ACM/IEEE 16th International Conference on
Model Driven Engineering Languages and Systems

September 30, 2013 – Miami, Florida (USA)

ME 2013 – Models and Evolution Workshop Proceedings

Alfonso Pierantonio, Bernhard Schätz (Eds.)

Published on September 22, 2013

Program Committee

Arnaud Albinet	Continental Automotive (France)
Abdelkrim Amirat	LINA Laboratory (France)
Vasilios Andrikopoulos	University of Stuttgart (Germany)
Salima Benbernou	Université Paris Descartes (France)
Mireille Blay-Fornarino	Université de Nice-Sophia Antipolis (France)
Jean-Michel Bruel	IRIT (France)
Antonio Cicchetti	Mälardalen University (Sweden)
Davide Di Ruscio	Università degli Studi dell'Aquila (Italy)
Anne Etien	LIFL - University of Lille 1 (France)
Jesus Garcia-Molina	Universidad de Murcia (Spain)
Ludovico Iovino	Università degli Studi dell'Aquila (Italy)
Ethan K. Jackson	Microsoft Research (USA)
Gerti Kappel	Vienna University of Technology (Austria)
Udo Kelter	University of Siegen (Germany)
Jochen Kuester	IBM Research (Switzerland)
Olivier Le Goaer	Université de Pau et des Pays de l'Adour (France)
Tom Mens	University of Mons (Belgium)
Richard Paige	University of York (UK)
Alfonso Pierantonio (co-chair)	Università degli Studi dell'Aquila (Italy)
Bernhard Rumpe	RWTH Aachen University (Germany)
Bernhard Schätz (co-chair)	fortiss Gmbh (Germany)
Martina Seidl	Vienna University of Technology (Austria)
Jonathan Sprinkle	University of Arizona (USA)
Dalila Tamzalit	University of Nantes (France)
Hans Vangheluwe	University of Antwerp (Belgium)
Stefan Wagner	University of Stuttgart (Germany)
Manuel Wimmer	Vienna University of Technology (Austria)

Table of Contents

Keynote

Learning on the Job: Supporting the Evolution of Designs

Bran Selic

Malina Software Corp., Canada

System design is the process of finding a suitable solution in the abstract space of possible design variants. As design progresses, we identify and evaluate potential design alternatives, learning in the process not only about possible solutions but, if we are doing it right, about the problem on hand. (A wise man once noted: If you think about a problem long enough, you will always find a better way of solving it.) Engineering models can and should play a fundamental role in this process, supporting both understanding and invention.

In this talk, we first present a view of design as a search problem (which clearly distinguishes it from the closely related project management process with which it is often confused). From this perspective, we identify and categorize the issues involved in design and focus in particular on where and how models and model-based technologies can help overcome them. The talk concludes with a list of related research challenges for the modeling community.

Bran Selic *is President and Founder of Malina Software Corp., a Canadian consulting and research enterprise, focused on model-based software and systems engineering. Bran has over 40 years of industrial experience in the design and development of complex software-intensive systems in various technical domains (robotics, aerospace, telecom, and industrial control). He was one of the primary contributors to the Unified Modeling Language (UML) and other modeling language standards. He is formally affiliated as an adjunct with several academic and research institutions.*

A Survey on Incremental Model Transformation Approaches*

Angelika Kusel[1], Juergen Etzlstorfer[1], Elisabeth Kapsammer[1], Philip Langer[2], 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. Steadily evolving models are the heart and soul of Model-Driven Engineering. Consequently, dependent model transformations have to be re-executed to reflect changes in related models, accordingly. In case of frequent, but only marginal changes, the re-execution of complete transformations induces an unnecessary high overhead. To overcome this drawback, incremental model transformation approaches have been proposed in recent years. Since these approaches differ substantially in language coverage, execution, and the imposed overhead, an evaluation and comparison is essential to investigate their strengths and limitations. The contribution of this paper is a dedicated evaluation framework for incremental model transformation approaches and its application to compare a representative subset of recent approaches. Finally, we report on lessons learned to highlight past achievements and future challenges.

1 Introduction

In Model-Driven Engineering (MDE), models are first-class artifacts throughout the software life-cycle [3]. Transformations of these models are comparable, in role and importance, to compilers for high-level programming languages, since they allow, e. g., to bridge the gap between design and implementation [14]. Like any other software artifact, models are subject to constant change, i. e., they evolve, caused by, e. g., changing requirements. Therefore, dependent models, which have been derived from the original models by means of transformations, have to be updated appropriately. A straight-forward way is to re-execute the transformations entirely, i. e., in batch mode. In case of minor changes on large models consisting of several thousand elements [20], however, re-execution of a

* This work has been funded by BMVIT under grants FFG BRIDGE 832160 and FFG FIT-IT 825070 and 829598, FFG Basisprogramm 838181, and by ÖAD under grant AR18/2013 and UA07/2013.

complete transformation is not efficient [12,13,18]. Consequently, it is of utmost importance, to transform those elements that have been changed, only, which is commonly referred to as *incremental transformation* [8,17].

Several incremental model transformation approaches focusing on different transformation languages have been proposed recently. Although all of them aim at the efficient propagation of changes, they differ substantially, not only since basing on different transformation languages. However, no dedicated survey has been brought forward so far, which would be essential to highlight past achievements as well as future challenges. To alleviate this situation, in this paper, first, a dedicated evaluation framework is proposed (cf. Section 2), whose criteria are not only inspired by the MDE domain (cf., e. g., [8]), but also by related engineering domains like data engineering, e. g., in terms of incremental maintenance of materialized views (cf., e. g., [10]), where incremental approaches are used since decades. Second, this framework is applied to a carefully selected set of incremental transformation approaches to achieve an in-depth comparison (cf. Section 3). Third, lessons learned are derived from this comparison and presented in Section 4 to highlight past achievements and future challenges. Finally, Section 5 concludes the paper.

2 Evaluation Framework

In this section, the proposed criteria for the evaluation of incremental model transformation approaches are presented. The set of criteria for evaluating the incremental transformation approaches has been derived by examining dedicated approaches in a bottom-up manner as well as in a top-down manner by reviewing surveys in related engineering domains (cf., e. g., [10]) – methodologically adhering to some of our previous surveys, e. g., [23]. This process resulted in an evaluation framework (cf. Fig. 1), comprising the three categories of (i) *language coverage* for explicating those parts of a transformation language that are considered for incremental execution (cf. Section 2.1), (ii) *execution phases* for highlighting how incrementality is achieved at run-time (cf. Section 2.2), and (iii) *overhead* for pointing out the additional efforts needed to achieve incrementality (cf. Section 2.3).

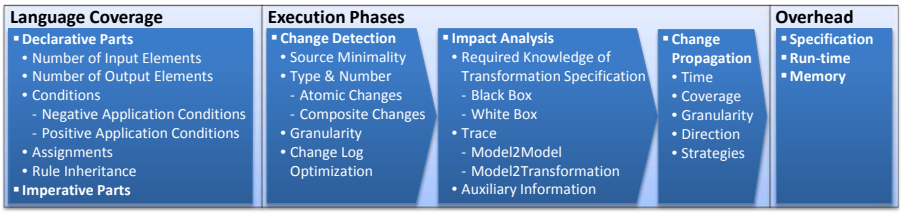


Fig. 1. Evaluation Criteria for Incremental Model Transformation Approaches

2.1 Language Coverage

The first set of criteria reveals the *language coverage*, i. e., which parts of a transformation language may be executed incrementally. In general, model transformations are specified by a set of rules that comprise input patterns with conditions that match source model elements, and output patterns that produce elements of the target model. Thereby, source and target models have to conform to their corresponding metamodels. Furthermore, transformation rules may be inherited for reuse (cf. [22] for an overview). Such transformation specifications may comprise *declarative* and *imperative* transformation parts. Since current incremental transformation approaches mostly focus on the support of incremental execution of declarative language parts [15], this category has been further broken down into the criteria *number of input and output elements*, *conditions*, *assignments*, and *rule inheritance*. Among those, conditions play a special role in incremental model transformations, since negative application conditions (NACs) may lead to *non-monotonicity*, e. g., insertion of elements in the source model entails a deletion of elements in the target model [11, 19], and positive application conditions (PACs) may cause *non-continuity*, i. e., a non-uniform treatment of model elements, e. g., new elements in a container may be transformed differently than previously added elements to the same container [19].

2.2 Execution Phases

Any incremental approach may be characterized by three dedicated execution phases, comprising (i) *change detection*, i. e., detection of changes in the source model, (ii) *impact analysis*, i. e., detection of parts of the transformation that must be executed to reflect the changes, and (iii) *change propagation*, i. e., the actual execution of the affected transformation parts as well as the update of the target model.

Criteria for Change Detection. For detecting changes, basically two approaches may be followed, comprising (i) *state-based* comparison of the changed source model to its previous version, or (ii) *operation-based* detection of changes by directly recording each change [7]. Although (i) may allow for tool independence, run-time performance depends on the size of the source model. Thus, with regard to run-time performance, (ii) is preferable and is referred to as *source minimality* [8], representing the first criterion. Change detection may be further characterized by the *type* of the detected changes, including *atomic* operations, i. e., insert and delete, and *composite* operations, e. g., update and move, which may be composed by combining atomic operations and may allow for reducing the number of change propagation operations to be executed, by substituting insert and delete operations. Another criterion is the *granularity* of the detected changed elements, ranging from coarse granularity, i. e., on the level of objects, only, to fine granularity, i. e., on the level of values and links, which is preferable, since it allows for a more precise detection. Finally, change detection may be capable of detecting single or multiple changes at once, whereby the latter utilizes a change log and may allow to reason on inter-dependencies between changes,

thereby facilitating *change log optimization*. In this context, for instance, change operations that cancel others may be detected and optimized, e. g., a rename operation for an element that is subsequently deleted may be removed.

Criteria for Impact Analysis. For impact analysis, the impact of changes on target models must be detected by (i) utilizing knowledge from the transformation specification and (ii) employing information that relates the transformations and the models during execution. Concerning the knowledge of the transformation, approaches, which require *white-box* knowledge, i. e., insights into the internal structure, may allow for a more precise detection of affected transformation parts compared to those, where *black-box* knowledge, i. e., the inputs and outputs, are considered, only. For (ii), it is of utmost importance to recognize those transformation parts that must be executed, requiring a *Model2Transformation* (M2T) trace. Such a trace may be either created at *compile-time* by analyzing the bound metamodel types or at *run-time* by keeping track of the execution of the transformation. Additionally, a dynamically created *Model2Model* (M2M) trace is obligatory to relate the elements of the source model to those in the target model. Besides these traces, additional *auxiliary information*, such as internal execution states or intermediate results may be utilized, to further improve incrementality.

Criteria for Change Propagation. Finally, changes detected in the first phase have to be propagated to the target model utilizing dependency information exposed in the second phase. Change propagation may differ in (i) *time*, i. e., when the changes are propagated, (ii) *coverage*, i. e., the amount of changes to be propagated, (iii) *granularity*, i. e., the degree of transformation code to be re-executed, (iv) *directionality*, i. e., the direction of change propagation between source and target, and, finally, (v) *strategies*, i. e., different plans to propagate the changes. Concerning time, *eager* means that changes in the source model are propagated synchronously to the target model, whereas *lazy* induces an asynchronous propagation. Propagation may cover either the *complete* set of changes at once or a *part* of it, only, e. g., single changes. The propagation of single changes is useful to allow for change propagation on demand, i. e., only when the affected target model elements are accessed. According to the granularity of the change propagation, either a complete *rule* or a single *binding* (or assignment) has to be re-executed, whereby the latter case results in modifying fewer elements in the target model. Furthermore, changes may be either propagated from a dedicated source to a target model, i. e., *unidirectional* or also vice versa, i. e., *bidirectional*. Bidirectionality can be either achieved by utilizing dedicated bidirectional model transformation languages (e. g., TGGs [21] or JTL [6]) or by connecting source and target model elements via traces and potentially restricting transformation language expressiveness to enable bidirectional propagation. Therefore, this criterion is included in the propagation phase. Finally, different *strategies* for updating the target model may be applied, e. g., executing multiple updates sequentially or in parallel, that may be selected automatically or manually.

2.3 Overhead

The final set of criteria aims at pointing out overheads that result from incrementality. Such overheads may arise with respect to three different areas, being (i) *specification*, i. e., whether the transformation designer must use a dedicated syntax resulting in a new specification, (ii) *run-time*, i. e., whether additional run-time is consumed in contrast to batch transformations, and (iii) *memory*, i. e., whether additional memory is consumed compared to batch transformations.

3 Comparison of Approaches

After having presented the evaluation framework in the previous section, in this section it is applied to selected approaches. The results of the application are summarized in Table 1. The selection of approaches for incremental model transformation comprises approaches that facilitate incremental execution of user-specified transformations. Hence, approaches that e. g., aim at efficient batch transformations or support incrementality for specific applications, only, are not covered in the comparison (e. g., [5, 6]). Eight approaches across different transformation languages that meet these criteria have been identified in the literature and, therefore, participate in the evaluation. Three of them are of declarative nature, whereby two base on Triple Graph Grammars (TGGs) [9,16] and one on the Tefkat transformation language [11]. The remaining five approaches allow for hybrid transformation code, i.e., declarative and imperative parts. Among them, one approach bases on the Atlas Transformation Language (ATL) [13], three on the graph-transformation-based Viatra2-framework [1, 2, 18], and one approach enables incrementality by abstracting from the actual transformation specification and by relating source and target model elements, only [19].

3.1 Language Coverage

In the context of declarative language parts, six of the eight approaches [1, 2, 9, 11, 16, 18] support an arbitrary number of input and output elements, i. e., M:N mappings. One approach [13] restricts itself to a single input element, but allows for an arbitrary number of output elements, i. e., 1:N mappings, and another approach [19] may handle 1:1 mappings, only. Interestingly, NACs and PACs, although quite challenging, are supported by all except one approach [19]. While assignments are naturally supported by all approaches, support for rule inheritance is not that widespread. Only one approach may handle rule inheritance in incremental transformations, since the transformation specification is regarded as black-box and, therefore, inherited rules do not impact the approach [19]. The remaining approaches do either exclude rule inheritance for incremental transformation [9, 11, 16], explicitly refer to the support of rule inheritance as future work by flattening the inheritance hierarchy [13], or do not support rule inheritance at all [1, 2, 18], i. e., also not in batch mode. In contrast to the comprehensive support for declarative language parts, imperative parts are either

completely re-executed [1, 2, 18], thereby resigning the advantages of incrementality, regarded as black box [19], which may violate correctness, or not allowed at all [13].

In summary, one may see that incremental model transformation approaches suffer from restricted language coverage in terms of rule inheritance and imperative parts, and thus, not all potential for incremental execution is exploited so far.

3.2 Execution Phases

Change Detection. Regarding change detection, all approaches detect changes operation-based and, therefore, comply with source minimality. Atomic changes are supported by all approaches. Concerning composite changes, six approaches support update operations [1, 2, 9, 13, 18, 19]. Move operations are explicitly supported by one approach, only [2]. Four approaches are able to detect multiple change operations [2, 9, 16, 18]. However, only two of them actually support an optimization of the change log [9, 18]. Considering the granularity of changes, all approaches enable the detection of fine-grained changes.

Summing up, current approaches mostly lack the detection of multiple, composite change operations and a change log optimization. Both would favor an optimized change propagation resulting in fewer operations on the target model.
Impact Analysis. Seven of the eight approaches require white-box knowledge of the transformation for analyzing rules and generating trace information [1, 2, 9, 11, 13, 16, 18], while one approach considers the specification of the batch transformation as a black box [19]. The M2T trace is generated by seven approaches [1, 2, 9, 11, 13, 16, 18] at run-time, while two of them generate parts of this trace at compile-time [1, 13]. One approach does not generate any M2T trace at all [19], but lacks comprehensive language coverage. All approaches dynamically generate M2M traces. Concerning auxiliary information, one approach stores the complete transformation context in terms of an *SLDNF-tree (Selective Linear Definite clause with Negation as Failure)* [11], one generates dedicated rules for change propagation at compile-time [13], and in [2] so-called *Change History Models* are used as input for the change propagation. One approach [1] creates database tables for each match pattern in a transformation specification and employs triggers for change detection, while in [19] a so-called *Concept Pool* is utilized, which holds similar concepts between source and target models to enable bidirectional change propagation.

In summary, one may see that all approaches require trace information and most approaches rely on auxiliary information to further leverage incrementality.
Change Propagation. Most approaches propagate changes in an eager manner [1, 9, 11, 13, 16, 18]. One approach allows for lazy propagation by letting the user decide when to apply the changes on the target model [2]. Finally, one approach allows for both, synchronous and asynchronous, change propagation [19]. The same approach also supports partial propagation. Five of the eight approaches re-execute a complete rule [1, 2, 9, 16, 18], while two approaches are capable of re-executing a single binding, only [11, 13]. Those two rely on

auxiliary information, being the complete execution context [11] or fine-grained compiler-generated rules [13]. One approach does not consider rules or bindings of the batch transformation specification to be re-executed, since the transformation is regarded as a black-box [19]. Concerning directionality, TGGs support bidirectionality [9, 16], while other approaches allow for unidirectional change propagation, only [1, 2, 11, 13, 18]. One approach aims at deriving a bidirectional model transformation from a unidirectional transformation specification, but is limited in terms of language coverage [19]. Solely a single approach offers manual selection of strategies in terms of serial or parallel execution of change propagation [18].

Summing up, current incremental model transformation approaches mostly propagate changes synchronously with a predefined strategy. Thereby, situations that may require different execution strategies for efficient incremental execution may not be handled satisfactorily.

3.3 Overhead

Two of the eight approaches require a new specification by the transformation designer to allow incremental execution [2, 18]. Concerning run-time overheads, three approaches state, that even in the worst case (i. e., the complete source model has been changed) incremental execution is as fast as batch transformation, i. e., no run-time overhead occurs [2, 9, 16], while the remaining lack any statement on this fact. Finally, all approaches accept memory overheads in terms of traces and auxiliary information.

In summary, one may see that most approaches introduce incrementality by changes to the execution engine instead of requiring a new syntax for transformation specifications, which is favorable, since then incrementality is transparent to the transformation designer. All approaches induce memory overheads due to the generation of traces and auxiliary information, which are, nevertheless, essential for incremental execution [12].

4 Lessons Learned

In this section, lessons learned from the evaluation and comparison of approaches are discussed along the different evaluation categories.

Insufficient Support for Imperative Parts. While all examined approaches support declarative language parts, they lack sufficient support for imperative parts, which are either disregarded at all or treated as black box, i. e., executed completely, instead of breaking them down into fine-grained parts. Consequently, extensive support for imperative parts, e. g., by exploiting incremental evaluation of conditions and bindings [4], may further leverage incrementality.

Focus on Basic Change Operations. Atomic change operations are supported by all approaches. However, support for composite change operations, such as update and move, is not that widespread, but would allow for a more

	Approach	Language / Framework	Language Coverage: Declarative Parts: Number of Input Elements	Number of Output Elements	Conditions: NACs	PACs	Assignment	Rule Inheritance	Imperative Parts	Execution Phases: Change Detection: Source Minimality	Type & Number: Atomic: Insert	Delete	Update	Composite: Move	Granularity: Object	Value	Link	Change Log Optimization	Impact Analysis: Req. Knowledge of Transformation Specification: Black Box	White Box	Trace: Model2Model	Model 2 Transform.: Compile-time	Run-time	Auxiliary Information	Change Propagation: Time: Eager (Synchronous)	Lazy (Asynchronous)	Coverage: Partial	Complete	Granularity: Rule	Binding	Direction: Unidirectional	Bidirectional	Strategy: Number	Selection	Overhead: No Specification Overhead	No Run-time Overhead	No Memory Overhead
Declarative	[9]	TGG	1..*	1..*	✔	✔	✔	-	n.a.	✔	1..*	1..*	1..*	-	✔	✔	✔	✔	-	✔	✔	-	✔	-	✔	-	-	✔	✔	-	-	✔	1	-	✔	✔	-
	[16]	TGG	1..*	1..*	✔	✔	✔	-	n.a.	✔	1..*	1..*	-	-	✔	✔	✔	-	-	✔	✔	-	✔	-	✔	-	-	✔	✔	-	-	✔	1	-	✔	✔	-
	[11]	Tefkat	1..*	1..*	✔	✔	✔	-	n.a.	✔	1	1	-	-	✔	✔	✔	n.a.	-	✔	✔	-	✔	SLDNF tree	✔	-	-	✔	-	✔	✔	-	1	-	✔	u.	-
Hybrid (Decl. + Imp.)	[13]	ATL	1	1..*	✔	✔	✔	-	-	✔	1	1	1	-	✔	✔	✔	n.a.	-	✔	✔	✔	✔	Compiler Generated Rules	✔	-	-	✔	-	✔	✔	-	1	-	✔	u.	-
	[18]	Viatra2	1..*	1..*	✔	✔	✔	n.a.	✔	✔	1..*	1..*	1..*	-	✔	✔	✔	✔	-	✔	✔	-	✔	-	✔	-	-	✔	✔	-	✔	-	2	m.	-	u.	-
	[2]	Viatra2	1..*	1..*	✔	✔	✔	n.a.	✔	✔	1..*	1..*	1..*	1..*	✔	✔	✔	-	-	✔	✔	-	✔	Change History Models	-	✔	-	✔	✔	-	✔	-	1	-	-	✔	-
	[1]	Viatra2	1..*	1..*	✔	✔	✔	n.a.	✔	✔	1	1	1	-	✔	✔	✔	n.a.	-	✔	✔	✔	✔	Database Tables	✔	-	-	✔	✔	-	✔	-	1	-	✔	u.	-
	[19]	Arbitrary	1	1	-	-	✔	✔	~	✔	1	1	1	-	✔	✔	✔	n.a.	✔	-	✔	-	-	Concept Pool	✔	✔	✔	✔	n.a.	n.a.	-	✔	1	-	✔	u.	-

Legend: ✔ ... supported - ... not supported ~ ... partially supported * ... multiple n.a. ... not applicable m. ... manual u. ... unknown

Table 1. Comparison Table of Incremental Model Transformation Approaches

efficient change propagation by utilizing according change propagation operations.

