rule is:

- Input pattern element: the type of any of the added element of the ΔIm model.
- Output pattern element: the type of one of the added element of the ΔOm.
- Rule name: the concatenation of both types.

Execution traceability data is used to search the rule that created the container element. This information is used to establish the binding that relates the new target element with its container.

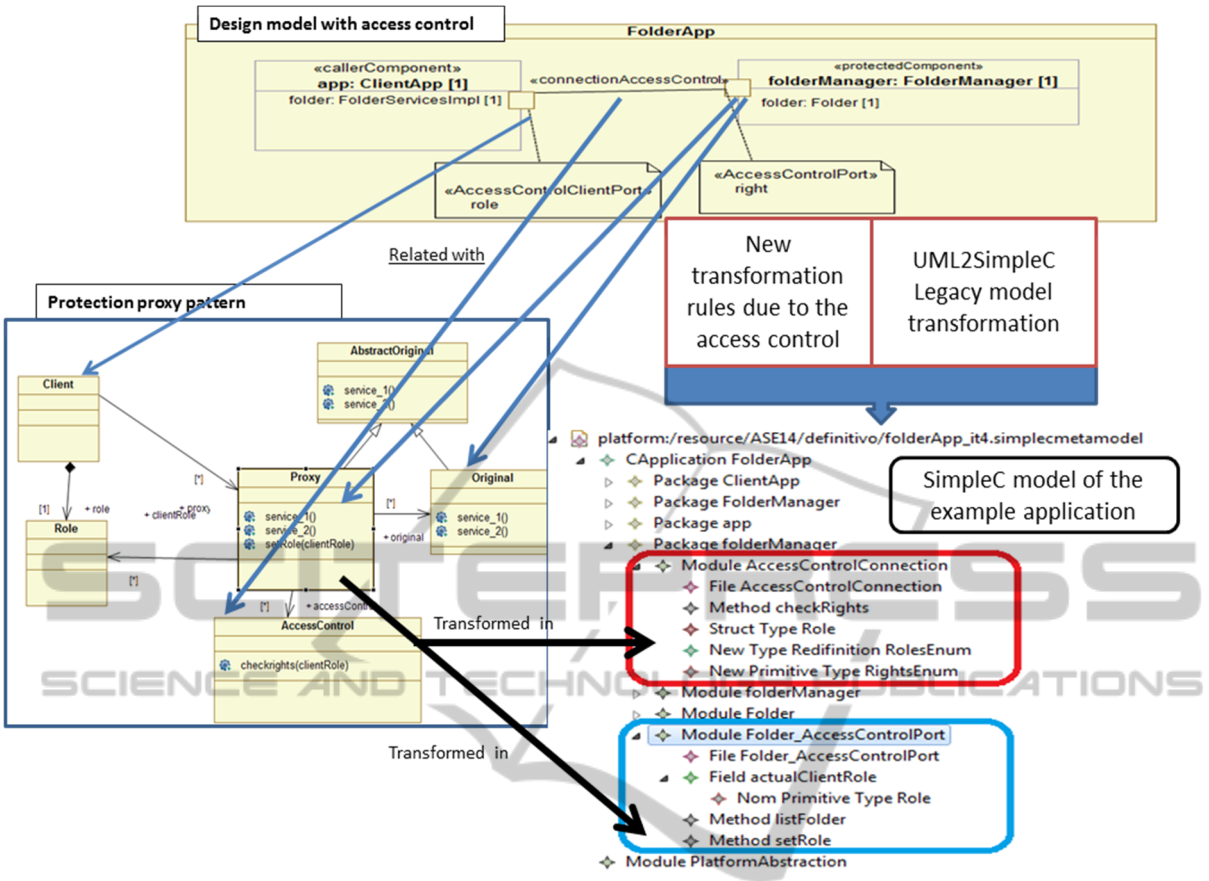


Figure 5: The design model with access control specification, the protection proxy pattern and the desired output model with the access control.

