P²E: A Tool for the Evolution Management of UML Profiles

Fadoi Lakhal[1], Hubert Dubois[1] and Dominique Rieu[2]
[1]*CEA, LIST, Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués, 91191 Gif-sur-Yvette Cedex, France*
[2]*Laboratoire d'informatique de Grenoble, Equipe SIGMA, 220 Rue de la Chimie, BP 53, 38041 Grenoble Cedex 9, France*

Keywords: Abstract Syntax, Modelling Language, UML Profile Evolution, Evolutions Classification, Impact Classification, Models Migration.

Abstract: UML profiles are a frequently used alternative to describe the abstract syntax of modelling languages. As any abstract syntax, UML profiles evolve through time. As the UML profiles are used by models, their evolutions may have a direct impact on them. In order to manage these evolutions, a specific treatment is needed. The models have then to be fitted to the new profiles version. The manual adaptation cost of these models may be as important as building the adapted models from scratch. In this paper, we deal with reducing the cost of models adaptation fitting the conducted evolution of the UML profiles. We provide an automatic treatment using a specific tool. The P²E tool has the ability to detect the changes occurred on the UML profiles, to classify them according to their impacts on the models and finally to adapt the models to the new version of the UML profile.

1 INTRODUCTION

In Model Driven Engineering, the abstract syntax of the modelling languages is usually described by means of metamodels (Kleppe, 2007). Creating new metamodels implies reusing the basic concepts, for example the *Class* concept, the *State* or *Operation* ones. This means that basic concepts have to be created as many as we need for a given metamodel we want to create. In order to avoid this problem, UML proposes the profile mechanism (UML 2.4, 2011). A UML profile consists in describing an abstract syntax of a modelling language by extending the UML language's concepts (defined in the UML metamodel). The UML concepts are then specialized by specific stereotypes to fit the concepts of the specific domain. More than just avoiding defining basic concepts, the profile mechanism gives the designer the ability to use the existing UML tools instead of building new ones.

As any abstract syntax, a profile may evolve regularly for several reasons such as the emergence of new concepts, modifications of existing concepts or reorganization of its structure. A manual management of these evolutions is usually tedious and complex. This complexity varies according to the evolution kind (atomic, composite) and the impact on the models that used the profile. Indeed, an evolution can be seen as an atomic operation (one independent change on the profile) or as a composite evolution (concatenation of atomic operations dependent on each other). If all the evolutions are treated as atomic operations, this dependence relation will be then lost. Information will be lost and the models adaptation in order to ensure their compliance with the new profile version will be then more complicated to manage. One of the major issues is that the existing models conform to the initial profile version become unusable if we do not manage atomic and composite evolutions.

In this paper, we propose to automate the profile evolution and their consequent impact on the models using it. The P²E tool aims at facilitating the profile and models co-evolution (Mens, 2008) by:

- Adapting the models to keep the compliance with their evolved profile. By definition, a model is complying with an abstract syntax described as a UML profile in our case.
- Improving the models for a better description of the modelled system. This means that when the profile evolution consists in an improvement, the models using this profile should then be enhanced as well.

P²E tool is implemented as a Papyrus plugin

(Papyrus, 2012) and is based on three main operations:

- The automatic detection of the atomic and composites evolutions for a better management of their impacts.
- The migration operation used to adapt the models to the new profile version. The migration should be as automated as possible.
- The optimization operation: it consists in improving the models in order to give a better representation of the system (i.e. to improve the model meaning). This operation is semi-automated by the fact that the alerts and recommendations are processed in interaction with the model designer.

This paper is organized as follows: section 2 presents some existing tools, technics and our approach positions. Section 3 gives a classification of the profile evolutions. Section 4 details the adaptation process in P²E tool on an illustrating example. Finally, section 5 concludes the paper and put some future research directions forward.

2 EXISTING TOOLS

Existing tools consider the evolutions when the abstract syntax is described as a metamodel. None of them treat the case of evolution of UML profiles. Nevertheless, we studied how adaptable they could be for this use. We evaluate these tools according to four criteria that interest us:

- The differences between two metamodel versions: are they collected during the evolution or *a posteriori* of the evolution?
- What is the role of the model designer in the approach?
- What kind of changes treats the tool (atomic or composite)?
- Do they propose a classification of the changes?