Trace Information is Mandatory. All approaches rely on trace information. Furthermore, the more trace information is generated to relate elements of source and target models as well as model elements and the transformation specification, the more accurateness in change propagation may be achieved. Of course, this is the general trade-off between space and time in computing.

Lack of Appropriate Propagation Strategies. Current incremental model transformation approaches do not provide different propagation strategies according to a given change situation. In case of many changes, re-execution of a batch transformation may outperform the run-time performance of incremental execution. An automatic selection of an appropriate propagation strategy could be achieved by analyzing the changes, transformation specification, and traces.

Lack of Proof of Correctness. Apart from one approach [16], correctness of the incremental model transformation approaches is not proven. Thus, formal proofs for correctness would be highly beneficial for the confidence of the incremental transformations.

5 Conclusion

In this paper, we presented a survey on incremental model transformation approaches. Therefore, criteria for evaluating and comparing incremental model transformation approaches have been proposed. Eight recent approaches have been compared according to the criteria and lessons learned have been presented.

Summing up, although several approaches for incremental model transformations exist, there is still space for improvements comprising (i) better support for imperative language parts, (ii) more efficient change propagation with dedicated change operators, (iii) provision and automatic selection of appropriate propagation strategies, and (iv) proving correctness of incremental model transformations.

References

1. Bergmann, G., Horváth, D., Horváth, A.: Applying Incremental Graph Transformation to Existing Models in Relational Databases. In: 6th Int. Conf. on Graph Transformations. Springer (2012)
2. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. SoSym 11(3) (Jul 2012)
3. Bézivin, J.: On the Unification Power of Models. SoSym 4(2) (May 2005)
4. Cabot, J., Teniente, E.: Incremental Evaluation of OCL Constraints. In: 18th Int. Conf. on Advanced Information Systems Engineering. Springer (2006)
5. Cicchetti, A., Ciccozzi, F., Leveque, T.: Supporting Incremental Synchronization in Hybrid Multi-view Modelling. In: Int. Conf. on Models in SE. Springer (2012)
6. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: JTL: A Bidirectional and Change Propagating Transformation Language. In: Int. Conf. on Software Language Engineering. Springer (2011)
7. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. ACM Comp. Surv. 30(2) (Jun 1998)
8. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal 45(3) (Jul 2006)
9. Giese, H., Hildebrandt, S.: Incremental Model Synchronization for Multiple Updates. In: 3rd Int. Workshop on Graph and Model Transformations. ACM (2008)
10. Gupta, A., Mumick, I.S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. IEEE Data Eng. Bull. 18(2) (1995)
11. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: 9th Int. Conf. on Model Driven Engineering Languages and Systems. Springer (2006)
12. Johann, S., Egyed, A.: Instant and Incremental Transformation of Models. In: 19th Int. Conf. on Automated Software Engineering. IEEE (2004)
13. Jouault, F., Tisi, M.: Towards Incremental Execution of ATL Transformations. In: 3rd Int. Conf. on Theory and Practice of Model Transformations. Springer (2010)
14. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model Transformation By-Example: A Survey of the First Wave. In: Conceptual Modelling and Its Theoretical Foundations. Springer (2012)
15. Kolovos, D., Paige, R.F., Polack, F.A.: The Grand Challenge of Scalability for Model Driven Engineering. In: Models in SE. Springer (2009)
16. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Efficient Model Synchronization with Precedence Triple Graph Grammars. In: 6th Int. Conf. on Graph Transformations. Springer (2012)
17. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. ENTCS 152 (Mar 2006)
18. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live Model Transformations Driven by Incremental Pattern Matching. In: 1st Int. Conf. on Theory and Practice of Model Transformations. Springer (2008)

19. Razavi, A., Kontogiannis, K., Brealey, C., Nigul, L.: Incremental Model Synchronization in Model Driven Development Environments. In: Conf. of the Center f. Adv. Studies on Collab. Research. IBM (2009)
20. Scheidgen, M., Zubow, A., Fischer, J., Kolbe, T.H.: Automated and Transparent Model Fragmentation for Persisting Large Models. In: 15th Int. Conf. on Model Driven Engineering Languages and Systems. Springer (2012)
21. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Graph-Theoretic Concepts in Computer Science. Springer Berlin Heidelberg (1995)
22. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., Wagelaar, D.: Surveying Rule Inheritance in Model-to-Model Transformation Languages. Journal of Obj. Techn. 11(2) (Aug 2012)
23. Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., Kapsammer, E.: A Survey on UML-Based Aspect-Oriented Design Modeling. ACM Comp. Surv. 43(4) (Oct 2011)

Co-evolution of Metamodels and Models through Consistent Change Propagation

Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed

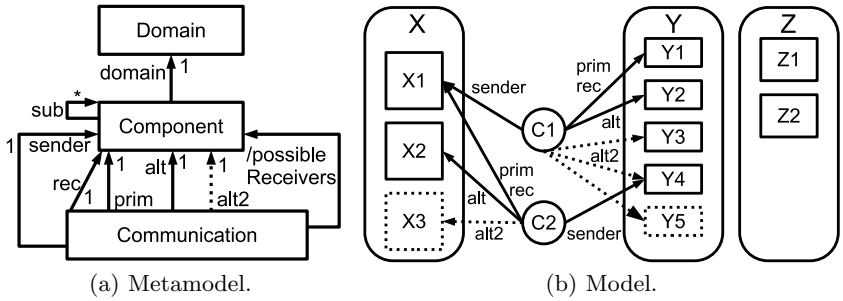
Institute for Systems Engineering and Automation
Johannes Kepler University (JKU)
Linz, Austria
{andreas.demuth|roberto.lopez|alexander.egyed}@jku.at

Abstract. In Model-Driven Engineering (MDE), metamodels and domain-specific languages are key artifacts as they are used to define syntax and semantics of domain models. However, metamodels are evolving over time, requiring existing domain models to be co-evolved. Though approaches have been proposed for performing such co-evolution automatically, those approaches typically support only specific metamodel changes. In this paper, we present a vision of co-evolution between metamodels and models through consistent change propagation. The approach addressed co-evolution issues without being limited to specific metamodels or evolution scenarios. It relies on incremental management of metamodel-based constraints that are used to detect co-evolution failures (i.e., inconsistencies between metamodel and model). After failure detection, the approach automatically generates suggestions for correction (i.e., repairs for inconsistencies). Preliminary validation results are promising as they indicate that the approach computes correct suggestions for model adaptations, and that it scales and can be applied live without interrupting tool users.

1 Introduction

In *Model-Driven Development (MDD)* [1], metamodels are key artifacts that represent real-world domains. Therefore, they define the language of models; that is, the different elements available for modeling along with their interdependencies. Metamodels thus impose structural and semantical constraints on models [2]. Although metamodels are often perceived as static artifacts that do not change, it has been shown that the opposite is the case: metamodels do evolve over time for various reasons. For instance, there is a trend for flexible design tools with adaptable metamodels that can be tailored to different domains (e.g., [3]). Another common source for metamodel evolution are refactorings that focus on improving a metamodel's structure and usability.

Co-evolution of models denotes the process of adapting models as a consequence of metamodel evolution [4, 5]. This is a non trivial process, and incorrect co-evolution may cause models to no longer comply with their metamodels. Several incremental approaches have been proposed to support this process (e.g., [6]). Unfortunately, proposed solutions are typically limited to specific



(a) Metamodel. (b) Model.

Fig. 1. Simple metamodel (a) and model (b). Metamodel evolution and valid model co-adaptations drawn dotted.

metamodels or do not fully support all kinds of possible changes (e.g., restriction of metaproperty) [7]. In particular, existing generic approaches do not take into account domain-specific model constraints. Therefore, co-evolution of metamodels and models remains an open issue.

In this paper, we outline a generic approach that does not try to automate co-evolution in general, but that detects co-evolution failures and suggests model adaptations to co-evolve a model correctly. In particular, the approach relies on incremental constraint management that allows for efficient detection of co-evolution failures (including the absence of co-evolution). If such failures are detected, the resulting inconsistencies between metamodel and model – along with other design constraints imposed on the model – are used for finding suitable model adaptations (repairs) that establish compliance of the model with the updated metamodel and thus lead to correct co-evolution.

2 Motivating Example

To illustrate our work, we use a simple metamodel for component-oriented systems with high availability requirements, as shown in Fig. 1(a). Note that, for now, we only consider elements drawn solid – dotted elements indicate evolution which we will discuss later. The metamodel defines three classes: `Component`, `Domain`, and `Communication`. Components can have an arbitrary number of `sub`-components and must belong to a `Domain`. Domains include components that are responsible for fulfilling a common task in the system. Communications occur between a `sender` and a receiver (`rec`) component. To increase the chance of a successful communication, different components can be specified as primary (`prim`) and as alternative (`alt`) target of a communication. All possible targets of a communication are aggregated by the derived reference `possibleReceivers`. Note that the defined reference cardinalities (e.g., `1` for `prim`) implicitly define model constraints. For example, an instance of `Communication` must reference exactly one `Component` via `prim`. However, to ensure that only intended models

can be built, the metamodel has been extended with the following three explicit constraints:

R1 All possible receivers of a `Communication` must be located within a single `Domain` (i.e., a set of components with a common purpose).
R2 A communication may only occur between components of different component domains.
R3 It is not permitted that a single component is used as primary and alternative target.

Note that we omit other possible domain-specific constraints as well as implicit syntactical constraints for simplicity reasons.

In Fig. 1(b), a model that complies with the metamodel is depicted. Again, ignore dotted elements for now as they indicate possible evolution which we will discuss later. The derived reference `possibleReceivers` is omitted for readability reasons. The model contains three domains: X, Y, and Z. Domain X consists of only two components ($X1$ and $X2$), Y consists of four components in total ($Y1$ – $Y4$), and Z consists of two components ($Z1$ and $Z2$). The model also contains two communications ($C1$ and $C2$), drawn as circles.

Let us now consider a simple metamodel evolution. To increase the availability of systems and reduce the chance of communication failures, a second alternative communication target (called `alt2`) is added to the metamodel, as indicated by the dotted arrow in Fig. 1(a). Intuitively, this metamodel evolution requires the model in Fig 1(b) to be co-evolved as a new mandatory reference was added to the class `Communication` that is instantiated twice in the model. While existing approaches can typically find model adaptations that produce a syntactically correct model – for example, by adding a new reference `alt2`, which points to an arbitrary `Component`, to every `Communication` – such adaptations may easily lead to a semantically incorrect model. In the next section, we will show how our approach handles this scenario and automatically provides user guidance that helps designers to easily find valid adaptions.

3 Co-evolution through Consistent Change Propagation

To address the issues discussed above, we propose to perform co-evolution through consistent change propagation. The approach consists of two phases:

1. Detect co-evolution failures.
2. Derive options for correction of failures.

In Phase 1, the approach detects locations where co-evolution is not performed correctly. This, of course, includes the situation of plainly missing co-evolution. In Phase 2, options for a correct propagation of the metamodel change to the affected model are derived. We will now discuss those phases in more detail and also show how the approach is applied to the evolution scenario presented above.

3.1 Phase 1: Co-evolution Failure Detection

Although metamodel evolution is likely to require model adaptations, this is not a necessity – a metamodel may also change in ways that do not affect the validity of existing models. For example, when an optional reference is added. Additionally, models may be changed manually by designers or automatically by tools after a metamodel evolution occurred, trying to co-evolve the model. Therefore, it is necessary after a metamodel change – and subsequent model adaptations – to determine whether an affected model is consistent with the updated metamodel. If it is, co-evolution was performed correctly and no further intervention is required. If, however, the model is inconsistent with the updated metamodel, co-evolution failed and additional model adaptations are necessary.

Constraint Management As we have shown in the running example, constraints can be used for ensuring both syntax and semantics. Therefore, when the metamodel evolves, constraints of both kinds may be affected. By using an incremental constraint management approach, it is possible to update constraints after metamodel changes – ensuring that models are always validated with constraints that are up-to-date.

In our example, the addition of `alt2` in Fig. 1(a) requires a new syntactical constraint to be enforced on models:

R4 Each `Communication` must reference a `Component` via `alt2`.

Consistency Checking After updating the set of constraints imposed by the metamodel, standard consistency checkings mechanisms (e.g., [8, 9]) can be used to detect inconsistencies. While the typical scenario is that an unchanged model becomes inconsistent after metamodel evolution, it is also possible that a previously inconsistent model becomes consistent without any adaptations. Additionally, model adaptations that are performed for the purpose of co-evolution may be incorrect and actually introduce new inconsistencies. If a consistency checker that uses an up-to-date set of constraints detects inconsistencies after a metamodel evolution, model adaptations are required and co-evolution was not done correctly.

After adding the new syntactical constraint defined above, any standard consistency checker will find that the model in Fig. 1(b) contains two inconsistencies: neither $C1$ nor $C2$ provide a second alternative target. Thus, our intuitive assumption that additional model adaptations are necessary for correct co-evolution has been confirmed.

3.2 Phase 2: Co-evolution Correction

Once co-evolution failures have been detected, our approach reaches Phase 2 in which those failures are corrected.

Repair Options To correct co-evolution failures and propagate the metamodel change correctly, it is necessary to find model adaptations (i.e., repair options) that transform the inconsistent model into a consistent one. Unfortunately, finding suitable adaptations is non trivial as every change to a model may not only eliminate the violation of a constraint, but it may also cause other constraints to be violated. However, single changes can of course also remove several inconsistencies at once. Due to those side effects, finding suitable corrections is a complex task that should not be performed in an ad-hoc manner. Our approach employs a reasoning mechanism that takes into consideration all design constraints present in a model to find suitable adaptations [10]. Note that using not only those constraints that are actually based on the metamodel but all design constraints, repairs can be computed with higher precision as more information is available for the reasoning engine and side effects can be computed.

Let us come back to our example. During Phase 1, two inconsistencies caused by the elements $C1$ and $C2$ were detected in the model. First, we consider the inconsistency involving $C1$. To correct the syntax and remove the violation of constraint $R4$, a reference $alt2$ to any component is sufficient. Moreover, in each domain a new component could be created and used as second alternative target for $C1$. Of course, it would also be possible to create a new component in an entirely new domain. Therefore, there are 12 options available in total: one for each of the eight existing (i.e., solid drawn) components in Fig. 1(b), one for each of the three domains, and one for a new domain with a new component. However, by also taking into account the domain-specific constraints $R1$ – $R3$ from Section 2, our approach computes side effects for each of those options. Due to constraint $R2$, adding either $X1$ or $X2$ as second alternative target to $C1$ is not a valid adaptation as this would violate $R2$. Additionally, the existing references `prim` and `alt` from $C1$ to components of domain Y disallow the use of any components that belong to a domain other than Y, according to constraint $R1$. This rules out any remaining options that involve a second alternative receiver in domain Z or in a newly created domain. Finally, constraint $R3$ disallows $Y1$ and $Y2$ as options because they are already possible receivers. Note that this means a reduction from 12 options – from which 9 are actually semantically incorrect – to only 3 options that co-evolve $C1$ correctly. Those are drawn dotted in Fig. 1(b).

For the communication $C2$, the constraints $R1$ – $R4$ can only be satisfied by adding a new component to domain X that is used as second alternative receiver, as indicated by the dotted drawn component $X3$ in Fig. 1(b).

Change Execution Although each derived repair option fixes a model, some of them may seem more intuitive and more logical to stakeholders than others. Therefore, stakeholders should choose manually which of the available repair options should be executed. However, repair options could of course be selected and executed automatically if model characteristics such as readability are of low importance.

In our example, the co-evolution of $C2$ can be done automatically as there is only one repair option. To repair the inconsistency of $C1$, a user has to decide

between only three options that propagate the metamodel change correctly to the model.

4 Discussion

Let us now briefly discuss the planned implementation of our approach and preliminary validation results.

4.1 Prototype Implementation

The individual parts of the approach have been implemented in previous work. For the constraint management part, we have implemented a template-based transformation engine that generates and updates metamodel-based model constraints [11]. For the consistency checking, we rely on the Model/Analyzer consistency checking framework [12] that allows for efficient incremental addition and removal of models constraints. Finally, for the repair option generation, we have implemented a generic inconsistency repair mechanism that builds upon the Model/Analyzer framework [10].

4.2 Preliminary Performance Results

We have demonstrated in [13] that constraint management through transformation is efficient and that constraints are updated within milliseconds after a metamodel change. In [12] and [9], we have shown that the Model/Analyzer is capable of validating constraints instantly, even for large industrial models of over 100,000 model elements. Moreover, it was demonstrated that adding constraints (or removing them) is handled efficiently. For repairing detected inconsistencies, we have observed that for typical UML models less than 10 suggestions were derived, also within milliseconds [10]. Moreover, we have previously found that by considering side-effects between different constraints, the number of suggestions can be reduced even further [14].

4.3 Applicability

We have illustrated how our approach updates constraints and derives options for correcting co-evolution failures. Although we have used the proposed solution in isolation to keep the example simple and focused, it is compatible with existing automatic co-evolution techniques. When used in isolation, our approach detects the absence of necessary model adaptations as co-evolution failures. When combined with other approaches, it also detects co-evolution failures that are based on incorrect model adaptations. Therefore, our solution is not a substitute but a complement to existing technologies.

5 Related Work

Let us now discuss how the presented approach relates to other work that has been done in the field of co-evolving metamodels and models. The necessity of support for efficient and automatic co-evolution of metamodel and models was identified as a major challenge in software evolution by Mens et al. [4], and various approaches have been published to address it. Instead of seeing a metamodel evolution step as a single, complex, and manually performed change that is performed in an ad-hoc manner, Wachsmuth [15] describes metamodel evolution as a series of transformational adaptations performed stepwise. Metamodel changes are traced and qualified based on properties such as semantics- or instance-preservation. Co-transformations for models can be generated based on transformation patterns that are instantiated with the performed metamodel transformations. Cicchetti et al. [16] similarly classify metamodel and model changes. They identified dependencies between different kinds of modifications and propose an automated approach that leverages these dependencies for performing co-evolution automatically. Herrmannsdoerfer et al. [17] investigated to which degree different metamodel adaptations can be handled automatically. Note that those approaches focus on decomposing metamodel adaptations into atomic steps that are used for finding suitable co-adaptations of models. While our approach also relies on atomic metamodel modifications, we use those modifications for updating the conditions that must hold in a valid model. Our approach in general does not try to automate co-evolution of metamodels and models. Instead, the fully automated co-evolution of metamodels and constraints allows our reasoning engine to provide tool users with specific guidance on how co-evolution can be performed. Note, however, that in some cases models may also be adapted automatically (e.g., if only a single repair option exists).

Wimmer et al. [18] merge different metamodel versions to a unified metamodel and then apply co-evolution rules to models. New metaclasses are instantiated and existing model elements that are no longer required are removed. Due to issues regarding typecasts and instantiation, their co-evolution rules had to be adapted. The components used in our prototype implementation are capable of handling arbitrary metamodel adaptations, including type changes.

6 Conclusion and Future Work

In this vision paper, we have presented the outline of a novel approach for supporting the co-evolution of metamodels and models. Our approach is generic and relies on the detection of inconsistencies that occur after metamodel evolution. Those inconsistencies serve as input for a reasoning mechanism that provides as output a set of possible model adaptations for repairing – that is, co-evolving – an affected model.

The preliminary validation results are promising and suggest that the presented approach is feasible and that it can be implemented efficiently. However, these were observed in tests that were performed with prototype implementations for the individual components involved in the approach. For a complete

validation, we have yet to conduct case studies with industrial models and a complete implementation that fully integrates the prototypes of individual components.

References

1. D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.
2. R. B. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *FOSE*, pp. 37–54, 2007.
3. E.-J. Manders, G. Biswas, N. Mahadevan, and G. Karsai, "Component-oriented modeling of hybrid dynamic systems using the generic modeling environment," in *MBD/MOMPES*, pp. 159–168, 2006.
4. T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *IWPSE*, pp. 13–22, 2005.
5. L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Enhanced automation for managing model and metamodel inconsistency," in *ASE*, pp. 545–549, 2009.
6. M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "COPE - automating coupled evolution of metamodels and models," in *ECOOP*, pp. 52–76, 2009.
7. A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *EDOC*, pp. 222–231, 2008.
8. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002.
9. I. Groher, A. Reder, and A. Egyed, "Incremental consistency checking of dynamic constraints," in *FASE*, pp. 203–217, 2010.
10. A. Reder and A. Egyed, "Computing repair trees for resolving inconsistencies in design models," in *ASE*, pp. 220–229, 2012.
11. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Supporting the co-evolution of metamodels and constraints through incremental constraint management," in *MoDELS*, 2013. Accepted for publication.
12. A. Reder and A. Egyed, "Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML," in *ASE*, pp. 347–348, 2010.
13. A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Constraint-driven modeling through transformation," *Software and System Modeling*, 2013. DOI: 10.1007/s10270-013-0363-3.
14. A. Nöhrer, A. Reder, and A. Egyed, "Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track," in *ICSE*, pp. 864–867, 2011.
15. G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, pp. 600–624, 2007.
16. A. Cicchetti, D. D. Ruscio, and A. Pierantonio, "Managing dependent changes in coupled evolution," in *ICMT*, pp. 35–51, 2009.
17. M. Herrmannsdoerfer, S. Benz, and E. Jürgens, "Automatability of coupled evolution of metamodels and models in practice," in *MoDELS*, pp. 645–659, 2008.
18. M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel, "On using inplace transformations for model co-evolution," in *MtATL*, INRIA & Ecole des Mines de Nantes, 2010.