3.3.2 One-to-many Mapping Scenario

This kind of scenario requires the creation of a new output pattern element or a new lazy rule. When different types of target elements are created, new output pattern elements are added to a rule. When instance of the same type are created for an input element type, lazy rules are required.

3.3.3 Many-to-many Mapping Scenario

A many-to-many mapping scenario is defined when a set of elements are added and both ΔImc and ΔOmc, are higher than one. Two strategies can apply to this scenario. The first strategy is to specify the transformation example with a set of one-to-many mapping examples, where ΔImc is equal to 1 in each step. When ΔImc is greater than 1 the algorithm aligns input elements with output elements using the similarity of its properties values. In those cases, false positives adaptation operations can be deducted. For those cases a warning message is used and manual intervention required.

3.3.4 Removal of an Output Model Element

Two removal scenarios are detected by the algorithm. A matched rule is removed when ΔOmc = -1. The other scenario occurs when ΔOmc = 0 and some *ModelElementChangeRight* appears (see table 3). This scenario requires a filtering operation in the input pattern element. In both cases the affected rule is founded searching in the execution trace the rule that generates the removed elements.

3.3.5 Change of an Element Container

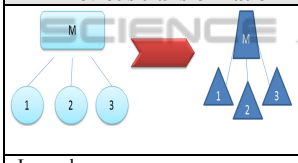
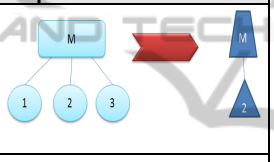
Sometimes without any modification in the input models (ΔImc = 1 and ΔIm=0) the model transformation evolves and requires to change the instance of the container of an output elements or even the container type. Both scenarios are detected by the algorithm. The first scenario involves a split rule operation. To split the affected rule a copy of the rule is done but filtering is added to the input pattern and a binding must be modified. When the type of the container changes a binding must be

deleted in the rule that created the previous container and a binding must be added in the rule that created the desired container. To search those affected rules the execution trace of the previously executed transformation is used.

3.3.6 Addition and Modification of Attributes or References

The operations related to these scenarios are modification of a binding or an addition of a binding. In these cases, the execution trace is used to search the affected rule. The information of the output elements that have the difference (*Updateattribute, UpdateReference, Reference ChangeLeftElement and AttributeChangeLeft Element*) is used to search the affected rule in the traceability data and to define the binding statement.

Table 3: Removing output elements.

Previous transformation	Expected transformation
	
Legend: • Arrow: Transformation • Geometric shapes (left side of the arrow): Elements of the input model • Geometric shapes (right side of the arrow): Elements of the Output model	

3.4 Metaclass Vs. Instances

Using the differences models and the traceability information the analysis of the transformation can be done. The difference model is based on model elements and not on metamodel elements, so several differences may be referred to the same change to be made in the transformation rules. We therefore must filter the adaptation operations to obtain the final adaptation operation model.

3.5 Dealing with Metamodel Extensions and NFR

A metamodel defines the languages and processes to form a model. On metamodeling when the domain language requires the integration of NFR, usually metamodel composition techniques are used. Metamodel composition techniques help reusing and adding metamodel elements in a new metamodel definition. The primary design goal of the metamodeling composition environment is to leave the original metamodels intact, still able to be used independently from any composition they may be a part of. Metamodel extension is an example of metamodel composition technique. When UML is used as design metamodel and new concepts are required on the modeling, metamodel extension can be used. UML profiles as MARTE (OMG, 2009) or UMLSec (Jürjens, 2002), extends UML with stereotypes that are related to NFR. On the case study, UML is extended with a profile to offer access control properties on the design level.

