ACM/IEEE 17th International Conference on
Model Driven Engineering Languages and Systems

September 28, 2014 • Valencia (Spain)

ME 2014 – Models and Evolution Workshop Proceedings

Alfonso Pierantonio, Bernhard Schätz and Dalila Tamzalit (Eds.)

Published on Sept 2014 v1.0

© 2014 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners.

Editors' addresses:

Alfonso Pierantonio
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica · Università degli Studi dell'Aquila (Italy)

Bernhard Schätz
fortiss GmbH (Germany)

Dalila Tamzalit
IUT - LINA · Université de Nantes (France)

Organizers

Alfonso Pierantonio (co-chair)	Università degli Studi dell'Aquila (Italy)
Bernhard Schätz (co-chair)	fortiss GmbH (Germany)
Dalila Tamzalit (co-chair)	Université de Nantes (France)

Program Committee

Anne Etien	University of Lille (France)
Antonio Cicchetti	Mälardalen University (Sweden)
Antonio Vallecillo	Universidad de Málaga (Spain)
Bernhard Rumpe	RWTH Aachen University (Germany)
Davide Di Ruscio	Università degli Studi dell'Aquila (Italy)
Eleni Stroulia	University of Alberta (Canada)
Gerti Kappel	Vienna University of Technology (Austria)
Jean-Marie Mottu	Université de Nantes (France)
Jeff Gray	University of Alabama (USA)
Jesus Garcia-Molina	Universidad de Murcia (Spain)
Jonathan Sprinkle	University of Arizona (USA)
Ludovico Iovino	University of L'Aquila (Italy)
Manuel Wimmer	Vienna University of Technology (Austria)
Martina Seidl	Johannes Kepler University Linz (Austria)
Mireille Blay-Fornarino	Université de Nice-Sophia Antipolis (Germany)
Olivier Le Goaer	LIUPPA,Université de Pau et des Pays de l'Adour (France)
Richard Paige	University of York (United Kingdom)
Stefan Wagner	University of Stuttgart (Germany)
Tihamer Levendovszky	Vanderbilt University (USA)
Udo Kelter	University of Siegen (Germany)
Vasilios Andrikopoulos	University of Stuttgart (Germany)
Zinovy Diskinz	McMaster University / University of Waterloo (Canada)

Table of Contents

Keynote

From Model Evolution to Evolution Models

Gerti Kappel

Vienna University of Technology

Inspired by the seminal paper on "Model Transformations? Transformation Models!" [1] we will have a fresh look on the evolution of model evolution. Model evolution is more and more seen as part of change management in general with change as first class principle in the whole software product lifecycle. Thus, a descriptive notion of evolution in terms of evolution models is necessary. In this talk, we will shed some light on current research endeavors pathing the way to a more systematic management of evolution.

Gerti Kappel *is a full professor at the Institute for Software Technology and Interactive Systems at the Vienna University of Technology, heading the Business Informatics Group. Until 2001, she was a full professor of computer science and head of the Department of Information Systems at the Johannes Kepler University of Linz. She received the Ms and PhD degrees in computer science and business informatics from the University of Vienna and Vienna University of Technology in 1984 and 1987, respectively. From 1987 to1989 she was a visiting researcher at Centre Universitaire d'Informatique, Geneva, Switzerland.*
She has been involved in national and international joint projects, both governmental and industry funded, as well as sponsored by the EU. From 2004 to 2007, she was also dean of student affairs for business informatics.

[1] Jean Bzivin et al., Model transformations? Transformation Models!, In: Proceedings of the 9th international Conference on Model Driven Engineering Languages and Systems (MoDELS'06), O. Nierstrasz et al. (eds.), Springer LNCS 4199, pp. 440-453

A Systematic Taxonomy of Metamodel Evolution Impacts on OCL Expressions*

Angelika Kusel[1], Juergen Etzlstorfer[2], Elisabeth Kapsammer[1], Werner Retschitzegger[1], Johannes Schoenboeck[3], Wieland Schwinger[1], and Manuel Wimmer[2]

[1] Johannes Kepler University Linz, Austria
[firstname.lastname]@jku.at
[2] Vienna University of Technology, Austria
[lastname]@big.tuwien.ac.at
[3] University of Applied Sciences Upper Austria, Campus Hagenberg, Austria
[firstname.lastname]@fh-hagenberg.at

Abstract. Metamodel evolution is prevalent in Model-Driven Engineering, necessitating the co-evolution of dependent artifacts like models and transformations. Whereas model co-evolution has been extensively studied, the co-evolution of transformations and especially its substantial ingredient in terms of OCL expressions has received little attention up to now. Thus, the goal of this paper is a systematic analysis of potential impacts of metamodel evolution on OCL expressions in model transformations. For this, a *complete and minimal* set of atomic metamodel changes has been derived from Ecore, which is analyzed with respect to its effects on structural complexity and information capacity. This analysis builds the basis for investigating the *impacts* concerning *syntactical conformance* and *scope* of affected OCL expressions. Finally, we report on lessons learned gained from establishing the set of changes and examining the impacts thereof.

1 Introduction

Model-Driven Engineering (MDE) proposes the use of models to conduct software development on a higher level of abstraction [1]. Thereby, model transformations play a vital role for systematic transformations of models conforming to different metamodels (MMs). Just like any other software artifact, MMs evolve, necessitating the co-evolution of dependent artifacts like models and transformations [10].

While the automated co-evolution of models has been subject to extensive research in the past (cf., [9] for a survey), the automated co-evolution of transformations has been less examined so far (cf., e.g., [4–6, 13]). Especially the co-evolution of Object Constraint Language (OCL) [18] expressions has not been a major focus up to now, despite the fact that OCL expressions are used to perform complex queries on the input models [2, 22]. Therefore, they represent a substantial ingredient in rule-based model transformation languages, such as ATL [11] or QVT [17].

* This work has been funded by the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) grant FFG BRIDGE 832160 and FFG FIT-IT 825070 and 829598, FFG Basisprogramm 838181, and by ÖAD grant AR18/2013 and UA07/2013.

To tackle this limitation, this paper focuses on the co-evolution of OCL expressions in model transformations by first proposing a *complete and minimal set of atomic changes* focusing on structure, which has been systematically derived from the Ecore[4] meta-MM, enabling the definition of arbitrary evolutions of Ecore-based MMs. All changes of this set are subsequently analyzed concerning their effects on the MM with respect to *structural complexity*, i.e., the number of instantiable MM elements, and *information capacity*, i.e., the potential number of instances of the MM, since these two criteria are significant for the impacts on OCL expressions which will be revealed in the remainder of this paper. Second, the *potential impacts* of these changes on OCL expressions are investigated by systematically analyzing the impacts of each of these changes concerning affected OCL expressions, revealing *non-breaking* and *breaking* impacts [7] with respect to *syntax* and scattering of impacts considering their *scope*, being *local*, in case that OCL expressions use the changed MM element itself, or *global*, if they use inherited versions thereof. Thus, this investigation builds the foundation for identifying resolution actions to co-evolve syntactically broken OCL expressions and serves as basis for implementing an impact analysis tool, constituting the first and fundamental step towards the automated co-evolution of OCL expressions in model transformations.

Outline: Section 2 systematically analyzes the impacts of MM evolution on OCL expressions. While lessons learned are presented in Section 3, related work is surveyed in Section 4. Finally, Section 5 concludes the paper with an outlook to future work.

2 Systematic Impact Analysis

In this section, role and importance of OCL in model transformations are highlighted, before the complete and minimal set of changes as well as the investigation of impacts of each change on OCL expressions are presented. Although OCL might also be used in other contexts, e.g., to specify MM constraints restricting the instantiability of the MM, we focus on the co-evolution of OCL in model transformations. Nevertheless, this work might also be applied to other application contexts. A detailed investigation of impacts on OCL constraints in MMs is, however, left to future work.

2.1 Role and Importance of OCL in Model Transformations

In order to illustrate the role and importance of OCL, Figure 1 shows an excerpt of the well-known Class2Relational transformation[5], serving as a running example throughout the paper. From the example one might see that OCL expressions are used in two indispensable roles [13]. First, OCL is used in *bindings* to query elements of the source model, which are used to produce the target model (cf., e.g., "cl.package+' '+cl.id" calculating the values for the target attribute Table.name). Second, OCL is utilized in *conditions* to steer the control flow (cf., e.g., "cl.abstract=false" to transform non-abstract classes, only). Through these two essential roles, OCL expressions constitute large parts of transformation definitions [2, 22], and thus, it is of utmost importance to consider OCL in detail in the context of transformation co-evolution.

[4] http://eclipse.org/modeling/

[5] For a complete example see: http://www.eclipse.org/atl/atlTransformations/

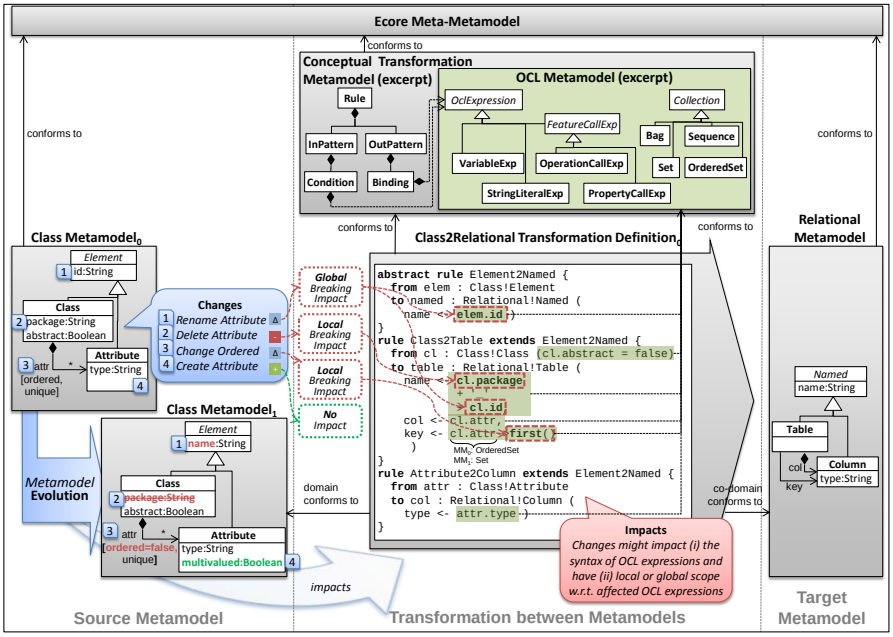


Fig. 1. Running Example: Class2Relational

In general, model transformations depend on three distinct MMs, being (i) the source MM, (ii) the target MM, and (iii) the transformation MM (cf. Fig. 1). Thereby, OCL expressions depend on the source MM by means of a so-called "domain conforms to"-relationship [15] and the OCL MM as part of the transformation MM by means of a "conforms to"-relationship [11]. This is, since OCL expressions are used to query source models and do not refer to concepts of the target MM. Therefore, this paper focuses on the evolution of the source MM and its impact on OCL expressions. For this, a systematic set of changes is needed, which will be the focus of the next two subsections.

2.2 Complete and Minimal Set of Changes

A *systematic* set of changes, as a prerequisite for investigating impacts, has to fulfill two criteria – *completeness* to allow for any possible change and *minimality* to avoid the analysis of redundant changes. To fulfill both, we focus on *atomic changes*, transferring the MM from one consistent state, i.e., conforming to Ecore [20], to another one.

Example. Before proposing the systematic set of atomic changes, four exemplary atomic changes (cf. Fig. 1) with their effects on structural complexity and information capacity of the MM are discussed. First, the attribute Element.id has been renamed to name (cf. ① in Fig. 1), being updative, i.e., a state change, by nature. Although this causes neither a change in structure nor a change in information capacity, OCL expressions are affected. Second, the attribute Class.package has been deleted (cf. ② in Fig. 1), being destructive by nature, thus, decreasing structural complexity and information capacity, which impacts OCL expressions significantly, since the deleted element

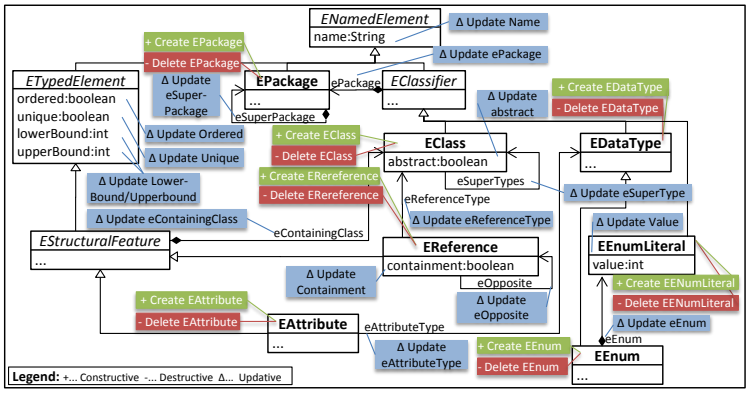


Fig. 2. Derived Set of Atomic Changes for Ecore-based MMs

must not be accessed anymore. Third, the reference Class.attr has been changed from ordered to unordered (cf. ③ in Fig. 1), being again updative by nature, leaving structural complexity and information unaffected, but, however, affecting OCL expressions. Finally, the attribute Attribute.multivalued of type Boolean has been created (cf. ④ in Fig. 1), being constructive by nature, therefore increasing both, structural complexity and information capacity, since this attribute might now be instantiated with true or false, without, however, affecting OCL expressions.

Systematic Set of Changes. Going beyond these four exemplary changes, Figure 2 shows the relevant excerpt of Ecore, including all elements for defining *structure*, while disregarding (i) properties for *code generation* (e.g., volatile), (ii) *derived properties* (e.g., required), since they may be led back from other properties (e.g., lowerBound), and (iii) *operations* (e.g., EOperation), since the focus is on MMs defining structure and not behavior. For deriving all *constructive* and *destructive* changes, one has to resort to all concrete meta-classes, e.g., EClass. For receiving all *updative* changes, i.e., state changes of features, one has to refer to all meta-features, e.g., EClass.abstract. The resulting set of atomic changes is shown in Figure 2 as well as in Table 2.

Criteria. Before analyzing the impacts of the changes on OCL expressions, their effects with respect to (i) *structural complexity* and (ii) *information capacity* are analyzed, being *increasing*, *neutral*, or *decreasing*. Changes affecting *structural complexity* indicate impacts in accessing MM elements in OCL expressions and might be evaluated by counting the number of instantiable MM elements [19]. In contrast, changes concerning *information capacity* indicate impacts on the results of OCL expressions and might be evaluated by counting the potential number of all valid instances of a MM [16].

Evaluation. In the following, the set of changes is evaluated (cf. Table 2).

Constructive/Destructive Changes: All constructive changes have an increasing effect on both, structural complexity and information capacity, since they increase the number of instantiable MM elements and by this also the potential number of valid instances. In contrast, all destructive changes have the exact opposite effect.

Updative Changes: Whereas all constructive as well as destructive changes behave equally with respect to our criteria, updative changes do not and might be further subdivided into four groups according to their behavior.

Group ① Renaming Updates: The first group includes updates on ENamedElement.name and EEnumLiteral.value, i.e., renames, being neutral with respect to both, structural complexity and information capacity.

Group ② Moving Updates: This group regards updates on containment references, i.e., EPackage.eSuperPackage, EClassifier.ePackage, EStructuralFeature.eContainingClass, as well as EEnumLiteral.eEnum, which enable the movement of a feature from one container to another one. Such updates increase structural capacity in the target container, but decrease structural complexity in the source container. Since the features are still available in the MM, yet at another position, the effect on the information capacity is neutral, i.e., not affecting the number of valid instances.

Group ③ Restricting/Relaxing Updates: The third group considers updates on restricting or relaxing the instantiability of MM elements, comprising the features abstract of EClass, upperBound, lowerBound, and unique of ETypedElement as well as all features of EAttribute and EReference. Their effect on structural complexity is neutral, but their effect on information capacity is either increasing or decreasing, depending on the concrete state change. For instance, in case of an increase of feature lowerBound, the number of valid instances decreases, since more values are required. In contrast, a decrease of lowerBound has the opposite effect. Furthermore, type specialization has decreasing effect on information capacity, since the set of valid instances decreases. In contrast, type generalization has increasing effect on information capacity. Please note that feature ETypedElement.ordered has neutral effect on both, structural complexity and information capacity, but impacts the underlying OCL datatype (cf. Sect. 2.3).

Group ④ Constructive/Destructive Updates: Finally, this group considers updates on types, i.e., EClasses themselves, or the datatypes of EStructuralFeatures. This group may be further subdivided into two categories according to their effects. First, the addition of eSuperTypes and pulling up of EStructuralFeatures have increasing effect on information capacity, while their effect on structural complexity is increasing for the addition of eSuperTypes and both, increasing and decreasing for EStructuralFeatures, analogously to moving updates. Second, the deletion of eSuperTypes and pushing down of EStructuralFeatures has decreasing effect on information capacity, while their effect on structural complexity is decreasing for the removal of eSuperTypes and again both, increasing and decreasing for EStructuralFeatures.

2.3 Impact Analysis

In the following, impacts of MM evolution on OCL expressions are exemplified and on basis of this, dedicated criteria are derived, which are finally evaluated with respect to the complete and minimal set of changes.

Example. To reveal impacts of MM evolution on OCL expressions, the running example is utilized again: first, the renaming of the attribute Element.id (cf. ① in Fig. 1) has no effect with respect to structural complexity and information capacity, but breaking impact on the syntax of all OCL expressions accessing the element either directly or indirectly via inherited versions thereof, i.e., the impact scatters. Second, the deletion of the attribute Class.package (cf. ② in Fig. 1) naturally has breaking impact on all OCL expressions accessing this element, since the structure has been changed in a destructive way, and since belonging to a leaf class, the impact does not scatter. Third, the

Ecore Meta-Feature		OCL Type				
		Scalar Type	Collection			
			Bag	Sequence	Set	OrderedSet
lowerBound		no impact on OCL type				
upperBound = 1	unique/ordered not applicable	✓				
upperBound > 1	unique = true and ordered = true					✓
	unique = true and ordered = false				✓	
	unique = false and ordered = true			✓		
	unique = false and ordered = false		✓			

Table 1. Resulting OCL Types out of Ecore Settings

change of the reference Class.attr from ordered to unordered, i.e., ordered=false (cf. ③ in Fig. 1), has breaking impact, although the structural complexity is unaffected, i.e., the feature is still accessible. However, it causes a change of the internally employed OCL collection type from OrderedSet to Set and by this, invalidates the usage of now undefined operations such as first(). In this context, Table 1 shows the possible Ecore settings related to collections and the resulting OCL collection type. Finally, the creation of the attribute Attribute.multivalued (cf. ④ in Fig. 1) naturally has no impact.

Criteria. As one might see from the exemplary discussion above, changes may have potential impact on the *syntax* of OCL expressions being either *non-breaking* or *breaking*. Moreover, a change exhibits a certain *scope*, i.e., the scattering of the impact, being *local*, i.e., OCL expressions using the MM element itself, or *global*, i.e., on OCL expressions using inherited versions thereof.

Evaluation. In the following, all changes are systematically evaluated with respect to these criteria. Please note that the evaluation assumes that changed MM elements have been used by at least one OCL expression and the worst case scenario is considered, i.e., changes are evaluated as breaking, if there exists at least one case that breaks the OCL expression. Since the vast majority of changes have local impact regarding the scope as long as they concern elements in leaf classes, this criterion is discussed for exceptional cases, only. The detailed results of the evaluation may be found in Table 2.