Automating Instance Migration in Response to Ontology Evolution

Mark Fischer[1], Juergen Dingel[1], Maged Elaasar[2], Steven Shaw[3]

[1]Queen's University, {fischer,dingel}@cs.queensu.ca
[2]Carleton University, melaasar@gmail.com
[3]IBM, sshaw@ca.ibm.com

Abstract

Of great importance to the efficient use of an ontology is the ability to easily effect change [9]. This paper presents an approach toward automating a method for instance data to be kept up to date as an ontology evolves.

1 Introduction and Motivation

As computers become more ubiquitous and the information they attain and store becomes vast, the benefit gained from formally representing that information grows. Ontologies are one of the key technologies which strive to give information well-defined meaning. This, in turn, allows computers and people to work more cooperatively [10].

While there are many applications which help develop and create ontologies, there are still very few which aid or facilitate the evolution of an ontology [7]. Changes in domain understanding and changes in application requirements often necessitate a change in the underlying ontology [11]. It may be impractical or impossible to predict how or even if an ontology may change after being deployed. Changes to an ontology may generate inconsistencies in dependent ontologies. As ontologies evolve and grow, it becomes increasingly attractive to find efficient ways of keeping dependent information up to date.

A change in an ontology requiring dependent ontologies to be updated is a common problem. IBM's Design Manager is a collaborative design management software. It works with domains which are specified using ontologies to store, share, search and manage designs and models. The work presented in this paper was inspired, in part, by work IBM has done and problems IBM has encountered while developing Design Manager. Figure 1 helps describe the problem being addressed.

In Figure 1, there exists an original ontology (O), an evolved or updated ontology (O') and a set of dependent ontologies (I_1, I_2, ..., I_n). For all j, I_j

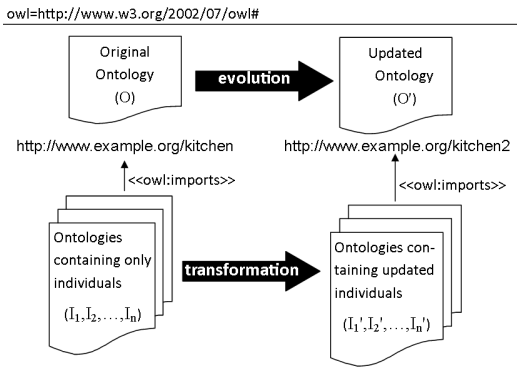


Figure 1: An overview of the problem setup

imports O and contains only individuals and facts asserted about those individuals.

Notice that classes and properties are kept separate from individuals. In an ontology, an individual is a resource which is an instance of a class and cannot contain any other resource (individuals may be related to one another through properties). Through this seperation, classes and properties are analogous to metamodels or schema while individuals are analogous to objects or data. Separating class definitions and property definitions from individuals is common practice when designing and building ontologies [3].

For any change that is made to O, there may exist an arbitrary number of j such that I_j is inconsistent. Furthermore, it is possible that there exist an a and a b such that a change in O will make I_a and I_b inconsistent for unrelated reasons (the axioms in O' that make I_a inconsistent are distinct from the axioms in O' that make I_b inconsistent). It is the hope that for all j, a transformation can coevolve or migrate I_j into a new ontology, I'_j, such that I'_j imports O' and remains consistent. In Figure 1, the arrow labeled 'transformation' represents the portion of this problem for which we propose a unique solution which automates the transformation process.

Currently, this work is often done manually, or put off entirely because of the effort required to devise and implement these sorts of migrations. Due to the complexity possible in an ontology and the semantic ambiguity behind changes being made, it is not possible to automatically generate the transformation that performs the migration for individuals in dependent ontologies. Since ontologies may be too complex for a user to fully comprehend and changes may be ambiguous and therefore difficult for a computer or algorithm to deal with [11], the solution is to automate the migration process with the help of some user input.

To help automate the migration process, we aid the user creation of a transformation which unambiguously defines how to perform a migration for any possible dependent ontology. We also develop a set of tools and techniques

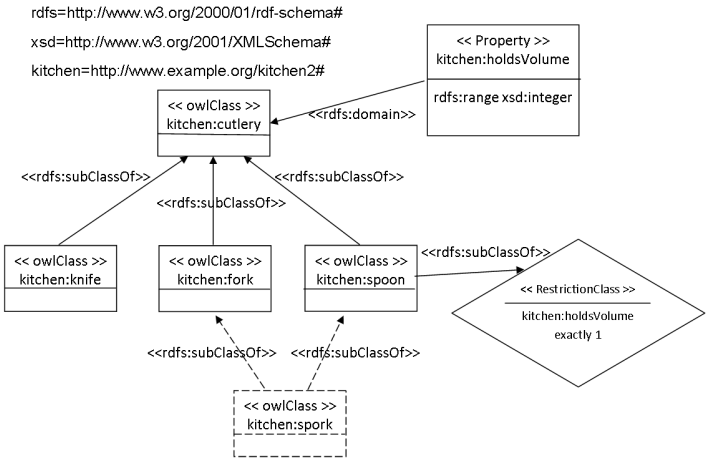


Figure 2: Kitchen example ontology and a possible evolution ontology. Solid components and their labels are from the original ontology while both solid and dashed components are from the evolved ontology. The original ontology has four named classes, one anonymous class, and a property. The evolved ontology has added a named class called *spork* which is a subclass of both *spoon* and *fork*. A spork is a pronged spoon and was first patented in the U.S. in 1874.

aimed at making it increasingly easier for users to create these transformations.

2 The Approach

The approach discussed here breaks the process into three phases which are incorporated into a single tool called Oital-T.

The first phase deals with comparing the two ontologies (O and O'). While not strictly necessary for the development of the transformation, an understanding of the difference between O and O' allows Oital-T to make suggestions and help the user to make choices regarding the transformation.

The second phase is the creation of the transformation itself. The transformation is authored in a language called Oital. Oital-T acts as an integrated development environment for Oital, allowing the syntax to be easier to learn and manipulate. The end goal of this entire process is to create a transformation written in Oital. This is the transformation that can then be run any number of times to migrate instances from O to O'.

Finally, the last phase involves analyzing the transformation. This phase of the process can be used to indicate which parts of the transformation may require correction or further development.

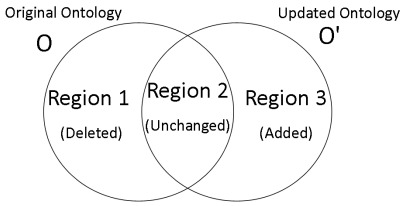


Figure 3: Venn Diagram showing regions where an axiom may be found

2.1 Analyzing the ontologies

In OWL, the separation of class and properties from individuals is expressed using axioms and facts. In relation to Figure 1, I_1 through I_n are all a series of facts with an inclusion reference to O while O and O' both consist of a series of axioms (possibly with inclusion references to other ontologies).

Axioms may either restrict or add to an ontology. Restrictive axioms add further clarification to the information encoded in an ontology (such as *holdsVolume exactly 1* in Figure 2) while additive axioms broaden the scope of the information (such as the creation of a new class).

Figure 3 depicts three regions in which an axiom may exist. Region 1 describes axioms which can be found in O and not in O' (deleted axioms), region 2 describes those axioms which can be found in both O and O' (unchanged axioms), and region 3 describes axioms which can not be found in O but can be found in O' (new or added axioms).

Axioms may influence one another. In Figure 2, the kitchen ontology has an axiom (A) stating that *cutlery* is the domain of the property *holdsVolume*. It also has the axiom (B) that there is a specific anonymous restriction class which contains only individuals that hold exactly one amount of volume. Axiom B is influenced by axiom A because a change in A is axiomatically a change in B as well. For any axiom, C, the axiom and all other axioms it is influenced by is denoted as C^* and called that axiom's context.

Table 1 gives an overview of how axioms are labeled while analyzing O and O'. If there exists axiom A such that its context, A^*, is entirely in Region 3, then A cannot be the cause of any inconsistencies. Specifically, if I_j is consistent with respect to O, then any inconsistencies I_j has with respect to O' cannot be due to axiom A. This means that aspects of the updated or evolved ontology which are entirely unique to the updated ontology are not of concern for the creation of a transformation. Similarly, for any A such that A^* is entirely in region 2, A cannot cause inconsistencies. This means that those parts of the original and updated ontologies which remain unchanged are not of concern either.

Axioms found in Region 1 may not, by default, be ignored. If the axiom is restrictive, such as the restriction class in Figure 2, then its removal cannot create an inconsistency and is considered OK. If, however, the axiom is additive, then its removal may create inconsistencies. For example, removing the *fork is a subclass of cutlery* axiom from the kitchen ontology in Figure 2 would result in

	Restrictive Axiom (and its context)	Additive Axiom (and its context)
Region 1	OK	Must Investigate
Region 2	OK	OK
Region 3	OK	OK
Region 1 & 2	Should Investigate	Must Investigate
Region 2 & 3	Must Investigate	Should Investigate

Table 1: Overview of which axioms require further investigation with respect to specific regions as outlined in Figure 3. *OK* means the axiom may be safely ignored as it is very unlikely to hinder a migration. *Should Investigate* means this axiom is unlikely to hinder migration, but it is probably semantically important. *Must Investigate* means this axiom needs to be considered carefully during migration as it may cause inconsistencies.

any facts which state that an individual fork holds a specific volume becoming inconsistent.

If, for a given axiom, A, the context, A^*, straddles two regions, then A is very likely to be semantically important. This means that even if the axiom may not cause inconsistencies, it may still be of interest to a user creating the transformation. Generally A^* having elements in region 1 and 2 suggests that a part of the ontology was removed but in such a way that the updated ontology had to be restructured to accommodate the removal (such as the removal of a class from a hierarchy). Similarly, A^* having elements in both region 2 and 3 suggests that a part of the ontology that has been added affects parts of the already existent ontology (Such as the creation of a new restriction class in a class hierarchy). Some of these changes may cause inconsistencies and are treated as more important, but all instances of these are shown to the user.

The following example illustrates how this analysis in performed. Consider the kitchen ontologies from Figure 2. The following is information that can be gathered from an analysis of these two ontologies. The *spork* class is new and therefore part of Region 3 from Figure 3. Its context includes the *fork* and *spoon* classes because of the subclass property. Because of the *spoon*, its context also involves the anonymous restriction class as well as the property *holdsVolume*. Because of *holdsVolume*, its context involves the *cutlery* class. Since *spork* is from Region 3, and it has a context which encompasses Region 2, this change straddles Region 2 & 3. The creation of a named class is additive, so (as shown in Table 1) this is unlikely to create an inconsistency during migration, but should still be brought to the user's attention as it may be important semantically.

2.2 Creating the transformation

A transformation that facilitates a migration of individuals from one ontology to another must, in some way, encode all the information that is still lacking after the two ontologies have been fully analyzed. While an analysis of the original

and evolved ontology can reveal how the original ontology has been changed, it cannot uncover the reasoning behind any given change.

A comparison of O and O' may uncover a difference for which there are many possible reasons. Consider the analysis of the evolution depicted in Figure 2, to know how to proceed, it is import to know why the *spork* class was created. If, for example, the class was added because of the discovery of a spork, then the migration can simply ignore this difference. If, on the other hand, the spork class was created because there were individuals within the class *fork* which held a volume, then a migration should take those forks which have the *holdsVolume* property and migrate them to the new *spork* class.

This very simple example illustrates why user interaction is required for the creation of the migration transformation. To express this transformation, we developed a domain specific transformation language called Oital.

2.2.1 Ontology Individual Transformation Authoring Language

Oital is a transformation language designed specifically for specifying the migration of individuals who are conformant to O such that they become conformant to O'. Currently, SPARQL update is used to perform tasks which alter an ontology. SPARQL update, however is an update language for RDF graphs. To use SPARQL update to effect change on an ontology requires an understanding of how ontologies are stored as RDF triples. Ontologies which are not stored in an RDF triple store must first be converted to one before SPARQL update can be used.

Oital alleviates these issues by using ontology concepts directly in the language (thus abstracting away from RDF). Instead of querying triples in a triple store the way SPARQL update does, Oital queries classes and properties in the ontology.

The current Oital compiler compiles Oital code into SPARQL update code. This means that ontologies queried using Oital must still be stored in RDF. It is, however, possible for Oital to be compiled in such a way that it can efficiently query and alter other formats as well. Details concerning the syntax and structure of Oital have been omitted due to space requirements.

To help make Oital easier to adopt, the syntax was heavily influenced by the Manchester OWL syntax. Oital transformation classes are defined using a syntax similar to the way OWL classes are defined. As the Manchester Owl Syntax has been adopted by World Wide Web Consortium [1], it should be fairly familiar to users who are already creating, updating, and evolving ontologies.

So far, no extensive evaluation of Oital has been conducted. However, preliminary results suggest that it is a convenient means of expression for the kind of ontology migration transformations we target.

2.3 Analyzing the transformation

Analyzing the created transformation is a helpful step toward inspecting if the transformation that has been created is correct. Currently, the only forms of

analysis supported by Oital's IDE (Oital-T) are traditional testing and a form of abstract interpretation which executes the transformation in some abstract fashion while collecting specific information [5][5].

We have developed and implements an abstract interpretation of Oital to track class membership. For instance, if after running an abstract interpretation of the transformation, the result shows that a specific class – such as *spoon* – is empty, then there exists no possible input such that any individuals from that input will have *rdf:type spoon*. Depending on the intent of the transformation, this may or may not be a desired result.

Traditional testing, abstract interpretation, and – in future – other forms of analysis may be used throughout the transformation authoring process in order to ease the development of complex transformations with fewer errors.

3 Related Work

The migration problem, as presented here, closely resembles the data migration problem found in database work [4] as well as the co-evolution problem found in model driven development (MDE) work [2].

Using a transformation language is an approach currently being used to solve the co-evolution problem. Languages such as ATL [6] (a QVT [8] compliant language) can be used to facilitate the automation of co-evolution in model-driven engineering [2]. When considering instance migration in ontologies, Oital is used in a similar capacity (in the domain of ontologies).

Not much work has been done on the migration problem for ontologies as has presented here, but similar work can be found concerning ontology evolution management. Much of the work done in this field, such as [11] relies on being able to access dependent ontologies (I_1, I_2, ..., I_n in Figure 1). They also depend on user involvement at the time of evolution or migration, which our approach does not require.

4 Conclusions and Future Work

In this paper, we have described an approach to facilitate the development of transformations that migrate individuals from the original ontology to an updated one. The approach is based on 1) differencing the ontologies, 2) transformation development using a novel, domain-specific language, and 3) analysis.

There is, however, still much outstanding work to be done. Along with continued development on Oital's IDE (Oital-T), we will attempt to identify transformation patterns and additional analysis that are scalable, yet still provide developers with useful information.

Aiding the development process will be sufficiently complex case-studies which shall help show the usability and capability of the approach presented in this paper. IBM has given us multiple versions of an ontology encoding of the UML specification along with a series of UML models stored as ontologies.

IBM has also offered access to a domain with IBM's Design Manager which may benefit from this approach.

References

[1] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Owl web ontology language reference. February 2004.

[2] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE*, pages 222–231, 2008.

[3] A. Gangemi and V. Presutti. Ontology design patterns. *Handbook on Ontologies*, pages 221–243, 2009.

[4] J. Hall, J. Hartline, R. A. Karlin, J. Saia, and J. Wilkes. On algorithms for efficient data migration. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2001.

[5] N. Jones and F. Nielson. Abstract interpretation: A semantics-based tool for program analysis. *Semantic Modeling, Clarendon Press, Handbook of Logic in Computer Science*, 4:527–635.

[6] F. Jouault, F. Allilaire, J. Bezivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *Object-oriented Programming Systems, Languages, and Applications*. 21st ACM SIGPLAN Symposium, 2006.

[7] A. M. Khattak, Z. Pervez, S. Lee, and Y. Lee. After effects of ontology evolution. In *Future Information Technology*. IEEE 5th International Conference, 2010.

[8] I. Kurtev. State of the art of qvt: A model transformation language standard. *Applications of Graph Transformations with Industrial Relevance. Springer Berlin Heidelberg*, pages 377–393, 2008.

[9] N. F. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 6(4):428–440.

[10] P. Plessers, O. De Troyer, and S. Casteleyn. Understanding ontology evolution: A change detection approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(1):39–49, 2007.

[11] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pages 285–300, 2002.

Generating Edit Operations for Profiled UML Models

Timo Kehrer, Michaela Rindt, Pit Pietsch, Udo Kelter

Software Engineering Group
University of Siegen
{kehrer,mrindt,pietsch,kelter}@informatik.uni-siegen.de

Abstract. A recurring challenge in constructing tools for model-driven engineering (MDE) is to guarantee the consistency of models because many MDE tools can process only consistent models. Consistency must be guaranteed by all tools which modify models, e.g. model editors, transformers, merge tools, etc. An obvious solution of this challenge is to use a common library of consistency-preserving edit operations for modifying models. Typical meta-models lead to several 1000 edit operations, i.e. it is hardly possible to manually specify and implement so many edit operations. This problem is aggravated by UML profiles: stereotyped model elements are implemented as complex data structures. This paper discusses several approaches how to implement edit operations on profiled models and how to generate complete sets of specifications and implementations of edit operations.

1 Introduction

In model-driven engineering (MDE) models are the primary development artifacts. Models are modified by many different tools, including model editors, transformers, patch and merge tools [7], test data generators [11] etc. Many tools can process only consistent models; tool constructors are thus faced with the challenge that all tools must preserve the consistency of the models.

An obvious solution to this challenge is to use a common library of edit operations, more precisely a common set of specifications and implementations of consistency-preserving edit operations. Section 2 will discuss our notion of consistency in greater detail.

Consistency-preserving edit operations (CPEOs) depend on the meta-model, i.e. they are not generic. The set of CPEOs available for a given meta-model must be *complete* in the following sense: it must be possible to construct any consistent model and to edit each consistent model to become another consistent model. Complete sets of CPEOs are quite large; their creation should therefore be supported by (meta-) tools. Section 2 introduces SERGE, our own tool to generate CPEOs, and discusses the limitations of automatically generating sets of CPEOs.

The challenge of defining CPEOs is aggravated in case of domain-specific modeling languages (DSMLs) which are defined using the UML profile mecha-

nism, e.g. SysML [9] and MARTE [10]: Stereotyped model elements are implemented as complex data structures, which create new consistency requirements. This issue is discussed in Section 4.

One main contribution of this paper is an analysis of how profiles affect (sets of) CPEOs of the underlying non-extended meta-model. This is discussed in Section 5.

Several approaches are available for implementing CPEOs for profiled meta-models and for semi-automatically generating complete sets of them. Section 6 presents these approaches and discusses their pros and cons.

Section 7 summarizes the contributions of this paper.

2 Consistency-preserving Edit Operations

Meta-Models. A model is conceptually considered as a typed, attributed graph which is known as the abstract syntax graph (ASG). Meta-models, e.g. the UML meta-model, define the types of nodes and edges allowed in an ASG. From a tooling point of view, meta-models must be implemented using a suitable technology. Following the common practice in MDE, we assume an object-oriented implementation of meta-models in this paper. Thus, nodes and edges of an ASG are represented by (runtime) objects and references between them. We distinguish containment and non-containment references.

Consistency of Models. An ASG is considered consistent if complies to the definitions and constraints specified by its meta-model, notably constraints concerning hierarchies, relationships and multiplicities. Constraints are typically specified using the OCL [8].

Standards such as the UML typically define strict, "ideal" consistency constraints; ASGs complying with them represent models which can be translated to source code and other platform-specific documents. Many model editors (e.g. the Eclipse UML Model Editor) de facto use less strict meta-models. For example, they enforce only multiplicity constraints which are required to produce a graphical representation of a model.

Edit Operations vs. Basic Graph Operations. Meta-models are just data models of models, they do not directly specify editing behavior. One obvious approach to modify models is to use 'basic graph operations' on the ASG including *deleting*, *creating*, *moving* and *changing* elements, their attributes or references.

Executing a single basic graph operation on an ASG can lead to a new state which violates the syntactic consistency. For example, a basic operation which creates a single UML *StateMachine* object leads to an inconsistent ASG: A *StateMachine* must always contain at least one *Region*. In contrast to this, a CPEO will create a *StateMachine* and create a contained *Region* at the same time. Thus, the model is transformed from one consistent state into another. This CPEO is *minimal* in the sense that it cannot be split into smaller parts which preserve the consistency of the model.

In general, an edit operation has an interface specifying input and output parameters. For example, the edit operation createStateMachine must be supplied with a container element in which the *StateMachine* is to be created, along with local attribute values for the new *StateMachine*. Objects created by an edit operation are handled as output arguments. Additionally, pre- and postconditions may precisely specify application conditions of an edit operation.

Basically, minimal CPEOs can be categorized into the same kinds of operations as basic graph operations, namely *creation*, *deletion*, *move* and *change* operations. However, minimal CPEOs usually comprise a set of basic ASG operations in order to create (or delete) 'atomic compositions' of ASG elements, such as a *StateMachine* with at least one nested *Region*, in a single transaction.

3 Generating Executable Specifications of Edit Operations

Comprehensive languages such as the UML require large sets of edit operations. The manual specification of such sets of CPEOs for a given modeling language is very tedious and prone to errors.