Transevol tool can detect metamodel extensions when UML is used as metamodeling language. When a stereotype is applied to an element of the example input model and the output model is affected, the model transformation adaptation can be implemented in several ways. For example the rule that deals with the stereotyped element can be split adding a filter to the input pattern element. Using the EMFDiff difference data the tool knows which element was stereotyped. The filter is implemented as a helper function, and uses the stereotype data. This way when the input element is stereotyped the new transformation rule will be executed and if not the original rule will be. The new rule is a copy of the original one and also creates the new elements required in the output model to fulfill the new requirement. Another possibility to adjust the model transformation is to add a condition in the binding statement of the transformation rule that deals with the stereotyped element. Transevol splits a rule when a new stereotype appears. Figure 6 resumes the split rule solution used to adapt the model transformation when a stereotype is detected by the tool. The figure shows how the tool adapt the transformation rule that creates the ports of the SW components when access control (property is integrated in the design modeling

To improve the quality of the split rule a super rule is extracted and inheritance is used on the original and the new rule, see figure 7. This way the common part of the split rule is located only in the super rule and it is not duplicated.

4 APPLYING THE TOOL TO THE CASE STUDY

This section describes how must be applied the Transevol tool to evolve a model transformation to integrate new non functional properties.. The aim of the case study is to add access control to the software components ports that requires protection.

The addition of the control access must be offered on the design level and on the code level, so the model transformation must be adapted. To demonstrate the transformation a minimal folder manager application example is used, see figure 3. The transformation is demonstrated in several steps. .Each step must be defined with a pair of input/ouput model increment. In each step after applying the tool, the automatically derived adaptation operations must be implemented (automatically using the HOT or manually) and the new transformation run trace must be obtain to use in next iteration.

In the first iteration the security profile is added to the example input model and the components are stereotyped with <<callerComponet>> and <<protectedComponent>> stereotypes. The example output model doesn't requires any change, so none changes are required on the model transformation. To simplify the example from here only the actions related to the <<protectedComponent>> are taken account. In the second step the connector that requires access control is stereotyped with <<accessControlConnection>>. In this step the desired output model must be specified. The output model is modified to offers the function that check the rights of the client of a port. A header (represented as module in SimpleC) and a file are added to the *folderManager* component directory (package). The *checkRights* method, the *role* structure and the *roles/rigths* enumerations are defined in this module. Figure 6 resumes the modified models defined in this step. The changes that present the desired output model in this step are detected using EMFCompare tool and are six *ModelElementChangeLeftTarget* (element aggregation). In the input models differential an aggregation of a stereotype is detected. Using the input and output differences data the tool detects a one-to-many mappings related with a stereotype applied to a *UML!Connector* element. The tools deduct that a split rule is required and that the new rule requires more output patterns element, one per each new element. To select the rule that must be split the algorithm searches in the execution trace of the previous iteration which rule created the *UML!Connector* that have been stereotyped. In this case the rule called *createConnector* is the affected one. The tool creates an adaptation operation that splits the affected rule, adds input pattern filter to both rules and finally add several output patterns to the new rule. To improve the quality of the transformation also a super rule is extracted as explained in the previous section.

The next task is to aggregate the access control to the ports of a protected component. In this case the input model and the output models, both, must be modified. This task is divided in two steps to obtain a more precise result, so two examples of input/models must be defined. First in the input model the ports that require security are stereotyped with the <<accessControlPort>> stereotype and in the output model the module that implements the proxy pattern is created for each port. In this step only the header file, the references to the interface that must be wrapped and the references to the module that implements the *checkrights* methods are added to the code model. With this information the tool detects again that a split rule operation is required. The new rule is not yet fully implemented; the methods that wrap the provided interface and the field that represent the role of the port user are left. The final pair example of input/output models is created only modifying the output models, so $\Delta Imc = 0$. In this case the tool detects that several elements have been added (the wrapper interface methods, the *setRole* method, the *actualClientRole* field and the file that implements the functions) and new output patterns are aggregate to the rule created in the previous step.

The result of applying the tool to the model transformation using the folder manager example is in figure 7 and the adaptation operations derivate in each step are collected in table 4.

4.1 Validation of the Generated Transformation Rules

Table 4: Adaptation operations derived in each demonstration step.