Hermandosfer et al. proposed a tool called COPE (Hermandosfer et al., 2008). This system records all the atomic changes detected during a metamodel evolution and attaches to each atomic change a migration operation. This migration operation is specific to a change and specified programmatically by the metamodel designer. By the fact that COPE treats the changes directly after the detection, it doesn't have interest to classify these changes. So, it doesn't propose a classification of changes impact. Our approach is dedicated to models designers who do not participate to the profile evolution but only have the different profile versions. So, COPE is not adapted to our goal.

Cicchetti et al. propose in (Cicchetti et al., 2008) a tool which is based on two transformations execution. The first transformation consists in transforming the metamodel as an input to a difference metamodel. By using this difference metamodel, the metamodel designer (who may be the model designer as well) specifies a difference model (containing all the changes between two versions of the same metamodel). From this model, the second transformation generates the corresponding migration transformation. Cicchetti et al. do not define a classification of the obtained changes but reuse the classification of Grushko et al. As for (Hermandosfer et al., 2008), this approach only treats atomic changes while we focus on all kind of changes (composites or atomics). Furthermore, our tool uses a difference model automatically obtained, while in (Cicchetti et al., 2008) they need a manual specification of differences. The metamodel designer should be able to identify the differences between two metamodel versions and then to specify them; but it is not systematically the case.

Levendovszky et al. in (Levendovszky et al., 2010) define a language called *Model Change Language* (MCL). Using this language, the metamodel designer (who is also the model designer) manually defines the rules that map the matching concepts between two metamodel versions. The difference detection here corresponds to the establishment of these mapping rules. The migration tool uses an algorithm specified by the designer as input. It interprets the rules set and executes them. The approach doesn't propose a difference model or a classification step. This approach is not adapted to our goal; we want to have the most automated process to reduce as much as possible the models designer intervention.

Grushko et al. (Grushko et al., 2007) focused on the management of Ecore-based metamodel. Their migration tool uses the *ChangeRecorder* facility in the EMF tool set to detect atomic changes between two versions of the metamodel. Their migration tool then generates a model migration in the Epsilon Transformation Language (ETL). But, the migration model is only generated for renaming changes. For other changes, the metamodel designer has to manually specify the appropriate transformation. We were interested in the classification proposed in (Grushko et al., 2007) that classifies the changes according to their impact on models (which is also our goal). Its decomposition defines three categories: ***Non-breaking*** change ("*does not require any* *adaptation of existing models*”), ***Breaking and resolvable*** changes (“*an algorithm can be defined to migrate existing instances to the new metamodel version*”) and ***Breaking and not resolvable*** change (“*manual interaction is required*”). We extended this classification to the domain of profile evolution. However, there are a few points that we will discuss in the next section.

The Table 1 synthetizes the previous approaches to the four criteria initially described and we complete it with our approach.

Table 1: Synthesis of the approaches and positioning.

CRITERION / APPROACH	*Changes Determination*		*Difference Model Achievement*		*Classification*		*Adaptation dedicated for:*	
	During the evolution	A posteriori	Specified manually	Automatic generation	yes	no	Atomic evolutions	Composite evolutions
Ciccheti et al.		√	√			√	√	
Levendovsky et al.		√	Ø	Ø		√	√	
Hermandosfer et al.	√			√		√	√	
Gruschko et al.		√		√	√		√	
Lakhal et al.		X		X	X		X	X

3 CLASSIFICATION

3.1 Evolvable Elements of an UML Profile

A UML profile consists in extending the UML metamodel by specializing the UML concepts. Each extended concept (represented by a *Stereotype*) allows defining a new concept for a particular domain. In addition to explaining each step of a profile definition, (Selic, 2011) explains that an extension of the UML language implies a semantic proximity between the created stereotype and the extended metaclass. It also states that the new characteristics of a stereotype should not conflict with those inherited from the extended metaclass. Under these conditions, we start by determining the profile elements that may evolve and may impact the models. The Figure 1 describes the extract of the UML metamodel which allows defining a profile. The grey elements represent the elements that may evolve. According to the UML standard, the *Stereotype* element is "*a restricted type of metaclass*". It can define properties, operations or relationships with other elements of the profile. These characteristics have not a fixed number or a default value. It is precisely their evolutions that will imply that the evolution of a stereotype will impact models. Therefore, it is important to consider the evolution of a *Stereotype* but also the evolution of its characteristics.