This problem is addressed by our *SiDiff Edit Rules Generator (SERGe)* [12]. SERGe derives sets of minimal CPEOs from a given meta-model with multiplicity constraints. These sets are complete in the sense that all kinds of edit operations, i.e. create, delete, move and change operations, are contained for every model element, reference and attribute. Edit operations generated by SERGe respect multiplicity constraints and maintain the consistency of a model in this sense.

No other approaches for constructing CPEOs are known to us. There are approaches to create certain kinds of edit operations or grammars which can construct or modify models [1, 3, 6, 13]. However, they either do not support all types of modifications (e.g. *setting attribute values* or *moving elements*) or they lead to consistency violations. In sum, they were unusable for our own practical work.

SERGe uses the Eclipse Modeling Framework (EMF) [4]. Generated edit operations are realized as in-place transformation rules in the model transformation language Henshin [2, 5]. A Henshin transformation rule can specify model patterns to be found and preserved, to be deleted, to be created and to be forbidden. We will refer to these implementations of edit operations as *edit rules*.

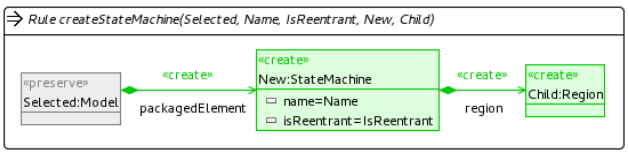


Fig. 1. Edit rule createStateMachine generated by SERGe

Figure 1 shows the edit rule createStateMachine. This edit rule is generated by SERGe when processing the implementation of the UML2 meta-model in EMF Ecore. For the sake of readability, the rule shown in Figure 1 is a simplified version of the rule actually generated.

This example illustrates that a Henshin rule can define variables serving as input or output parameters. The input parameter *Selected* determines the context object to which an edit rule shall be applied. In our example, the context object will be the *Model* in which the *StateMachine* shall be created. The *StateMachine* object and its mandatory *Region* will be returned as output parameters *New* and *Child* when the rule is applied. Additionally, the *name* and the property *isReentrant* of the *StateMachine* to be created must be provided by the input value parameters *Name* and *IsReentrant*, respectively.

Manual Adaptions of Generated Operations. SERGe is not capable of interpreting arbitrary well-formedness constraints (OCL Constraints) attached to a meta-model element. Hence, some of the generated edit rules have to be complemented by additional application conditions.

For example, the UML meta-model specifies that all the members of a *Namespace* (which includes the packaged elements of a *Model*) are distinguishable by their names. Thus, a *StateMachine* can only be created in a *Model* if there is no *NamedElement* with the same name. This precondition can be implemented in Henshin by a negative application condition (NAC) as shown in Figure 2.

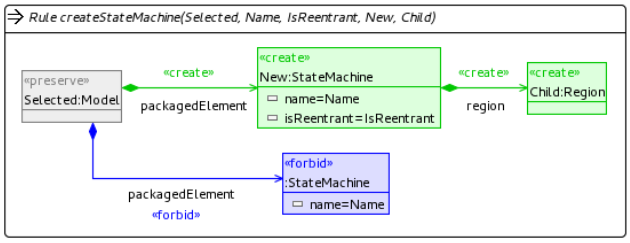


Fig. 2. Manually adapted edit rule createStateMachine

Conclusion. We can conclude that the most important innovation of SERGe is that the generated edit operations comply to multiplicity constraints and preserve the consistency of a model in this respect. If required, the generated edit operations can be extended manually additional well-formedness constraints; this usually requires only a limited effort.

However, SERGe currently does not address the generation of edit operations for UML profiles. Several design variants to extend a generator such as SERGe to profile definitions will be discussed in the remainder of this paper.

4 The UML Profile Mechanism

UML profiles provide a light-weight approach to implement domain-specific languages by reusing existing meta-models.

Profile Definition. Profiles are basically defined as follows: A profile must import the base meta-model which contains the reused element types. Principally, any MOF-based meta-model can be extended by a profile definition. In practice, the UML meta-model serves as the base meta-model. Figure 3 shows an excerpt of the SysML [9] profile definition. The profile defines stereotypes which extend one or more of the imported element types. These extended classes are called 'meta-classes'. The attribute *required* of a meta-class extension defines whether an instance of the extending stereotype must be attached to any instance of this meta-class. A stereotype can have new attributes or references, which are called 'tagged values'.

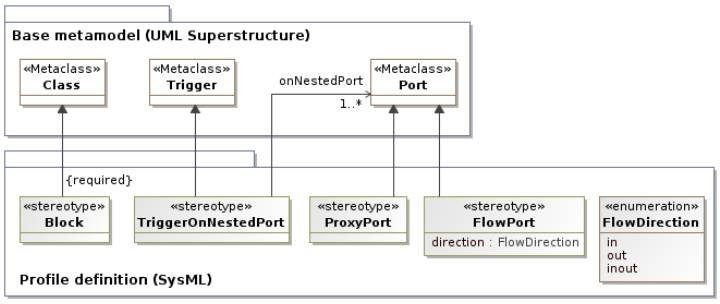


Fig. 3. SysML-Profile excerpt

Profile Application. A profile can be applied to a model which is an instance of the base meta-model, and later be revoked. The application or revocation of a profile can also be regarded as an edit operation: it causes stereotypes to be added to or to be removed from appropriate model elements of a model.

The UML profile mechanism is designed in a way that all data related to a profile are separated from the extended model; i.e. one or more profiles can be applied to a UML model without destroying its previous structure. Thus, the extended UML model always remains processable by UML tools.

An example of a profile application is shown in Figure 4; a very simple SysML model is shown in concrete syntax (left) and abstract syntax (right). It illustrates how stereotype objects of type *Block* and *FlowPort* (colored in light gray) are attached to instances of the meta-classes *Class* and *Port*, respectively.

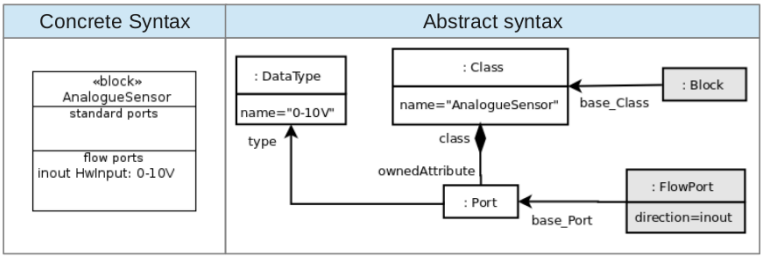


Fig. 4. Sample SysML model in concrete syntax (left) and abstract syntax (right)

5 Edit Operations for Profiled UML Models

In this section we analyze how UML profile definitions influence the set of CPEOs. The complete set of edit operations can be divided into four disjoint subsets, which are described below.

A. Edit Operations modifying UML Model Elements. The first set of edit operations operates on model elements that are instances of meta-classes of the base meta-model, i.e. the UML meta-model. For the sake of readability, we refer to them as (pure) **UML model element operations**. In principle, no knowledge about applicable profiles is required to generate these operations.

Nevertheless, these model elements can have stereotypes. Although the effect of a UML model element operation does not depend on whether or not there is a stereotype, it can be necessary to omit or rename some of the generated edit operations, notably in case of *required* stereotypes. Thus, we further divide the set of UML model element operations into two main subsets:

1. Edit operations which *create* or *delete* UML model elements and their containment references.
   These edit operations must be omitted if the type of the involved model element has a *required* stereotype in the profile definition. In this case, the base model element and the stereotype object have to be handled consistently by hybrid edit operations (s. Section 5.D). Other edit operations which modify the containment hierarchy, notably relocations of model elements caused by *move* operations, are not omitted.
2. Edit operations which *change* attributes or non-containment references of model elements. They are independent on whether or not a profile is applied. It can be helpful to rename these operations for the sake of understandability, e.g. from setClassIsAbstract to setBlockIsAbstract.

In both cases, edit operations are omitted if they operate on instances of meta-model elements which are excluded by the profile definition. For example, SysML does not support interaction diagrams as defined in UML2. Thus, edit operations modifying elements of type *Interaction*, *InteractionFragment* or *Lifeline* are omitted.

B. Edit Operations modifying Tagged Values. This subset comprises edit operations which *change* attributes or references of stereotypes. For the sake of simplicity, we refer to this subset as **tagged value modifications**. These edit operations modify only parts of a model, specifically the stereotype instances of the profile. They can have arguments whose type is defined by the base meta-model; these arguments remain unchanged.

Tagged value modifications are easy to define and implement if their domain is a primitive data type. For example, the edit operation setBlockIsEncapsulated simply sets the tagged value *isEncapsulated* of a SysML *Block*. Tagged value modifications are more complicated if they change references of stereotype objects. Here, multiplicity constraints which define mandatory neighbours or children must be taken into account. These cases can be handled in the same way as modifications of references on UML model elements (see Section 2).

C. Edit Operations for Stereotype Application. Stereotype applications apply or revoke a stereotype at/from a UML model element. From a users' point of view, this can appear as a conversion of a model element to another type. On the ASG level, a stereotype application is an edit operation that *creates* or *deletes* a stereotype object together with the reference to an instance of its base meta-class.

These edit operations are not permitted for stereotypes that are declared as required for their base meta-class. In such cases they are omitted from this set of edit operations. For example, the stereotype application operations applyStereotypeBlock and unapplyStereotypeBlock are not applicable to SysML models. The stereotype *Block* must always be created/deleted together with an instance of its base meta-class *Class*, but not without. This kind of modification is achieved by a hybrid edit operation.

D. Hybrid Edit Operations for Required Stereotypes. Hybrid edit operations concurrently modify instances of base meta-classes *and* stereotype instances. Typically, they *create* or *delete* model elements that have required stereotypes. An example is given in Section 5.C; creating a SysML *Block* requires the creation of a UML *Class* to which it must be attached as stereotype.

6 Generating Edit Operations

The previous section introduced 4 different kinds of edit operations. The complete set of operations, which is necessary to consistently edit profiled UML models, is the union of these 4 sets and represented by the dotted rectangle in Figure 5.

The edit operations specified in subsections 5.A can be generated by SERGe. The same is true for edit operations specified in subsections 5.B and 5.C because a profile definition is handled as a usual MOF-based meta-model.

However, some extensions are necessary for the generation of hybrid edit operations (Section 5.D). Generally, hybrid edit operations can be implemented

using two different approaches: a *higher-order transformation approach* and a *meta-model driven approach.*

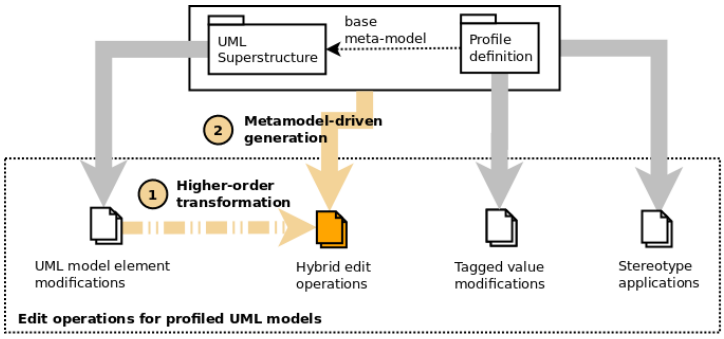


Fig. 5. Generation of different types of edit operations

A. Higher-order Transformation Approach. This approach uses a set of existing UML edit operations as primary input (see arrow 1 in Figure 5). It does not matter whether the edit operations have been generated and/or constructed manually. The basic idea is to modify them by adding appropriate stereotypes.

We assume that edit operations are defined as executable specifications in the form of Henshin transformation rules as explained in Section 2. Henshin rules are technically represented as models and can be transformed automatically, i.e. their modification can be considered as a higher-order transformation (HOT).

We have implemented this HOT in Henshin as follows: Basically, stereotypes as defined by the profile are applied to all instances of a UML base meta-class which occur in the given set of UML edit rules. A parametrized HOT rule is provided in order to attach an instance of a stereotype, which is given as first rule parameter, to all instances of a UML meta-class (second rule parameter) which occur in the given set of UML edit rules. We have implemented three variants of this HOT rule: they attach stereotype objects to UML model elements which are (1) to be created, (2) to be deleted and (3) to be preserved by an edit rule. A working example of a HOT rule can be found at [12].

Since a UML meta-class can be extended by several stereotypes, different variants for every stereotype will be created. Such variants can also consist of combinations of multiple stereotypes, if more than one meta-class is contained in the given UML edit rule. Thus, the scheduling algorithm which applies the HOT rules with appropriate invocation arguments is responsible for generating all possible combinations of stereotype attachments.

A big advantage of this approach is that efforts of manual adjustments on UML edit operations are not lost. A disadvantage is that it supports only simple profiles: It cannot handle stereotypes which have mandatory references to other

stereotypes or UML model elements. However, important profile-based standards such as SysML and MARTE rarely contain such scenarios.

B. Metamodel-driven Approach. This approach does not require any previously generated sets of edit operations, all operations are generated from scratch; the profile and the base meta-model are the only input data here (see arrow 2 in Figure 5).

This approach can consider multiplicity constraints of (non-) containment references which emanate from stereotypes. However, in contrast to the HOT approach, all manual adjustments on an existing set of UML edit operations (modifications, creations and deletions) are lost. Manually created edit operations for the base language are not automatically adapted.

Two alternative patterns are available for implementing a hybrid operation; Figures 6 and 7 illustrate them using the creation of a SysML *Block* as an example:

1. **Sequentially boxed operations** (Figure 6): Here, the UML edit operation and the profile application operation are applied in sequential order; Initially, a class instance is created. Then the required stereotype is added in a separate step.
2. **Concurrent operation** (Figure 7): Basically, all edit operations used separately in the first approach are 'merged' into one. For Henshin transformation rules, such a merge can be implemented by concurrent rule construction.

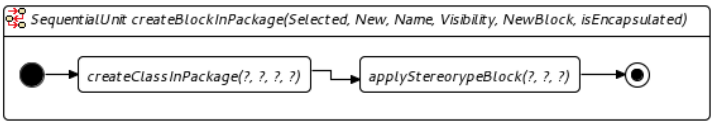


Fig. 6. Implementing a hybrid edit operation using a sequential transformation unit

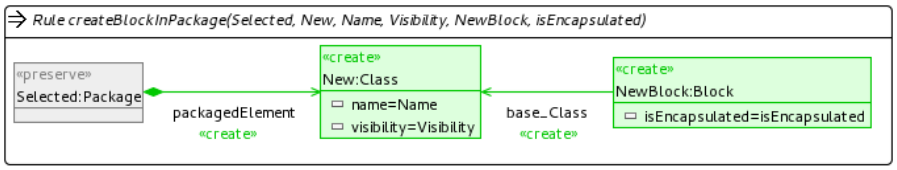


Fig. 7. Implementing a hybrid edit operation using a single transformation rule

7 Conclusion

This paper has presented our approach to construct complete sets of consistency-preserving edit operations on models, Existing approaches for generating edit operations do not support consistency preservion and DSMLs using UML Profiles such as SysML or MARTE, which are important DSML standards for embedded systems.

We addressed an important practical requirement: how to consistently maintain operation sets for the base language without profiles and for the profiled language. Our solution is based on a clear separation of all edit operations in 4 categories. The most complex type of edit operations are hybrid ones; we showed that they can be implemented in such a way that manual optimizations of the base edit operations can be preserved.

Acknowledgement. This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

References

1. Alanen, M.; Porres, I.: A relation between context-free grammars and meta object facility metamodels; Technical Report 606, TUCS Turku Center for Computer Science; 2003;
2. Arendt, T.; Biermann, E.; Jurack, S.; Krause, C.; Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations; in: Proc. MoDELS 2010, Oslo; LNCS 6394, Springer; 2010
3. Ehrig, K.; Küster, J.M.; Taentzer, G.; Generating instance models from meta models; SoSym Volume 8:4, p.479-500; 2009
4. EMF: Eclipse Modeling Framework; http://www.eclipse.org/emf; 2012
5. EMF Henshin Project; http://www.eclipse.org/modeling/emft/henshin
6. Hoffmann, B.; Minas, M.: Generating instance graphs from class diagrams with adaptive star grammars. Intl. Workshop on Graph Computation Models, 2011
7. Kehrer, T.; Kelter, U.; Taentzer, G.: Consistency-Preserving Edit Scripts in Model Versioning; in: Proc. 28th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE 2013); ACM; 2013
8. Object Constraint Language: Version 2.0; OMG, Doc. formal/2006-05-01; 2006
9. Systems Modeling Language: Version 1.3; OMG, Doc. formal/2012-06-01; 2012
10. UML Profile For Marte - Modeling And Analysis Of Real-time Embedded System: Version 1.1; OMG, Doc. formal/2011-06-02; 2011
11. Pietsch, P.; Shariat Yazdi, H.; Kelter, U.: Generating Realistic Test Models for Model Processing Tools; p.620-623 in: Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11); ACM; 2011
12. The SiDiff EditRule Generator - A tool to automatically derive consistency-preserving edit operations of any ecore meta model; http://pi.informatik.uni-siegen.de/Mitarbeiter/mrindt/SERGe.php; 2012
13. Taentzer, G.: Instance Generation from Type Graphs with Arbitrary Multiplicities; in: Electronic Communication of the EASST 47; 2012

Evolution of Model Clones in Simulink

Matthew Stephan, Manar H. Alalfi, James R. Cordy, and Andrew Stevenson

Queen's University,
Kingston, Ontario, Canada
{stephan,alalfi,cordy,andrews}@cs.queensu.ca

Abstract. A growing and important area of Model-Based Development (MBD) is model evolution. Despite this, very little research on the evolution of *Simulink* models has been conducted. This is in contrast to the notable amount of research on UML models, which differ significantly from *Simulink*. Code clones and their evolution across system versions have been used to learn about source-code evolution. We postulate that the same idea can be applied to model clones and model evolution. In this paper, we explore this notion and apply it to *Simulink* models. We detect model clones in successive versions of MBD projects and, with a new tool, track the evolution of model clones with respect to their containing clone classes. When there is a change in classification of a model-clone, we investigate what specifically evolved in the model to cause this classification change.

Keywords: model evolution, model clone detection, model clone evolution, Simulink

1 Introduction

Understanding software model evolution in **Model-Based Development** (MBD) is important as it can improve our ability to adapt to change, allows us to refactor more efficiently, and increases the quality and amount of analysis we can do on MBD projects. While still a relatively young area, there is a notable amount of work that discusses the evolution of UML models [5, 8, 10]. In contrast, there is very little research related to the evolution of *Simulink* models, a data-flow modeling language that is widely used in the automotive and communication industries, as well as other embedded areas. *Simulink* models differ significantly from anything in UML, with the most analogous diagram being a UML activity diagram, which is still quite different.

As noted in [13], there are a number of instances where code clones are used in order to perform source-code evolution analysis. Specifically, once a relationship can be established between two versions of a system, it can be employed as a means to understand the evolution of the system. Such a relationship can be realized by extracting code clones from different versions and then identifying and analyzing similar groups and how they have changed. We believe the same holds true for models. Thus, as a first step towards understanding *Simulink*

model evolution, we introduce the notion and use of *Simulink* **model clone evolution** (MCE). We focus on *Simulink* because there is sparse research on its evolution, it is of interest to our industrial partners, and *Simulink* **model clone detection** (MCD) is the most mature form of MCD.

In this paper, we use our near-miss MCD tool, *SIMONE* [1], to detect clones from successive versions of both industrial and publicly available models. Using a new tool we develop, we are able to track the evolution of a model-clone class' clone instances throughout multiple versions. When there is a change in the classification of the model-clone instance, we then delve deeper into the model itself to see and illustrate what exactly has evolved that caused this change and then explain it. The paper begins by providing background in Sect. 2 and defining key terms in Sect. 3. We then present the tool we developed for this work in Sect. 4 and our experiments with it, along with examples of models and their evolution, in Sect. 5. We present related and future work in Sect. 6 and conclude in Sect. 7.

2 Background

2.1 Simulink

Simulink models consist of 3 levels of granularity: whole models, (sub) systems, and blocks. Models contain systems, and systems contain other (sub) systems and blocks. This is similar to source files: models are like programs; systems are like methods, functions, and classes; and blocks are like statements in traditional programming languages. An important characteristic of Simulink models is "All block names in a model must be unique and must contain at least one character." [1]

2.2 Clone Genealogies

Kim et al. [9] define the notion of genealogy for code clone groups as the way in which a collection of clones evolves over multiple versions of a system. The clone group evolution they describe is in terms of code snippets, which are comprised of both text and location. For Kim et al.'s genealogies, each code clone group contains identical (exact) code clones which are matched to other clone groups based on textual similarity. Saha et al. [14] later consider clone groups containing non-identical (near-miss) code clones and match them to other groups by matching functions (code blocks) containing the code clones. We extend and modify the concepts and approaches from these works in order to apply them to *Simulink* models.

2.3 Model Clone Detection

Much like its counterpart, code clone detection, model clone detection entails discovering identical or similar fragments of model elements [4]. We recently

[1] http://www.mathworks.com/help/simulink/ug/changing-a-blocks-appearance.html

developed a model clone detector, called *SIMONE*, that is capable of detecting both exact and near-miss clones in *Simulink* models [1]. Its clone detection algorithm uses a sorted and filtered version of the underlying internal textual representation of the models stored in the Simulink MDL files. This is in contrast to *CloneDetective* [4], which treats *Simulink* models as graphs. Both techniques identify clones and group them together into clone classes, however, in *SIMONE*, a user-specified similarity threshold can be specified, such as 70% for near-miss and 100% for exact clones. There are other less-mature MCD techniques as well [17].