Step	Description	Differences	Adaptation operations
1	The access control profile is applied and the <<callerComponent>> and <<protectedComponent>> are used	Input model:2 Output model:0	None
2	<<AccessControlConnection>> is applied to the connector in the design model and the access control module is created in the output model	Input model:1 Output model:6 ModelElementChangeLeft	1 Split rule 1 Extracted super rule 6 new output patterns
3	<<AccessControlPort>> stereotype is applied to a port and the proxy module is created in the ouput SimpleC model. The includes that requires the proxy module are also aggregate.	Input model:1 Output model:1 ModelElementChangeLeft and 2 UpdateReferences	1 Split rule 1 Extracted super rule 2 new bindings
4	Methods and fields of the proxy module are created.	Input model: 0 Output model:4	4 new output patterns

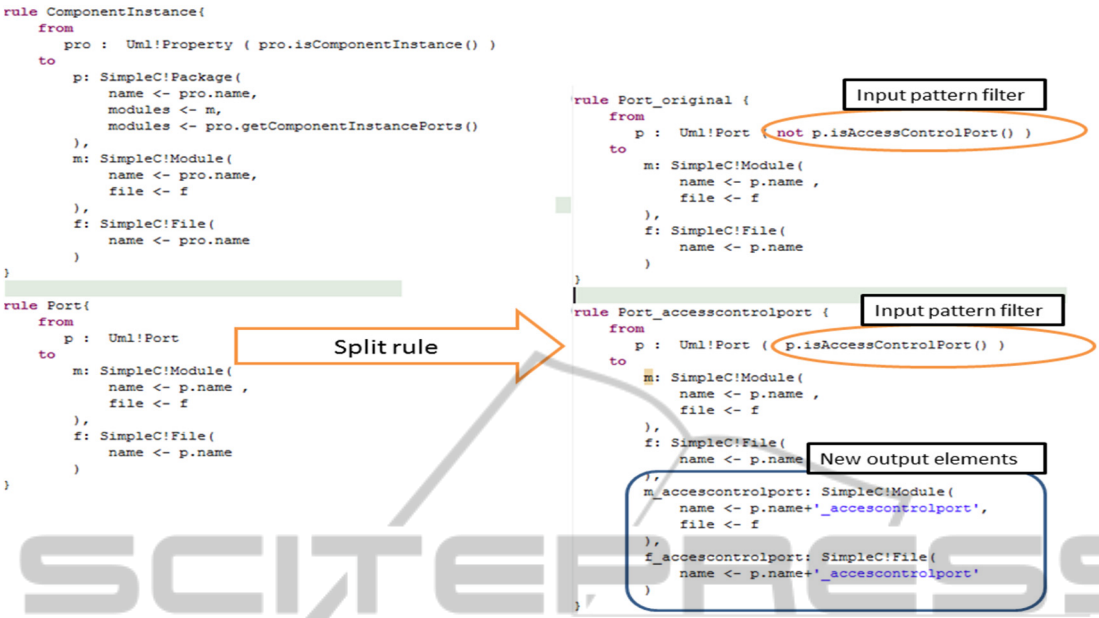


Figure 6: An example of a split rule adaptation operation.