Constructive/Destructive Changes: Constructive changes do not have any impact on OCL expressions, since newly created elements can not have been referred to. In contrast, destructive changes always have breaking impact on OCL expressions, since having a destructive effect on the structure.

Updative Changes: Updative changes are evaluated on basis of the groups introduced in Section 2.2, in the following.

Group ① Renaming Updates: Although renames do neither affect structural complexity nor information capacity, their impact is nevertheless breaking, since renamed elements are no longer accessible under their original name.

Group ② Moving Updates: Since moves change the structure of instances by changing the position of elements, the impact on OCL is always breaking.

Group ③ Restricting/Relaxing Updates: Although these updates leave the structural complexity unaffected, they impact information capacity, which may also break the syntax of OCL expressions. This is since different settings of features such as ETypedElement.ordered result in different OCL datatypes (cf. Table 1). For example, changing the feature ordered from true to false implies a change from the OCL collection type OrderedSet to Set, thereby invalidating, e.g., the usage of the operation first().

Group ④ Constructive/Destructive Updates: This group is divided into two categories concerning their effect with respect to information capacity as already mentioned

	Change on Ecore-based Metamodel											Impact on OCL			
	Ecore Meta-Class	Name of Atomic Change	Meta-Feature	State Change of Meta-Feature	Group	Effect on Structural Complexity: Increase	Structural Complexity: Neutral	Structural Complexity: Decrease	Information Capacity: Increase	Information Capacity: Neutral	Information Capacity: Decrease	Syntax: Non-breaking	Syntax: Breaking	Scope: Local	Scope: Global[1]
Constructive	EPackage	Create EPackage	*n.a.*	n.a.	n. a.	+			+			no impact possible			
	EClass	Create EClass	*n.a.*	n.a.		+			+						
	EAttribute	Create EAttribute	*n.a.*	n.a.		+			+						
	EReference	Create EReference	*n.a.*	n.a.		+			+						
	EDataType	Create EDataType	*n.a.*	n.a.		+			+						
	EEnum	Create EEnum	*n.a.*	n.a.		+			+						
	EEnumLiteral	Create EEnumLiteral	*n.a.*	n.a.		+			+						
Destructive	EPackage	Delete EPackage	*n.a.*	n.a.	n. a.			-			-		x	x	
	EClass	Delete EClass	*n.a.*	n.a.				-			-		x	x	
	EAttribute	Delete EAttribute	*n.a.*	n.a.				-			-		x	x	
	EReference	Delete EReference	*n.a.*	n.a.				-			-		x	x	
	EDataType	Delete EDataType	*n.a.*	n.a.				-			-		x	x	
	EEnum	Delete EEnum	*n.a.*	n.a.				-			-		x	x	
	EEnumLiteral	Delete EEnumLiteral	*n.a.*	n.a.				-			-		x	x	
Updative	*ENamedElement* (inherited by: EClass, EPackage, EAttribute, EReference, EDataType, EEnum EEnumLiteral)	Update Name	*name*	oldName → newName and oldName <> newName	①		o			o			x	x	
	EPackage	Update ESuperPackage	*eSuperPackage*	oldeSuperPackage → neweSuperPackage	②	+		-		o			x	x	
	EClassifier (inherited by: EClass, EDataType, EEnum)	Update EPackage	*ePackage*	oldePackage → newePackage	②	+		-		o			x	x	
	EClass	Update Abstract	*abstract*	true → false	③		o		+			x		x	
				false → true			o				-	x		x	
		Update ESuperType	*eSuperType*	add	④	+			+			x		x	
				remove				-			-		x	x	x
	ETypedElement (inherited by: EAttribute, EReference)	Update UpperBound	*upperBound*	x → y, x>1, y>x	③		o		+			x		x	
				1 → >1			o		+				x	x	
				x → y, y>1, y<x			o				-		x	x	
				>1 → 1			o				-		x	x	
		Update LowerBound	*lowerBound*	x → y, x>0, y>x			o				-	x		x	
				0 → >0			o				-	x		x	
				x → y, y>0, y<x			o		+			x		x	
				>0 → 0			o		+			x		x	
		Update Ordered	*ordered*	false → true			o			o			x	x	
				true → false			o			o			x	x	
		Update Unique	*unique*	false → true			o				-		x	x	
				true → false			o		+				x	x	
	EStructuralFeature (inherited by: EAttribute, EReference)	Update EContainingClass	*eContainingClass*	push down	④	+		-			-		x	x	
				pull up		+		-	+			x		x	
				along reference with upperBound = 1	②	+		-		o			x	x	
				other	④	+		-	+		-		x	x	
	EAttribute	Update EAttributeType	*eAttributeType*	specialize	③		o				-	x		x	
				generalize			o		+				x	x	
				other			o		+		-		x	x	
	EReference	Update Containment	*containment*	false → true			o				-	x		x	
				true → false			o		+			x		x	
		Update EReferenceType	*eReferenceType*	specialize			o				-	x		x	
				generalize			o		+				x	x	
				other			o		+		-		x	x	
		Update EOpposite	*eOpposite*	add			o				-	x		x	
				remove			o		+			x		x	
	EEnumLiteral	Update Value	*value*	oldValue <> newValue	①		o			o		x		x	
		Update EEnum	*eEnum*	oldeEnum <> neweEnum	②	+		-		o			x	x	

Legend: x ... True n.a. ... Not applicable [1] ... Always true when changed in superclass +... Increase o... Neutral -... Decrease

Group: ①... Renaming updates ②... Moving updates ③... Restricting/relaxing updates ④... Constructive/destructive updates

Table 2. Set of Atomic Changes and their Impacts on OCL Expressions

above. First, updates increasing information capacity are comparable to constructive changes and thus, non-breaking. Second, updates decreasing information capacity are comparable to destructive changes and are thus, breaking. The scope of these updates is local, unless eSuperTypes are removed, affecting inheriting elements and therefore, having global impact.

3 Lessons Learned

This section discusses lessons learned gained from (i) establishing the complete and minimal set of changes as well as from (ii) investigating impacts.

Universal Applicability of Change Set Derivation Procedure. Although we focused on one specific meta-MM, i.e., Ecore, the approach of deriving constructive and destructive changes from concrete meta-classes as well as updative changes from all meta-features is universally applicable since it might be applied to any meta-metamodel.

Atomic Changes Allow for Non-redundant Impact Analysis. Since the employed change set is minimal comprising atomic changes, only, it allows for non-redundant impact analysis. In contrast, a set of composite changes might include overlaps, e.g., "Extract Class" and "Extract Superclass" both include the change "Create EClass". Composite changes, however, might held more information to be exploited for co-evolution.

State-Changes of Meta-Features are Pivotal. As might be seen in Table 2, all changes of meta-features have been broken down into several cases, explicating different state changes. This has been necessary, since different state changes entail different effects on structural complexity and information capacity and, consequently, impact OCL expressions differently, e.g., the state change of upperBound from 1 to > 1 has breaking impact, whereas the state change from > 1 to another number > 1 has not.

Increase of Structural Complexity Breaks Models, but not OCL. All constructive changes as well as updates with constructive effects (cf. part of group ④) that increase structural complexity never break OCL expressions. This is in contrast to model co-evolution where the introduction of required elements has breaking impact on models, since models rely on a different kind of relationship to their MM, i.e., "conforms to", while transformations "domain conform to" their source and target MMs.

Changes not Affecting Structural Complexity may Break OCL. Impact analysis revealed that changes not affecting structural complexity, e.g., updates of group ③, may nevertheless induce a syntactical breakage of OCL expressions in certain cases as explicated above. This is, since changes of group ③ may induce implicit type changes of the underlying OCL datatypes and by this, change the set of valid operations.

4 Related Work

Subsequently, related work is evaluated with respect to its focus, supported changes, impact analysis on OCL, and support by a prototypical implementation (cf. Table 3).

Regarding the *focus of co-evolution* in a specific *technical space*, two groups of approaches exist. Most closely related, the first group of approaches targets the co-evolution of transformations employing OCL expressions [5, 6, 13] in the *technical space* of Ecore, whereby the co-evolution of the OCL-part is considered particularly

Approach	Focus of Work					Supported Changes					Impact on OCL		Implementation
	Co-Evolution of		Technical Space			Classes of Supported Changes			Complete Set	Minimal Set			
	OCL in Model Transformations	OCL Constraints in Metamodels	Ecore	UML	MOF	Constructive	Destructive	Updative			Syntax	Scope	
García et al. [6]	~ (ATL)		✓			~	~	~	× (23 atomic & composite changes)	×	~	×	✓
Garcés et al. [5]	× (ATL)		✓			~	~	~	× (number of atomic & composite changes unkown)	×	×	×	✓
Kruse [13]	× (ATL)		✓			~	~	~	× (16 atomic & composite changes)	×	✓	×	✓
Hassam et al. [8]		✓			✓	~	~	~	× (17 atomic & composite changes)	×	✓	×	✓
Markovic et al. [14]		✓		✓		×	×	~	× (6 atomic & composite changes)	×	✓	×	✓
Kosiuczenko [12]		✓		✓		~	~	~	× (number of atomic & composite changes unkown)	×	×	×	×
Correa et al. [3]		✓		✓		~	×	~	× (number of atomic & composite changes unkown)	×	×	×	✓
Own work	✓ (ATL)		✓			✓	✓	✓	✓ (30 atomic changes)	✓	✓	✓	Future work

Legend: ✓ ... true × ... false ~ ... partially true

Table 3. Comparison of Related Approaches

by one of them [6], only. More widely related since not basing on Ecore and by this entailing a different set of changes, but nevertheless facing similar challenges, the second group concentrates on the co-evolution of OCL constraints as parts of UML class diagrams [3, 12, 14], with one exception basing on MOF [8].

Considering the *supported changes*, six approaches [3, 5, 6, 8, 12, 13] partially allow for constructive changes, five of those [5, 6, 8, 12, 13] partially consider destructive changes, and updative changes are partially supported by all approaches. Thus, no approach covers a *complete* change set. However, the surveyed approaches additionally consider composite changes, which will be one line of future work as detailed below. By concentrating on composite changes, no approach presents a minimal change set, which is different to our work providing a systematically derived, minimal set of changes.

Regarding the *impact on OCL*, four approaches [6, 8, 13, 14] consider breaking and non-breaking impacts on the *syntax*, whereby one of them [6] considers impacts partially, only. Considering the *scope* of impact on OCL, no approach regards this. Finally, six approaches [3, 5, 6, 8, 13, 14] provide an *implementation*, while a sole approach is conceptual, only, like the work presented in this paper.

In summary, one might see that the work presented in this paper is unique with respect to the complete and minimal set of changes and a systematic in-depth investigation of impacts. This is in contrast to related approaches, which rather concentrate on fully supporting co-evolution for smaller sets of selected composite changes.

5 Conclusion & Future Work

This paper provided a systematic investigation of impacts of MM evolution on OCL expressions in model transformations. Basing thereupon, several lines of future work remain open. First, resolution actions to resolve violations caused by MM evolution have to be identified. Their goal will be to perform local repairing by establishing a view simulating the old MM version, e.g., in case of decreasing the upperBound from > 1 to 1, the now single-valued feature will be wrapped into a collection. In case that multiple changes have been performed on a single MM-element, the resolution actions should be chained analogous to the idea presented in [21]. This chaining will represent a first step towards the support for composite changes out of atomic changes, which will be the next step in our research agenda. Moreover, we plan to investigate impacts and

resolution actions for complete transformation definitions, i.e., not only parts written in OCL, thereby also focusing on impacts caused by an evolution of the target MM. Finally, we will implement the conceptual approach presented in this paper.

References

1. Bézivin, J.: On the Unification Power of Models. SoSym 4(2) (2005)
2. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): A Definitive Guide. In: Formal Methods for Model-Driven Engineering. Springer (2012)
3. Correa, A., Werner, C.: Applying Refactoring Techniques to UML/OCL Models. In: UML 2004. Springer (2004)
4. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In: ICGT. Springer (2012)
5. Garcés, K., Vara, J., Jouault, F., Marcos, E.: Adapting transformations to metamodel changes via external transformation composition. SoSym (2013)
6. García, J., Diaz, O., Azanza, M.: Model Transformation Co-evolution: A Semi-automatic Approach. In: Software Language Engineering. Springer (2013)
7. Gruschko, B., Kolovos, D., Paige, R.: Towards Synchronizing Models with Evolving Metamodels. In: Int. Workshop on Model-Driven Software Evolution (2007)
8. Hassam, K., Sadou, S., Gloahec, V.L., Fleurquin, R.: Assistance System for OCL Constraints Adaptation during Metamodel Evolution. In: SMR. IEEE (2011)
9. Herrmannsdörfer, M., Wachsmuth, G.: Coupled Evolution of Software Metamodels and Models. In: Evolving Software Systems. Springer (2014)
10. Iovino, L., Pierantonio, A., Malavolta, I.: On the Impact Significance of Metamodel Evolution in MDE. JoT 11(3) (2012)
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1–2) (2008)
12. Kosiuczenko, P.: Redesign of UML class diagrams: a formal approach. SoSym 8(2) (2009)
13. Kruse, S.: On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In: Int. Workshop on Models and Evolution (2011)
14. Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. SoSym 7(1) (2008)
15. Méndez, D., Etien, A., Muller, A., Casallas, R.: Towards Transformation Migration After Metamodel Evolution. In: Int. Workshop on Models and Evolution (2010)
16. Miller, R.J., Ioannidis, Y.E., Ramakrishnan, R.: The use of information capacity in schema integration and translation. In: VLDB. vol. 93. Morgan Kaufmann (1993)
17. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/1.1 (2011)
18. Object Management Group: OMG Object Contraint Language (OCL). http://www.omg.org/spec/OCL/2.4 (2014)
19. Rossi, M., Brinkkemper, S.: Complexity Metrics for Systems Development Methods and Techniques. Information Systems 21(2) (1996)
20. Schönböck, J., Kusel, A., Etzlstorfer, J., Kapsammer, E., Schwinger, W., Wimmer, M., Wischenbart, M.: CARE - A Constraint-Based Approach for Re-Establishing Conformance-Relationships. In: APCCM (2014)
21. Wimmer, M., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Sánchez Cuadrado, J., Guerra, E., De Lara, J.: Reusing Model Transformations across Heterogeneous Metamodels. In: Int. Workshop on Multi-Paradigm Modeling (2011)
22. Wimmer, M., Martínez, S., Jouault, F., Cabot, J.: A Catalog of Refactoring for Model-to-Model Transformations. JOT 11(2) (2011)

A Generic Framework for Analyzing Model Co-Evolution

Sinem Getir[1], Michaela Rindt[2] and Timo Kehrer[2]

[1]Reliable Software Systems, University of Stuttgart, Germany
sinem.getir@informatik.uni-stuttart.de
[2]Software Engineering Group, University of Siegen, Germany
{mrindt,kehrer}@informatik.uni-siegen.de

Abstract. Iterative development and changing requirements lead to continuously changing models. In particular, this leads to the problem of consistently co-evolving different views of a model-based system. Whenever one model undergoes changes, related models should evolve with respect to this change. Domain engineers are faced with the huge challenge to find proper co-evolution rules which can be finally used to assist developers in the co-evolution process. In this paper, we propose an approach to learn about co-evolution steps from a given co-evolution history using an extensive analysis framework. We describe our methodology and provide the results of a case study on the developed tool support.

Keywords: Model-driven engineering, model evolution, multi-view modeling, model co-evolution, model synchronization, model differencing

1 Introduction

The multi-view paradigm is a well-established methodology to manage complexity in the construction of large-scale software systems. In Model-driven Engineering (MDE), this paradigm leads to the concept of multi-view modeling; different modeling notations are used to describe different aspects such as structure, behavior, performance, reliability etc. of a system.

Iterative development and changing requirements lead to continuously changing models. Consequently, this entails the special challenge to consistently co-evolve different views of a system [12]. In practice, this challenge usually appears as a synchronization problem; different (sub-)models, each of them representing a dedicated view on the system, are usually edited independently of each other. This occurs if they are assigned to different developers or due to the fact that a developer concentrates on a single aspect at a specific point of time [13]. Thus, changes to one model must be propagated to all related models in order to keep the views synchronized and to avoid inconsistencies.

We assume a setting as shown by the bottom-left part of Figure 1, the terminology is partly adopted from related work on model synchronization and model co-evolution [5, 6]: A source model $M_{src,n}$ is related to a target model $M_{tgt,n}$ via traces. A source model is the model that undergoes changes and a target

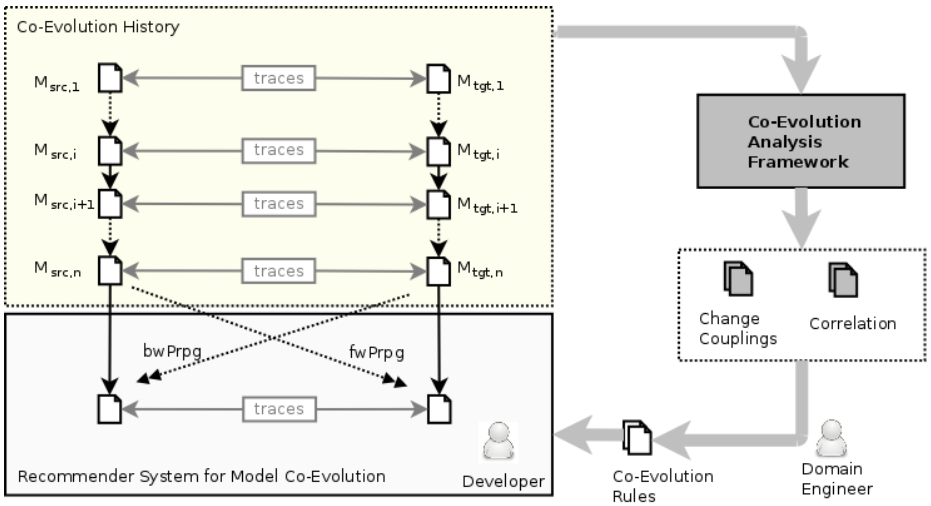


Fig. 1. Overview of the overall co-evolution process

model is the model to which these changes have to be propagated. Finally, a trace is a relationship between elements in these two different models. Forward propagation (*fwPrpg*) denotes the migration of the target model in response to changes occurring in the source model. Backward propagation (*bwPrpg*) denotes the migration of the source model in response to changes occurring in the target model. We refer to both kinds of propagations as *co-evolution steps*. From a technical point of view, co-evolution steps can be (semi-)automated via bidirectional model transformations. We call the transformation rules from which propagation rules can be derived as *co-evolution rules*.

However, due to the multitude of different modeling notations, the manual specification of co-evolution rules is a tedious and challenging task. Domain engineers, who have to find proper co-evolution rules, are faced with two essential questions: (1) Do certain changes on a source model correlate with changes on the target model? (2) If so, how are the changes coupled with each other? There are several domains for which no simple and straightforward co-evolution exists. The only viable solution is to pre-define possible co-evolution rules which can be offered to developers as possible options. For instance, this is the case for software architecture and quality of service models [4]. In Section 2, we introduce software architecture models and state charts as another example of co-evolving models which demonstrates the aforementioned research questions. We use the same example to serve as a running example throughout the paper.

This paper reports on our ongoing work on the semi-automated co-evolution of models of arbitrary source and target domains. The general process is illustrated by Figure 1. We propose to observe the co-evolution history in order to learn about developer decisions and to finally predict the co-evolution steps with a certain degree of probability. The more evolution steps are analyzed, the more accurate prediction results are expected. The contribution of this paper is the co-

evolution analysis framework which serves as a foundation for this co-evolution process. The analysis results can be used to generate co-evolution rules for a recommender system to interactively support model co-evolution. We describe our approach in Section 3. Tool support and early evaluation results which demonstrate the feasibility of our approach are briefly discussed in Section 4. Related work is analyzed in Section 5. We draw some conclusions and give an outlook on future work in Section 6.