The *Extension* element links the ends (*ExtensionEnd*) of a metaclass and a stereotype. It is used to assign to a stereotype the adequate UML concept (UML metaclass). The stereotype inherits the metaclass characteristics but also its implementation. So, a stereotype evolution may be inherits these characteristics.

The *Profile* element is a specialization of the *Package* element. A profile evolves because its contained elements evolve. Furthermore, these contained elements use the profile name to create their own qualified names. Thus, a renaming of the profile can have side effects on the use of the profile elements. A profile allows also the import of external elements (*ElementImport*) or external packages (*PackageImport*).

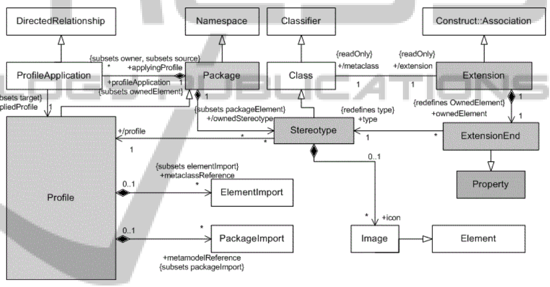

Figure 1: Extract of the UML metamodel, profile package.

3.2 Classification of Profile Evolutions

The classification of profile evolutions seems to be the key step for a better management of their impact on the models. Then, we determine the possible evolutions of a UML profile. The study was conducted on a profile containing 17 stereotypes, 17 generalization links, 37 properties, 2 operations, 5 SysML stereotypes specialized and 5 UML extended metaclasses. We identified 84 possible evolutions (composites or atomics) for these elements that we classify into four possible categories (we don’t treat the elements *ElementImport* and *PackageImport*):

CATEGORIE 1: ***Impact-free category***. This category corresponds to the evolutions which do not impact models compliance. The models do not need an adaptation to be compliant with the new version of the profile (no migration operation is required). Grushko and al. consider these evolutions as “Non-breaking” because no migration is required. In our approach, we consider that this evolution impacts the representation quality of the models. The addition of optional concepts is not insignificant for model designer. They can improve the models clarity or ensure a better satisfaction of system requirements. For this evolution category, we sent to the model designer improvement messages. They identify the parts of models that can be improved.

CATEGORIE 2: ***Automatic evolution category***. To maintain the models compliance with the evolved profile, a migration operation is required. But it can be fully automated. For example, for the addition of a new property, all the characteristics of this property (multiplicity, type, initial value and container) are defined during the profile evolution. The migration operation will be fully automated but a question remains: for each instance of this property, do they have to be equal to the initial value? Improvement messages are then sent to the model designer in order to alert him to potential improvements.

CATEGORIE 3: ***Monitored evolution category.*** These are evolutions that require a migration operation with the support of the designer. This type of evolution is restrictive for the migration operation because some information misses to complete the operation. These blocking constraints can be resolved by an interaction with the model designer. Let's take the example of the adding of a property for which an initial value was not defined. What will be the value of the property instantiated in a model?

CATEGORIE 4: ***Manual evolution category.*** This category includes evolutions which required migration can not be automated. They require a manual migration by the model designer. In this case, alert messages are sent to identify the elements which need a manual operation.

4 PROCESS OF MODELS ADAPTATION IN P²E

Our approach implemented in P²E is divided into four main phases: the determination of the differences between two profiles versions (1), their classification according to the categories described above (2), the adaptation consisting in automatically generating the migration operation (3) and the tracking of improvement messages for optimization of the models description (we assist the model designer in his improvement choices). The Figure 2 illustrates our adaptation process and the tasks of the models designer during its execution. To illustrate our approach, we consider an evolution of the EAST-ADL2 language. The EAST-ADL2 standard is used in automotive domain to design systems at a high abstraction level. EAST-ADL2 is representative of our approach. Indeed, they define the metamodel of their abstract syntax as a UML profile.