3 Definitions

A *Simulink* clone is essentially a similar subgraph of a larger *Simulink* system and is comprised of *Simulink* blocks (including sub-system blocks) and the lines that connect them. The basic units in our model-clone genealogy are these subgraphs, which we term **model clone instances** (MCIs). The attributes of an MCI are its list of blocks and lines, and its location. Location refers to the specific *Simulink* model and system(s) the MCI is contained in. For example, both *CloneDetective* and *SIMONE* produce XML clone reports that contain this information in some form. A **model clone class** (MCC) is a collection of MCIs grouped together by a model clone detector based on some measure of classification. All the *Simulink* MCD tools we have encountered thus far identify clone classes explicitly.

In order to trace a specific MCI across different versions, we can use the combination of (1) the model containing the MCI, and (2) the fully qualified path to the system (or sets of systems, for clones that span systems) comprising the MCI, that is, the trail of enclosing Simulink (sub) systems that contain the MCI's blocks and lines. Because all blocks, including those of type "subsystem" must have unique names in a *Simulink* model, this is a suitable source of clone traceability. This is analogous with Saha et al.'s code clone mapping where they determine if a code clone fragment is located within a function.

As Saha et al. have noted, with near-miss clones it is not possible to simply map one class to another in successive versions [14]. Analogously to what they do with functions and code clone classes, we extend their ideas to the modeling domain by taking a specific MCC, say MCC_v from version v, and seeing what MCCs in future versions contain MCIs from MCC_v. In contrast, however, while they are interested primarily with counting occurrences of code-clone evolution patterns, we are concerned mainly with how individual model clone instances evolve to cause a change in MCC classification. As such, we need to identify only if MCC_v yields one MCC in a future version, v+1; multiple MCCs in version v+1; or no MCCs in version v+1; and focus on the specific evolution of the MCIs involved in each case.

4 Tool Description

For the first step of our analysis, we developed a tool, called **Simulink Clone Class Tracker** (SIMCCT), that allows a user to select a specific MCC from one version of a system in order to display, in a GUI, what MCCs in future versions contain its MCIs. As input, the program takes in an ordered set of MCD results in XML form, with each XML file representing a different version in the evolution. We used a TXL [3] source transformation to change the XML output of *SIMONE* to a form more conducive to evolution analysis. The same can be done for *CloneDetective*. As demonstrated in a simplified version of the input in Fig. 1, the file contains a list of clones, sorted by classes. Each class contains sources, which correspond to the MCIs. As mentioned previously, these are comprised of blocks and lines, each with their own attributes of interest.

In brief, SIMCCT begins by parsing the input XML file we describe above and extracting the required information, treating each file as a version. It then identifies unique MCIs across all versions using our earlier definition and assigns each a unique ID number. This ID number is used in the GUI to represent the MCI, as a textual ID would be too long and unwieldy. Each time an MCC from the first version is selected by a user, related MCCs for successive versions are discovered and displayed by searching for the MCCs in future versions that contain the MCIs belonging to the selected MCC. So, for example, let us consider an MCC with class ID 4 from a first version, MCC_{v1c4}. It is selected and contains a set of MCIs, MCI_{v1c4}. In future version 'x' and class 'y', MCC_{vxcy} is displayed if MCI_{vxcy} contains any element from MCI_{v1c4}.

5 Experiment

We ran SIMCCT on both publicly available models and private models from our industrial partners. The public models include the **Automotive Power**

```
<clones>
    <class classid="#" nclones="#" similarity="#" ...>
        <source file="..."subsystem="..." ... >
             <block path="..." type="..." ...Block attributes.../>
             ...More Blocks...
             <line ...Line attributes"/>
             ...More Lines...
        </source>
         ...More Sources...
     </class>
     ...More Classes...
</clones>
```

Fig. 1: General form of SIMCCT input

Window (PW) System that comes with the *Simulink* example set and a large open-source **Advanced Vehicle Simulator**(AVS) [2].

To start, we analyzed the 3 systems using *SIMONE* with our best-fit [1] settings of 70% similarity and blind-renaming. Table 1 displays statistics about the results. The PW system is a smaller, compact, and simple system. AVS is quite large and complex, and has more clone pairs and MCCs than our industrial system set. Thus, we believe it is a fairly representative and rich system.

After transforming the MCD results into the SIMCCT format, we execute SIMCCT. As mentioned, we are looking specifically to note what MCCs in future versions contain MCIs from a user-selected MCC in an earlier/earliest version, v_1. For each MCC in v_1, its relation to future MCCs with respect to another version can be classified in one of five ways: (1) **1 to 1**; (2) **1 to 1***, which is the same as "1 to 1" except there are additional MCIs, missing MCIs, or a combination of both; (3) **1 to many**, which has no additional or missing elements in future MCCs; (4) **1 to many***, which has additional or missing elements in future MCCs; and (5) **1 to 0**, meaning the MCIs from the original MCC are no longer in any MCC. We can then use this information as grounds to investigate what model evolution has transpired on the MCIs to cause this relation.

Table 2 classifies the MCC evolution we observed in the three systems as it pertains to each MCC's MCIs in each system's first version. In our sample systems, we found no instances of "1 to many", that is, every time an MCC later had its constituent MCIs split into multiple MCCs, there were always additional elements present. For "1-to-many*" relationships, the AVS system had a "1 to 2" and our industrial set had a "1 to 2" and a "1 to 4".

5.1 Examples

We now showcase a set of examples from our experiment demonstrating different cases. We illustrate the examples by extending the representation Göde used for

[2] http://sourceforge.net/projects/adv-vehicle-sim/?source=dlp

Table 1: Systems Analyzed by SIMCCT

System Name	Version #	Model Files	SubSystems	Clone Pairs	MCCs
PW	1	1	18	7	5
	2	1	29	15	5
	3	1	33	23	6
	4	1	25	13	4
	5	1	45	39	6
AVS	r0000	69	861	1916	18
	r0080	69	1621	5693	35
	r0116	72	1714	5951	38
Industrial Set	55	9	977	600	20
	56	9	977	618	21
	57	9	986	624	23

Table 2: Relationship Classifications of MCCs w.r.t. Earliest Versions

System Name	Version	1 to 1	1 to 1*	1 to many	1 to many*	1 to 0
PW	2	1	4	0	0	0
	3	1	4	0	0	0
	4	1	3	0	0	1
	5	1	2	0	0	2
AVS	r0080	12	5	0	1	0
	r0116	9	8	0	1	0
Industrial Set	56	14	4	0	2	0
	57	14	4	0	2	0

the evolution of type-1 code clones [6], with MCCs being rectangles and MCIs being circles. In addition, we provide figures of some of the examples showcasing the specific evolution that has transpired. This is in order to highlight samples of changes that form various evolutionary MCC relationships.

We choose examples from public models as they adequately exhibit the cases and are available to all. We then investigate what evolution has taken place on the models themselves that caused the observed MCE. A reminder, each number within a circle refers to a uniquely identified key that corresponds to a unique MCI across all versions.

Power Window - Model Clone Class 3: This example is presented in Figs. 2 and 3. It contains MCC3, which begins with two MCIs, 5 and 6, that are 81% similar. In version 2, represented by the part underneath the dashed line in Fig. 3, MCI6 has 2 additional blocks and no longer belongs to any MCC. Conversely, MCI5 is simplified by replacing three blocks with one and is now 71% similar to MCI7 and other MCIs, causing a reclassification with them. Starting in version 4, MCI5 is simplified even further by removing more blocks and no longer belongs to any MCC. As such, MCC3 has a "1-to-0" relation to versions 4 and 5.

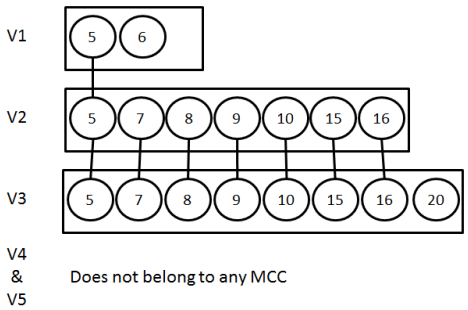


Fig. 2: PW MCC3 SIMCCT Trace

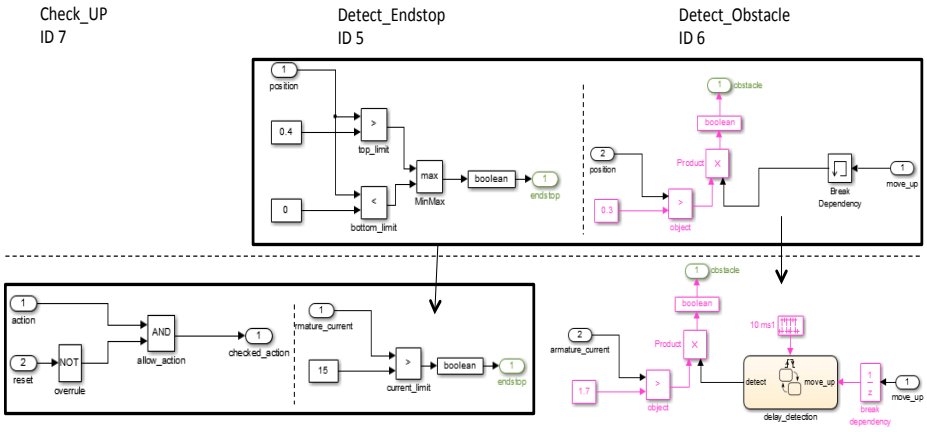


Fig. 3: Sample Models from PW MCC3

Power Window - Model Clone Class 2: We demonstrate, in Figs. 4 and 6, a "1-to-1*" variant where a single MCI is removed from MCC2 in version 1 while the remaining MCIs remain grouped together. In version 1, the three MCIs; 2,3, and 4; are 74% similar to each other with MCIs 3 and 4 being identical in this case. Fig. 6 shows the evolution of MCI2, the Window_System. As shown, it was changed significantly from version 1 to 2 in terms of its ports, the amount and types of blocks, and its lines. This was done in order to include power electronics and to incorporate bodies, joints, and actuators. It changed again in version 4, albeit it not as radically, but it was not enough to reunite it with the original MCIs from MCC2 from version 1.

Advanced Vehicle Simulator - Model Clone Class 7: Fig. 5 presents an example of a "1-to-many*" MCC trace. In version 1, MCI17, which is the system "Energy Storage <ess> RC", is 71% similar to MCIs 16,18,19. In the next version, MCI17 becomes 76% similar to MCI325 and is reclassified with that.

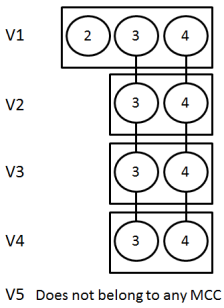


Fig. 4: PW MCC2 SIMCCT Trace

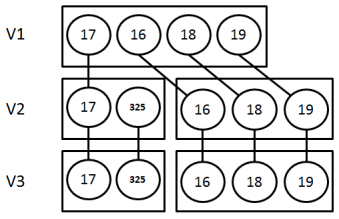


Fig. 5: AVS MCC7 SIMCCT Trace

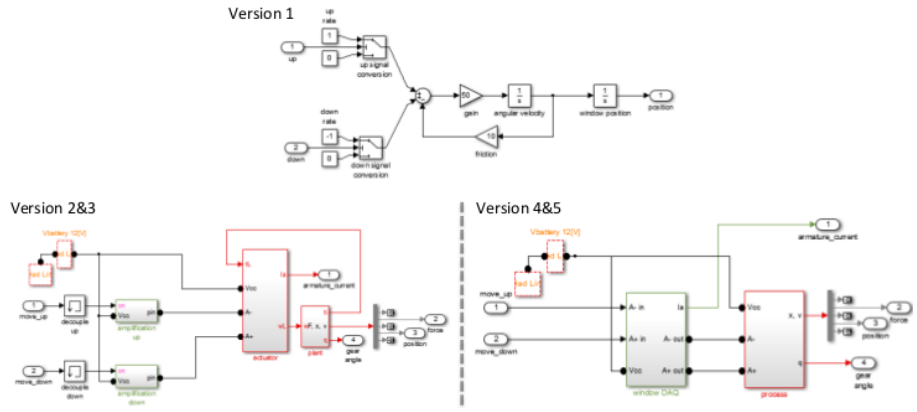


Fig. 6: Evolution of the PW Window_System

MCI17's evolution involved many low-level structural changes, and as such, is not worth showing a diagram of. So, while the model remained the same in terms of layout, there were some key non-visual changes including modification of block types, the addition or explication of ports, and changing a block's key parameters, for example, changing a Gain block's multiplication mode.

6 Related and Future Work

6.1 Related Work

As mentioned already, Saha et al. [14] are focused on counting the occurrences of code clone genealogies. In contrast, we are more interested in reasoning about the changes to models that cause MCE and use the genealogy of the MCCs as the starting point only.

There are some language-agnostic model and metamodel evolution approaches [7, 12] that can track both evolution and co-evolution. However, in order to use techniques like this for MCE, we would essentially have to create a system containing only the clones of interest. As such, we developed a tool explicitly intended to perform model clone class evolution analysis.

There are some model comparison approaches [16, 17] that can find similarities and differences among models for versioning and other purposes, but there are no attempts to explicate the structural evolution of *Simulink* models. That is, to define what are the potential structural changes that can occur to a *Simulink* model and their prevalence. Model evolution and MCE are strongly related to model comparison and versioning, but can be viewed as a longer-term analysis over multiple versions with a focus on how a specific artifact or clone has changed. None of the model comparison techniques we surveyed previously were ideal for tracking MCE.

The only work that deals with any form of *Simulink* evolution is from Tran and Kreuz, who focus on refactoring *Simulink* [11]. Specifically, they look at forms of antipatterns in *Simulink* and discuss tool support for correcting them.

6.2 Future Work

One area of future work is providing better differencing and visualization of differences for *Simulink* models. So far, we've been doing it relatively manually. While there are many model comparison tools [16, 17] that provide visualization functionality, there are none well-suited for our purposes and the provided *Simulink* XML comparison functionality is inadequate as it does not capture the information we desire. As such, devising and automating the differencing and visualization for MCE purposes and incorporating it into SIMCCT would be ideal.

In addition, we are currently working on MCD for other model types, including *Stateflow* and behavioral UML models [2]. We believe our work on MCE can be applied to other model types; Specifically, as long as an MCD tool identifies both MCCs and MCIs, these concepts can be extended and an appropriate clone class tracker can be developed. This is something that we will investigate once our MCD techniques for these other model types are more mature.

In the long term, we plan on enumerating a set of *Simulink* model evolutions as they relate to model clone evolution. The purpose in doing this is to find a sufficient set for performing MCD evaluation in a mutation-based framework as discussed in [15]. So, in addition to all the previously mentioned benefits of observing model evolution, we plan on using this work to help with mutation research intended to realize model-clone tool evaluation.

7 Conclusions

We believe MCE research is quite valuable as it can be a useful tool for better understanding how *Simulink* and other data-flow models evolve. In this paper, we took some first steps towards understanding *Simulink* MCE. We began by defining key terms, including model clone classes and instances. SIMCCT is a tool we introduced that is capable of tracking an MCI's evolution with respect to its containing MCCs across different system versions. We used this tool on three systems, share our findings, and go into details for a few examples. These examples included looking at the model evolution that transpired causing the specific MCE observed. In the future, we plan on automating the differencing and visualization of the model evolution for a given MCE trace as well as enumerating the *Simulink* model evolution steps that cause MCC changes. This and other *Simulink* and data-flow MCE work can go a long way towards improving our relatively underdeveloped understanding of the model evolution of these technologies.

Acknowledgments

This work is supported by NSERC, the Natural Sciences and Engineering Research Council of Canada, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

References

1. Alalfi, M.H., Cordy, J.R., Dean, T.R., Stephan, M., Stevenson, A.: Models are code too: Near-miss clone detection for Simulink models. In: ICSM. pp. 295–304 (2012)
2. Antony, E., Alalfi, M., Cordy, J.: An approach to clone detection in behavioural models. In: WCRE. p. 5 (2013), (to appear)
3. Cordy, J.: The TXL source transformation language. Science of Computer Programming 61(3), 190–210 (2006)
4. Deissenboeck, F., Hummel, B., Juergens, E., Schaetz, B., Wagner, S., Girard, J.F., Teuchart, S.: Clone detection in automotive model-based development. In: ICSE. pp. 603–612 (2009)
5. France, R., Bieman, J.M.: Multi-view software evolution: a uml-based framework for evolving object-oriented software. In: ICSM. pp. 386–395 (2001)
6. Göde, N.: Evolution of type-1 clones. In: SCAM. pp. 77–86 (2009)
7. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope-automating coupled evolution of metamodels and models. In: ECOOP 2009, pp. 52–76 (2009)
8. Keienburg, F., Rausch, A.: Using XML/XMI for tool supported evolution of UML models. In: HICSS. vol. 9, p. 9064 (2001)
9. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. ESEC/FSE-13 30(5), 187–196 (2005)
10. Mens, T., Lucas, C., Steyaert, P.: Supporting disciplined reuse and evolution of UML models. UML98: Beyond the Notation pp. 378–392 (1999)
11. Minh Tran, Q., Kreuz, I.: Refactoring of simulink models. In: MathWorks Automotive Conference, Stuttgart (2012)
12. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: Model migration with epsilon flock. In: Theory and Practice of Model Transformations, pp. 184–198. Springer (2010)
13. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Tech. Rep. 2007-541, Queen's University (2007)
14. Saha, R.K., Roy, C.K., Schneider, K.A.: An automatic framework for extracting and classifying near-miss clone genealogies. In: ICSM. pp. 293–302 (2011)
15. Stephan, M., Alafi, M., Stevenson, A., Cordy, J.: Using mutation analysis for a model-clone detector comparison framework. In: ICSE. pp. 1277–1280 (2013)
16. Stephan, M., Cordy, J.R.: A survey of methods and applications of model comparison. Tech. Rep. 2011-582 Rev. 3, Queen's University (2012)
17. Stephan, M., Cordy, J.R.: A survey of model comparison approaches and applications. In: MODELSWARD (2013)

Proactive Quality Guidance for Model Evolution in Model Libraries

Andreas Ganser[1], Horst Lichter[1], Alexander Roth[2], and Berhard Rumpe[2]

[1] RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
{ganser, lichter}@swc.rwth-aachen.de,
home page: http://www.swc.rwth-aachen.de
[2] RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
{roth, rumpe}@se.rwth-aachen.de,
home page: http://www.se.rwth-aachen.de

Abstract. Model evolution in model libraries differs from general model evolution. It limits the scope to the manageable and allows to develop clear concepts, approaches, solutions, and methodologies. Looking at model quality in evolving model libraries, we focus on quality concerns related to reusability.
In this paper, we put forward our proactive quality guidance approach for model evolution in model libraries. It uses an editing-time assessment linked to a lightweight quality model, corresponding metrics, and simplified reviews. All of which help to guide model evolution by means of quality gates fostering model reusability.

Keywords: Model Evolution, Model Quality, Model Libraries

1 The Need for Proactive Quality Guidance

Modeling is one of the traditional disciplines in computer sciences and other sciences. Therefore, computer scientists have been creating models for decades and have seen models incarnate in a lots of different forms. Interestingly enough that the general modeling theory was not developed by a computer scientist but by Herbert Stachowiak in the seventies [Stachowiak, 1973]. His work found impact in a lot of domains in computer sciences, e.g. databases, resulting in research like generic model management [Melnik, 2004]. This transforms Stachowiak's rather abstract theories to an applicable approach offering concepts and algorithms, e.g., `diff`, `merge`, `similarity`, and `match` operations. As a result, pure operations on models did not seem to be challenging any more and a broader perspective was investigated.

A similar development took place in object oriented modeling which brought up UML as a suitable modeling language. Today, the success of UML is often accredited for two reasons. First, UML is believed to be an effective language, because "for larger and distributed projects, UML modeling is believed to contribute to shared understanding of the system and more effective communication" [Chaudron et al., 2012]. Second, UML is considered as the de facto

standard in modeling. Due to that, a lot of tools were developed around UML including code generators. They bolster approaches like rapid prototyping or model driven development (MDD) and allow modelers to deal with complexity on an appropriate level of abstraction.

Consequently, UML models are widely used and can be regarded as project assets that should be reused. Moreover, it is believed that model reuse could decrease development time while increasing software quality [Mens et al., 1999, Lange and Chaudron, 2005], because best practices and experience would be leveraged. But the question is how to store models in a way that their quality is maintained or even improved over time. Certainly, model reuse requires an infrastructure enabling to persist models in a library or knowledge base. Furthermore, it needs a means to control model evolution and quality in the long run. Unfortunately, quality is a matter of subjectivity, often relative to requirements and sometimes hard to measure [Moody, 2005]. Moreover, all-at-once quality assessments result in endless quality reports that are hard to work through. One way out is edit-time quality assessment and guidance for assuring a certain level of quality in model libraries, we call proactive quality guidance. To the best of our knowledge we could not find such an approach for model libraries.

Hence, we looked into recent research (section 2) and found that model evolution is often considered as a goal. That would be self-defeating in model libraries. So, we adopted the meaning of model evolution to fit model libraries and developed an approach (section 3) that explains how models should evolve in model libraries. This enables us to discuss model evolution in model libraries on a more formal and qualitative level. In detail, we introduce our understanding of model quality and quality gates. After that we explain our proactive approach including tool support by defining a mapping between a lightweight quality model and metrics (section 4). This mostly deals with syntactic aspects, so we introduced simple reviews (section 5) for the mostly semantic and pragmatic aspects.