2 Co-Evolution of Multi-View Models

Component diagrams and state charts are widely used notations to model structure and behavior in component-based software engineering. Intuitively, there are several relations between model elements of both views. For example, every state usually has a relation to a component, not necessarily the other way round. Transitions between states somehow reflect the interfaces and connections of the corresponding components in the component diagram. The hierarchy of composite states is expected to correspond to the hierarchical structure of components and their respective sub-components. Despite those rather intuitive relationships, consistently co-evolving both views is not a straight forward process, which is illustrated by the following example.

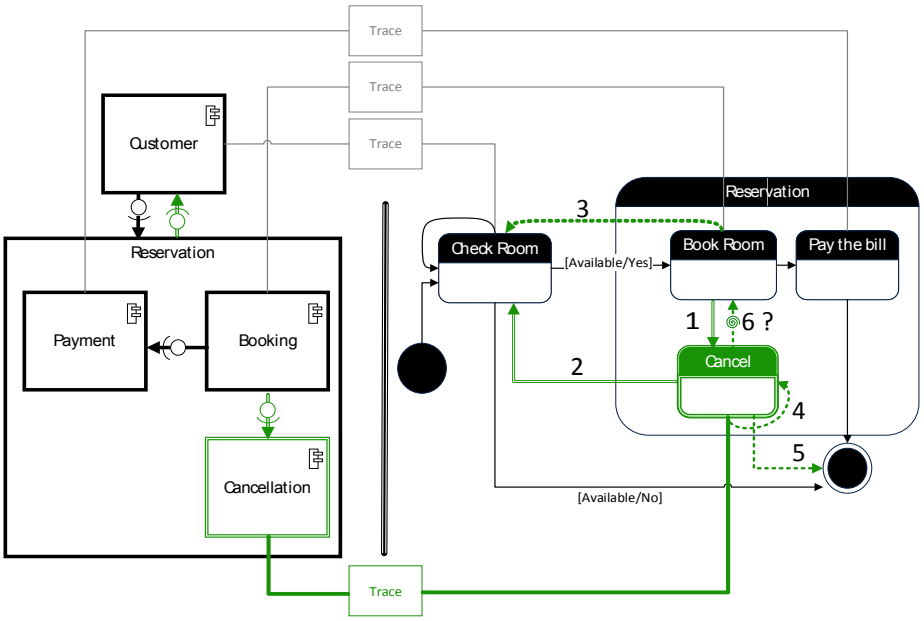


Fig. 2. Sample hotel reservation system modeled from two different viewpoints

Figure 2 shows a simple hotel reservation system modeled from two different viewpoints. The initial version of the system architecture consists of three

components, namely *Customer*, *Booking* and *Payment*. Relations between corresponding states and components are explicitly given by trace links. The system evolves at some point of time because it requires a new function to cancel a reservation process. In general, we assume that models are edited by means of a set of language-specific *edit operations*. An *edit step* invokes an edit operation and supplies appropriate actual parameters, which are also referred to as arguments. In our example, the revised version of the component model is obtained in three edit steps, namely the creation of the component *Cancellation* and two connectors. The new component and its connections to other components are highlighted in Figure 2 by doubled lines.

State chart elements printed in doubled lines indicate the developer's intention of co-evolution steps in response to the changes in the component diagram (1,2). We discuss several additional co-evolution steps which are possible on the state chart (M_{tgt}) in response to the changes in the component diagram (M_{src}). Note that these co-evolution steps are only assumptions which are based on domain knowledge, they are not meant to be a result of an empirical analysis.

Elements printed in dotted lines represent expected co-evolution steps which are, however, not intended by the user (3,4,5). Finally, a dotted line with spiral indicates an unexpected co-evolution step which is nonetheless intended by the user (6). We do not claim the set of possible options (1)-(6) to be complete. Nevertheless, it demonstrates the huge challenge of predicting the proper co-evolution steps:

- As the component *Cancellation* is added as a sub-component of reservation, a new state called *Cancel* is expected to be created as a sub-state of the corresponding composite state *Reservation*.
- The creation of transition (1) is expected due to the creation of port and interface relations of the corresponding components in the component model.
- Although there is no explicit relation between the components *Cancellation* and *Customer*, the creation of transition (2) is expected. Because the newly created relation between the composite component *Reservation* and the top-level component *Customer*, as a result of the creation of *Cancellation*. However, the new component may lead to an interaction between the components *Booking* and *Customer* indirectly via interfaces as well, therefore we should also consider the transition (3) with a small expectation.
- The required information for the proposed transitions (4) and (5) cannot be gathered from the component diagram. However, taking general state chart semantics into account, they can be presented to the developer as a possible option.
- Finally, we point out transition (6). The developer wants to create a loop between the states *Book Room* and *Cancel* which cannot be clearly anticipated from the component diagram since we observe only one direction for communication. Nonetheless, this option can be offered to the developer with a low probability.

We can conclude that each edit step on the component model may lead to many arbitrary co-evolution steps on the state chart. Some forward propagations

can be expected with a high probability based on the changes in the component diagram, others can only be offered as a set of possible choices.

3 Co-Evolution Analysis Framework

In Section 2, we have demonstrated a running example as a motivation of our analysis framework. We have presented possible co-evolution steps and observed that there are highly expected, less expected and unexpected changes for state charts when the component diagram evolves.

To study such changes and their relations, our co-evolution analysis framework takes a co-evolution history as illustrated by Figure 1 as input. Each pair of successive versions $i \rightarrow i+1$ from the given history is referred to as *evolution scenario* $ev_{i \rightarrow i+1}$. We assume that the co-evolution history includes consistent views for every evolution scenario. We further assume a model differencing engine to be available, which, given a set of possible edit operations for instances of a meta-model MM and successive model versions M_i and M_{i+1}, calculates a difference diff(M_i, M_{i+1}). A difference diff(M_i, M_{i+1}) is defined to be a partially ordered set of edit steps $s_1 \ldots s_k$. We finally offer two kinds of analysis functions; the *correlation analysis* is described in Section 3.1, the additional *coupling analysis* is presented in Section 3.2.

3.1 Correlation Analysis

We use the well-known Pearson correlation coefficient to assess the dependency between edit operations which are applicable to the source and target models. The basic processing steps of our correlation analysis are shown by Figure 3. For each evolution scenario $ev_{i \rightarrow i+1}$ of the co-evolution history, we first compute the differences diff($M_{src,i}$,$M_{src,i+1}$) and diff($M_{tgt,i}$,$M_{tgt,i+1}$). Subsequently, we count the edit steps contained by each of the obtained differences and group them by evolution scenarios and edit operations invoked by the respective edit steps. The sets of edit operations, which are available for instances of MM_{src} and MM_{tgt}, are given as additional input parameters of the correlation analysis.

Based on the calculated differences, we basically construct two matrices. For source model changes, we construct an e-×-s matrix where e denotes the number of evolution scenarios in the history (i.e., $e = n-1$), s denotes the number of edit operations available for instances of MM_{src}. A variable $a_{i,j}$ ($i \in \{1, ..., e\}$, $j \in \{1, ..., s\}$) represents the number of edit steps of type j (i.e. edit steps invoking edit operations represented by j, e.g. *createComponent* in our running example) in evolution scenario i. Analogously, an e-×-t matrix is being constructed for target model changes, where t denotes the number of edit operations available for instances of MM_{tgt}.

Let $X = \langle x_1, x_2, ..., x_e \rangle$ be a column vector of the e-×-s matrix, and $Y = \langle y_1, y_2, ..., y_e \rangle$ be a column vector of the e-×-t matrix. Then we can compute the Pearson correlation coefficient $r_{X,Y}$ for each combination of column vectors X and Y in order to quantify the linear relationship between edit operations that have been applied to the source and target models.

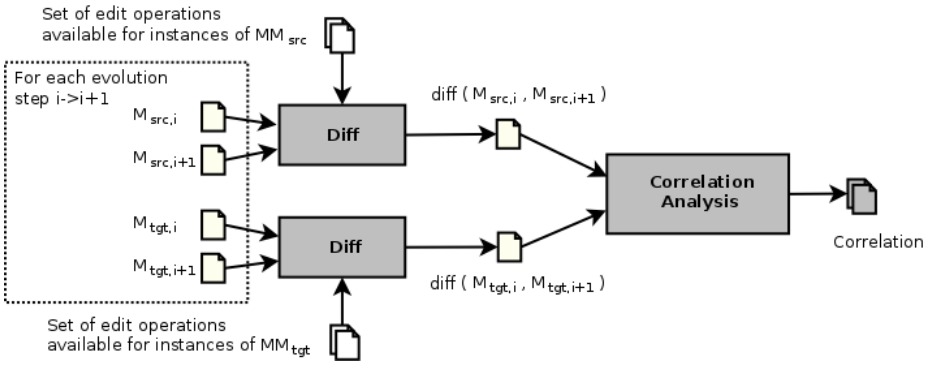


Fig. 3. Correlation analysis: basic proceeding, input and configuration parameter

3.2 Coupling Analysis

The correlation analysis has the advantage that it only requires the source and target models of each evolution scenario $ev_{i \to i+1}$. Thus, this approach can also be applied to study the co-evolution history in cases where no explicit trace links between the observed source and target model exist. However, a correlation between edit operations does not imply that the respective edit steps are actually coupled. In other words, they can have a dependency by coincidence such that none of the involved arguments are actually related by a trace. Hence, we also provide a second analysis function which is capable of identifying *coupled changes*. Such an analysis can provide knowledge about user's modeling intentions enhancing correlation analysis results, for example learning of the loop intention by the user, as provided in Figure 2 with transition (6).

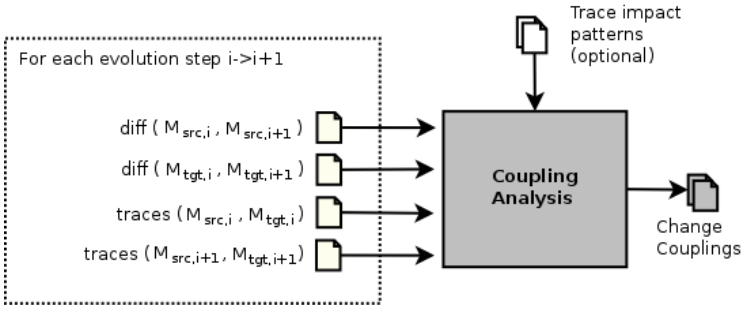


Fig. 4. Coupling analysis: basic proceeding, input and configuration parameter

In general, a coupled change identifies a pair of edit steps which have happened in the same evolution scenario. It also identifies the changed model elements which are connected (either directly or indirectly) to each other and were not just coincidentally changed in the same evolution scenario. We assume here that trace links identify related model elements of the source and target model.

These are, together with the model differences for each evolution scenario, provided as additional input parameters of the coupling analysis (see Figure 4).

Let $args(s)$ be the set of arguments of an edit step s. Basically, a pair of edit steps (s_{src}, s_{tgt}) is considered to be a coupled change, if we can find a pair of arguments (a_{src}, a_{tgt}), with $a_{src} \in args(s_{src})$ and $a_{tgt} \in args(s_{tgt})$, which are connected via a trace link.

Additionally, domain-specific *trace impact patterns* can be specified as optional inputs of the coupling analysis. These patterns allow to extend the search for coupled edit steps to the "neighborhood" of elements which are directly connected by a trace link. Consider for instance our running example shown in Figure 2. Here, trace links are only provided for related states and components. However, component connectors and state transitions are also to be considered as related if the connected components/states are related. This can be specified by a trace impact pattern as shown in Figure 5, i.e. the component connector labelled as *cc* and the state transition labelled as *st* are implicitly related. Consequently, a pair of edit steps modifying occurrences of *cc* and *st*, respectively, are to be considered as coupled.

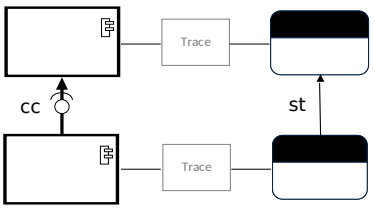


Fig. 5. Example of a trace impact pattern

Coupled changes are summarized over all evolution scenarios of the history as follows: We construct a s-$\times$-t matrix where s denotes the number of edit operations available for instances of MM_{src} and t denotes the number of edit operations available for instances of MM_{tgt}. A variable $a_{i,j}$ ($i \in \{1, ..., s\}$, $j \in \{1, ..., t\}$) is computed as the fraction of coupled edit steps of types i and j (i.e. edit steps invoking edit operations represented by i and j, respectively) with respect to all edit steps of type i being observed in the source model history.

4 Tool Support

We have prototypically implemented the analysis framework proposed in Section 3 on the widely used Eclipse Modeling Framework (EMF) and the model differencing engine SiLift [8, 9]. It is made available to the general public at the SiLift website[1] in order to enable other researchers to study the co-evolution of any EMF-based models.

Adaption of the generic framework. In order to adapt the generic framework to new modeling languages, i.e., to adapt it to a given source and target domain, one has to configure the SiLift differencing tool chain. Primarily, suitable edit operations for the source and target domain have to be provided. In SiLift, we use the model transformation language and system Henshin [1] to implement edit operations as declarative transformation rules, to which we refer to as *edit*

[1] http://pi.informatik.uni-siegen.de/Projekte/SiLift/coevolution.php

rules. Domain engineers can make use of the EMF-based meta-tool SERGe (SiDiff Edit Rule Generator) [10] in order to generate basic edit rules, which can be derived from MM_{src} and MM_{tgt}, respectively. Basic edit rules can be complemented by semantically rich complex edit rules such as refactoring operations. Typically, many complex edit rules can be composed of basic edit rules generated by SERGe.

Optionally, a set of trace impact patterns can be specified as additional input for the coupling analysis. Trace impact patterns are also specified in Henshin. We refer to these pattern specifications as *trace impact rules*. Trace impact rules do not implement in-place transformations, but serve as specifications of graph patterns which are to be found by the Henshin matching engine. Obviously, trace impact rules have to be specified manually by a domain engineer.

PPU Case Study. In order to demonstrate the feasibility of our approach, we have adapted the analysis framework to be used in the PPU(Pick and Place Unit) case study [11], which provides several evolution scenarios of a laboratory plant. In our previous work [4], we modeled each of the scenarios from two different viewpoints using two types of modeling languages: A simple architecture description language (SA) was used to model the system architecture, fault trees (FT) were used to model undesired system states and their possible causes.

All configuration artifacts which are needed to adapt the analysis framework to SA and FT models are available at the EnSure website[2]. In summary, we identified 82 edit rules available for FT models, 69 of them could be generated with SERGe. For SA models, we identified 42 suitable edit rules of which only one had to be specified manually, all other 41 edit rules could be generated with SERGe. In addition, we specified 6 trace impact rules serving as additional input of the coupling analysis. Consequently, we were able to automatically generate the results that have been produced by a manual analysis in our previous work [4].

5 Related work

Most approaches to model co-evolution address the migration of different types of MDE artifacts in response to meta-model adaptions. MDE artifacts which have to be migrated are, for example, instance models [7], model transformations [14], or syntactic and semantic constraints [3].

Only a few approaches address the evolution of multi-view models, which is most often considered as a model synchronization problem. Solutions are often based on the principle of bidirectional model transformations which are used to derive incremental change propagation rules, e.g., [5, 16, 6]. Among them, the approaches of Giese et al. [5] and Hermann et al. [6] are based on Triple Graph Grammars (TGGs). TGG rules describe correspondences between elements of source and target models together with the according forward and backward

[2] http://www.iste.uni-stuttgart.de/rss/projects/ensure/co-evolution

editing behavior. Bergmann et al. [2] present a novel type of model transformation to which they refer to as change-driven transformations. Change-driven transformations are directly triggered by complex model changes and thus can be utilized to specify sophisticated co-evolution patterns. A similar approach is presented by Wimmer et al. [15].

In contrast to our approach, TGG rules and change-driven transformations must be specified manually, whereas we intend to generate our co-evolution rules. In fact, we believe that existing approaches based on TGGs, change-driven transformations or similar techniques, can be also supported by our co-evolution analysis framework. Up to the best of our knowledge, we are not aware of any approach providing a framework to empirically study co-evolution by analyzing the history of co-evolving models.

6 Conclusion and Future Work

Many approaches for consistently co-evolving models and other related MDE artifacts have been proposed recently. Some are tailored to fixed source and target domains while others are more generic and adaptable.

However, correlation and coupling of changes has not been researched in-depth for many types of co-evolving (sub-)models. In order to close this research gap, we focus on establishing a co-evolution analysis framework to analyze the history of co-evolving models of arbitrary types. This will provide the foundation for synthesizing co-evolution rules in an automated way. Although the analysis framework still needs some configuration data as input, we conclude from the PPU case study that this adaption to a dedicated source and target domain can be done with moderate effort. Currently, the co-evolution rules which we finally intend to use as input of a co-evolution framework (see Figure 1) still have to be manually synthesized based on the information which is produced by the analysis framework. Larger case studies are needed to evaluate how far we can push the generation of co-evolution rules and how much training data is needed to derive appropriate co-evolution rules.

On the one hand, these co-evolution rules can be used to configure existing model synchronization frameworks in cases of domains where the co-evolution process can be fully automated. On the other hand, co-evolution rules serve as basis for a recommender system, which is able to handle semi-automated co-evolution of models. The latter one is subject to our future work.

Acknowledgments. This work is supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution. The authors would like to thank André van Hoorn, Matthias Tichy and Lars Grunske for the initial discussions and valuable reviews.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place emf model transformations. In: Intl. Conf. on Model Driven Engineering Languages and Systems, pp. 121–135 (2010)
2. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations - change (in) the rule to rule the change. Software and System Modeling 11(3), 431–461 (2012), http://dx.doi.org/10.1007/s10270-011-0197-9
3. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Supporting the co-evolution of metamodels and constraints through incremental constraint management. In: Intl. Conf. on Model Driven Engineering Languages and Systems. pp. 287–303 (2013)
4. Getir, S., Van Hoorn, A., Grunske, L., Tichy, M.: Co-evolution of software architecture and fault tree models: An explorative case study on a pick and place factory automation system. In: Intl. Workshop on Non-functional Properties in Modeling: Analysis, Languages, Processes. pp. 32–40 (2013)
5. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Int. Conf. on Model Driven Engineering Languages and Systems, pp. 543–557 (2006)
6. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y., Gottmann, S., Engel, T.: Model synchronization based on triple graph grammars: correctness, completeness and invertibility. Software & Systems Modeling pp. 1–29 (2013)
7. Herrmannsdörfer, M., Wachsmuth, G.: Coupled evolution of software metamodels and models. In: Evolving Software Systems, pp. 33–63 (2014)
8. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Intl. Conf. on Automated Software Engineering. pp. 163–172 (2011)
9. Kehrer, T., Kelter, U., Taentzer, G.: Consistency-preserving edit scripts in model versioning. In: Intl. Conf. on Automated Software Engineering. pp. 191–201 (2013)
10. Kehrer, T., Rindt, M., Pietsch, P., Kelter, U.: Generating edit operations for profiled UML models. In: Intl. Workshop on Models and Evolution. pp. 30–39 (2013)
11. Legat, C., Folmer, J., Vogel-Heuser, B.: Evolution in industrial plant automation: A case study. In: Proc. of IECON 2013. IEEE (2013)
12. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: Intl. Workshop on Principles of Software Evolution. pp. 13–22 (2005)
13. Ruhroth, T., Gärtner, S., Bürger, J., Jürjens, J., Schneider, K.: Versioning and evolution requirements for model-based system development. In: Intl. Workshop on Comparison and Versioning of Software Models (2014)
14. Taentzer, G., Mantz, F., Lamo, Y.: Co-transformation of graphs and type graphs with application to model co-evolution. In: Graph Transformations, pp. 326–340. Springer (2012)
15. Wimmer, M., Moreno, N., Vallecillo, A.: Viewpoint co-evolution through coarse-grained changes and coupled transformations. In: Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings. pp. 336–352 (2012), http://dx.doi.org/10.1007/978-3-642-30561-0_23
16. Xiong, Y., Song, H., Hu, Z., Takeichi, M.: Synchronizing concurrent model updates based on bidirectional transformation. Software & Systems Modeling 12(1), 89–104 (2013)

Dealing with the coupled evolution of metamodels and model-to-text transformations

Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio

DISIM University of L'Aquila,
{name.lastname}@univaq.it

Abstract. In Model-Driven Engineering (MDE) the modification of a metamodel typically can invalidate many different sorts of artifacts. In order to mitigate the pragmatic consequences of such problem, several coupled evolution techniques have been introduced over the last few years mainly focussing on restoring the validity of models, transformations, and editors. However, none of the proposed techniques addressed the coupled evolution of metamodels and template-based code generators, which has been largely neglected despite the relevance of such systems. In an attempt to fill the gap, this paper presents an approach for the coupled evolution of Acceleo-based templating including the OCL embedded in its notation. The approach has been implemented and illustrated by means of a running example.