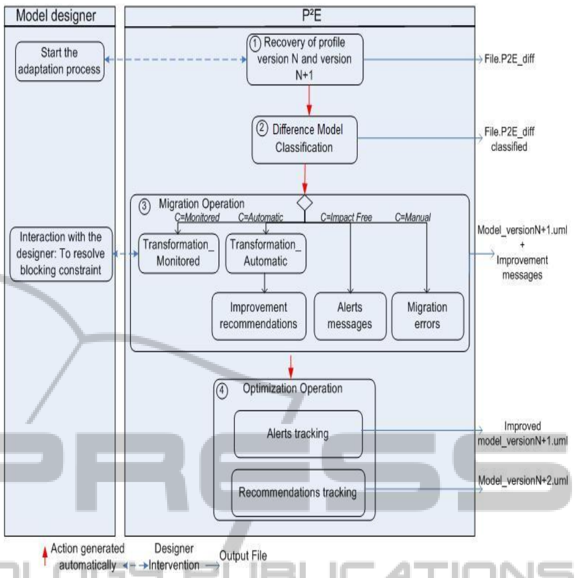
Model designer
P²E
Start the adaptation process
① Recovery of profile version N and version N+1
File.P2E_diff
② Difference Model Classification
File.P2E_diff classified
③ Migration Operation
C=Monitored
C=Automatic
C=Impact Free
C=Manual
Interaction with the designer: To resolve blocking constraint
Transformation_ Monitored
Transformation_ Automatic
Model_versionN+1.uml + Improvement messages
Improvement recommendations
Alerts messages
Migration errors
④ Optimization Operation
Alerts tracking
Improved model_versionN+1.uml
Recommendations tracking
Model_versionN+2.uml
Action generated automatically
Designer Intervention
Output File

Figure 2: Adaptation process in P²E.

4.1 Step 1: Difference Model Generation

Since 2010, the EAST-ADL2 standard evolved towards three different versions. To detect the evolutions between two profile versions, we have chosen to reuse the *EMFCompare* technology (EMFCompare, 2011). *EMFCompare* is an open source tool dedicated to the EMF-based models comparison. The lasts improvements allow comparing two UML profiles or two EMF-based metamodels. *EMFCompare* is based on a *match* engine (that looks for corresponding concepts) and a *diff* engine to determine differences. We estimate that *EMFCompare* is sufficient to detect atomic changes between two profile versions and so we extended it mechanisms to obtain a profile comparison adapted to our approach. Indeed, currently, *EMFCompare* is not able to detect composite changes which are not systematically an atomic changes sum. So, we will add this feature to *EMFCompare*. Furthermore, it doesn't provide a usable change model as output. To resolve this issue, we define a structured difference metamodel and then we extend *EMFCompare* to generate a usable difference model that encompasses our difference metamodel.

Between the version 2.0 and the version 2.1 of EAST-ADL2, we established that: 580 elements were added, 529 elements were deleted and 82 concepts were modified. A total of 1191 changes can be detected. Due to a lack of place, let us consider one of them. In the version 2.0 of EAST-ADL2 (Figure 3 (a)), the system functions are represented by the *ADLFunctionType* concept. It may be composite to describe the functions hierarchy and own *ADLFlowPort* (i.e. *Port* concept) to communicate with others functions. In the version 2.0, the *ADLFlowPort* concept is abstract which means it can't be instantiated. So, it is specialized into three sub-stereotypes corresponding to the possible direction of the port: *ADLInFlowPort*, *ADLOutFlowPort* and *ADLInOutFlowPort*. In the version 2.1 of the standard (Figure 3 (b)), the *ADLFlowPort* concept evolves towards a more compacted concept. Indeed, the tree sub-stereotypes are removed and replaced by an enumeration property (*"direction:EADirectionKind"*) in the *ADLFlowPort* stereotype. The *EADirectionKind* enumeration contains the literals corresponding to the possible direction of a port (`in`, *inout* and *out*).

The Table 2 presents the difference model of the evolution in a schematic way (simplified).

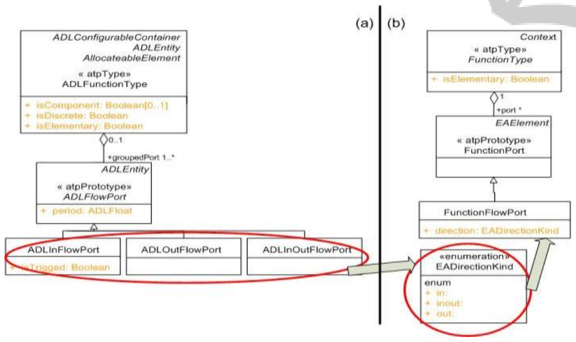

Figure 3: Port concept in EAST-ADL2 (version 2.0 (a) and version 2.1 (b)).