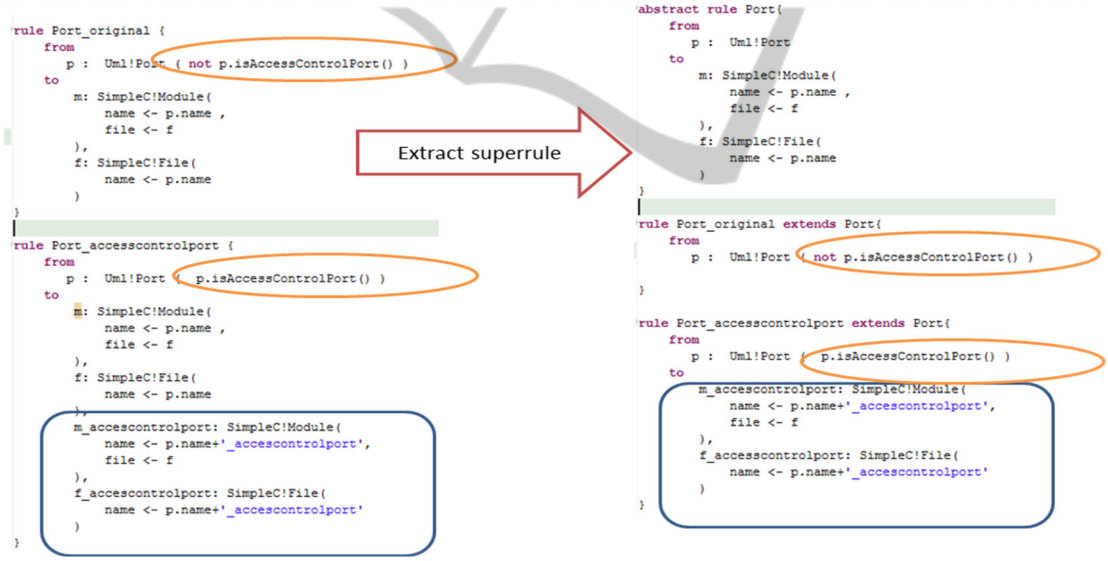


Figure 7: An example of how a super rule is extracted from a split rule adaptation operation.

The model transformation modifications were first validated applying the transformation to the RBAC folder access design. To validate the new transformation rules the transformation was applied to the folder access example model. The generated output model was compared with the expected output model defined manually and used in the adaptation operations deduction phase. Both output models were compared using the EMFCompare tool. The output models were identical. After the first validation a criteria to create more testing input/output pair models was defined. The criteria used to specify the test models was to instantiate all the RBAC extension meta-classes, combine different values for the properties, and combine instances of different meta-classes. Fifteen different input models with their associated output models were defined. In some testing models all the connections between components were stereotyped with RBAC extensions. In some others a set of connections were stereotyped as secure. The number of ports and connectors for component depending on the model

could be one or two. Finally a model without security connections was defined. Each input models had different security settings. The largest model consisted of 5 components. The validation results were satisfactory in all the tests.

5 CONCLUSIONS AND FUTURE WORK

A MTBE approach and a tool to evolve ATL transformation have been presented. A metamodel for expressing adaptation operations for transformation rules and how the adaptation operations for M2M transformations are derived have been described. The tool can be used for adapting exogenous legacy model transformations to new transformation requirements. Even the case study is an exogenous model transformation the tool can be also used with endogenous transformations. The used endogenous transformations were model refinements: the flattening of state machines and the introduction of the bridge design pattern on UML class diagrams.

It has been demonstrate that the tool can be used to integrate NFR on a model transformation. The tool has been used successfully to add RBAC security mechanism in a legacy model transformation. Although initial case study show promising results, as all the transformation rules have been correctly identified, algorithm should be proved in more complex and different examples to improve the coverage of the validation.

In the example case study the NFR affects only the communication between SW components. For some NFRs the architectures and the implementations may be drastically changed. Validation with more NFR case studies is required like e.g. performance / scalability. In these cases bottlenecks have to be removed by e.g. introducing new architectural elements and patterns like caches and/ or clustering. Actually the approach have been only used when the NFR are expressed as UML extension and on PIM to PSM model transformations. The NFR taken account are here applied to express security concepts on the SW architecture. It has not been analyzed how can be address issues related to secure programming standards that affect the code. Those details must be treated mainly in model to text transformations.

Our approach derived correct transformation rules. But the code of the helper functions used to implement the filtering on the matching rules are only defined and called but not implemented. Actually the code of the helper functions must be completed manually. To apply the tool it is enough knowing the changes that are necessary in the M2M transformation input and output models. Previous knowledge of the model transformation implementation is not required, so the time required to adapt the M2M transformation is reduced

In short-tem the tool is going to be used in several model transformations to aggregate different NFR. The aim is to extend the tool to deal not only with NFR expressed as UML profile.