2 Related Work

Model evolution, as we will present, can be discussed closely related to model libraries and model quality. In the following, we present the current understanding of model evolution, model repositories, and model quality.

Model evolution is often investigated as a goal to be achieve automatically by tool support; as it is for software. There are several tools and research prototypes available. First, COPE supports evolution and co-evolution by monitoring changes in an operation based way. These can be applied as editing traces to other models, i.e., forwarded [Herrmannsdoerfer and Ratiu, 2010]. Our approach differs in a regard that we do not trace changes but focus on edit-time changes and their impact on quality aspects. Second, MoDisco [Eclipse, 2012] (hosted with AM3 [Allilaire et al., 2006]) tries to provide means to support evolution of legacy systems applying model-driven ideas. That means, MoDisco is a tool for re-engineering legacy software by means of models and starting a model driven development from gleaned models. Moreover, co-evolution is discussed. We keep

to plain model evolution, but the main distinction to our approach is that we want evolution to be guided and directed instead of being aimlessly.

Regarding model repositories one needs to bare in mind their functionality. Often, they allow for querying, conflict resolution, and version management but no more. This means, evolution and co-evolution are not considered. Examples are, first, MOOGLE, a user friendly model search engine offering enhanced querying [Lucredio et al., 2010]. Second, ReMoDD which focuses on comunity building by offering models in a documentary sense to the community [France et al., 2007]. Mind that all of these model libraries do not consider model evolution. Consequently, we have implemented an enhanced model library [Ganser and Lichter, 2013], offering model evolution as presented below.

This evolution support was enhanced by ideas regarding quality in modeling by Moody [Moody, 2005], because we wanted to establish a common understanding of model quality in our library avoiding that "quality is seen as a subjective and rather social than a formally definable property lacking a common understanding and standards" [Moody, 2005]. This is why we took the quality dimensions by Lindland et al. [Lindland et al., 1994] and applied them in our environment. They comprise *syntactic*, *semantic*, and *pragmatic quality* bearing in mind that these quality dimensions can influence each other as presented by Bansiya et al. [Bansiya and Davis, 2002]. Looking at UML models, there exist manifold model qualities. We chose the work of Lange et al [Lange, 2006] to be most suitable and linked it with metrics. There we employed the work of Genero et al. [Genero et al., 2003], Wedemeijer et al. [Wedemeijer, 2001], and Mohagheghi et al. [Mohagheghi and Dehlen, 2009]. Furthermore, we needed means to assess some semantical and pragmatical aspects. Here we root our ideas on reviews but found Fagans approach to heavyweight [Fagan, 1976]. So we subdivided review tasks using an adoption of the thinking hats as proposed by De Bono [De Bono, 2010].

3 Quality Staged Evolution

General model evolution is to be distinguished from model evolution in model libraries [Roth et al., 2013] since the purpose differs. While general model evolution is considered aimless, evolution in model libraries does not make sense if it is undirected. This is due to the reuse focus of model libraries and the fact that these models mostly represent a starting point for modeling. Consequently, models in model libraries do not strive for perfection in every possible deployment scenario but rather an eighty percent solution that will need adaption. Still, evolution in model library needs a sound foundation and tool support.

We developed an approach for quality assured model evolution in a gated and staged manner [Roth et al., 2013]. It defines our approach to model evolution in model libraries and how it can be guided. In the remainder of this section, we will shortly describe the quality staged model evolution approach.

3.1 Quality and Evolution in Model Libraries

During modeling some parts of a model might seem so generic or generally reusable that a modeler decides to put them in a model library. This means, some parts of the model are extracted, prepared for reuse, and stored in a model library. At the same time, the modeler annotates the extracted model with a simplified specification, we call *model purpose*. It is supposed to grasp the main intention of the model and reflect the general idea in a few words so this model is explained in a complementary way. As all of this is done, we consider this the moment model evolution of this particular model starts.

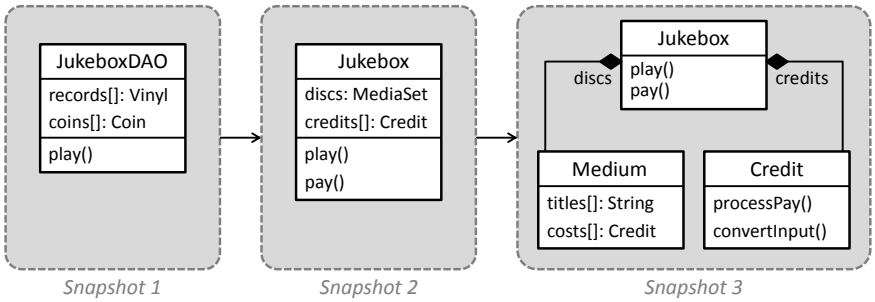


Fig. 1. Model Evolution Example

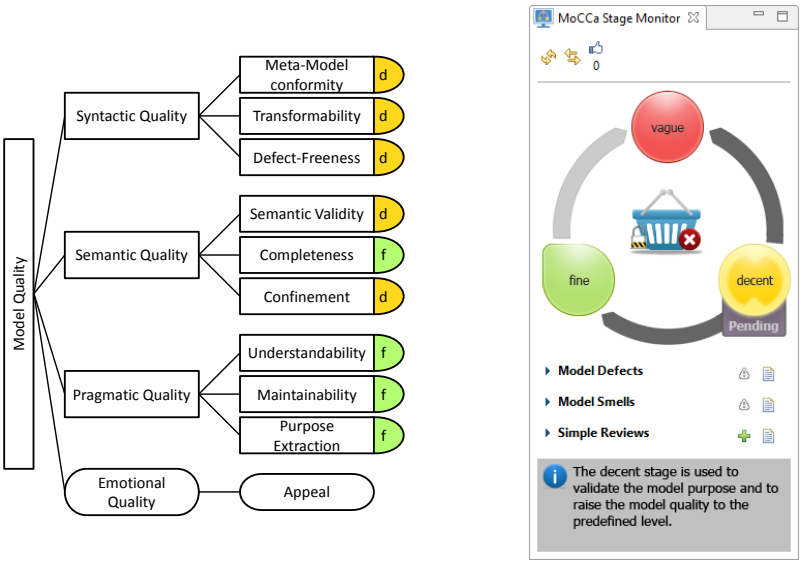
Reasons for model evolution in model libraries can be best explained considering figure 1 as an example. Certainly, the first snapshot of this model is reusable, but some modeling decisions might be questionable. Furthermore, the model is not free from technological details. The "DAO" suffix in the class name is a clumsy leftover that should be removed quickly. Due to that, we offer editing support so models can be overwritten in a versioning style and call a version of a model in our approach a model snapshot.

A few snapshots might be necessary to get a well designed and reusable model. Since all of them are persisted, one can order these snapshots as shown in figure 1 and assign numbers to each snapshot forming an *evolution sequence*.

This evolution sequence can be subdivided into subsequences annotated with stages that make a statement regarding reusability. We conducted a field study about the number and the names and found that "vague", "decent", and "fine" are the best representatives and assigned the colors "red", "yellow", and "green" respectively (cf. figure 2(b)). This is meant to provide an intuitive representation of the model's reusability and the underlying formalities [Roth et al., 2013], since we do not want to bother modelers with the state machine formalizing the states.

The modeler just needs to know the semantics behind each stage. First, a "vague" model might contain some awkward design or leftovers from the originating environment saying: "Be careful reusing this!". Second, a "decent" model is considered reusable in general, but might contain some pragmatic or semantic mismatch between the given purpose and the actual model. This stage is best

characterized by: “The devil is in the details.” Finally, a “fine” model should be easily reusable and might offer additional information, e.g. template information. So, one could informally characterize it by: “Go ahead and enjoy.”.



(a) Quality Model and Stages (b) Eclipse Prototype

Fig. 2. Quality Model, Stages, and Prototype similar to [Roth et al., 2013]

The more formal idea behind the quality stages is a quality model that defines criteria (cf. figure 2(a)), which need to be met to gain a certain stage. This is why we talk about quality gates that need to be passed between the stages. In more detail, each quality attribute in figure 2(a) is mapped to a stage in figure 2(b) indicating which quality attributes are required for a certain stage. For example, *completeness* is only required if a model should be regarded “fine”, therefore we attached an “f” to that quality attribute in figure 2(a)

Some of the criteria of a quality model might be checked automatically and some might depend on modeler interaction. As a result, the formalization underneath is non-deterministic [Roth et al., 2013], partly because some semantic and most of the pragmatic quality attributes are a matter of subjectivity. For example, contradicting attributes are very unlikely found by tools. If one of the required quality attributes of a gate is not met any more the model loses its status automatically and falls back to the next lower stage.

3.2 Quality Measurement Instruments

Evaluating quality attributes of models shows that some of them are automatically assessable and some are not. Clearly, syntactic errors can be found easily

by parsers, but completeness is in the eye of the beholder. Consequently, we make a distinction regarding model quality measurement instruments in three categories: *strong*, *medium*, and *weak characteristics*. This classification enables a mapping from model qualities (cf. figure 2(a)) to quality measurement instruments, where each attribute is used to derive a feedback with respect to the attribute name.

Strong characteristics form the strictest type. They can be measured precisely using model metrics. A model metric is formulated with respect to models and provides clear feedback including the reason for the improvement and the suggested solution. For example, a model including a class without a name. Besides, model metrics strong characteristics can be measured with external tools, e.g. EMF validator and EMF generator [Steinberg et al., 2009]. With respect to our quality model in figure 2(a), strong characteristics can be used to derive feedback of the following model quality characteristics: defect-freeness, meta-model conformity, and transformability.

Medium characteristics are based on Fowler's idea of smells [Fowler, 1999]. A smell is something that does not seem to be right and can be measured in some way. For example, a model with hundreds of classes is harder to understand than one with only a few. Such characteristics can be measured with metrics, which define a clear threshold. However, this threshold can be overridden, if the modeler does not agree. Medium characteristics can be used to derive information for confinement, understandability, and maintainability.

Finally, weak characteristics can be compared to hunches. A hunch is something that does not seem to be right because of gut feelings, experience, or intuition. Clearly, it is hard to measure such weak characteristics using metrics. We present simplified reviews in section 5 that enable assessing weak characteristics in a quick and precise way. Such model reviews allow to derive qualitative feedback on semantic validity, completeness, purpose extraction, and appeal.

4 Proactive Quality Assessment

Quality measurement instruments and a quality model are used to assess the quality of a model. Such an assessment is, generally, triggered manually at a certain point in time. At this point, the model is analyzed and a report is created, which identifies improvements of the model. Clearly, such improvements can be very vague making the cause for an improvement or a suggested solution hard to understand. Additionally, such events that trigger model assessment are mostly of manual nature, i.e., triggered by someone. Consequently, to prevent long assessment reports with dozens improvement suggestions, such assessment events should be triggered automatically and more importantly periodically.

Proactive quality guidance is an approach that triggers assessment events automatically and regards the iterative nature of model creation, i.e., the final model is created in multiple iterations. During model creation the assessment is triggered whenever the model has been changed. The model is analyzed and feedback is presented to the modeler. Because the assessment is triggered when

a model is changed, the resulting assessment iterations are kept small avoiding large reports and improvement suggestions. Due to constant and precise feedback during model creation the model evolution is guided and such detected violations with respect to the quality model in section 3.1.

The main parts of proactive quality guidance are (a) automatic and constant assessment of the model, which is currently created, and (b) clear instructions on where the improvements have to be made and why. Automatic assessment is executed when the model is changed but clear and instructive feedback is challenging, because it identifies areas of improvement and their cause and must always be correct. Otherwise, modelers will be annoyed by false feedback. However, the subjective nature of model quality makes it hard to always derive correct feedback without manual interaction.

As the underlying source of information for feedback are quality measurement instruments, we applied the classification of quality measurement instruments, as presented in section 3.2, to structure feedback and, thereby, to loosen up the restriction of always correct feedback. Always correct feedback relies on strong characteristics, which can be measured precisely by using metrics, e.g. if a class has duplicate methods. Furthermore, feedback relies on medium characteristics, which are less precise than strong characteristics, are only suggestions and can be ignored by the modeler. For instance, methods with long parameter lists should be avoided to not pass everything as a parameter. A list of all strong and medium characteristics metrics is listed in [Roth, 2013]. Finally, weak characteristics regard the subjective nature of model quality and, consequently, are hard to measure. In consequence, we present an approach to simplify reviews and to enable measurement of weak characteristics. Such weak quality measurement instruments can then be used to provide feedback.

5 Simplified Reviews

Metrics enable proactive quality assessment for strong and medium characteristics but for some weak characteristics proactive quality assessment is difficult. At best heuristics can support modelers but they are unlikely to overrule experience or gut feeling. For example, purpose extraction can be checked partially by keyword comparison but the modeler must have the last word. This is why we researched on simplifying reviews as a means to quickly quality check these and weaker characteristics.

Our result is an approach, we call simplified reviews, that separates different aspects of reviews by altering a technique used in parallel thinking [De Bono, 2010]. These "Six Thinking Hats" provide a separation of concerns for each role which is behind each hat directing tasks clearly. In our simplified reviews this leads to reviews that take no longer than absolutely necessary:

In total five review roles remained because a role for controlling is not necessary for this approach. The hats are designed as follows: A *Yellow Hat Review* (Good points judgment) considers positive aspects of a model and a high number indicates better quality. For example, a review might emphasize that a model is

of high benefit in maintenance. The *Black Hat Review* (Bad points judgment) can be regarded as the most known type of reviews. It is used to criticize pointing out difficulties, dangers, defects, or bad design. A black hat review indicates that the corresponding model needs to be patched immediately. A *White Hat Review* (Information) is used to provide information or ask for information, which cannot be gleaned from the model. For example, if a modeler has expertise on limitations of the model, this should be documented by a white hat review. The *Green Hat Reviews* (Creativity) are a means to provide information about possible improvements or new ideas. For a model library this review type is an integral part to foster evolution and to keep modelers satisfied in the long run. Finally, in a *Red Hat Review* (Emotions) a reviewer can express a general attitude in terms of like and dislike. For example, this can be a dislike based on experience that might help improving a model in future.

All of the simple reviews need no more than very simple tool support. A look at figure 2(b) shows an entry that reads "Simple Reviews" with a plus button right next to it. Clicking this button opens a small window that allows to select the type of the simple review and entering additional text. Moreover, the tree editor can be unfolded if there are simple reviews related to this model. Then every review can be inspected and checked "done" or "reopened". This is a bit similar to a very lightweight issue tracking system.

6 Proactive Quality Guidance in a Nutshell

General model evolution is to be distinguished from model evolution in model libraries as we briefly discussed above. This is due to unguided evolution being self-defeating for model libraries. Due to that guidance is required to keep models reusable. Moreover, a model library with a focus on reuse puts constraints on a quality model for models that makes it manageable.

All in all, we have shown how models should evolve in model libraries with proactive quality guidance. Therefore, we illustrated how a quality model for model library can be used to guide and stage model evolution in model libraries. To achieve this, we broke down the stages "vague", "decent", and "fine" to quality characteristics which are assured in different ways. While strong characteristics are checked automatically, medium and weak characteristics require user interaction. But this interaction is supported in two ways. On the one hand, for medium characteristics some metrics provide assessments that only need to be judged by a user because certain constraints like thresholds might not hold true for a particular case. On the other hand, for weak characteristics, we introduced simple reviews that allow quick and guided evaluations.

Since all this takes place during editing time, we call this approach proactive quality guidance. But there is more, because a lot of metrics allow to derive recommendations how to fix certain issues. We use other published experience to do so and implemented a prototype that realizes the entire approach. It looks simple and clean, because we tried to avoid as much noise for modelers as possible so the modeler does not get distracted while modeling.

Acknowledgements

We would like to thank all our anonymous reviewers for their comments and effort. We would also like to thank all our participants in our surveys and studies.

References

[Allilaire et al., 2006] Allilaire, F., Bezivin, J., Bruneliere, H., and Jouault, F. (2006). Global Model Management in Eclipse GMT/AM3. In *Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference*.

[Bansiya and Davis, 2002] Bansiya, J. and Davis, C. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on SE*.

[Chaudron et al., 2012] Chaudron, M., Heijstek, W., and Nugroho, A. (2012). How effective is UML modeling? *Software and Systems Modeling*, pages 1–10.

[De Bono, 2010] De Bono, E. (2010). *Six Thinking Hats*. Penguin Group.

[Eclipse, 2012] Eclipse (2012). MoDisco. http://www.eclipse.org/MoDisco/.

[Fagan, 1976] Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211.

[Fowler, 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code (Object Technology Series)*. Addison-Wesley Longman, Amsterdam.

[France et al., 2007] France, R., Bieman, J., and Cheng, B. (2007). Repository for Model Driven Development (ReMoDD). In Kuehne, T., editor, *Models in Software Engineering*, volume 4364 of *LNCS*, pages 311–317. Springer Berlin / Heidelberg.

[Ganser and Lichter, 2013] Ganser, A. and Lichter, H. (2013). Engineering Model Recommender Foundations. In *Modelsward 2013, Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain,19.-21- February 2013*, pages 135–142. SCITEPRESS.

[Genero et al., 2003] Genero, M., Piattini, M., Manso, M. E., and Cantone, G. (2003). Building uml class diagram maintainability prediction models based on early metrics. In *IEEE METRICS*, pages 263–. IEEE Computer Society.

[Herrmannsdoerfer and Ratiu, 2010] Herrmannsdoerfer, M. and Ratiu, D. (2010). Limitations of automating model migration in response to metamodel adaptation. In Ghosh, S., editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 205–219. Springer Berlin Heidelberg.

[Lange and Chaudron, 2005] Lange, C. F. and Chaudron, M. R. (2005). Managing model quality in uml-based software development. *Software Technology and Engineering Practice, International Workshop on*, 0:7–16.

[Lange, 2006] Lange, C. F. J. (2006). Improving the quality of uml models in practice. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 993–996, New York, NY, USA. ACM.

[Lindland et al., 1994] Lindland, O., Sindre, G., and Solvberg, A. (1994). Understanding quality in conceptual modeling. *Software, IEEE*, 11(2):42 –49.

[Lucredio et al., 2010] Lucredio, D., de M. Fortes, R., and Whittle, J. (2010). MOOGLE: a metamodel-based model search engine. *Software and Systems Modeling*, 11:183–208.

[Melnik, 2004] Melnik, S. (2004). *Generic Model Management: Concepts and Algorithms*. Lecture Notes in Computer Science. Springer.

[Mens et al., 1999] Mens, T., Lucas, C., and Steyaert, P. (1999). Supporting disciplined reuse and evolution of UML models. In Bezivin, J. and Muller, P.-A., editors, *Proc. UML'98 - Beyond The Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 378–392. Springer-Verlag. Mulhouse, France.

[Mohagheghi and Dehlen, 2009] Mohagheghi, P. and Dehlen, V. (2009). Existing model metrics and relations to model quality. In *Proceedings of the Seventh ICSE conference on Software quality*, WOSQ'09, pages 39–45, Washington, DC, USA. IEEE.

[Moody, 2005] Moody, D. L. (2005). Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. *Data Knowl. Eng.*, 55(3):243–276.

[Roth, 2013] Roth, A. (2013). A Metrics Mapping and Sources. http://goo.gl/ruqFpi.

[Roth et al., 2013] Roth, A., Ganser, A., Lichter, H., and Rumpe, B. (2013). Staged evolution with quality gates for model libraries. In *1st International Workshop on:(Document) Changes: modeling, detection, storage and visualization, September 10th, 2013, Florence, Italy.*

[Stachowiak, 1973] Stachowiak, H. (1973). *Allgemeine Modelltheorie.* Springer-Verlag.

[Steinberg et al., 2009] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley, 2nd edition.

[Wedemeijer, 2001] Wedemeijer, L. (2001). Defining metrics for conceptual schema evolution. In *Selected papers from the 9th International Workshop on Foundations of Models and Languages for Data and Objects, Database Schema Evolution and Meta-Modeling*, FoMLaDO/DEMM 2000, pages 220–244, London, UK. Springer.

Towards a Novel Model Versioning Approach based on the Separation between Linguistic and Ontological Aspects

Antonio Cicchetti and Federico Ciccozzi

School of Innovation, Design and Engineering
Mälardalen University, SE-721 23, Västerås, Sweden
antonio.cicchetti@mdh.se, federico.ciccozzi@mdh.se

Abstract. With the increasing adoption of Model-Driven Engineering (MDE) the support of distributed development and hence model versioning has become a necessity. MDE research investigations targeting (meta-)model versioning, conflict management, and model co-evolution have progressively recognized the importance of tackling the problem at higher abstraction level and a number of solving techniques have been proposed. However, in general existing mechanisms *hit the wall* of semantics, i.e. when not only syntax is involved in the manipulations the chances for providing precision and automation are remarkably reduced.
In this paper we illustrate a novel version management proposal that leverages on the separation between linguistic and ontological aspects involved in a (meta-)modelling activity. In particular, we revisit the main versioning tasks in terms of the mentioned separation. The aim is to maximize the amount of versioning problems that can be automatically addressed while leaving the ones intertwined with domain-specific semantics to be solved separately, possibly by means of semi-automatic techniques and additional precision.

1 Introduction

Model-Driven Engineering (MDE) promises to reduce software development complexity by shifting the focus from coding to modelling. Models become first-class citizens and they represent abstractions of real-phenomena tailored to a specific purpose. In this respect they are an appropriate composition of concepts, whose well-formedness is specified by means of a metamodel. Moreover, model transformations are exploited to manipulate models to perform analysis and generate code. Given the relevance gained by models, they are expected to be affected by the same evolutionary pressure source code experienced in the past. Therefore, if MDE approaches are not able to provide evolution support at least comparable with the one existing for text-based software development, MDE adoption would be remarkably hindered.