Table 2: Difference model simplified.

Evolution Operation	Element
Remove	ADLInFlowPort ADLOutFlowPort ADLInOutFlowPort
Modify	ADLFlowPort *abstract → actif*
Addition	Property « direction » *Type : Enumeration EADirectionKind* *Owner : ADLFlowPort* Enumeration « EADirectionKind » *Literals: in, out, inout*

4.2 Step 2: Classification of Detected Changes

This step consists in reorganizing the difference model according to the four categories we defined in the section 3. The difference model (table 2) shows that three evolution operations were made on our example. The *remove* operation can be fully automated, the *modify* operation can also be automated. But, the *addition* operation can't be automated if all of their characteristics are not defined: the initial value of the *direction* property is missing. By this example, we notice that the decomposition of one global evolution into successive atomic operations can reduce the automation of a migration operation. In our approach, we propose to make researches on the difference model, in order to detect patterns of composite evolutions. We have thus defined a catalogue of the most common profile evolutions in the form of evolution patterns. For each pattern, we associate a category. For this example, the associated pattern is called "*Removal of sub stereotypes that become enumerated type*". Indeed, the three merged operations become one composite evolution to handle. The type of each port can be associated (mapping) to a literal and can be implicitly used as an initial value. So, it allows adding automatically the *direction* property. The management of composite patterns allows automating a migration that wouldn't be automated if decomposed into atomic operations. More generally, assigning a category to each detected pattern will be made by filtering.

4.3 Steps 3 and 4: Operations of Migration and Optimization

For *Impact-free* evolutions, *recommendation messages* are generated for a treatment during the step 4.

For *Automatic* and *Monitored* evolutions, we automatically generate the transformation rule specific to the detected evolution pattern. Indeed, we studied each evolution pattern to give the corresponding migration rule (to treat non atomic patterns). It will consist then in instantiating the adequate migration rule for each evolution pattern contained in the catalogue take into account the blocking constraints that require a designer intervention. After the generation of the migration rules, *alert and recommendations messages* are then created.

One main objective of our approach being to avoid the intervention of the models designer during the adaptation process, we want to reduce *Manual* evolutions to have as much as possible *a posteriori* evolutions that belong to the first three categories. If no solution is determined, the model elements that can't be migrated will be identified (generation of *warning messages*).

To illustrate the migration operation required in the evolution described above, we specified a model using the *ADLInFlowPort* concept (Figure 4). The *Engine* element represents the engine function of the system. It owns a port *accelerator* stereotyped by *ADLInFlowPort* representing the *accelerator* sensor. The migration operation will consist in instantiating the corresponding migration rule: 1/ the mapping of the type of the sub stereotypes with the literals (example: *ADLInFlowPort* = in). 2/ the replacement of the stereotype *ADLInFlowPort* by the stereotype *FunctionFlowPort*. 3/ the creation of the new property *direction*. 4/ the information of the property by the specific value (*in*). The Figure 5 illustrates the result of the adaptation process executed on the model in Figure 4.

For this example, we don't consider the renaming of the concepts and we don't detail the optimization step.

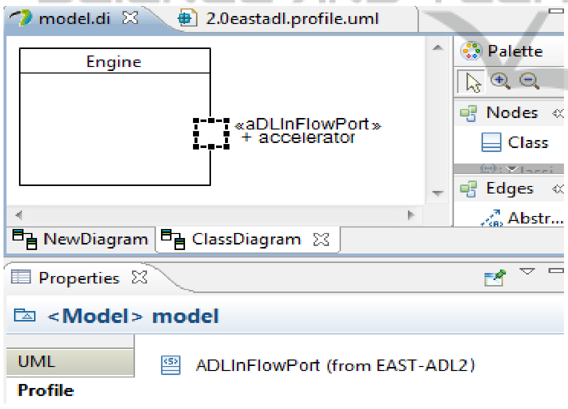
model.di
2.0eastadl.profile.uml
Engine
«aDLInFlowPort»
+ accelerator
Palette
Nodes
Class
Edges
Abstr...
NewDiagram
ClassDiagram
Properties
<Model> model
UML
Profile
ADLInFlowPort (from EAST-ADL2)

Figure 4: Model before the adaptation process.