The definition of a methodology for the correct specification of example models is a priority task. The example pair models generated manually in this work were defined intuitively. A methodology to generate automatically the example models is required. Actually exists several works on the area of model transformation validation were the testing and oracle models are generated automatically. Defining a correct set of models to be used as input of model transformations for testing is a difficult problem. In (Guerra et al., 2013) a formal language (PAMOMO) for the specification of the transformations based on invariants pre and post conditions is used to generate the test input model and the corresponding test oracle model. In (Sen et al., 2009) the meta-model and its constraints are used to generate the test input models using the metamodel coverage criteria. The generation of correct example input/ouput pair models has similarities with the automatic generation of testing models. Actually we are analyzing how must be specified the new transformation requirement to generate the example input and ouput models correctly.

6 RELATED WORK

The presented approach is highly related to MTBE. By-example approaches define transformations using examples models. Examples are easier to write than a transformation rule. In MTBE starting from pairs of example input/output models the transformation rules are derived. By example approaches for model transformation are classified in two types (I) demonstration based (II) correspondences based. Model transformation by demonstration (MTBD) (Sun et al., 2009) specifies the desired transformation using modifications performed on example models. MTBE based on correspondences, uses pairs of input/output models

and also a mapping between them to derive the transformation rules.

There are previous MTBE approaches which already deal with automatic generation of model transformations starting from pairs of example models. Most of the approaches are based on formal mapping to derive the transformations (Balogh and Varró, 2009). (Strommer and Wimmer, 2008) approach uses correspondence model between input and output model to generate ATL transformation rules. Instead offering a mapping model (García-Magariño et al., 2009) annotates with extra information the source metamodel and the target metamodel to derive the required ATL transformation rules. Our approach also creates ATL transformation rules but a mapping between the desired input and output model or extra information besides the models differentials is not required.

In (Faunes et al., 2013) a genetic programming based approach to derive model transformation rules (implemented with JESS) from input/output models is presented. This approach doesn't require fine-grained transformation traces. This approach is a self-tuning transformation so it cannot be used with legacy model transformations. TransEvol tool can be used with legacy ATL model to model transformations.

MTBD are based on defining the desired transformation by editing a source model and demonstrating the changes that evolve to a target model. Most of the MTBD are used on endogenous model transformation (Sun and Gray, 2013) not as MTBE, based on correspondences, which can be used with exogenous transformations. (Langer et al., 2010) presents a MTBD approach that can be applied to exogenous model transformation. This approach uses a state-based comparison to determine the executed modification operations after modeling the desired transformation. Using an incremental approach, in each step using a small transformation rule demonstration, internal templates representing the transformation rules are created. Because the approach uses templates created by transformation rules demonstrations it is not easy to apply this approach to legacy model transformations.

Recently a MTBD approach for automating the maintenance of non functional system properties was presented (Sun et al., 2013). The approach can only be applied to endogenous transformation while Transevol can be applied to exogenous model transformations.

Metamodel and transformation co-evolution solution also exists. In (Iovino et al., 2012) weaving between metamodels and transformation rules is used to analyze the impact on the transformation rules due to input metamodel evolution. These works only derives the modification on the transformation rules when regular metamodel evolution, as attribute modification or metaclass rename, occurs. When new elements on the input metamodel appear, the approach cannot derive the transformation rules.

ACKNOWLEDGEMENTS

This work has been developed in the DA2SEC and UE2014-12 AURE projects context funded by the Department of Education, Universities and Research of the Basque Government. The work has been developed by the embedded system group supported by the Department of Education, Universities and Research of the Basque Government.

REFERENCES

Agirre J., Sagardui, G., Etxeberria, L., 2010. Plataforma DSDM para la Generación de Software Basado en Componentes en Entornos Empotrados. *In Jornadas de Ingeniería del Software y Bases de Datos,* JISBD (pp. 7- 15).

Agirre, J., Sagardui, G., Etxeberria, L.m, 2012. A flexible model driven software development process for component based embedded control systems. *III Jornadas de Computación Empotradas JCE*, *SARTECO*.

Balogh, Z., Varró, D., 2009. Model transformation by example using inductive logic programming. *Software and System Modeling, 8*(3): 347-364.