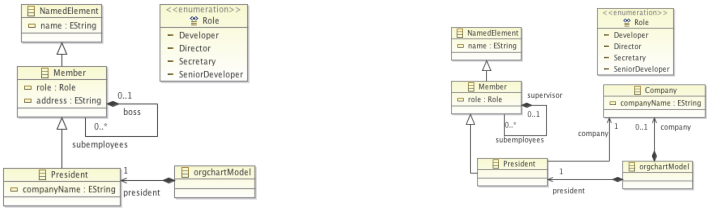
1 Introduction

In Model-Driven Engineering [1] (MDE) the employment of metamodels is pervasive. They are used to formally describe a wide range of artifacts, including models, transformations, concrete syntaxes, and editors. In essence, metamodels are at the core of any modeling ecosystem and their management is therefore key to success. Similarly to any software component metamodels are expected to evolve during their lifecycle [2]. However, because of the dependencies between metamodels and the related artifacts, modifying a metamodel might compromise the validity of the latters. Unfortunately, restoring the validity of such artifacts in a semi-automated manner is intrinsically difficult because it must consider both the purpose of the metamodel modification (i.e., updative, adaptive, performance, corrective or reductive) and the technical aspects (i.e., the when, where, what and how of changes) [3]. While several approaches already investigated the coupled evolution of models (e.g., [4,5,6]), transformations (e.g., [7,8,9]), and editors (e.g., [7,10]), the coupled evolution of template-based code generators has been largely neglected at the expense of reduced pragmatics.

In this paper, we propose an approach to the coupled evolution of metamodels and template-based code generators. In particular, we provide a solution to the problem of adapting Acceleo[1]-based templates when the metamodels of the input models are changed. The proposed approach is able to adapt corrupted Acceleo templates and it is performed by means of an ATL *adaptor*, i.e., a model transformation which takes the metamodel changes (represented in an opportune model-based notation), the model

[1] http://www.obeo.fr/pages/acceleo

II



(a) Initial version

(b) Evolved version

Fig. 1: OrgChart simple metamodel evolution

associated to the corrupted Acceleo template and returns the adapted template. The approach presents many similarities to other approaches focussing on the adaptation of other kinds of artifacts, however it presents also specific difficulties: a) all the possible metamodel refactorings (see [11]) must be dealt by the adaptor, and b) Acceleo provide model traversing facilities by means of the Object Constraint Language[2] (OCL) whose expressions must be adapted as well. Interestingly, to the best of our knowledge none of the current approaches focussed on code-generators, despite the relevance they have in a wide range of projects.

The paper is organized as follows. In Section 2 we clarify the context of this work, exposing briefly Acceleo and the co-evolution problem using a real case study. In Section 3 we propose the process for the resolution of the presented problem and we show the models before and after the resolution. In Section 4 a related section is organized for the comparison of the existing works and the presented one. We conclude also including the future works in Section 5.

2 The coupled evolution problem in model-to-text transformations

Almost any artifact involved in a model-driven development processes at different extent depends on the considered metamodels. Dependencies can emerge at different times during the metamodel life-cycle, and with different degrees of causality depending on the nature of the considered artifact. An ecosystem of modeling artifacts and tools developed atop of a metamodel, like the OrgChart *metamodel* shown in Fig. 1.a and presented later in the paper, is a living set of artifacts working together and built on the top the considered domain. For example *graphical* and *textual editors* might be developed to support the specification of OrgChart *models*. *Model transformations* might be also developed to generate target models or code out of source OrgChart models. Moreover, the Acceleo-based *WebPages code generator* is one of the possible code generator used to generate target Web pages from source OrgChart models.

When metamodels are changed e.g., to satisfy unforeseen requirements or simply to better represent concepts of the considered domain, the already existing artifacts might be compromised and they have to be adapted in order to recover their conformance with the new version of the changed metamodel. The metamodel evolution problem and the consequent ecosystem migration has been discussed [12] and to the best of our knowledge the adaptation of Acceleo-based generators has not been investigated yet and it represents the main goal of this paper.

[2] `http://www.omg.org/spec/OCL/`

```
1  [comment encoding = UTF-8 /]
2  [module generate('http://www.disim.univaq.it/orgchart')]
3  [template public generateElement(aPresident : President)]
4  [comment @main/]
5  [file (aPresident.companyName.concat('.html'), false, 'UTF-8')]
6  <html>
7    <head>
8      <script type='text/javascript' src='https://www.google.com/jsapi'></script>
9      <script type='text/javascript'>
10       google.load('visualization', '1', {packages: [ '[' /]'orgchart' [ ']' /]});
11       google.setOnLoadCallback(drawChart);
12       function drawChart() {
13         var data = new google.visualization.DataTable();
14         data.addColumn('string', 'Name');
15         data.addColumn('string', 'Manager');
16         data.addColumn('string', 'ToolTip');
17         data.addRows( [ '[' /]
18         [ '[' /]{v:'[aPresident.name/]', f:'[aPresident.name/]<div style="color:red; font-style:italic">President</div>'}, '', 'The
19           [for (subemp : Member | aPresident.subemployees) separator (',')]
20           [generateSubMembers(subemp)/]
21           [/for]
22         [ ']' /]
23         );
24         var chart = new google.visualization.OrgChart(document.getElementById('chart_div'));
25         chart.draw(data, {allowHtml:true});
26       }
27     </script>
28   </head>
29   <body><center><h2>[aPresident.companyName/] Orgchart</h2></center><div id='chart_div'></div></body>
30 </html>
31 [/file]
32 [/template]
33 [template public generateSubMembers(m : Member)]
34 [if (m.subemployees->size()>0)]
35 [ '[' /]{v:'[m.name/]', f:'[m.name/]<div style="color:red; font-style:italic">[m.role.toString()/]</div>'}, '[m.boss.name/]',
36 '[m.address/]'[ ']' /],
37    [for (subm : Member | m.subemployees) separator (',')]
38       [generateSubMembers(subm)/]
39    [/for]
40 [else]
41  [ '[' /]{v:'[m.name/]', f:'[m.name/]<div style="color:red; font-style:italic">[m.role.toString()/]</div>'}, '[m.boss.name/]',
42  '[m.address/]'[ ']' /]
43 [/if]
44 [/template]
```

Fig. 2: Sample Acceleo templates

The remaining of the section is organized as follows: an overview of Acceleo is given in Section 2.1. Section 2.2 introduces the problems related to the adaptation of Acceleo-based transformations according to the changes operated on the corresponding metamodels.

2.1 Acceleo-based model-to-text transformations

Acceleo is a model-to-text transformation tool typically used to develop code generators. Acceleo generators are based on templates that identify repetitive and static parts of the applications, and embody specific queries on the source models to fill the dynamic parts. Fig. 2 shows a fragment of the Acceleo template that has been developed to generate graphical representations of source OrgChart models as shown in Fig. 3. In particular, the template generates HTML and Javascript code that uses the Google Chart API[3] to get the chart representations like the one in Fig. 3.b from source organizational models. Each Acceleo-based text generator starts with the specification of a *module* referring to the metamodel that will be used during the generation process (see line 2 in Fig. 2). Lines 3-32 consist of the *template* used to transform instances of the *President* metaclass in Fig. 3a. In particular, for each president in the source model a corresponding HTML file is generated (see line 5). The file name is obtained from the attribute *companyName* in the *aPresident* variable, that is an instance of *President*.

From line 8 to 17 the template consists of static JavaScript code to create the chart. Interestingly, lines 18 to 21 consist of an iteration that creates the rows related to the

[3] `https://developers.google.com/chart/`

members to be represented as boxes in the charts. Each row contains information taken from the *Member* instance e.g., the *name* and the *role*. Once the president element has been transformed, the other members are generated by means of the template *generateSubMembers* (see lines 33-44)specifically developed to manage instances of the metaclass *Member*. Such a template iterates on the *subemployees* relations in order to generate corresponding boxes.

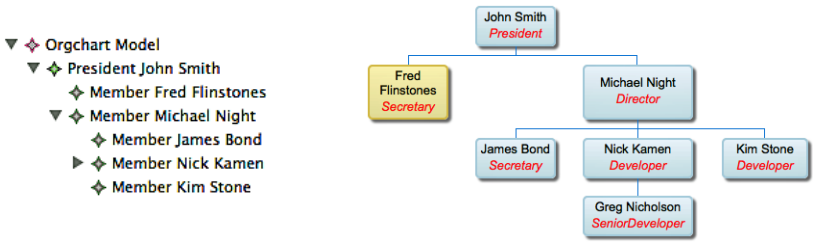
2.2 Invalidating Acceleo-based templates with metamodel evolution

As previously mentioned, existing modeling artifacts might be affected by the changes operated on the corresponding metamodels, and Acceleo templates are not an option. For instance, the metamodel evolution shown in Fig. 1b compromises the Acceleo templates previously analyzed that become invalid in different points. In particular, the new version of the OrgChart metamodel has been obtained by operating the following metamodel change patterns [6]:

i) a new metaclass with name *Company* has been added with the attribute *companyName* moved into it. Such a modification refers to the *extract metaclass* pattern;
ii) the attribute *address* in the initial *Member* metaclass has been removed by applying the *remove attribute* pattern;
iii) the reference *boss* has been renamed as *supervisor* by applying the *rename reference* pattern.

Because of such changes the templates presented in the previous section are invalid and cannot be applied on models conforming to the new version of the OrgChart metamodel. In particular, the references to the *companyName* attribute (e.g., see the expression *"aPresident.companyName"* at line 5 in Fig. 2) have to be migrated since the attribute has been moved to a new metaclass. A similar problem occurs at lines 36-42 that refers to the attribute *address* that has been removed. Finally, lines 35-41 have to be also migrated since they refer to the feature *boss* that has been renamed as *supervisor*.

Adapting Acceleo templates without a proper supporting can be a strenuous and error-prone task. In the next section we propose an approach based on model-differencing and model transformations that under certain conditions is able to automatically adapt affected Acceleo templates.



(a) Source OrgChart model

(b) Generated HTML5+JS chart

Fig. 3: Simple OrgChart model and corresponding graphical representation

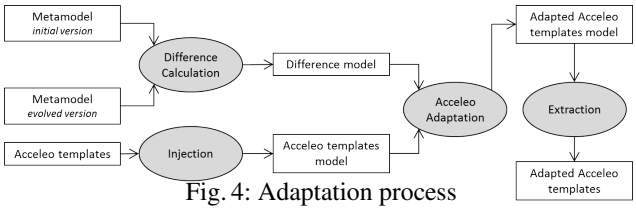


Fig. 4: Adaptation process

3 Adaptation of Acceleo templates

In this section we propose an approach able to adapt Acceleo templates that have to be migrated because of changes operated on the corresponding metamodels. The approach is based on the process shown in Fig 4 and it resembles the techniques we have already applied to adapt ATL transformations [7], TCS specifications [13], and GMF editors [10], that are different kinds of artifacts having the same co-evolution problem. In particular, the adaptation process consists of the following main activities:

▷ *Difference calculation*, given two subsequent versions of the same metamodel, their differences are calculated to identify the changes which have been operated on the first version of the model to obtain the last one. The calculation can be operated by any of the existing approaches able to detect the differences between any kind of models, like EMFCompare [4];

▷ *Difference representation*, the detected differences have to be represented in a way which is amenable to automatic manipulations and analysis. To take advantage of standard model driven technologies, the calculated differences should be represented by means of another model[14];

▷ *Generation of the adapted Acceleo templates*, the differences represented in the difference model are taken as input by the *Acceleo Adapter Transformation* able to adapt the source templates with respect to the operated metamodel modifications.

Concerning the last step of the process it is important to recall that metamodel changes can be classified as follows [6,9]:

▷ *non-breaking changes*: changes that do not break existing Acceleo templates that are still valid with the new version of the metamodel;

▷*breaking and resolvable changes:* changes that affect the validity of existing Acceleo templates but that can be automatically adapted to be applied on models conforming to the new version of the metamodel;

▷ *breaking and unresolvable changes:* changes that affect the validity of existing Acceleo templates and user intervention is required to migrate them.

By considering the previous classification, the adaptation process shown in Fig. 4 is able to migrate Acceleo templates only in case of breaking and resolvable changes. In case of unresolvable changes, comments are added in the generated templates in order to highlight the parts of the templates that have to be fixed by developers. Because of space limitations, in this paper we do not list the catalogue of metamodel changes according the classification above and interested readers can refer to [6] for a detailed discussion. In the remaining of the section we give some details about the representation

[4] EMFCompare: `http://www.eclipse.org/emf/compare/`

VI

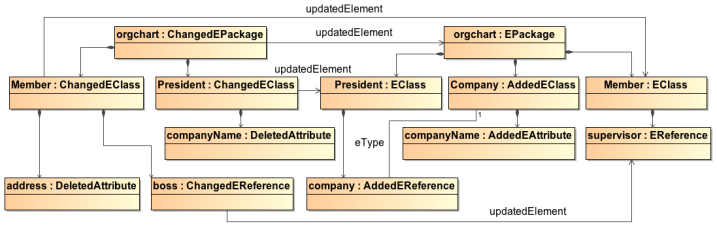


Fig. 5: Delta model representing the differences between the metamodels in Fig. 1

of metamodel differences (Section 3.1) and about the management of some breaking and resolvable changes (Section 3.2).

3.1 Representation of metamodel changes

The differences between different versions of a same metamodel are represented by exploiting the *difference metamodel* concept, presented by some of the authors in [15]. In particular, given two Ecore metamodels, their difference conforms to a difference metamodel *MMD* derived from Ecore by means of a model transformation as follows: for each class *MC* of the Ecore metamodel, the additional classes *AddedMC*, *DeletedMC*, and *ChangedMC* are generated in the extended Ecore metamodel by enabling the representation of the possible modifications that can occur on domain models and that can be grouped as follows: *i) additions*, new elements are added in the initial metamodel; *ii) deletions*, some of the existing elements are deleted, and *iii) changes*, some of the existing elements are updated. Fig. 5 shows a fragment of the model representing the differences between the metamodels in Fig. 1. For instance, the renaming of the reference *boss* into *supervisor*, is represented by means of the *ChangedClass* named *Member* that has the *ChangedReference boss* referring by means of the reference to the new *updatedElement* to the reference named *supervisor*.

3.2 Acceleo Adapter Transformation

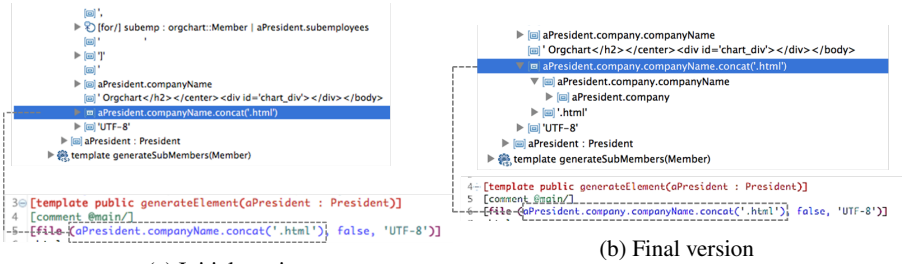
The adaptation of affected Acceleo templates is performed by means of an ATL transformation that takes as input the model of the affected Acceleo templates, the difference model representing the evolution of two subsequent versions of the same metamodel, and generates the adapted Acceleo templates. The transformation consists of a *conservative copy* part, including rules that simply copy the not affected elements in the template, and a *migration* part, consisting of one rule, each devoted to the management of a specific metamodel change. A fragment of the developed transformation is shown in Listing 1.1.Due to space limitations, Listing 1.1 reports only the rules managing the change patterns discussed in Section 2.2. Following we will describe the rules of the *Acceleo Adaptater* showing how the migration has been done on the template of our case study. As can be seen in Fig. 6 the corrupted Acceleo template is injected in a model with extension ".emtl", compliant to the Acceleo metamodel. In Figures 6a, 6b, excerpts of the templates and corresponding models have been reported (top is the model injected from the template source in the bottom).

```
1 module AcceleoAdapter;
2 create OUT:AcceleoMM from IN:AcceleoMM, InitialMM:ECORE, DELTA:DELTAMM, EvolvedMM:
      ECORE;
3 ...
4 rule PropertyCallExpExtractMC {
5  from s : AcceleoMM!PropertyCallExp  (s.existExtractMC())
6  to t : AcceleoMM!"ecore::PropertyCallExp" (
7   ...
8  referredProperty <- s.getReferenceExtractMC(),
9   source <- t1
10 ),
11 t1 : AcceleoMM!PropertyCallExp(
12  source <- thisModule.VariableExpExtractMC(s.source),
13  referredProperty <- s.getReferredPropertyExtractMC()
14 )
15 }
16 lazy rule VariableExpExtractMC
17 {
18  from s : AcceleoMM!VariableExp
19  to t: AcceleoMM!"ecore::VariableExp"(
20      ...
21      referredVariable <- t1
22  ),
23  t1 : AcceleoMM!"ecore::Variable"(..)
24 }
25 rule PropertyCallExp {
26  from s : AcceleoMM!PropertyCallExp
27  (not s.deletedEStructuralFeature() or not s.referredProperty.
       isChangedEStructuralFeature())
28  to t : AcceleoMM!"ecore::PropertyCallExp" (
29  ...
30  referredProperty <- s.getReferredProperty()
31 )
32 }
33 rule PropertyCallExpChanged {
34  from s : AcceleoMM!PropertyCallExp
35  (s.referredProperty.isChangedEStructuralFeature())
36  to t : AcceleoMM!"ecore::PropertyCallExp" (
37  ...
38  referredProperty <- s.referredProperty.getReferredPropertyChanged()
39 )
40 }
41 ...
```

Listing 1.1: Fragment of the Acceleo adapter transformation

PropertyCallExpExtractMC (lines 4-15 in Listing 1.1) is the matched rule responsible for managing attributes involved in extract metaclass changes, like the attribute *companyName* of the metamodel (Fig. 1a), cited in the template highlighted in Fig. 6a line 5. The filter condition in this rule calls the *existExtractMC* helper that checks if he considered attribute is involved in some extract metaclass change pattern. In this specific case, such a change pattern occurs because of the addition of the metaclass *Company*, the deletion of the attribute *companyName*, the addition of the reference *company* and the changed changed metaclass *President*. The execution of such rule on the considered template generates two nested *PropertyCallExp*s (see lines 8-14) for the proper navigation of the extracted attribute, see Fig. 6b, where the expression *"aPresident.company.companyName"* corresponds to a nested element in the related model. The *PropertyCallExpExtractMC* rule calls the lazy rule *VariableExpExtractMC* (see line 12) that creates new *VariableExp* and *Variable* OCL elements. The value of *referredProprerty* (line 13) binding is set by the helper *getReferredPropertyExtractMC* that retrieves the new reference according to the information available in the difference

VIII



(a) Initial version

(b) Final version

Fig. 6: Fragment of the *generateElement* template

model. This rule is able then to migrate the expression *aPresident.companyName* to the new expression *aPresident.company.companyName*.