In the latest years the need for appropriate support of model evolution has been largely recognized and addressed by a number of research works, including differencing, storing versions, managing merges and possible conflicts, supporting metamodel evolution and corresponding model migrations. In particular, model differencing covered both language-specific and agnostic cases, model changes have been tackled both

in a state-based and operation-based manner, mechanisms have been introduced to detect divergences between concurrent manipulations of the same model and provide possible reconciliation strategies. Moreover, techniques have been developed to detect metamodel changes, classify them in terms of effects on existing model instances, and provide corresponding migration countermeasures ranging from manual to automatic.

Given the high abstraction level of modelling activities, mixing syntax and semantics is unavoidable; unfortunately, when semantics comes into play, versioning problems become more complex to manage and very often they cannot be dealt with automatically. In other words, automation support has typically to be reduced and user intervention is required to keep the desired degree of precision.

In this paper we propose to enhance automation opportunities by defining a novel methodology for all activities involved in model versioning. The main idea is to exploit the separation between linguistic and ontological aspects of a model and address them separately. In particular, linguistic aspects are those related to the structural correctness of a model, while ontological aspects pertain to the specific domain taken into account (please, see Section 2 for a more precise definition of these aspects). In this way the evolution of the linguistic part, that is expected to be mainly syntactic and hence easier to manage, can be supported by automated mechanisms, whereas the ontological part can be provided with more precise domain-specific versioning support and possibly offer semi-automatic management. Based on the separation mentioned so far, this work revisits the current techniques developed for model version management and outlines a research agenda to cover all the aspects of model versioning.

The structure of the paper is as follows. Section 2 depicts the motivations underneath the proposed methodology as well as the related works in the area. In Section 3 we propose a research agenda to cover the different aspects of model versioning with our novel methodology and we conclude the paper with an outlook in Section 4.

2 Background and Related Work

In MDE, models are commonly defined as abstractions of real phenomena, by means of a given modelling purpose in mind, pursuing a simplification of the reality [1]. In this respect, (meta-)modelling activities carry along not only the syntax by which concepts are expressed (either textual, graphical, or a combination of both), but also the underlying semantics of the application domain taken into account. In general these two aspects are not clearly distinguishable, since part of the semantics can be intertwined with the adopted syntax and structural constraints.

Kühne proposed an alternative separation between those two aspects by introducing *linguistic* and *ontological* matters of (meta-)modelling [2]. In particular, the linguistic can be referred to constraints and rules that define the structural correctness of a model. For instance, a class must have a unique name within a model, or a relationship shall have a source and a target model element. On the contrary, ontological aspects are those pertaining to the domain taken into account, and exploit structural compositions to prescribe domain-specific well-formedness. In other words, they create a new logical abstraction level by specializing (groups of) concepts at lower levels of abstraction. Notably, the class `person` must have a name, a surname, and an age greater than 0 to be

a valid ontological instance of the type `person`. It is worth noting that while linguistic aspects are invariants of the modelling activity, the ontological part is strictly coupled with the domain taken into account, and hence the purpose the modelling activity is devoted to. More importantly, ontologies implicitly define a set of semantic relationships which would need to be explicitly specified otherwise, as proposed in existing works on semantic model versioning [3]. Therefore, the semantics can be considered as direct consequence of adopting a given ontology for the domain taken into account.

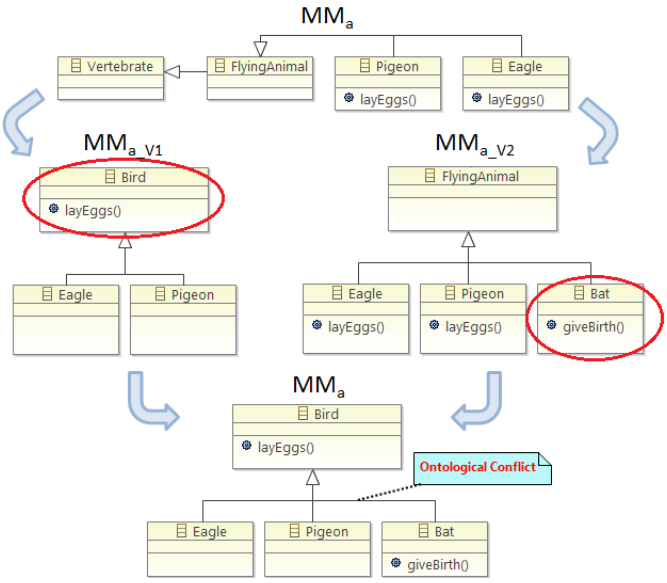


Fig. 1. Motivating Scenario

The idea of separating linguistic and ontological matters to reduce the complexity of modelling management, and/or to enhance reuse chances, is not new. In general these techniques have been referred to as deep or multilevel metamodelling [4, 5], to stress the fact that it could be useful to consider more than two (fixed) metamodelling layers. Based on the partition of linguistic and ontological aspects, it has been possible to support generic modelling language and transformation specifications [6, 7], and to lower the complexity of language evolution [8]. Notably, even if by exploiting different terminologies, all the mentioned works leverage on the distinction between linguistic and ontological aspects to define generic operations that are later on applied to the metamodel taken into account. In particular, linguistic manipulations can be replicated directly, while ontological ones have to be bound to the concepts pertaining to the considered applicative domain. This allows to create operators over models, define constraints, model transformations, and manage the need for language evolution. In [9] this methodological approach is the foundation for a framework supporting generic model management operators, which have been implemented within the Epsilon family

of (meta-)modelling languages [10]. A similar solution is developed by means of the Melanie tool [11]; also in this case the modelling environment uses a multilevel modelling approach. Moreover, the ATLAS Transformation Language has been extended to encompass predicates distinguishing between the various modelling levels.

Despite the growing research interest and effort in this area, so far there has been little effort in the definition of evolution management support based on the separation between linguistic and ontological aspects. In this respect, with this paper we propose to revisit the main model versioning features in terms of such a separation to improve their efficacy. In order to better grasp the potentials of this idea, in Fig. 1 is depicted a sample evolutionary scenario. Metamodel $\texttt{MM}_a$ defines a language for the definition of vertebrates with focus on flying animals. As it can be noticed in its original form the metaclass `FlyingAnimal` is specialised by the sub-types `Eagle` and `Pigeon`, both containing a `layEggs` operation. Let us now suppose that $\texttt{MM}_a$ is exposed, and concurrently evolves, in two different views, resulting in two versions of $\texttt{MM}_a$, namely $\texttt{MM}_{a\_V1}$ and $\texttt{MM}_{a\_V2}$. Both view-specific metamodels undergo modifications. In case of $\texttt{MM}_{a\_V1}$, the metaclass `FlyingAnimal` is renamed to `Bird` and the operation `layEggs` is moved from the sub-types to the super-type. In $\texttt{MM}_{a\_V2}$, the new sub-type `Bat` is added together with its operation `giveBirth` as specialisation of `FlyingAnimal`. These modifications result in an ontological conflict from the perspective of $\texttt{MM}_a$, since a bat is both a vertebrate and a flying animal giving birth to live young but NOT a bird laying eggs.

3 A Research Agenda

In the latest years a considerable research work has been devoted to all the activities involved in evolution management, notably *model differencing*, *conflict management*, as well as *metamodel evolution* and *model co-evolution*. Providing a survey on all those investigations goes far beyond the scope of this work, however in the next sections we outline some common principles and problems characterizing the current available solutions. In general current approaches for model versioning that can be found in the literature have to fight intrinsic semantics issues entailed by the modelling level of abstraction.

This paper aims at providing the guidelines for a novel version management methodology that takes into consideration the separation between linguistic and ontological aspects involved in modelling activities. Our belief is that, by means of such a separation, domain-specific issues can be better managed thus improving degree of automation and accuracy of current version management. In this respect, the next sections also illustrate a research agenda to revisit current versioning solutions based on the separation between linguistic and ontological aspects, discussing foreseeable benefits and needs.

At this point, it is worth noting that this proposal is not excluding the current available techniques, whereas it provides additional means to better exploit those solutions. By embracing the MDE principles, versioning artefacts are models conforming to corresponding metamodels and are manipulated by means of model transformations [1]. In this respect, this work relies on a model-based representation of differences as the ones proposed in [12, 13]. In particular, we exploit the difference representation proposal

in [12] because of its generative approach that allows to adapt the already existing solutions to our separation in a smooth way.

3.1 Model Differencing

Model differencing has been firstly addressed by means of language-specific solutions and then generalized to language-agnostic cases. Typically, difference detection, representation, and visualisation are performed at model level of abstraction for being able to grasp user's intentions [14, 15]. *State-based* techniques deduce modification operations based on the old and new state of a model [16]. The element matching can result very complex and hard to make arbitrarily precise, since it is reduced to the graph homomorphism problem [17]. A way to reduce such an inherent intricacy is to adopt *operation-based* approaches which keep track of the operations performed by the users to modify the model. The drawback of such approaches is that the differences detection is tightly coupled to the tool, since anything happening outside the tool cannot be tracked in terms of evolution information.

Regardless being state- or operation-based, differencing approaches rely on structural similarities to determine evolution operations, thus requiring user intervention whenever the semantics involved in the changes is misinterpreted or cannot be grasped at all. Notably, if we consider the example shown in Fig. 1, a differencing engine would detect a rename of the `FlyingAnimal` class towards `Bird` since their subgraphs are matching from a structural perspective. However, this information does not provide any additional detail about the domain-specific consequences of such a modification. Therefore, we propose to exploit ontological information as part of the differencing result. By exploiting a model-based mechanism like the one proposed in [12], analogously to what is performed for the linguistic part, each metaclass of the ontology involved in the modelling activity can be extended with corresponding evolution means as shown in bottom-right part of Fig. 2. Notably, in this case we would have an `AddedFlyingAnimal`, `ChangedFlyingAnimal`, and `DeletedFlyingAnimal` for representing evolutions involving the `FlyingAnimal` class, an `AddedBird`, `ChangedBird`, and `DeletedBird` for `Bird`, and so forth. Consequently, differences would carry not only modifications from a structural point of view, but also the information related to the ontological evolution. Those details can be useful for detection and visualization purposes: the ontological relationships among elements could help in distinguishing between a rename and a delete/add evolution and to show changes from a domain-specific perspective, respectively. Moreover, they become very relevant when dealing with version merging and/or co-evolution management, as discussed in the remainder of the paper.

3.2 Conflict Management

Conflict management followed the same development differencing techniques did, that is detection and resolution have been addressed firstly at atomic operation level [18], then by considering refactoring modifications [19], and eventually by supporting arbitrary semantics divergences [20, 21]. Also in this case, and possibly even more important than for differencing, when domain aspects get involved in the management of

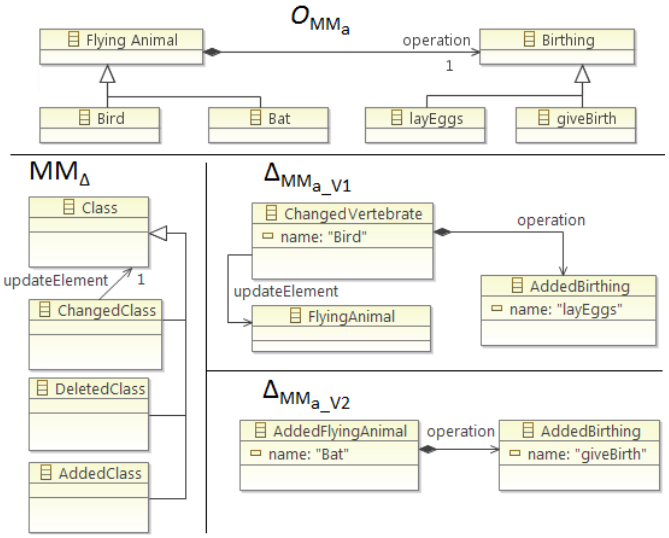


Fig. 2. Ontology and Difference Models

concurrent modifications the precision of structure-based solutions degrades remarkably and user intervention is unavoidable.

By referring to the example illustrated in Fig. 1, a linguistic merging operation would not reveal any problem, since the involved subgraphs structures are perfectly compatible. However, by taking into account also ontological evolution information it is possible to discover deeper issues. In particular, `Bat` becomes a specialization of `Bird` and `layEggs()` is inherited in the `Bat` class, thus arising an ontological conflict. It is very important to notice that, while the latter conflict could be solved by structural constraints (e.g., only one operation per class is admitted), the former has to be explicitly defined by the user. Even more important, in both cases the ontological aspects disclose the possibility to grasp the rationale behind the problems: `Bat` is not a `Bird` and, being a mammal, `Bat` does not `layEggs()`.

From a conflict management perspective, the separation between linguistic and ontological aspects discloses very interesting research directions. Notably, while linguistic conflicts have to be solved since they affect the well-formedness of the merge result itself, ontological divergences can be *tolerated*. Therefore the separation proposed in this paper could be very useful, especially in the early stages of development, to allow collaborative development without forcing the users in taking domain-specific design decisions when their side effects are not completely clear [22].

3.3 Metamodel Evolution and Model Co-Evolution

In MDE, metamodels are subject to the same evolutionary pressure models do. Metamodel evolutions trigger model co-evolutions, i.e. model instances have to be migrated

to the newer version of the metamodel in order to recover their conformance [23]. In this scenario a correct interpretation of metamodel manipulations is of critical importance to adopt appropriate migration countermeasures. Moreover, model co-evolution may require user information to resolve particular migration cases [24]. As a consequence, a number of approaches have been introduced, supporting from (semi-)automated to manual model co-evolution approaches [25].

In the case of metamodel evolution domain-specific issues can heavily affect the migration process, especially when the whole metamodel ecosystem is involved in the evolution [26]. Therefore, recent investigations have been devoted to relax the metamodel-model conformance relationship, even by separating linguistic and ontological aspects involved in the language definition [8]. Our idea is based on the same principle of this latter work, but instead of relaxing the conformance relationship we propose to separate linguistic and ontological aspects and address their co-evolution separately. Analogously to the model merging problem, also in this case structural co-evolution has to be performed in order to re-establish the linguistic well-formedness. Whereas, ontological issues can be solved in a separate way and by means of domain-specific solutions.

It is not expectable that the separation between linguistic and ontological aspects will guarantee full automation of co-evolution operations. However, by knowing the metamodel evolution in ontological terms can help in managing it in a better way. In particular, by noticing that `Bird` and `Bat` are not compatible, a migration operation would add a new metaclass `Bird` instead or renaming `FlyingAnimal`. In turn, `Bat` would keep its correctness after the migration being still a valid instance of `FlyingAnimal`. Interestingly, since `Bat` is still a `FlyingAnimal` a tool co-evolution countermeasure could also decide to re-use the same icon, or ask for a new one specific for bats. In the same way, it could be possible to notice that using the `Bird` icon would be erroneous from a domain semantics perspective.

4 Outlook

This paper proposed the guidelines for a novel model versioning methodology based on the separation between linguistic and ontological aspects of (meta-)modelling. The idea is not ignoring what already existing and developed in the latest years for model evolution investigations; rather, it aims at enriching current solutions by adding ontological details to the manipulation information. Given this characteristics, it has been possible to conduct small experiments by means of already available techniques (notably [12] and the Melanie tool [11]) and the results are encouraging. However, the methodology has to be validated against real-life systems to prove its efficacy. Moreover, it is not possible to exclude future needs for addressing ontological-specific evolutions, both for the detection, representation, and management, beyond the general additions, deletions, and changes.

References

1. Bezivin, J.: On the Unification Power of Models. SoSym **4** (2005) 171–188
2. Kühne, T.: Matters of (meta-)modeling. SoSym **5** (2006) 369–385

3. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting metamodels to ontologies: A step to the semantic integration of modeling languages. In: Procs. of MoDELS, Genova (Italy). LNCS, Springer (2006) 528–542
4. de Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. In: Proc. of TOOLS, Málaga (Spain). LNCS (2010) 1–20
5. Atkinson, C., Gutheil, M., Kennel, B.: A Flexible Infrastructure for Multilevel Language Engineering. IEEE TSE **35** (2009) 742–755
6. de Lara, J., Guerra, E., Cuadrado, J.S.: Abstracting Modelling Languages: A Reutilization Approach. In: Proc. of CAiSE, Gdansk (Poland). LNCS, Springer (2012) 127–143
7. Cuadrado, J.S., Guerra, E., de Lara, J.: Generic Model Transformations: *Write Once, Reuse Everywhere*. In: Procs. ICMT, Zurich (Switzerland), 2011. LNCS, Springer (2011) 62–77
8. Gómez, P., Sánchez, M., Florez, H., Villalobos, J.: Co-creation of models and metamodels for enterprise architecture projects. In: Procs. of XM, ACM (2012) 21–26
9. Rose, L.M., Guerra, E., de Lara, J., Etien, A., Kolovos, D.S., Paige, R.F.: Genericity for model management operations. SoSym **12** (2013) 201–219
10. : Epsilon. http://www.eclipse.org/epsilon/ (2013)
11. : Melanie - multi-level modeling and ontology engineering environment. http://code.google.com/a/eclipselabs.org/p/melanie/ (2013)
12. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. JOT **6** (2007) 165–185
13. Rivera, J., Vallecillo, A.: Representing and Operating with Model Differences. In: Procs. TOOLS EUROPE. (2008)
14. Kolovos, D., Paige, R., Polack, F.: Model comparison: a foundation for model composition and model transformation testing. In: Procs. GaMMa, Shanghai (China). (2006) 13–20
15. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional (2008)
16. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. ACM Computing Surveys **30** (1998) 232–282
17. Kolovos, D.S., Di Ruscio, D., Paige, R.F., Pierantonio, A.: Different models for model matching: An analysis of approaches to support model differencing. In: Proc. 2nd CVSM'09, ICSE09 Workshop, Vancouver, Canada (2009)
18. Alanen, M., Porres, I.: Difference and Union of Models. In: UML 2003 - The Unified Modeling Language. Volume 2863 of LNCS., Springer-Verlag (2003) 2–17
19. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci **127** (2005) 113–128
20. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. International Journal of Web Information Systems (IJWIS) **5** (2009) 271 – 304
21. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Procs. MoDELS. (2008) 311–325
22. Wieland, K., Langer, P., Seidl, M., Wimmer, M., Kappel, G.: Turning conflicts into collaboration. Computer Supported Cooperative Work **22** (2013) 181–240
23. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software **20** (2003) 42–45
24. Gruschko, B., Kolovos, D., Paige., R.: Towards Synchronizing Models with Evolving Metamodels. In: Procs of the Work. MODSE. (2007)
25. Rose, L.M., Herrmannsdoerfer, M., Williams, J.R., Kolovos, D.S., Garcés, K., Paige, R.F., Polack, F.A.C.: A Comparison of Model Migration Tools. In: Procs. MoDELS. LNCS, Springer (2010) 61–75
26. Iovino, L., Pierantonio, A., Malavolta, I.: On the impact significance of metamodel evolution in mde. Journal of Object Technology **11** (2012) 3: 1–33

Analyzing Behavioral Refactoring of Class Models

Wuliang Sun, Robert B. France, Indrakshi Ray

Colorado State University, Fort Collins, USA

Abstract. Software modelers refactor their design models to improve design quality while preserving essential functional properties. Tools that allow modelers to check whether their refactorings preserve specified essential behaviors are needed to support rigorous model evolution. In this paper we describe a rigorous approach to analyzing design model refactorings that involve changes to operation specifications expressed in the Object Constraint Language (OCL). The analysis checks whether the refactored model preserves the essential behavior of changed operations in a source design model. A refactoring example involving the *Abstract Factory* design pattern is used in the paper to illustrate the approach.
Keywords: Behavioral refactoring, UML/OCL, Alloy

1 Introduction

In Model-Driven Development (MDD) projects, one can expect design models to evolve as developers explore design spaces for high quality solutions. Class models are among the most popular models used in practice and given their pivotal roles, there is a need to manage their evolution. Software refactoring [4][14] is an important class of changes that is applicable to class models. The goal of a refactoring is to improve software qualities such as maintainability and extensibility, while preserving essential structural and behavioral properties. A number of model refactoring mechanisms have been proposed (e.g., see [2][5][12][19][20][21]), and many (e.g., see [19][21]) provide support for checking whether structural properties are preserved in refactored models. However, we are not aware of any approach that supports rigorous analysis of behavioral properties when operation specifications in class models are added, removed, or modified. In this paper we describe a rigorous approach to analyzing the refactoring of design class models that involve changes to operation specifications expressed in the Object Constraint Language (OCL) [15].

The model on which a refactoring is performed is called the *source* model, and the model produced by the refactoring is called the *refactored* model. A refactoring that involves making changes to operation specifications is called a *behavioral refactoring*. In this paper, we present an approach to analyzing behavioral refactorings to check that changes to operation specifications preserve the net effect of the operation (i.e., its essential behavior) as specified in the source model. The net effect of an operation can be expressed using the OCL pre-/post-conditions.

As an example, consider a case in which the operation $FlightManager$ :: $bookFlight()$ in a flight reservation system class model is refactored into the following four operations in the refactored model: $Airline$ :: $getAvailableFlights()$ returns all flights that are available on a given day and airport, $Flight$ :: $getAvailableSeats()$ returns all seats that are available on the flight on a given day and airport, $Flight$ :: $reserveSeat()$ reserves a seat on the flight, and $FlightManager$ :: $bookFlight()$ books a flight by calling the previous three operations. The net effect of the $FlightManager$ :: $bookFlight()$ operation in the source model is specified using an OCL pre-/post-condition stating that if there exists available flight seats, at the end of the operation execution a seat will be reserved by a flight manager. The behavioral refactoring performed on the source model redistributes the functionality of $FlightManager$ :: $bookFlight()$ across different classes (i.e., $Airline$, $Flight$, and $FlightManager$). It is tedious to manually determine if the above behavioral refactoring preserves the net effect of the original operation because it involves manually building a description of the global net effect of a behavior by composing operation specifications that define sub-behaviors in local contexts (i.e., classes in which the operations are located).