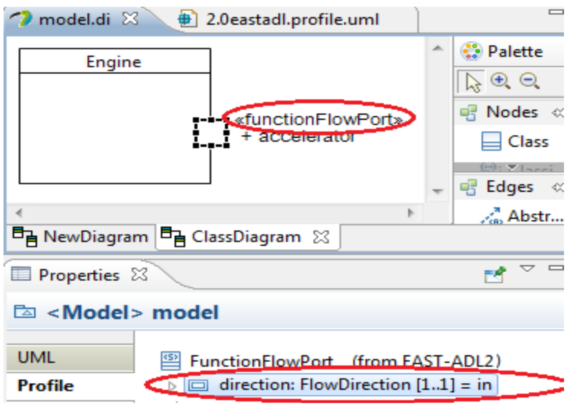
model.di
2.0eastadl.profile.uml
Engine
«functionFlowPort»
+ accelerator
Palette
Nodes
Class
Edges
Abstr...
NewDiagram
ClassDiagram
Properties
<Model> model
UML
Profile
FunctionFlowPort (from EAST-ADL2)
direction: FlowDirection [1..1] = in

Figure 5: Model after the adaptation process.

5 CONCLUSIONS AND FUTURE WORKS

In this paper, we presented P²E tool (a Papyrus plugin). The adaptation process implemented in P²E was based on an automatic detection of evolution patterns. To each detected pattern, a category is assigned (according to the classification that we propose) allowing then to adapt the models (by using a migration operation) specifically to the impact of the detected evolution. Then, P²E assists (optimization step) the model designer in his choices to improve models (by the tracking of improvement messages).

P²E should be completed by the implementation of the filtering method used for the detection of all evolution patterns (composites). P²E should also be able to define the correct order between the atomic operations which compose a composite pattern. Indeed, this order may differentially affect the level of automation of the migration and may increase the time of the migration operation. P²E should take into account the relations of *import* or *merge*. These relations imply that there exist two main impacts to manage. The first one when an imported profile is evolving, what are the evolution impacts on profile that are imported? And the other one on the models using the profile that make the import? (respectively on a merging profile). For this, we will measure the evolution impacts of these imports into a profile to then offer a migration strategy for the adaptation of the models as automatically as possible.

REFERENCES

Cicchetti, A., Ruscio, D. D., Eramo, R., Pierantonio, A., 2008. Automating Co-evolution in Model-Driven Engineering. *In: IEEE Enterprise Distributed Object Computing Conference*, pp. 222–231.Washington.

EAST-ADL, 2010. Available from World Web: *http:// www.atesst.org/home/liblocal/docs/*.

EMF Compare, 2011: Available from World Web: *http:// wiki.eclipse.org/ index.php/EMF_Compare*.

Gruschko, B., Kolovos, D., Paige, R., 2007. Towards Synchronizing Models with Evolving Metamodels. *In: Procs of the Work. MODSE.*

Herrmannsdoerfer, M., Benz, S., Juergens E., 2008. Automatability of Coupled Evolution of Metamodels and Models in Practice. *In: 11th international conference on Model Driven Engineering Languages and Systems*, pp. 645–659, Berlin.

Kleppe, A. G., 2007. A Language Description is More than a Metamodel. *In: Workshop on Software Language Engineering.* Nashville, USA.

Levendovszky, T., Rumpe, B., Schätz, B., Sprinkle, J.,

2010. Model evolution and management. *In: Model-Based Engineering of Embedded Real-Time Systems*, pp. 241–270. Heidelberg.

Mens, T. Demeyer, S., 2008. *Software Evolution*. Springer.

Papyrus, 2012. Available from World Web: *www.papyrusuml.org.*

Unified Modeling Language (UML), 2011. Available from World Web: *http://www.omg.org/spec/UML/2.3/*.

Selic, B., 2007. A Systematic Approach to Domain-Specific Language Design Using UML. *In: Object-Oriented Real-Time Distributed Computing*, pp. 2-9.