The matched rule *PropertyCallExp* (line 25-32) is one of the conservative copy rule for the elements in the templates that are not affected by the operated metamodel changes. In fact, the rule condition at line 27 checks if the considered element is not deleted or changed according to the information available in the difference model. At line 30, the value of *referredProperty* is set by *getReferredProperty* that derives the right reference. In our example this rule is responsible for the deletion of the expression *m.address* (line 36 in Fig. 2) related to the deleted attribute *address* in the metamodel.

The matched rule *PropertyCallExpChanged* (line 33-40) manages reference renaming changes as in the case of the reference *boss* renamed as *supervisor* in our metamodels (Figures 1a and 1b). The condition of the rule checks if the renaming is occurring in the difference model, by using the helper *isChangedEStructuralFeature*. In this case the new *PropertyCallExp* is created, setting the *referredProperty* with the new value coming from the renamed element in the difference model, by using the helper *getReferredPropertyChanged*. For instance, the expression "*m.boss*" in Fig. 2 (line 41), referring to the renamed reference, is replaced with the expression "*m.supervisor*".

4 Related Work

Metamodeling ecosystems and coupled evolution have been presented in [12], exploring the problem artefact by artefact and including the relation definition among them. Coupled evolution of models and metamodel has been previously exhaustively treated in [6,5,16]. These works focus on the problem of models migration when metamodel changes. They use a model migration language, or an higher order transformation for migrating models. These approaches use a conservative copy for the non-breaking changes, like in our approach. Obviously the atifact kind is different but the intent is the same. Moreover, transformations and metamodel co-evolution is another interesting topic investigated in [8,17,9]. They propose methods and discussions about the problem that we have changing the metamodel which the transformations refer to. All those works use the similar definitions for the classification of changes. Other kinds of artefacts defined on the top of the metamodel can be concrete syntaxes definitions, like diagrammatic or textual. Also these artefacts have dependencies to the metamodel that need to be restored when the metamodel evolves. In [7] and [10] those problems are respectively treated proposing automation similar to the one described in this paper.

This work is strictly related the OCL definition [18], since the migration part is inherent to the Object Constraint Language used by Acceleo for the model navigation in the templates. The work developed for the migration of acceleo templates can be the inspiration or partially reused in other migration of artifacts using OCL, like OCL Query or OCl expression in ATL and so on. In [19] the authors have dealt with the problem of constraints adaptation in order to be compliant to the evolution of their associated metamodels. Since maintaining OCL constraints can be a tedious task, Kahina Hassam et al. propose to assist the developer to rewrite them after each evolution of the associated metamodels. In [20], the authors presented an architecture to automate coupled evolution on an arbitrary software domain. They compute equivalences and differences between any pair of metamodels (e.g., representing schemas, UML models, ontologies, grammars) to derive adaptation transformations from them, and they apply these adaptations as step wise automatic transactions on the initial metamodel, to obtain the final metamodel. These works are related for the OCL part that is in common with our work. In [21] presents *ChainTracker*, a general conceptual framework, and model-transformation composition analysis tool, that supports developers when maintaining and synchronizing evolving code-generation environments. *ChainTracker*, gathers and visualizes model-to-model, and model-to-text traceability information for ATL and Acceleo model-transformation compositions.

5 Conclusions and future work

The problem of coupled evolution in Model-Driven Engineering represents one of the major impediment for the practice of such software discipline. Several approaches have been already proposed mainly focussing on the adaptation of models, transformations, and - at different extent - editors. This paper extended existing techniques to the adaptation of template-based code generators, because such kind of code generators are widely used and part of routinary practices. In particular, the paper proposed an ATL adaptor to consistently migrate Acceleo templates in accordance to the changes operated on the corresponding metamodel. The main contribution of the paper consists in a) the refactoring coverage which is extensive and considers the major refactorings classified in [11]; and b) the migration of OCL expressions which are used by Acceleo for model traversing. The approach has been implemented and is illustrated throughout the paper by means of a running example. To the best of our knowledge, this is the first attempt in addressing the problem of the coupled evolution of template-based code generators. Future work includes the possibility of covering a part of the breaking and non resolvable cases by introducing models with partiality and uncertainty borrowed from the area of requirement engineering.

References

1. Schmidt, D.C.: Guest NOOPeditor's Introduction: Model-Driven Engineering. Computer **39** (2006) 25–31
2. Di Ruscio, D., Iovino, L., Pierantonio, A.: Coupled Evolution in Model-Driven Engineering. Software, IEEE **29** (2012) 78–84
3. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a Taxonomy of Software Change: Research Articles. Volume 17., New York, NY, USA, John Wiley & Sons, Inc. (2005) 309–332

4. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E., ed.: ECOOP 2007 Object-Oriented Programming. Volume 4609 of LNCS., Springer (2007) 600–624
5. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming. Genoa, Berlin, Heidelberg, Springer-Verlag (2009) 52–76
6. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE. (2008) 222–231
7. Di Ruscio, D., Iovino, L., Pierantonio, A.: A Methodological Approach for the Coupled Evolution of Metamodels and ATL Transformations. In Duddy, K., Kappel, G., eds.: Theory and Practice of Model Transformations. Volume 7909 of LNCS., Springer (2013) 60–75
8. Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A Novel Approach to Semi-automated Evolution of DSML Model Transformation. In van den Brand, M., Gaevi, D., Gray, J., eds.: Software Language Engineering. Volume 5969 of LNCS. Springer (2010) 23–41
9. Garca, J., Diaz, O., Azanza, M.: Model Transformation Co-evolution: A Semi-automatic Approach. In Czarnecki, K., Hedin, G., eds.: Software Language Engineering. Volume 7745 of LNCS. Springer (2013) 144–163
10. Di Ruscio, D., Lmmel, R., Pierantonio, A.: Automated Co-evolution of GMF Editor Models. In Malloy, B., Staab, S., van den Brand, M., eds.: Software Language Engineering. Volume 6563 of LNCS. Springer (2011) 143–162
11. The MDE Research Group: The Metamodel Refactorings Catalog. `http://www.metamodelrefactoring.org` (2013) University of L'Aquila.
12. Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary Togetherness: How to Manage Coupled Evolution in Metamodeling Ecosystems. In: Procs.of the 6th Int. Conf. on Graph Transformations. ICGT'12, Berlin, Heidelberg, Springer-Verlag (2012) 20–37
13. Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the Coupled Evolution of Metamodels and Textual Concrete Syntax Specifications. In: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conf. on. (2013) 114–121
14. Di Rocco, J., Iovino, L., Pierantonio, A.: Bridging State-based Differencing and Co-evolution. In: Procs.of the 6th Int. Workshop on Models and Evolution. ME '12, New York, NY, USA, ACM (2012) 15–20
15. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology **6** (2007) 165–185
16. Rose, L., Kolovos, D., Paige, R., Polack, F.: Model Migration with Epsilon Flock. In: Proc. ICMT. Volume 6142 of LNCS., Springer (2010) 184–198
17. Wagelaar, D., Iovino, L., Ruscio, D., Pierantonio, A.: Translational Semantics of a Co-evolution Specific Language with the EMF Transformation Virtual Machine. In: Theory and Practice of Model Transformations. Volume 7307 of LNCS. Springer (2012) 192–207
18. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. 2 edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
19. Hassam, K., Sadou, S., Gloahec, V.L., Fleurquin, R.: Assistance system for OCL constraints adaptation during metamodel evolution. In: Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, IEEE (2011) 151–160
20. Vermolen, S., Visser, E.: Heterogeneous Coupled Evolution of Software Languages. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Vlter, M., eds.: Model Driven Engineering Languages and Systems. Volume 5301 of LNCS. Springer (2008) 630–644
21. Guana, V., Stroulia, E.: ChainTracker, a Model-Transformation Trace Analysis Tool for Code-Generation Environments. In Di Ruscio, D., Varr, D., eds.: Theory and Practice of Model Transformations. Volume 8568 of LNCS. Springer International Publishing (2014) 146–153

User-defined Signatures for Source Incremental Model-to-text Transformation

Babajide Ogunyomi, Louis M. Rose, and Dimitrios S. Kolovos

Department of Computer Science, University of York
Deramore Lane, Heslington, York, YO10 5GH, UK.
[bjo500, louis.rose, dimitris.kolovos]@york.ac.uk

Abstract. Model-to-text (M2T) transformation is an important part of model driven engineering, as it is used to generate a variety of textual artefacts from models, such as build scripts, configuration files, documentation and code. Despite the importance of M2T transformation, building M2T transformations that scale with the size of the input model(s) remains challenging because most contemporary M2T transformation languages do not provide adequate support for incremental transformations. We have previously proposed the use of automatic signatures, as a technique for source incremental transformations. In this paper, we introduce user-defined signatures, which outperform automatic signatures. We perform a comparative analysis of user-defined signatures with automatic signatures, and non-incremental transformation by application to an existing M2T transformation.

1 Introduction

Model-to-Text (M2T) transformation is a model management operation that involves generating text (e.g., source code, documentation, configuration files, reports, etc.) from models. As M2T transformations become increasingly popular for generating textual artefacts in software projects, so also is the concern for building scalable M2T transformations [1]. According to Bennett et. al. [2], software evolution is inevitable and it involves activities that are necessary to fulfil the requirements of end-users. However, software evolution incurs costs associated with finding a subset of changed parts of the system model, analysing the impact of the change, implementation of the change, re-validation of the system [3]. For example, re-generation of text files upon making changes to a source model should not take as much time as it took to generate the text files in the first instance, and the process of re-generation should also be devoid of redundant re-computations, i.e., files that are not affected by changes need not be re-generated.

In our previous work [4], we proposed *signatures* for constructing efficient, scalable M2T transformations. Signatures can be used to detect changes in source model(s) and limit the execution of a transformation to the parts of the transformation that are affected by the change(s). Signatures must be derived from the transformation that is to be executed incrementally, and we have previously

proposed automation for deriving signatures (which we now term *automatic signatures* and discuss further in Section 3.3). In this paper, we reiterate and then address the shortcomings of automatic signatures via *user-defined signatures*. User-defined signatures have the further advantage of being more efficient than automatic signatures (Section 5).

2 Related Work

Model transformation has been described as the heart and soul of MDE [5]. Although little work has been done on incremental M2T transformation, there have been a few published techniques on incremental model-to-model (M2M) transformation. For example, Hearnden et. al. in [6] proposes an incremental method which represents the trace of a transformation execution as a tree. The Hearnden approach maintains an entire transformation context throughout all transformation executions, which allows propagation of changes between source and target models by re-executing the transformation on computed model deltas. Other incremental M2M approaches like PMT [7] synchronises models via trace links, which contain information relating to the provenance of target model elements with respect to source model elements. PMT is a rule-based M2M transformation language and the transformation engine uses identifiers to match source model elements to target model elements.

To the best of our knowledge, Xpand[1] is the only contemporary M2T language that supports source incremental transformation. Incremental generation in Xpand is a threefold process: generating trace links; performing model differencing; and analysing the difference model with respect to generated files. The generated trace links specify how source model elements are mapped to generated files. The difference model enables the transformation engine to identify the elements of the model that have changed. Model differencing is achieved in one of two ways: either by listening to changes made by the user in a model editor, or by comparison of the current and previous versions of the input model. Once the difference model is constructed, impact analysis is performed to determine which changed model elements are used in which templates. A template is re-executed if it uses a model element that has changed. The approach to incrementality employed by Xpand cannot utilise additional information about change, which might be known only to the developers. As we shall see in Section 3.3, user-defined signatures can utilise domain-specific information to increase the efficiency of incremental transformation.

3 Background: Incremental M2T

Incrementality in software engineering refers to the process of reacting to changes in an artefact in a manner that minimises the need for redundant re-computations.

[1] `http://eclipse.org/modeling/m2t/?project=xpand`

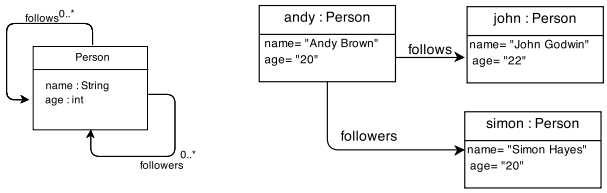
Generally, incremental model transformations reduce the amount of time and computation expended on propagating changes from inputs to outputs.

In the context of M2T transformation, incrementality provides mechanisms for the transformation engine to only re-invoke templates that are affected by changes to the input model. Incrementality in M2T transformation is categorised into 3 types [8]: user-edit preserving, target, and source incrementality. User-edit preserving incrementality prevents loss of user-crafted content by mixing generated text with manually written text. Target incrementality updates already generated targets with output from the current transformation execution after having invoked all templates. Source incrementality, unlike target incrementality, isolates modified model elements and invokes only those templates that are affected by the changes.

Many contemporary M2T transformation languages support user-edit preserving and target incrementality, but not source incrementality. Developing a sound theory of source incrementality is an open research challenge, and is crucial for significantly improving the efficiency of transformations that are complex (i.e., operate on large or densely connected models).

3.1 Running Example

In the following section, we demonstrate the use of signatures in an M2T template language. Figure 1 is a simplified model of a person's contact list on a social media platform (e.g. Twitter), which consists of: persons, the people that a person follow (follows), and that people that follow a person (followers). The follows - followers relationship is asymmetric: following a person does not imply that they must follow you. The model serves as input to the template in Listing 1.1. The generation of a person's follower list or following list represents a simple example of a process that would benefit from incremental generation. In the remaining sections, we demonstrate the use of signatures with our running example.



(a) Metamodel. (b) Example input model.

Fig. 1. Example input model.

3.2 Signature-Based Source Incrementality

Signatures are concise, lightweight proxies for templates that indicate whether or not a change to an input model will affect the output of a template. Signatures can be used to detect changes in input model(s) and limit the execution of a transformation to the templates that are affected by the change(s). Signatures represent a dynamic identifier for a model element in relation to a particular template that consumes data from the model element. The composition of a model element's signature depends on what attributes of the model element are accessed in a template. For example, in Figure 1, a person has *age* and *name* attributes. Consider the template shown in Listing 1.1, that is used to generate a person's followers and followed list from the person model in Figure 1. The template accesses the name attributes of the person, the person's followers and persons followed by the person. Therefore, the signature of *person* with respect to the template can be the values of the properties that are accessed by the template: $person.name$, $person.follows.name$ and $person.followers.name$. Note that the *age* attribute does not form any part of the signature, as this template is not dependent on the value of the any person's *age*. That is to say, a signature is a proxy for a specific template, and captures only those parts of the model that are used by that template.

```
1  [template public personToFile(p : Person)]
2  [file (p.name)/]
3  Person [p.name] has the following followers:
4  [p.followers.name] and follows the following persons:
5  [p.follows.name].
6  [/file]
7  [/template]
```

Listing 1.1. Simple template-based M2T transformation template specified in OMG MOFM2T syntax

We represent a signature as a sequence. Each element in the sequence is either a primitive value, or a further sequence. During the computation of a signature, each model element property value that makes up the composition of a signature is added to the signature sequence. For instance, considering the input model in Figure 1, suppose that a *person*'s signature is calculated from the name of person and the name attribute of person's followers and persons followed by the person. The initial signature for 'andy' using a flat structure will be the sequence: `{"Andy Brown", {"John Godwin"}, {"Simon Hayes"}}`. Note that the second and third elements of the sequence are also sequences because the follows and followers references are multi-valued. At the end of each transformation execution, the signatures (i.e., sequences) are persisted in non-volatile storage, such as a relational database or XML document.

During the first execution of an M2T transformation, a template's signature is evaluated each time the template is invoked, and the resulting signature value

is written to disk along with a unique identifier (typically, the model element's id) for the model elements that produced that signature value. In subsequent executions of the M2T transformation, the previous signature values are used in determining which templates need to be re-executed, and for which model elements. More specifically, a change in the model results in a signature value that differs from the equivalent signature value in the previous execution of a template on that model element. A signature value that has changed indicates that a template must be re-executed on a model element in order to propagate the change (to the generated text).

We have seen how signatures are stored and how they are used, but not how they are computed. The subsequent sections discuss two approaches to styles of signature which differ by the way in which that they are computed: automatic signatures and user-defined signatures.

3.3 Automatic Signatures

Automatic signatures are computed by concatenating the dynamic text-emitting sections of a template. The transformation engine strips a template of its static sections and invokes the template made up of only the dynamic sections, essentially capturing property accesses of model elements specified or expressed in templates. The templates are analysed at runtime. For example, the automatic signature computed by executing template in Listing 1.1 on 'andy' is the sequence: {"Andy Brown",{"John Godwin"},{"Simon Hayes"}}.

Automatic signatures require that a template is invoked at least once to re-compute a signature and compare the newly computed value to the signature value stored during the last successful transformation execution. Through an empirical study in our previous work [4], automatic signatures have been demonstrated to be an effective way of achieving source incremental transformations, leading to significant (30 - 50%) reduction in execution time of re-transformations.

```
1  [template public personToFile(p : Person)]
2  [file (p.name)/]
3  [p.name/]
4  [if (p.followers.isEmpty())]
5  has no followers
6  [else]
7  has some followers
8  [/if]
9  [/file]
10 [/template]
```

Listing 1.2. Example of a template-based M2T transformation, specified in OMG MOFM2T syntax.

Automatic signatures however do not always guarantee the correctness of re-generated text. For example, in Listing 1.2, the only dynamic text-emitting

section in the template emits the name of a *person*. Therefore, the signature of *person* is always *person.name*. However, this signature is not sensitive to all possible changes to *person* that could result in different text being generated, such as the followers reference becoming non-empty. Suppose that the model evolves such that person 'andy' has no followers. The signature of 'andy' will remain constant (`"Andy Brown"`) despite the change to the model, and the obvious need to re-generate the text file. The transformation engine using the automatic signature cannot detect the change, and thus, no template invocation is performed.

Templates that access properties in static sections, such as the one shown in Listing 1.2, tend to result in the computation of signature values that do no always accurately reflect a change in the model that necessitate re-generation of text.

4 User-defined Signatures

User-defined signatures give more control to the developer by allowing them to express the way in which a signature is computed. Ideally, a user-defined signature accesses precisely the same model elements (and precisely the same properties of those model elements) as the template for which the signature is a proxy. The responsibility for ensuring the signatures are representative of the templates rests with the transformation developer. Unlike automatic signatures, user-defined signatures give more control to the developer, and are more lightweight. As such user-defined signatures are heavily reliant on the developer's knowledge of the transformation.