Bouaziz, R., Hamid, B., Desnos, N., 2011. Towards a better integration of patterns in secure component-based systems design. *In Computational Science and Its Applications*-ICCSA 2011 (pp. 607-621). Springer Berlin Heidelberg.

Brun, C., Pierantonio, A., 2008. Model differences in the eclipse modeling framework. UPGRADE, *The European Journal for the Informatics Professional*, 9(2), 29-34.

Bunse, C., Gross, H. G., Peper, C., 2009. Embedded System Construction–Evaluation of Model-Driven and Component-Based Development Approaches. *In Models in Software Engineering* (pp. 66-77). Springer Berlin Heidelberg.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M., 1996. Pattern-oriented software architecture, volume 1: A system of patterns. John Wiley & Sons.

Faunes, M., Sahraoui, H., Boukadoum, M., 2013. Genetic-Programming Approach to Learn Model

Transformation Rules from Examples. *In Theory and Practice of Model Transformations* (pp. 17-32).

Fleurey, F., Baudry, B., Muller, P. A., Le Traon, Y., 2009. Qualifying input test data for model transformations. *Software & Systems Modeling*, 8(2), 185-203.

García-Magariño, I., Gómez-Sanz, J. J., Fuentes-Fernández, R., 2009. Model transformation by example: an algorithm for generating many-to-many transformation rules in several model transformation languages. *In Theory and Practice of Model Transformations* (pp. 52-66). Springer Berlin Heidelberg.

Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schwinger, W. 2013. Automated verification of model transformations based on visual contracts. *Automated Software Engineering, 20*(1), 5-46.

Iovino, L., Pierantonio, A., Malavolta, I., 2012. On the Impact Significance of Metamodel Evolution in MDE. *Journal of Object Technology*,*11* (3): 3: 1-33.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. *Science of computer programming*, 72(1), 31-39.

Jürjens, J., 2002. UMLsec: Extending UML for secure systems development. In *≪ UML≫ 2002*—The Unified Modeling Language (pp. 412-425). Springer Berlin Heidelberg.

Kolovos, D. S., Paige, R. F., Polack, F. A., 2008. The epsilon transformation language. *In Theory and practice of model transformations* (pp. 46-60). Springer Berlin Heidelberg.

Langer, P., Wimmer, M., Kappel, G., 2010. Model-to model transformations by demonstration. *In Theory and Practice of Model Transformations* (pp. 153-167).

Object Management Group (OMG), 2009. Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0, http://www.omg.org/spec/MARTE/1.0/.

Object Management Group (OMG), 2011. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.1.

Sen, S., Baudry, B., & Mottu, J. M. 2009. Automatic model generation strategies for model transformation testing. *In Theory and Practice of Model Transformations* (pp. 148-164). Springer Berlin Heidelberg.

Strommer, M., Wimmer, M., 2008. A framework for model transformation by-example: Concepts and tool support. *In Objects, Components, Models and Patterns* (pp. 372-391). TOOLS.Springer Berlin Heidelberg.

Sun, Y., Gray, J., 2013. End-User support for debugging demonstration-based model transformation execution. *In Modelling Foundations and Applications* (pp. 86-100). Springer Berlin Heidelberg.

Sun, Y., Gray, J., Delamare, R., Baudry, B., White, J., 2013. Automating the maintenance of non-functional system properties using demonstration-based model transformation. *Journal of Software: Evolution and Process, 25*(12): 1335-1356.

Sun, Y., White, J., Gray, J., 2009. Model transformation by demonstration. *In Model Driven Engineering Languages and Systems* (pp. 712-726). Springer Berlin Heidelberg.

Toulmé, A., 2006. Presentation of EMF Compare Utility. *Eclipse Modeling Sympossium*.

Varró, D., 2006. Model transformation by example. *In Model Driven Engineering Languages and Systems* (pp. 410-424). Springer Berlin Heidelberg.

Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., 2013. *Model-driven software development: technology, engineering, management*. John Wiley & Sons.