For example, the transformation in Listing 1.2 which the automatic signature finds problematic can be addressed with the user-defined signature shown on line 2 in Listing 1.3. (Note that user-defined signatures necessitate extending the M2T language with an additional language construct). The user-defined signature is computed using the same parts of the model as the template, including whether or not the person has any followers. When the template is executed on person 'andy', the signature evaluates to `{"Andy Brown",false}`, which is a complete reflection of the property accesses made in the template. The signature expression instructs the transformation engine to evaluate the signature from the expression provided by the developer. In this case, the developer is careful to include all model element properties whose change are likely to result in re-execution of the template. The advantage of this approach is that signatures can include parts of the model that are accessed in static sections of templates, as well as those that are accessed in dynamic sections.

4.1 Drawbacks of User-defined Signatures

Despite the effectiveness of user-defined signatures at addressing the drawbacks of automatic signatures, they are not without their own drawbacks. In particular, user-defined signatures are prone to human error. For example, a transformation

```
1  [template public personToFile(p : Person)]
2  [signature : Sequence{p.name,p.followers.isEmpty()} /]
3  [file (p.name)/]
4  [p.name/]
5  [if (p.followers.isEmpty())]
6  has no followers
7  [else]
8  has some followers
9  [/if]
10 [/file]
11 [/template]
```

Listing 1.3. Example of user-defined signature in a template-based M2T transformation, specified in OMG MOFM2T syntax.

author might specify a signature expression that is incomplete. An incomplete signature expression omits at least one property access made in the template, and cannot be relied upon to produce signatures that are correct or that are true reflections of the property accesses made in a template. We address this challenges by applying runtime analysis of templates to provide helpful hints to the developer, which the developer can use to assess the correctness and completeness of their signature expressions.

Additionally, although user-defined signatures may appear to be simple, specifying signature expressions that are complete and correct reflections of model element property accesses in a template can be a onerous task, especially for complex templates involving large models. Furthermore, writing signature expressions for templates that access a large number of model element properties can result in very long lists of attributes in the signature expression, which may be unappealing to the developer and difficult to manage. Addressing this challenge remains as future work, but we anticipate providing built-in operations that make it easier for developers to declaratively express which parts of the input model should be traversed to compute a user-defined signature.

4.2 Runtime analysis for User-defined Signatures

Contemporary M2T languages limit the applicability of static analysis techniques to the languages, because most M2T languages are dynamically typed and support features such as dynamic dispatch [4]. Instead we have applied partial analysis of templates at runtime to determine model element properties accessed in a template. The property accesses made in the template then serve as useful hints to the developer for assessing the correctness and completeness of the specified signature expression composition. Property access hints are particularly useful during initial execution of a transformation, because the first transformation execution is not incremental, but the hints help the developer immediately assess the signature expression, perhaps still with little knowledge of the transformation. Therefore, on subsequent transformation executions, the developer is less concerned about the correctness of the signature expressions, provided the template has not been modified.

In addition to this, property access hints can capture model element property accesses used to control template execution flow. For instance, in the example shown in Listing 1.2, the runtime analysis will capture and suggest to the developer to include 'person.followers' in the signature expression.

5 Experimental Evaluation

To assess the performance benefits of user-defined signatures, and compare their performance with automatic signature generation and non-incremental transformation, we extend our experiment from our previous work [4]. This work used the Pongo[2] M2T transformation, which is implemented in EGL and used to generate data mapper layers for MongoDB relational databases. For this experiment, we generate Java code from the GmfGraph Ecore model obtained from the Subversion repository of the GMF team. GmfGraph Ecore model is a prime candidate for this experiment because it represents a project that has evolved independent of the signature implementation in EGL, which also means that our knowledge of GmfGraph in relation to the Pongo transformation templates is limited. User-defined signature was prototyped by extending the syntax of EGL with support for a user-defined "signature" expression per transformation rule. An example of user-defined signature expression is shown on Line 2 in Listing 1.3.

The results in Table 1 show the difference in the number of template invocations and total execution time between non-incremental transformation, and incremental transformation using automatic and user-defined signatures. Expectedly, due to the initial overhead of computing, processing, and storing signatures, the first execution of the transformation in incremental mode takes longer to execute than in non-incremental mode. However, in subsequent executions of the transformation, the incremental mode out-performs the non-incremental mode: on average execution of the incremental mode requires 66% of the time taken for non-incremental mode when using automatic signatures and 52% when using user-defined signatures. It was also observed that in two instances (versions 1.25 and 1.30 of the input model), the user-defined signatures resulted in more template invocations than the automatic signatures. This suggests that the automatic signatures, in these particular instances were insensitive to changes in the input model, and it also highlights the shortcoming (discussed in Section 3.3) of automatic signatures.

Furthermore, the results of the experiment indicate that signatures, generally, are effective means of achieving source incrementality. For instance, signatures allow the transformation engine to selectively invoke only the templates that are affected by changes to an input model. This was observed in versions 1.31 and 1.32 of the input model, when the transformation did not invoke any template using the signatures (both user-defined and automatic), whereas the non-incremental transformation invoked all the templates in the transformation.

[2] https://code.google.com/p/pongo/

		Non-Incremental		Incremental (Auto)		Incremental (User-def)	
Version	Changes (#)	Inv. (#)	Time (s)	Inv. (#)	Time (s; %)	Inv. (#)	Time (s; %)
1.23	-	72	2.16	72	2.69 (125%)	72	2.17 (100%)
1.24	1	73	1.93	1	1.38 (93%)	1	1.10 (57%)
1.25	1	73	1.89	**2**	1.47 (78%)	**4**	0.95 (50%)
1.26	1	74	2.09	1	**0.73 (35%)**	1	0.79 (38%)
1.27	10	74	1.89	44	1.27 (67%)	44	1.11 (59%)
1.28	10	74	2.16	44	1.63 (75%)	44	1.19 (55%)
1.29	14	74	2.05	14	1.67 (81%)	14	1.00 (49%)
1.30	24	77	2.21	**35**	1.45 (66%)	**37**	1.29 (58%)
1.31	1	77	2.13	0	0.97 (46%)	0	0.90 (42%)
1.32	1	77	2.13	0	0.81 (38%)	0	**0.48 (23%)**
1.33	3	79	1.88	3	0.71 (38%)	3	0.72 (38%)
			22.52		14.78 (66%)		11.70 (52%)

Table 1. Results of using non-incremental and incremental generation for the Pongo M2T transformation, applied to 11 historical versions of the GMFGraph Ecore model. (Inv. refers to invocations)

5.1 Discussion

The results from our experiments indicate that signatures are viable means of providing source incremental M2T transformations. For the Pongo transformation, we observed that the lowest execution time for the automatic signatures was as little as 35% of the execution time in non-incremental mode, and for the user-defined signatures, the execution time was as little as 23% of the execution time in non-incremental mode.

User-defined signatures often execute faster than automatic signatures. Apart from the overhead of storing, retrieving, and comparing signatures, automatic signatures also incur an additional overhead of invoking templates to calculate signatures. On the other hand, user-defined signatures are concise EOL[3] expressions that result in relatively small sized string values, compared to automatic signatures that contain all dynamic sections of a template.

6 Conclusion

We have proposed user-defined signatures, which provide source-based incremental M2T without the downsides of automatic signatures which were the subject of our previous work. We have illustrated, with the aid of an example, that user-defined signatures can effectively handle transformation templates that automatic signatures find problematic. Additionally, through empirical evaluation, we have showed that user-defined signatures are often more efficient than both non-incremental transformations, and incremental transformations that use automatic signatures.

In future work, we will make it easier for developers to specify user-defined signatures by providing built-in operations that traverse a subset of a model and collect appropriate signatures for all of the elements that have been traversed.

[3] http://www.eclipse.org/epsilon/doc/eol/

These operations will relieve the developer of the responsibility of specifying large or complex signatures. Additionally, we will extend our empirical evaluation to include several further M2T transformations that are used in industry, such as those found in the EMF[4] and GMF[5] projects.

Acknowledgements. This work was partially supported by the European Commission, through the Scalable Modelling and Model Management on the Cloud (MONDO) FP7 STREP project (grant #611125).

References

1. Giese, H. and Wagner, R. Incremental Model Synchronization with Triple Graph Grammars. In *MoDELS*, pages 543–557. Springer, 2006.
2. K. Bennett and V. Rajlich. Software Maintenance and Evolution: a Roadmap. In *Proc. of the Conf. on the Future of Software Engineering*, pages 73–87. ACM, 2000.
3. A. Etien and C. Salinesi. Managing Requirements in a Co-evolution Context. In *RE, 2005. Proc.. 13th IEEE Int'l Conf. on*, pages 125–134. IEEE, 2005.
4. B. Ogunyomi et. al. On the Use of Signatures for Incremental Model-to-Text Transformation. In *MoDELS 2014, Valencia, Spain*, LNCS. Springer, to appear.
5. S. Sendall and W. Kozaczynski. Model transformation: The Heart and Soul of Model-Driven Software Development. *Software, IEEE*, 20(5):42–45, 2003.
6. D. Hearnden et al. Incremental model transformation for the evolution of model-driven systems. In *Proc. MoDELS*, LNCS, pages 321–335. Springer, 2006.
7. L. Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–126, 2008.
8. K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[4] http://www.eclipse.org/modeling/emf/?project=emf
[5] http://wiki.eclipse.org/Graphical_Modeling_Framework/Models/GMFGen

Grammar Maturity Model

Vadim Zaytsev

Universiteit van Amsterdam, The Netherlands, vadim@grammarware.net

Abstract. The evolution of a software language (whether modelled by a grammar or a schema or a metamodel) is not limited to development of new versions and dialects. An important dimension of a software language evolution is maturing in the sense of improving the quality of its definition. In this paper, we present a maturity model used within the Grammar Zoo to assess and improve the quality of *grammars in a broad sense* (structural models) and give examples of activities possible for each level.

1 Motivation

Grammar Zoo [41] is an initiative aimed at systematic collection of *grammars in the broad sense* — structural definitions of software languages; annotation of each grammar with information about its source, original format and authors; complementing each grammar with details about how it was obtained; documenting usages of each grammar — its derivatives, tools, documents and other grammars; and finally making all these grammars publicly available in a variety of formats. It has many possible uses:

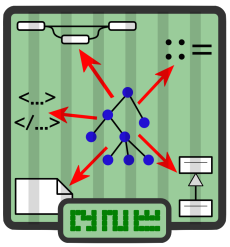


Fig. 1. Grammar Zoo, http://slps.github.io/zoo

Interoperability testing. Suppose that we have identified multiple grammars of the same intended language that correspond to its different frontends. To test their interoperability, one can do code reviews or develop a test suite, but a better, more systematic, way is to generate such a suite and compare or converge those grammars automatically. An approach for that has been proposed in [9] and evaluated by two case studies involving 4 Java grammars and 33 TESCOL grammars, which were extracted from parser specifications and became one of the early fragments of the Grammar Zoo. Its replication with a simplified algorithm applied to adapted grammars, used 28 grammars of different languages from the ANTLR grammar repository [3].

Grammar recovery heuristics. There have been many successful attempts of reusing grammatical knowledge embedded in various software artefacts like parser specifications, data format descriptions or metamodels. Some focused on idiosyncratic properties of the source notation, others tried to generalise the relaxed ways of treating the baseline artefact with heuristic rules for splitting/combining names, matching parentheses, etc [39]. The more grammars can be recovered with heuristics, the better validated and motivated they become.

Empirical grammar analysis. Grammar metrics is a mature field of research, but more elaborate characterisations such as "top" or "bottom" nonterminals are common in grammar-based papers. When the repository of various grammars has grown to a reasonable size and comprised over 500 grammars, the micropattern mining methodology [11] was applied to infer characterisations by mining this repository [40]. They can in turn be used for clustering grammars based purely on statistical data about sets of indicators and for triggering grammar mutations [3].

Sample-based grammar engineering. Crafting a good grammar suitable for the intended task is not an easy activity: is must not be too restrictive or too permissive, must be compatible with the intended technology, reasonable in terms of performance, readable for a human expert, machine processable, etc (some of the aspects obviously being more relevant under different circumstances). Having access to a large corpus of existing successful grammars (together with the information about their actual applications, of course), could aid the grammar engineer either implicitly (essentially by "code reading") or explicitly (by reusing an existing grammar as a starting point for developing a domain-specific language).

Grammar components. There is ongoing work on identifying *semantic components* of software languages that correspond to concepts like loops, variables, exception throwing, etc [16]. By using a combination of program slicing and clone detection techniques on a large enough corpus, we can identify *syntactic components* of software languages and investigate whether there is any correspondence with semantic components.

Since the first days of the Grammar Zoo it became painfully clear that there is no general agreement about maturity of grammars: even a grammar in a narrow sense (say, a parser specification) can be optimised with one parsing technology in mind while rendering it useless for other technologies, and for grammars in a broad sense the meaning behind measurements becomes even more obscure. In early publications around the project there was also mentioning of the Grammar Tank, a sibling project collecting smaller grammars of DSLs — it seemed unreasonable to put a tiny grammar extracted from a ten-line Haskell ADT definition right next to a "real" grammar defining a programming language like C# or C++, extracted from a 1000+ page manual. Later we imported grammars from the Atlantic Metamodel Zoo, from the Relax NG collection and many other places, adding to the complexity and heterogeneity. In the end, all grammars have been merged into one collection, each annotated with all kinds of metadata. One of the important annotations is a maturity level, which we define essentially as the level of details and consistency and a measure of how close the grammar is to actual use, notwithstanding its domain and intent.

The next session introduces the usual tools available for grammar engineering activities. §3 actually presents the levels of the maturity model. In §4, we give an example of how a grammar can be used in practice, referring to its possible maturity level. §5 discusses related work and §6 concludes the paper.

2 Grammar Manipulation

The arsenal of grammar manipulation is rather large, and for the sake of better understanding of the rest of the paper, we name a few important methods. All of them are to some extent represented in the GrammarLab[1] library and useful for programming in the Rascal metaprogramming language [21].

XBGF [25] is an operator suite for programmable grammar transformations. It consists of more than 50 operators like ***renameN*** for renaming a nonterminal or ***unfold*** for unfolding a nonterminal reference to its definition. Operators have universally defined semantics and known properties — some can be proven to be language-preserving refactorings, others extend the defined language or restrict it. Transformations are programmed against this operator suite by choosing an operator and providing appropriate arguments for it to make it applicable to the grammar being transformed — i.e., ***unfold***(n) is a valid transformation step, if n is a nonterminal present in the grammar. Similar techniques exist for other grammar manipulation frameworks [6,22,23].

SLEIR [42] is a similar suite for grammar mutations [38], which are generalised transformations automatically applicable on a large scale: enforcing a normal form, changing a naming convention, systematically refactoring harmful constructions into equivalent harmless ones are examples of grammar mutations.

Grammar analysis is a research direction on its own, where based on observable facts obtained from grammar metrics [31] or micropatterns [40] we can make estimations and draw conclusions about a grammar's suitability for specific tasks, compatibility with a technology, backwards compatibility through versions, interoperability among various tools, etc.

The last mentioned group of instruments was important for creating the Grammar Zoo and growing it, and less so for using it as such. Grammar extraction obtains grammars in a broad sense from software artefacts containing grammatical knowledge (source code, documentation, protocol definitions, algebraic data types, databases, etc). Grammar recovery [24,34,39] works similarly with sources using deceivingly familiar notations in an inconsistent or otherwise unexpected way — manually written, out of date, incomplete, etc.

3 A Quality/Maturity Model for Grammars

We distinguish among the following grammar levels:

- A grammar is **fetched** if it can be put in a file which we claim to contain grammatical knowledge. A fetched grammar is usually written in an (E)BNF-like notation, but it can also be an XML Schema schema, an Ecore model, a parser specification, etc. A grammar from an undisclosed ISO standard or a grammar built in a proprietary tool is *not* fetched, since we have no possible way to extract the knowledge from it. A compiler is therefore not a fetched grammar since the grammatical knowledge is ingrained too deep

[1] GrammarLab: `http://grammarware.github.io/lab`.

in it and requires special techniques to be fetched. The sources of a compiler, however, can be considered fetched, since further extraction can be semi-automated, and the result will depend mostly on the source and not on the extraction algorithm. Hence, a corpus of programs in a given language is also *not* a fetched grammar, but an APTA (Augmented Prefix Tree Acceptor) or a DFA (Deterministic Finite Automaton) constructed from it by a grammatical inference algorithm, is a fetched grammar. A fetched grammar can contain unreadable symbols, incorrect indentation, parts written in an unknown notation or a natural language, or even be present in a form of an image or a manuscript.

- A grammar is **extracted**, if it was fetched and then successfully processed by a (hopefully automatic) grammar extractor, possibly also corrected of typographical, text recognition and similar errors and converted into a context-free grammar or, more broadly speaking, to a Boolean grammar [29]. (Not venturing beyond context freedom is simply a consequence of the current lack of theoretical foundations for linking classic context-dependent grammars to generalised types — in general, aligning the Chomsky hierarchy [4] with Barendregt λ-cube [2]). An extracted grammar is suitable for automated processing: it can be pretty-printed in a range of different ways and transformed by general means, without writing a tool specific for its peculiar notation or format: syntax diagrams, Relax NG schemata, algebraic data types, parser specifications all lose their notational differences when they are being extracted, but they retain all structural peculiarities such as using a particular style of recursion (syntax diagrams are incapable of expression left recursion, and some parsing algorithms tend to avoid it as well), the lack or presence of terminal symbols (anything that defines an abstract syntax, has no terminals) or nonterminal symbols (classic regular expressions [20] have no notion of named subcomponents), etc.
- A grammar is **connected**, if it was extracted and then processed to not contain unwanted top sorts (defined but never used) and bottom sorts (used but not defined). These two quality indicators were proposed in [33,34] and discussed in more detail in [23] before being formalised as micropatterns [40]. Connecting a grammar can be done automatically with a mutation [38,42], semi-automatically with a sequence of transformation steps, or by editing it inline. Connecting is a simple procedure that allows to start making some claims about the grammar, since it enables its formalisation (the classic $\langle \mathcal{N}, \mathcal{T}, \mathcal{P}, s \rangle$ model of a grammar requires it to have one known starting symbol) and possible application of grammar-based algorithms (in particular grammar-based test data generation expects the grammar to be connected because otherwise it is futile to use any coverage criteria). In a broader sense, a connected grammar always relies on some underlying mechanism of testing or validation which ensures its general quality — as opposed to the extracted grammar which can be an output of an automated extractor and never checked nor inspected further.
- A grammar is **adapted**, if it is connected and then transformed towards satisfying some constraints: it could be complemented with a lexical part,

or its naming convention can be adjusted, or certain metaconstructs can be introduced to or removed from its syntax. The adaptation has a clear intent: adding a lexical part can lead to automated generation of a parser or at least a recogniser; conforming to a naming convention can enable the use of the grammar in specific language workbenches, etc.

- A grammar is **exported**, if it was adapted and then a piece of grammarware was generated from it. An exported grammar bidirectionally and possibly nontrivially corresponds to a real piece of grammarware such as a compiler or a code analysis or transformation tool.

Each Grammar Zoo entry has one *fetched* grammar: ones with less than one are "non-entries" that can perhaps be referred to, but can under no circumstances be machine processed; having more than one fetched grammar can happen for cases such as multiple websites mirroring one another, because an additional check is required to assert them to be equal. If several extractors are available (e.g., one straightforward one and one heuristic-based error-correcting one), there can be multiple *extracted* grammars per entry. Similarly, there can be several grammars of level *connected* and up per entry, varying per their extracted source and methods of processing. Especially different *adapted* grammars per entry are common, since each of them corresponds to a specific intentional adaptation project. At this point in the history of the Grammar Zoo we have not yet experienced the need to explicitly distinguish the reason for adaptation of each grammar: some are massaged for better readability, some adjusted with parsing in mind, some are disambiguated [35], some adapted for testing purposes [3,9,32], etc. We intentionally leave the hierarchy as general as it is, and leave its extension to future work.

4 A Maturation Path

Suppose we would like to have a piece of grammarware to parse and analyse programs in a particular software language — say, COBOL or PHP. Being constrained in time, we usually start by looking for existing grammars: once we find one that seems reasonably suitable for our needs, we can declare it *fetched*. If a fetched grammar of our intended language is already in the Grammar Zoo, it can save us the search, the frustration from websites having been taken down, as well as the ambiguity about the true source of the grammar.

In the simplest cases, grammar extraction methods and tools can be applied to a fetched grammar with reasonable success. There is quite a collection of them readily available within GrammarLab, and it is fairly straightforward to use notation-parametric grammar extraction [39], if the input notation is anything like BNF or EBNF; write out XSLT mapping templates, if the input notation is XMI, XSD or anything from the XMLware technical space. If all available methods fail, we can attempt to apply grammar recovery tools, which have heuristics known to overcome frequent erroneous patterns. Once some reasonable kind of non-empty grammar is obtained or if it was in the Grammar Zoo to begin with, the grammar can be considered to be *extracted*.

An extracted grammar can be processed further, analysed, transformed, exported, imported, visualised etc — there are many tools in the GrammarLab that can do it directly, and they can also help to export it to a format readily consumable by other metagrammarware. However, it does not mean that this grammar would "work" there, whatever that might mean. There are some sensible metrics, constraints and grammar analyses established in state of the art grammar recovery [24,34], that are almost universally useful in improving the quality of a grammar. For instance, we would like to identify the starting symbol of a grammar, establish it being unique. Furthermore, for each parts unreachable from it, we would like to make a decision and either remove them or connect to the rest of the grammar. This is usually done by programming the corresponding steps in XBGF [25], SLEIR [42], GDK [22], TXL [6] or any other grammar manipulation language. This usually implies manual examination of a grammar and its metrics by an expert, making the appropriate decisions and then documenting the changes. Once this is completed, we speak of having a *connected* grammar.

The next step is grammar adaptation [23]: a goal-specific continuation of grammar transformation activities. For example, if we have decided to parse and analyse code in COBOL or PHP, this is our goal, and in case of Rascal [21] it will mean having a complete concrete syntax specification and a suitable algebraic data type. Both can be obtained from a connected grammar, but the adaptation strategies are different. For a syntax specification, we need to add the lexical part, specify layout, increasingly disambiguate the grammar, etc. For a data type, we should think of its suitability for specifying our analyses later, and we can easily eliminate all terminal symbols and massage the remaining abstract grammar to enable more concise and readable patterns. These streaks of activity end up with an *adapted* grammar each.

Finally, our two grammars (or a syntax specification and a data type, or a grammar and a schema — terminology may vary) are ready to be *exported* — we do this with out of the box renderers, possibly followed by manual polishing such as adding documenting annotations and inserting copyright notices. It is not unusual for an exported grammar to be linked to a specific tool which it forms a part of.

5 Related Work

Lämmel and Verhoef [24] were the first to propose the notion of a *grammar level*[2] to specify a quality level or a recovery status of a grammar. We have conceptually inherited that hierarchy and extended it to accommodate more important details. Their level 1 broadly corresponds to an extracted grammar in our model, level 2 to connected, levels 3 and 4 (depending on how thorough it has been tested) to adapted. An exported grammar is at level 5 if it either demonstrates the absence of manual steps in grammar deployment, or documents them by its

[2] These "grammar levels" are essentially CMM-like levels applied to grammars, unrelated to well-known "grammatical levels" used for a range of grammar metrics [18,31].

existence. Our contribution lies in rethinking and generalising this hierarchy for all grammars in a broad sense and empirically applying it to hundreds of grammars (as opposed to CFGs of one or two programming languages).

El-Attar and Miller [7] have shown how antipattern detection can be used to improve the quality of models (in their case, use case models, but a similar technique for metamodels is not unthinkable). This approach is conceptually very similar to a typical grammar engineering activity when the language engineer identifies which metaconstructs are incompatible with the technology indended for use, and refactors them away. At the current state of software language engineering, the first part is the most appropriate to do with micropatterns [40] and the second part with grammar mutations [42].

France and Rumpe [10] have investigated the relation between quality and MDE from the pragmatic point of view and found out that one of the biggest advantages comes from the opportunity to reuse previously assessed models (and submodels) of known quality in the development of new ones. The same argumentation, lifted to the metalevel, can be found in the first section of this paper when we show some possible uses for the Grammar Zoo.

It is impossible to talk about quality without mentioning ISO/IEC 9126 [15], an official standard specifying quality of a software system as a product (hence, also a grammarware system). It identifies quality characteristics such as functionality, reliability, usability, efficiency, maintainability, portability, and continues to break them down into smaller pieces. There have been some attempts to formalise and detail parts of it up the the point of implementability — in particular, maintainability has received a lot of attention [13], but the general agreement is to treat the standard as a set of guidelines and considerations, not as an immediately implementable model. An extensive review of research activities concerning the quality models in the particular context of model-driven software development, was made about papers in top conferences in 2000–2009 [28].

Welker's maintainability index in the Coleman-Oman model is often claimed useful for quickly assessing the maintainability (and hence also quality, per ISO 9126 [15]) of software. Its formula exists in various slightly adjusted incarnations in the academic literature and is commonly denoted as either just "maintainability" [1, p.155][5, p.46] or "maintainability index" / "MI" [12, p.255][26, p.15]. As it turns out, MI is inappropriate for grammarware purposes. Suppose we apply aggressive normalisation and unchain all chain productions and inline all nonterminal symbols that are used only once and have only one production rule. Such a transformation preserves stability of the grammar, but obviously reduces its analysability, changeability and testability. Yet MI shows improvement. Since stability, analysability, changeability and testability are the main components of maintainability per ISO 9126 [15], MI does not adequately measure maintainability.

Discussions on language quality are abundant in the context of general purpose programming languages [36,14,37], modelling languages [30,19] and domain-specific languages [27,17]. Their contributions are mostly in a form of sets of guidelines that are with some evidence and expertise linked to the quality of the

final product. Our maturity model can be seen as an attempt to formalise and standardise a part of software language quality — namely, its structural model. There is some strong evidence that this syntactic aspect is not dominating when considering software language quality in general [8].

6 Conclusion

We have briefly introduced the field of grammarware manipulation and a project of collecting grammars in a broad sense — structural definitions of structure found in software systems. We have also presented a maturity model of several distinct levels on which grammars can reside — the model gradually came into existence during several years of research on grammar analysis and improvement. The renovated Grammar Zoo with this new maturity model is about to be deployed and made available for public use. This model was essential in its growth from a dozen self-made grammars to over 1500 entries of fetched level and above.

There are many reasons for models to evolve: some are externally motivated and concern the natural evolution (improvement as a response to contextual changes), some concern the actual use of the models. In this particular paper we have treated quality as basically the level of details in a model extracted from its real-life counterpart, which was directly linked to the possibility of automated processing. This made sense in our context — collecting and analysing grammars in a broad sense — but it stands to reason that the same considerations would apply for any fact extraction models and software models in general.

There is some evidence in adjacent fields that models which evolved in one dimension (e.g., a programming language grammar extracted from a book, cleaned up, polished and turned into a validation tool) can be very profitably (re)used for improving or constructing models that evolved in another dimension (e.g., a programming language grammar from the next edition of the book). For us, proving this or even collecting substantial evidence by convincing case studies, remains future work.

References

1. D. Ash, J. Alderete, P. W. Oman, and B. Lowther. Using Software Maintainability Models to Track Code Health. In *Proceedings of the International Conference on Software Maintenance (ICSM' 94)*, pages 154–160. IEEE Computer Society, 1994.
2. H. P. Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991.
3. E. Butrus. Satisfying Coverage Criteria by Grammar Mutations and Purdom's Sentence Generator. Master's thesis, Universiteit van Amsterdam, Aug. 2014.
4. N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167, 1959.
5. D. Coleman, D. Ash, B. Lowther, and P. Oman. Using Metrics to Evaluate Software System Maintainability. *Computer*, 27:44–49, 1994.

6. T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. Grammar Programming in TXL. In *Proceedings of the Second IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2002)*, pages 93–102. IEEE, 2002.
7. M. El-Attar and J. Miller. Improving the Quality of Use Case Models Using Antipatterns. *Software and System Modeling*, 9(2):141–160, 2010.
8. M. Erwig and E. Walkingshaw. Semantics First! In *Proceedings of the Fourth International Conference on Software Language Engineering*, SLE'11, pages 243–262. Springer, 2012.
9. B. Fischer, R. Lämmel, and V. Zaytsev. Comparison of Context-free Grammars Based on Parsing Generated Test Data. In U. Aßmann and A. Sloane, editors, *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *LNCS*, pages 324–343. Springer, 2012.
10. R. B. France and B. Rumpe. Modeling to Improve Quality or Efficiency? An Automotive Domain Perspective. *Software and System Modeling*, 11(3):303–304, 2012.
11. J. Gil and I. Maman. Micro Patterns in Java Code. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'05)*, pages 97–116. ACM, 2005.
12. J. H. Hayes, S. C. Patel, and L. Zhao. A Metrics-Based Software Maintenance Effort Model. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 254–258. IEEE Computer Society, 2004.
13. I. Heitlager, T. Kuipers, and J. Visser. A Practical Model for Measuring Maintainability. In *Proceedings of the Sixth International Conference on Quality of Information and communications Technology (QUATIC'07)*, pages 30–39. IEEE, 2007.
14. C. A. R. Hoare. Hints on Programming Language Design. Technical report, Stanford University, Stanford, CA, USA, 1973.
15. ISO/IEC. *ISO/IEC 9126. Software Engineering — Product Quality.* ISO/IEC, 2001.
16. A. Johnstone, P. D. Mosses, and E. Scott. An Agile Approach to Language Modelling and Development. *Innovations in Systems and Software Engineering*, 6(1-2):145–153, 2010.
17. G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schneider, and S. Völkel. Design Guidelines for Domain Specific Languages. In M. Rossi, J. Sprinkle, J. Gray, and J.-P. Tolvanen, editors, *Proceedings of the Ninth OOPSLA Workshop on Domain-Specific Modeling (DSM 2009)*, pages 7–13, Mar. 2009.
18. A. Kelemenová. Grammatical Levels of the Position Restricted Grammars. In *Proceedings on Mathematical Foundations of Computer Science*, pages 347–359. Springer, 1981.
19. S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, July 2009.
20. S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, pages 3–42, 1956.
21. P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2009)*, volume 6491 of *LNCS*, pages 222–289. Springer, Jan. 2011.

22. J. Kort, R. Lämmel, and C. Verhoef. The Grammar Deployment Kit. System Demonstration. *Electronic Notes in Theoretical Computer Science*, 65(3):117–123, 2002. Workshop on Language Descriptions, Tools and Applications (LDTA).
23. R. Lämmel. Grammar Adaptation. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*, pages 550–570. Springer, 2001.
24. R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, Dec. 2001.
25. R. Lämmel and V. Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ)*, 19(2):333–378, Mar. 2011.
26. A. Liso. Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model. *Software Engineering Technology*, pages 15–17, August 2001.
27. M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
28. P. Mohagheghi, V. Dehlen, and T. Neple. Definitions and Approaches to Model Quality in Model-based Software Development - A Review of Literature. *Information and Software Technology*, 51(12):1646–1669, Dec. 2009.
29. A. Okhotin. Boolean Grammars. *Information and Computation*, 194(1):19–48, 2004. http://users.utu.fi/aleokh/papers/boolean_grammars_ic.pdf.
30. R. F. Paige, J. S. Ostroff, and P. J. Brooke. Principles for Modeling Language Design. *Information and Software Technology*, 42(10):665–675, 2000.
31. J. F. Power and B. A. Malloy. A Metrics Suite for Grammar-based Software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:405–426, Nov. 2004.
32. P. Purdom. A Sentence Generator for Testing Parsers. *BIT*, 12(3):366–375, 1972.
33. A. Sellink and C. Verhoef. Generation of Software Renovation Factories from Compilers. In *Proceedings of 15th International Conference on Software Maintenance (ICSM 1999)*, pages 245–255, 1999.
34. M. P. A. Sellink and C. Verhoef. Development, Assessment, and Reengineering of Language Descriptions. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pages 151–160, Mar. 2000.
35. M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In N. Horspool, editor, *Compiler Construction 2002 (CC 2002)*, pages 143–158, 2002.
36. A. van Wijngaarden. Orthogonal Design and Description of a Formal Language. MR 76, SMC, 1965.
37. N. Wirth. On the Design of Programming Languages. In *IFIP Congress*, pages 386–393, 1974.
38. V. Zaytsev. Language Evolution, Metasyntactically. *Electronic Communications of the EASST; Bidirectional Transformations*, 49, 2012.
39. V. Zaytsev. Notation-Parametric Grammar Recovery. In A. Sloane and S. Andova, editors, *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA 2012)*. ACM Digital Library, 2012.
40. V. Zaytsev. Micropatterns in Grammars. In M. Erwig, R. F. Paige, and E. V. Wyk, editors, *Proceedings of the Sixth International Conference on Software Language Engineering (SLE 2013)*, volume 8225 of *LNCS*, pages 117–136. Springer, 2013.
41. V. Zaytsev. Grammar Zoo: A Repository of Experimental Grammarware. *Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5)*, 2014. In print.
42. V. Zaytsev. Software Language Engineering by Intentional Rewriting. *Electronic Communications of the EASST; Software Quality and Maintainability*, 65, 2014.

Towards a Model-Driven Dynamic Architecture Reconfiguration Process for Cloud Services Integration

Miguel Zuñiga-Prieto, Javier Gonzalez-Huerta, Silvia Abrahao, Emilio Insfran

ISSI Research Group, Department of Information Systems and Computation
Universität Politècnica de Valencia
Camino de Vera, s/n, 46022, Valencia, Spain
{mzuniga, jagonzalez, sabrahao, einsfran}@dsic.upv.es

Abstract. Cloud computing is a paradigm that is transforming the computing industry and is receiving more attention from the research community. The incremental deployment of cloud services is particularly important in agile development of cloud services, where successive cloud service increments must be integrated into existing cloud service architectures. This requires dynamic reconfiguration of software architectures, especially in cloud environments where services cannot be stopped in order to apply reconfiguration changes. This paper presents a model-driven dynamic architecture reconfiguration process to support the integration of cloud services. Models are used to represent high-level architecture reconfiguration operations as well as adaptation patterns. Adaptation patterns allow us to describe reconfiguration operations independently of a specific cloud platform technology. On the other hand, model transformations are used: i) to support compatibility checking of increments; ii) to generate software adaptors that solve incompatibilities between architectures; and iii) to generate reconfiguration plans specific of cloud provider, that include reconfiguration actions to be applied on cloud service instances at runtime. The proposed process is illustrated with a dealer network system development example, where cloud services are deployed in an incremental way.

Keywords: Model Driven Development, Model Transformations, Cloud Computing, Dynamic Reconfiguration, Model Based Evolution

1 Introduction

Cloud computing is a software engineering paradigm that has the potential of change large part of the IT industry; becoming a research topic with innovative proposals to design, develop and deploy cloud-based systems [1]. Cloud applications are delivered as services over the Internet. Among distinguishable characteristics of cloud computing parading are measured service and rapid elasticity and scalability [2]. The former allows billing based on real usage of resources. The later allows acquiring more resources during a peak of demand and releasing them once they are no longer required. In addition, services can be redeployed on different provider-specific platforms depending on Quality of Service (QoS), Service Level Agreement (*SLA*) or other business criterion.

Service-oriented architecture approach is a way of designing, developing and deploying loosely coupled distributed applications using coarse-grained services. Developing service-oriented applications (such as cloud services) facilitates reconfiguration of software architectures at runtime, what is known as dynamic architecture reconfiguration. Organizations that adopt this approach will be able to i) manage business evolution and/or upgraded services can be introduced with minimum impact on existing systems, and ii) implement loosely-coupled integration approaches [3]. As stated before, cloud services could be deployed in different provider-specific platforms; which often leads to tight coupling of developed cloud services to a specific cloud provider technology. In order to avoid the dependence on cloud providers, the cloud service architectural design must facilitate the use of different environments for execution [4].

Model-Driven Development (MDD) is an approach for developing software systems that promotes a new form of building systems based on the construction and maintenance of models at different levels of abstraction to drive the development process. In this approach, a software system is developed by refining models and it is implemented through model to text transformations.

Software adaptation patterns represent generic and repeatable solutions to manage change in recurring architectural adaptation problems, and prescribe the steps needed to dynamically adapt a software system at runtime from one configuration to another [5]. The use of adaptation patterns is a trend to support reuse in evolution for dynamic adaptive software architecture [6]. Adaptation of software architectures is not only supported by change management proposals, but also by proposals for solving the problems that arise when the interacting entities do not match properly. Software adaptation promotes generation of software adaptors to bridge incompatibilities among services (e.g., different names of methods and services, different message ordering, etc.) in an nonintrusive way [7,8, 9]. Generation techniques for software adaptors are beginning to be used in cloud environments [10].

Cloud applications integrate and compose different cloud services. The cloud services to be integrated may come from the delivering of a software increment in an incremental development approach, or just may be product of maintenance/evolution phases. The integration/update of increments may trigger the dynamic reconfiguration of the existing cloud service architecture. Dynamic reconfiguration creates and destroys architectural elements instances at runtime; being particularly important for cloud services be able to manage instances in different cloud platforms and continue working while reconfiguration takes place. However almost no or little attention has been paid on supporting this reconfiguration at runtime, and only in recent years software engineering research started focusing on these issues [11]. In addition, as far as we know, the incremental and dynamic deployment of cloud services into existing services in the cloud has not been studied yet.

In this paper, we introduce a process to support the dynamic reconfiguration of cloud service architectures due to the integration of software increments. This process will allow software developers to specify how the integration of the architecture of a software increment affects the current cloud service architecture. Additionally, after applying model-driven techniques, software developers will obtain the software artifacts needed to dynamically reconfigure the current cloud service architecture. We define the

architecture of a software increment as a portion of an architecture that corresponds to the architectural description of the increment whose integration into current architecture triggers the current architecture reconfiguration.

The remainder of the paper is structured as follows. Section II discusses related work on proposals to support the dynamic architecture reconfiguration. Section III presents our dynamic architectural reconfiguration approach. Finally, Section IV presents our conclusions and future work.

2 Related work

Software evolution based on reconfiguration of software architectures is an active area of research; however, there are gaps that still need to be covered. Some of these gaps where identified in a systematic literature review performed by Jamshidi et al. [6]. The authors took into account the stage of the software lifecycle where evolution mechanisms were active; findings showed lack of support during the integration and provisioning stage, but also during deployment stage. In our work, we give support to the dynamic reconfiguration of software architectures at the deployment stage of the software life cycle.

In this section, we analyze how researchers and practitioners address the dynamic reconfiguration of software architectures to support the development of cloud/service applications. The most relevant works [11, 12, 13] we have found are analyzed below.

Baresi et al.[12] propose a methodology for deriving service-oriented architectures from high-level business-oriented architecture descriptions. They use formal representations to describe both application specific types as well as runtime configurations of concrete instances. They also, use graph transformations rules and define refinement relation from a generic style of component-based systems to the SOA style.

MOdel-based SElf-adaptation of SOA systems (MOSES) [13], proposes a methodology aimed at driving the self-adaptation of a SOA system to fulfill non-functional QoS requirements. This framework uses linear programing to formulate the identification of the most suitable adaptation according to the detected changes in the environment.

Self-architecting Software Systems (SASSY) [14] uses the application requirements captured by domain experts to derive automatically a base software architecture. Then, SASSY derives an optimized architecture from the base architecture by selecting the most suitable service providers and by applying QoS architectural patterns. In addition, for each QoS architectural pattern, they apply adaptation patterns that specify how the system self-adapts to incorporate the pattern into the configuration. Unlike previous cited approaches, SASSY deploys the coordination logic.

All the works described above i) take into account structural and behavioral aspects for reconfiguration; ii) use SLA or QoS negotiation to discover and select the most suitable service implementation (instance); and iii) apply dynamic binding for reconfiguration. This means that reconfiguration improves non-functional qualities through perfective changes. However, adaptive changes (e.g., software increments due to new functionalities) that require architecture reconfiguration are not taken into account.

They abstract models of business requirements or derive high level architectures ; however, they do not take into account the importance of architectural aspects in agile/incremental development processes [15]. Despite the fact that cited approaches propose consistency or compatibility checking task, they do not provide solutions to support the deployment in different cloud platforms.

In summary, as far as we know, there is a lack of support to incremental and dynamic deployment of cloud services into existing cloud service architecture. Our approach allows incremental reconfiguration of software architectures, and promotes compatibility between the architecture of software increments with the existing cloud architecture, according to cloud specific deployment platform.

3 A Process for Dynamic Architecture Reconfiguration of Cloud Services

In this section, we introduce our motivation example and continue with the reconfiguration process description.

3.1 Motivating Example

The proposed motivation example is based on "*Acme Manufacturing*" dealer network system scenario [16]. *Acme* is a manufacturer company, which wants to improve service to its dealers and partners. With this purpose in mind, Acme considers building and deploying cloud services in an incremental way. The first increment aims to do a better job of fulfilling dealers' orders and provides cloud services for dealers to place and manage their orders. This allows a direct interaction between customer's I.T. systems and Acme's systems. Acme also needs to improve its shipping process to increase delivery speed, and thus, in a second increment provides its local transport partner with cloud services. The transport partner uses cloud services to retrieve orders that need to be shipped as well as to inform dealers about shipping status. This second increment also updates the cloud services deployed in the first increment, providing dealers with information about last bought of items included in the order. Finally, after the third increment Acme needs to be able to manage international deliveries. However, since the international partner has its own custom systems based on exposed web services *Acme* uses the partner's web service to make shipping requests.

3.2 Reconfiguration Process

The model-driven Dynamic Incremental Architectural Reconfiguration (*DIARy*) process has been defined using model-driven and adaptation techniques. This process aims to support software developers during the deployment phase, on activities related to integration of software increments into existing services in the cloud. We support the integration process from an architectural point of view. *DIARy* proposes activities to support the management of dynamic reconfiguration of existing cloud services architectures, produced due to the integration of architectural elements. The main activities

of *DIARy* process are: i) *Specify Increments*; ii) *Check Increment Compatibility*; and iii) *Reconfigure Architecture*. Fig. 1 shows the activities of the *DIARy* process.

Specify Increments. *DIARy* may be incorporated into existing development processes and this activity serves as a glue that allows its incorporation. *Software Architects* perform this activity not only to specify the architecture that corresponds to the architectural description of the increment to be deployed, but also the impact that the integration of the increment has on current architecture. The latter is specified describing how the elements of the *architecture of the software increment* collaborate to reconfigure the current architecture in order to provide the required functionality. This activity generates as output the Increment's Architecture Model; uses as inputs artifacts the Current Architecture, *Design Artifacts*, and *SLA*s. Additionally, uses Increment Description Guidelines as guide and an architecture description language to describe both the current architecture as the *architecture of the software increment*. Each of these artifacts is explained below:

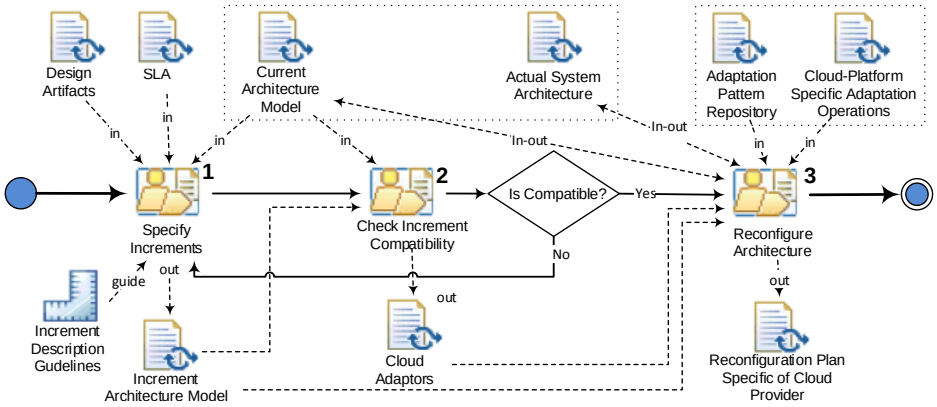


Fig. 1. Overview of the *DIARy* approach

1. *Current Architecture Model (CurAM):* This model allows representing the current architecture (i.e., before increment deployment) using design artifacts. *CurAM* includes information about services, connectors, configuration as well as cloud software architecture related information. *CurAM* evolves after each increment integration; however, in this activity it is used only as input, helping *Software Architects* to identify elements of the current architecture to be affected by the integration.
2. *Design Artifacts:* This input artifact represents design artifacts generated during the development process. Depending on the development process to which *DIARy* is applied, this artifact could be i) the original system architecture designed during the development process; ii) any form of architecture description that describes the increment; iii) Architectural backlogs generated during an agile development process. This artifact helps *Software Architects* to identify the elements of the current architecture that will be affected by the integration.

3. *Service Level Agreements (SLA):* This artifact contains the conditions and parameters that compromise the service provider to meet certain levels of quality. *Software Architects* use it to take design decision during specification.
4. *Increment's Architecture Model (IAM)*: Software Architects participate in generating this output artifact**.** *IAM* allows representing the *architecture of the software increment* and includes information about services, connectors, and configuration. Furthermore, *IAM* allows represent how the elements of the *architecture of the software increment* collaborate to reconfigure the current architecture. To do this, *IAM* includes references to *CurAM* elements. *Software Architects* use references to point out the elements of current architecture affected by the increment (elements added, updated or deleted) as well as the elements used as Integration Points (*IP*). We call *IP* to the interfaces of the current architecture elements that interact with interfaces of the elements of the *architecture of a software increment* in order to allow interaction and provide the required functionality. Finally, in order to support reconfiguration on cloud environments, *IAM* includes information related aspects of Cloud Software Architectures [17]. For instance, the *IAM* associated to the second increment of motivating example (see section 3.1) will include:
   (a) Information about Shipping Request Service, Ship Status Service and connections that need to be added.
   (b) References to the interfaces of Place Order service and its interaction protocol (i.e., both elements need to be updated in order to satisfy the new requirement that establish that: the Place Order service must provide information about last bought of items included in order).
   (c) References to the service interfaces required and provided by/to the current architecture that will serve as *IP*.
   (d) Information related to cloud software architecture such as interaction pattern between dealer and transport partner (e.g., publish/subscribe connector, request/response connector).
5. *Increment Description Guidelines:* Help *Software Architects*: i) to identify impact of increment integration on current architecture and; ii) to take design decisions. These guidelines give support about how to specify increments using *IAM* and *CurAM*. In addition, we have begun to work in an *Increment Description Language* (*IDL*). This language will allow *Software Architects* to use high-level architecture reconfiguration operations to specify impact of the integration on current cloud service architecture. Service Oriented Architecture Modeling Language (SoaML) [18] leverages Model Driven Architecture (MDA) and provides a UML profile and meta-model for the specification of services. However, SoaML does not allow to represent how the architecture of a software increment affects the existing cloud architecture nor to specify information related to cloud software architectures. *IAM* and *CurAM* are part of this *IDL* and their meta-models will extend the SoaML meta-model. *IAM* and *CurAM* artifacts are input for the next activity, which is described below.

Check Increment Compatibility. This activity helps to verify whether the *architecture of a software increment* can be integrated into the current architecture. Its main objective is to reduce the risk of incompatibilities between service interfaces that could

avoid integration (e.g., different names of methods and services, different message ordering, etc.). Despite the fact that in previous activity *Software Architects* specified the impact of the increment integration on current architecture, in practice, we cannot expect that any given software component perfectly matches the needs of a system nor that the components being assembled perfectly fit one another [7]. The same may happen during the integration of the increment, where incompatibilities may exist between *IAM* and *CurAM* elements.

We will apply software adaptation techniques to correct incompatibilities, generating software adaptors when needed. We have chosen to follow Cámara et al. approach [19] because i) it is a model driven approach; ii) it gives support the automatic creation of adaptors from abstract specifications; iii) and, it provides tools that fully support the adaptation approach from start to finish (including compatibility checking). To follow this approach we need to provide service interfaces described by signatures (operation names and types) and interaction protocol. The former must be described as a WSDL representation and the latter as an Abstract BPEL (ABPEL) representation. *Integration Designers* will specify model transformations to obtain these representations from the increment's architecture (*IAM*) as well as from the current architecture (*CurAM*).

This activity results in generation of software adaptors to be used in a specific cloud platform (*CloudAdaptors*) using *CurAM* and *IAM* as input. *CloudAdaptors* allow correcting incompatibilities between services interaction protocols (i.e., incompatibilities among *IP* operations). If discrepancies exist, *Software Architects* apply model-to-text (M2T) transformations to generate skeletons of *CloudAdaptors*. Then, *Software Developers* complete *CloudAdaptors* skeletons, implementing code to solve discrepancies according to deployment platform. Depending on cloud platform, *CloudAdaptors* may be scripts, configuration files, packages, services, or any cloud platform specific artifact. For instance, regarding to our motivating example (see Section 3.1), in order to allow interaction with web services provided by international transport partner, the integration of the third increment will require i) compatibility verification of interfaces requested by manufacturer and interfaces provided by international transport partner. The first interfaces are already deployed and belong to *CurAM*; whereas the latter, that are going to be deployed, are described in *IAM*; ii) generation of *CloudAdaptors*. Assuming that the deployment platform is Windows Azure, the generated *CloudAdaptors* will be a cloud service Worker Role.

Reconfigure Architecture. This last activity supports the execution of the integration operations, resulting in incorporation of the *architecture of the software increment* into current architecture and the corresponding dynamic architecture reconfiguration (see Fig. 2). This activity is composed of the following main steps:

Select Adaptation Pattern. In this step, *Software Architects* participate in the selection of the adaptation patterns best suited to integrate the *architecture of the software increment* into the current architecture. This step results in the generation of a List of Patterns Model, using *CurAM* and *IAM* information to select patterns from Adaptation Pattern Repository Model. The Output artifact and input *AdaPRepM* are explained below:

1. *Adaptation Pattern Repository Model* (*AdaPRepM*): *Integration Designers* use this to represent prescriptions at a high level of abstraction of steps required to integrate architectural elements into current architecture. *Integration Designers* define adaptation patters for possible integration scenarios. We consider scenarios where the elements of the *architecture of a software increment*: i) do not need interconnection with any current architecture element; ii) require establish interconnection with current architecture elements without updating them; and iii) require establish interconnections and update current architecture elements. Adaptation patterns is a research field by itself, in our work we will extend current proposals to define the *AdaPRepM* meta-model. To be specific, we will extend the Meta-model for Adaptation Pattern Composition proposed by Ahmad et al. [20].

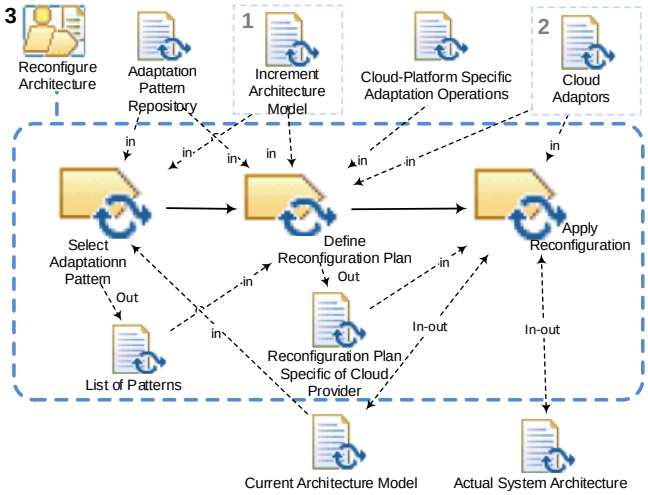


Fig. 2. Reconfigure Architecture Activity

2. *List of Patterns Model* (*LisPatM*): This output provides a list with the most suited adaptation patterns that must be applied to integrate the *elements of the architecture of the software increment* into current architecture.

Define Reconfiguration Plan. This activity aims to generate a plan with the sequence of reconfiguration operations needed to integrate the elements of the *architecture of the software increment* into the current architecture. For doing this, a two-step models transformation strategy must be applied. On the first step, *Integration Designers* specify M2M transformations that generate a Reconfiguration Plan Independent of Cloud Provider technology. This plan includes high-level reconfiguration actions needed to change cloud service architectures. In the second step, *Integration Designers* specify M2T transformations to operationalize reconfiguration actions into Reconfiguration Plans Specific of Cloud Provider. *Software Architects* execute these transformations and *Software Developers* complete the generated plans if required. This activity has as inputs *IAM*, *AdaPRepM*, *LisPatM* and *Platform Specific Adaptation Operations Model*.

1. *Cloud-Platform Specific Adaptation Operations Model*: This model represents at a high level of abstraction cloud artifacts and reconfiguration operations independently of a specific cloud platform technology. This model and *LisPatM* are used to generate a Reconfiguration Plan Independent of Cloud Provider.
2. *Reconfiguration Plan Specific of Cloud Provider*: This artifact is specific of a cloud provider technology. This artifact includes sequence of commands that create, update, or destroy architectural elements instances and their links at runtime. Examples are scripts, packages, configuration files and so on.

Apply reconfiguration. In the last step, the *Cloud Specialist*, expert in deployment, integrates the increment into the current architecture by deploying *CloudAdaptors* and by using dedicated services to apply the *Reconfiguration Plan Specific of Cloud Provider* artifacts in the corresponding cloud platform. Integration dynamically reconfigures instances of the running *Actual System Architecture*.

4 Conclusions and Future Work

We introduced the *DIARy* process to support the dynamic software architecture reconfiguration triggered by the deployment of new cloud services. *DIARy* uses model-driven and adaptation techniques to allow integration of cloud services into current architecture at runtime. We believe this process provides a solution to cover the lack of support to incremental and dynamic deployment of cloud services into existing cloud service architecture. *DIARy* shows the steps that software developers must follow to specify how the *architecture of a software increment* will affect the existing cloud architecture. In addition, model transformations are used for: i) promoting the compatibility between the architecture of the increment with the existing cloud architecture; and ii) generating cloud-platform specific reconfiguration plans that apply adaptation patterns to reconfigure existing cloud architecture. Activities and artifacts included in *DIARy* were described, and a motivation example was used to illustrate some related aspects.

At this moment, we experimented with several small examples to test the viability of the approach. As further work, we plan to empirically validate DIARy through controlled experiments and case studies with medium-sized real-world projects. We are also working on: i) the definition of an Increment Description Language to specify increment's architectures and their impact on actual system architecture; ii) the definition of a reference architecture to support the reconfiguration process, and iii) the implementation of different model transformations to automate the DIARy process.

Acknowledgments. This research was supported by the Value@Cloud project (MICINN TIN2013-46300-R); the Scholarship Program Senescyt, Ecuador; the Faculty of Engineering, University of Cuenca, Ecuador; and the ValI+D program (ACIF/2011/235), Generalitat Valenciana.

5 References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A View of Cloud Computing. Commun. ACM. 53, 50–58 (2010).
2. Motta, G., Sfondrini, N., Sacco, D.: Cloud Computing: An Architectural and Technological Overview. Int. Joint Conf. on Service Sciences. pp. 23–27. IEEE, Shanghai (2012).
3. Bastida, L., Berreteaga, A., Cañadas, I.: Adopting Service Oriented Architectures Made Simple. Springer, London (2008).
4. Fehling, C., Leymann, F., Retter, R.: An Architectural Pattern Language of Cloud-Based Applications. 18th Conf. on Pattern Languages of Programs. pp. 1–11. ACM Press, New York (2011).
5. Gomaa, H., Hashimoto, K., Kim, M., Malek, S., Menascé, D.: Software Adaptation Patterns for Service-Oriented Architectures. ACM Symposium on Applied Computing. pp. 462–469. ACM, New York (2010).
6. Jamshidi, P., Ghafari, M., Ahmad, A., Pahl, C.: A Framework for Classifying and Comparing Architecture-Centric Software Evolution Research. 17th European Conference on Software Maintenance and Reengineering. pp. 305–314. IEEE, Genova (2013).
7. Canal, C., Poizat, P., Salaun, G.: Model-Based Adaptation of Behavioral Mismatching Components. Softw. Eng. IEEE Trans. 34, 546–563 (2008).
8. Yellin, D.M., Strom, R.E.: Protocol Specifications and Component Adaptors. ACM Trans. Program. Lang. Syst. 19, 292–333 (1997).
9. Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., Tivoli, M.: Towards an Engineering Approach to Component Adaptation. Springer Berlin Heidelberg (2006).
10. Miranda, J., Guillen, J., Murillo, J.M., Canal, C.: Assisting Cloud Service Migration Using Software Adaptation Techniques. 6th Int. Conf. on Cloud Computing. pp. 573–580 (2013).
11. Baresi, L., Ghezzi, C.: The Disappearing Boundary Between Development-time and Run-time. FSE/SDP Workshop on Future of software Engineering Research. pp. 17–21 (2010).
12. Baresi, L., Heckel, R., Thöne, S., Varr´o, D´.: Style-Based Modeling and Refinement of Service-Oriented Architectures. Softw. Syst. Model. 5, 187–207 (2006).
13. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: QoS-Driven Runtime Adaptation of Service Oriented Architectures. Proc. 7th Jt. Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. 131–140 (2009).
14. Menascé, D.A., Gomaa, H., Malek, S., Sousa, J.P.: SASSY: A Framework for Self-Architecting Service-Oriented Systems. Software, IEEE. 28, 78–85 (2011).
15. Babar, M.A., Brown, A.W., Mistrik, I.: Making Software Architecture and Agile Approaches Work Together: Foundations and Approaches. Agile Software Architecture: Aligning Agile Processes and Software Architectures. pp. 1–22. Morgan Kaufmann (2013).
16. Casanave, C.: Enterprise Service Oriented Architecture Using the OMG SoaML Standard. Model Driven Solut. Inc., White Pap. 1–21 (2009).
17. Hamdaqa, M., Livogiannis, T., Tahvildari, L.: A Reference Model for Developing Cloud Applications. CLOSER. pp. 98–103. Citeseer (2011).
18. Object Management Group: Service Oriented Architecture Modeling Language (SoaML), http://www.omg.org/spec/SoaML/.
19. Cámara, J., Martin, J.A., Salaun, G., Cubo, J., Ouederni, M., Canal, C., Pimentel, E.: Itaca: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. 31st Int. Conf. on Software Engineering. pp. 627 – 630. IEEE, Vancouver, BC (2009).
20. Ahmad, A., Babar, M.A.: Towards a Pattern Language for Self-Adaptation of Cloud-Based Architectures. Proc. WICSA Companion Volume. pp. 1–6. ACM Press, New York (2014).