The above motivates the need for an automated analysis technique that supports rigorous analysis of behavioral refactorings. In the approach described in this paper, an analysis of a behavioral refactoring involves determining whether a sequence of operations in the refactored model preserves the net effect of an operation in the source model. The net effect of a source model operation is preserved by a sequence of refactored operations if the sequence starts in all the states that satisfy the pre-condition of the source model operation, and leaves the system in a state that satisfies the post-condition of the source model operation. The analysis approach requires the software modeler who performed the behavioral refactoring to provide a sequence diagram that describes the sequence of refactored operations. The approach takes the sequence of refactored operations, applies all the states that satisfy the pre-condition of the source model operation, and checks if the sequence of refactored operations produces any state that does not satisfy the post-condition of the source model operation. The net effect of the source model operation is not preserved by a sequence of refactored model operations if the sequence of refactored model operations starts in a state that satisfies the pre-condition of the source operation and produces a state that does not satisfy the post-condition of the source model operation.

The *Alloy Analyzer* [9] is used at the back end to statically analyze a behavioral refactoring. The analysis involves using the Alloy trace mechanism to determine whether operations in the refactored model can preserve the net effect of a changed operation specification in the source model. Since the Alloy Analyzer requires users to specify a bounded scope for each class, that is, the maximum number of instances that can be produced for a class, the analysis is performed within a bounded scope of class objects. The approach uses a UML-to-Alloy transformation to shield the software modeler from the "back-end" use of the Alloy Analyzer. Our transformation extends prior work on transforming UML to Alloy models [1][3][7][11][17] by providing support for transforming a

class model and a sequence diagram to an Alloy model that specifies behavioral traces.

The approach described in the paper is lightweight in that (1) it does not expose the modeler to any formal notation other than the OCL, and (2) the net effect preservation analysis is checked within a bounded domain. More heavyweight formal analysis techniques are needed in a setting where the net effect preservation checking requires more exhaustive analysis.

The rest of the paper is organized as follows. Section 2 provides an overview of the approach and Section 3 illustrates its use on a small example. Section 4 presents a research prototype to support the analysis approach. Section 5 describes related work, and Section 6 concludes the paper.

2 Approach Overview

The analysis approach is used to determine whether the net effect associated with a behavior specified in a source model can be preserved by distributed behaviors specified in a refactored class model. The net effect preservation property that is checked is defined as follows:

Definition 1: Net Effect Preservation. A sequence of operation invocations, $OpSeq$, in a refactored model is said to preserve the net effect of an operation, $Op0$, in the source model if the set of net effects (i.e., start and end system states associated with an operation invocation) characterized by the specification of $Op0$ is included in the set of net effects (i.e., start and end system states associated with a sequence of operation invocations) characterized by the sequence $OpSeq$. More precisely, a set of operations specified in a refactored model, $\{Op1$, $Op2$, ..., $OpN\}$, is said to preserve the net effect of an operation $Op0$ specified in the source model if there exists an invocation sequence of the refactored model operations, $OpSeq = [Op1; Op2; ...; OpN]$, such that the following holds:

1. $OpSeq$ starts in all the states that satisfy the pre-condition of $Op0$.
2. If $OpSeq$ starts in a state that satisfies the pre-condition of $Op0$ then the sequence of operation invocations leaves the system in a state that satisfies the post-condition of $Op0$.

The analysis approach requires a software modeler to provide the following as inputs:

1. The specification of the source model operation, $Op0$, that is refactored.
2. The result of a refactoring (i.e., a refactored class model), and a sequence diagram that describes how $Op0$'s redistributed behavior is used. The sequence diagram provides the sequence of refactored operations that will be analyzed against the source model specification of $Op0$.

The intermediate output of the approach is an analyzable model that can be used to check the net effect preservation property between $Op0$ and $OpSeq$. In this approach, the analyzable model takes the form of an Alloy model that is

produced from (1) the refactored class model, and (2) a sequence diagram that describes $OpSeq$.

The specifications for $Op0$ and the operations involved in $OpSeq$ are also included in the Alloy model. The inclusion of $Op0$ in an Alloy model produced from the refactored class model can be problematic when $Op0$ refers to elements not included in the refactored model. For this reason the first step of the approach checks that the elements referenced in the $Op0$ operation specification also appear in the refactored model.

The second step of the approach generates the base Alloy model that is extended in following steps to check the preservation property. We use a UML-to-Alloy transformation that builds upon our previous work on rigorous analysis of UML class models [17].

The third step of the approach takes as input the specification of $Op0$ and a sequence diagram, and produces an Alloy assertion (or predicate) that is used to determine whether the sequence described in the sequence diagram ($OpSeq$) preserves the net effect of $Op0$. The assertion (or predicate) is added to the Alloy model generated in the second step of the approach. If a check of the assertion (or predicate) by the Alloy Analyzer produces an Alloy instance then the net effect specified by $Op0$ cannot be preserved by the operation sequence.

More details on the major steps of the approach can be found in [18].

3 An Illustrating Example

A maze game class model from [6] (see Figure 1) is used in this paper to illustrate the analysis approach. The $MazeGame$ class is responsible for creating different types of mazes (e.g., $BombedMaze$ and $EnchantedMaze$) and their parts (e.g., $RoomWithBomb$ and $EnchantedRoom$). A maze room consists of four sides that can be doors, walls, or other rooms.

The operation *createBombedMaze()* in class $MazeGame$ is used to create a bombed maze that consists of four walls. Its net effect in the form of OCL specification is given below:

Context *MazeGame::createBombedMaze() : BombedMaze*
// Pre-condition: no maze has been created
Pre: *self.maze→isEmpty()*
// Post-condition: a bombed maze has been created, and it includes a room
// with four walls
Post: *result.oclIsNew() and self.maze.bRooms→size() = 1 and*
self.maze.bRooms→forAll(r : RoomWithBomb | r.bwalls→size() = 4)

If a new type of maze, maze room, door or wall were added, the structure of the class model would need to be changed significantly. Incorporating the *Abstract Factory* pattern [6] into the class model results in a more flexible design in which the maze creation responsibilities are localized in factories that the $MazeGame$ class can access.

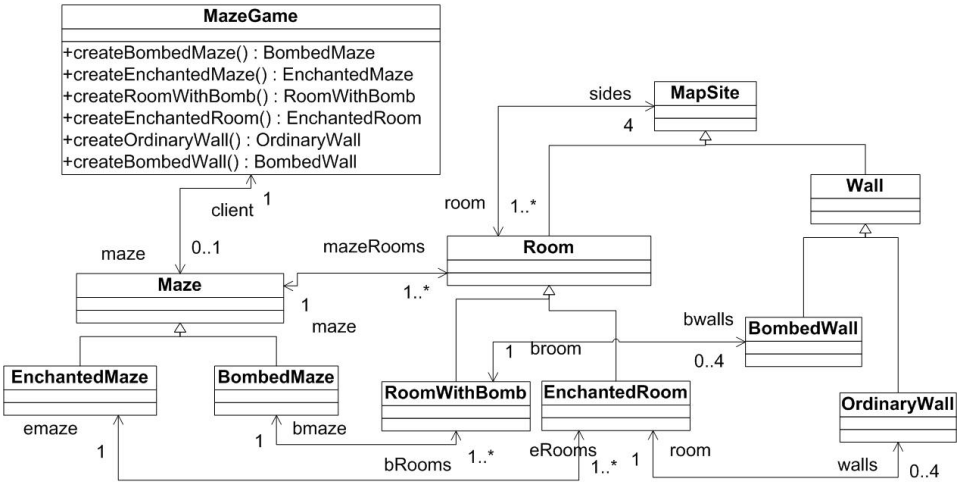


Fig. 1: Maze Game Class Model

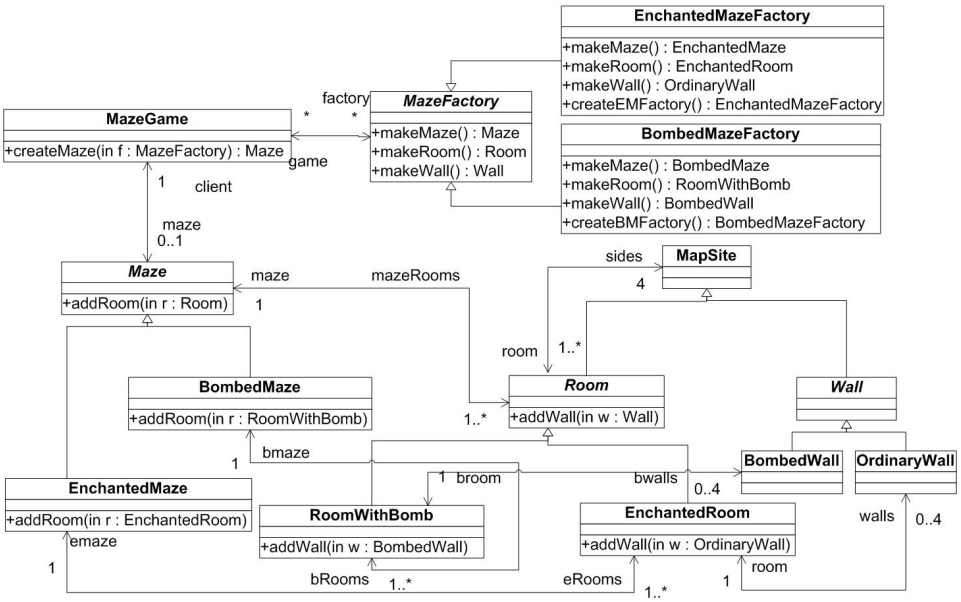


Fig. 2: Refactored Maze Game Class Model

Figure 2 shows a refactored maze game class model that incorporates an instantiation of the *Abstract Factory* pattern. The original $createBombedMaze$ and $createEnchantedMaze$ operations in $MazeGame$ have been replaced by the

$createMaze(f : MazeFactory) : Maze$ operation, that uses a factory to create a specific type of maze. The net effects of the original operations in $MazeGame$ need to be preserved by the behavioral refactoring. The analysis approach described in this paper can be used to check if the net effect of $createBombedMaze$ is preserved by relevant operations in the refactored model.

The OCL specifications for $createMaze$ and *addRoom* are given below:

Context *MazeGame::createMaze(f:MazeFactory) : Maze*
// Pre-condition: a maze factory has been associated with a maze game
Pre: *self.factory→includes(f)*
Post: *true*

Context *Maze::addRoom(r:Room)*
// Pre-condition: a room has not been associated with a maze
Pre: *self.mazeRooms→excludes(r)*
// Post-condition: a room has been associated with a maze
Post: *self.mazeRooms→includes(r)*

Unlike the $createBombedMaze$ operation, the $createMaze$ operation delegates its responsibility to other operations (i.e., $makeMaze$, $makeRoom$, $addRoom$, $makeWall$, and $addWall$) in the refactored class model. Due to space limitations, only the specifications of $createMaze$ and *addRoom* are given in the paper (see above). More operation specifications can be found in [18]. A sequence diagram (see Figure 3) is used to describe the result of the behavioral refactoring. It describes an invocation sequence of the refactored model operations that is intended to preserve the net effect of the $createBombedMaze$ operation in the source model.

The analysis showed that if we removed an operation (e.g., *addRoom*) from the operation sequence in Fig. 3, the net effect of $createBombedMaze$ cannot be preserved by the rest of operations in Fig. 3. We also used the same analysis approach to check if the net effects of other source model operations are preserved by refactored model operations. Our analysis results showed that all the operations in the source model (e.g., $createEnchantedMaze$, $createRoomWithBomb$, $createEnchantedRoom$, $createOrdinaryWall$ and $createBombedWall$) can be preserved by relevant operations in the refactored model.

4 Tool Support

We developed a research prototype to investigate the feasibility of developing tool support for the approach. The prototype consists of an Eclipse OCL parser, an Ecore/OCL transformer and an Alloy Analyzer. The Ecore/OCL transformer is developed using Kermeta [13], an aspect-oriented metamodeling tool. The inputs of the prototype are (1) an EMF Ecore [16] file that specifies a refactored class model, (2) a textual OCL file that specifies the pre-/post-conditions of $Op0$ and

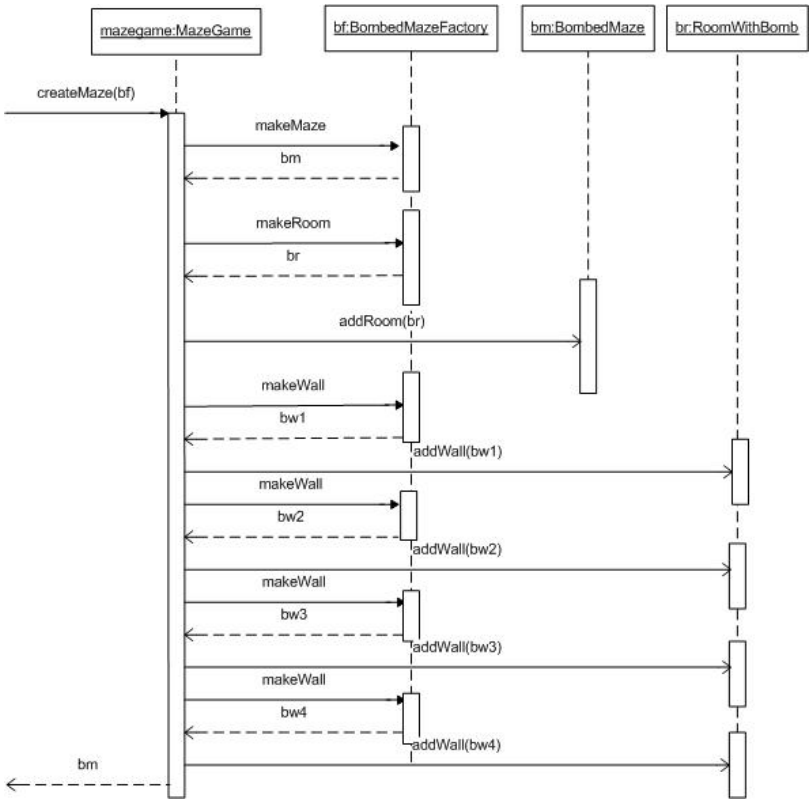


Fig. 3: A Sequence Diagram that Describes an Invocation Sequence of the Refactored Model Operations

operations involved in *OpSeq*, and (3) a textual file that describes a sequence diagram.

The inputs are automatically transformed to an Alloy model consisting of signatures and predicates. The prototype then uses the APIs provided by the Alloy Analyzer to pass the Alloy model to the Alloy SAT solver. The result returned by the Alloy SAT solver is interpreted by the prototype. The interpreted result provides the net effect preservation property between $Op0$ and *OpSeq*.

The prototype implementation uses a visitor pattern to transform a class model with operation specifications into an Alloy model. The traditional visitor design pattern keeps the separation of the structure (i.e., the metamodel elements) and the behavior (i.e., the visitor) by using a specific class for the visitor, and thus results in ping-pong calls between the objects of the structure and the objects of the visitor. The Kermeta [10] language provides an aspect weaving mechanism to simplify the visitor pattern by allowing a user to define a *visit* method for each model element being visited in an aspect class that is

woven into an existing base class at runtime. There is thus no need to keep a visitor class that is used to traverse each model element of a metamodel.

5 Related Work

Two broad categories of related work are discussed in the section: work on model refactoring and work on UML-to-Alloy transformation.

5.1 Model Refactoring

Refactoring has attracted much attention from the MDE community since it was first introduced by Opdyke in his PhD dissertation [14]. Boger et al. [2] applied the idea of refactoring to UML class diagrams, statechart diagrams, and activity diagrams. Their approach, however, does not provide support for rigorously reasoning about a behavioral refactoring.

Both Sunye et al. [19] and Van Gorp et al. [21] used OCL to formally specify the refactoring for UML models. An operation is defined for each type of the refactoring and its OCL pre-/post-condition specifies the model structure that must be satisfied before and after the refactoring associated with the operation. Their approach, however, can only be used to verify the refactoring involving the changes to model structures.

France et al. [5] described a metamodeling approach to pattern-based model refactoring in which refactorings are used to introduce a new design pattern instance to the model. Mens and Tourwe [12] used logic reasoning to detect if a design pattern instance that is introduced to a class model, limits the applicability of certain refactorings.

Straeten et al. [20] proposed a behavior preserving refactoring approach for UML class models. Unlike our approach, the behavior of a class model in their approach is expressed using state machines and sequence diagrams. Gheyi et al. [8] described a rigorous approach to verifying the refactoring for Alloy models.

However, based on our knowledge, none of the above approaches can be used to analyze operation-based model refactoring that involves changes to operation specifications.

5.2 UML to Alloy Transformation

Georg et al. [7] used both Alloy and UML/OCL to specify the runtime configuration management of a distributed system. An ad-hoc comparison between Alloy and UML/OCL is discussed in their paper.

Dennis et al. [3] used the Alloy Analyzer to uncover the errors in a UML model of a radiation therapy machine. The operations in the design model are specified using OCL. An informal description of OCL-to-Alloy transformation is described in their approach. Their approach, however, does not provide support for automated transformation between UML/OCL and Alloy.

Anastasakis et al. [1] described a tool, namely UML2Alloy, that automatically transforms a UML class model with OCL invariants into an Alloy model. Their tool builds upon a formal mapping between UML/OCL metamodel and Alloy metamodel. Unlike their approach, our approach leverages Alloy's trace mechanism to generate an Alloy model with trace features from a UML/OCL model.

Maoz et al. [11] developed a tool that implements the transformation between UML class models and Alloy models. Unlike the approach described in [1], Maoz's tool produces a single Alloy module from two class models. Maoz's approach, however, does not provide support for class models with OCL invariants and operation specifications.

6 Conclusion

We presented an approach to rigorously analyzing a behavioral refactoring that involves making changes to operation specifications expressed in the OCL. The behavioral refactoring analysis involves checking whether relevant operations in the refactored model can preserve the net effects of the operations targeted by the refactoring in the source model. The net effect preservation checking technique described in the paper builds upon the Alloy Analyzer and thus requires a translation from UML class models and OCL operation specifications to Alloy models. We developed a prototype for transforming UML+OCL models to Alloy models with traces to support the net effect preservation check. We applied the approach to a pattern-based model refactoring to demonstrate how software modelers can use the approach to analyze a behavioral refactoring.

We plan to extend the behavioral refactoring analysis approach by providing support for more complex OCL operators. Specifically we are currently investigating how we can use SMT solvers (e.g., Microsoft Z3) at the back-end to analyze the OCL specifications. Our future work will also explore how mappings between equivalent source and refactored forms can be used to support the net effect preservation checking.

ACKNOWLEDGMENT

The work described in this report was supported by the National Science Foundation grant CCF-1018711.

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.
2. M. Boger, T. Sturm, and P. Fragemann. Refactoring browser for UML. *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 366–377, 2003.

3. G. Dennis, R. Seater, D. Rayside, and D. Jackson. Automating commutativity analysis at the design level. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 165–174. ACM, 2004.
4. M. Fowler and K. Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.
5. R. France, S. Chosh, E. Song, and D.K. Kim. A metamodeling approach to pattern-based model refactoring. *Software, IEEE*, 20(5):52–58, 2003.
6. E. Gamma, H. Richard, J. Ralph, and V. John. Design patterns: elements of reusable object-oriented software. *Reading: Addison Wesley Publishing Company*, 1995.
7. G. Georg, J. Bieman, and R. France. Using Alloy and UML/OCL to specify run-time configuration management: a case study. *Practical UML-Based Rigorous Development Methods-Countering or Integrating the eXtremists*, 7:128–141, 2001.
8. R. Gheyi, T. Massoni, and P. Borba. A rigorous approach for proving model refactorings. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 372–375. ACM, 2005.
9. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
10. J.M. Jézéquel, O. Barais, and F. Fleurey. Model driven language engineering with kermeta. *Generative and Transformational Techniques in Software Engineering III*, pages 201–221, 2011.
11. S. Maoz, J. Ringert, and B. Rumpe. Cddiff: Semantic differencing for class diagrams. *ECOOP 2011–Object-Oriented Programming*, pages 230–254, 2011.
12. T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 570–579. IEEE, 2001.
13. P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. *Model Driven Engineering Languages and Systems*, pages 264–278, 2005.
14. W.F. Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks.* PhD thesis, PhD thesis, University of Illinois at Urbana-Champaign, 1992.
15. O.M.G.A. Specification. Object constraint language, 2007.
16. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework.* Addison-Wesley Professional, 2008.
17. W. Sun, R. France, and I. Ray. Rigorous analysis of UML access control policy models. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 9–16. IEEE, 2011.
18. W. Sun, R. France, and I. Ray. *Analyzing Behavioral Refactoring of Class Models.* Technical Report CS-13-104, Colorado State University, http://www.cs.colostate.edu/TechReports/, 2013.
19. G. Sunye, D. Pollet, Y. Le Traon, and J.M. Jezequel. Refactoring UML models. *UML 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 134–148, 2001.
20. R. Van Der Straeten, V. Jonckers, and T. Mens. A formal approach to model refactoring and model refinement. *Software and Systems Modeling*, 6(2):139–162, 2007.
21. P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. *UML 2003-The Unified Modeling Language. Modeling Languages and Applications*, pages 144–158, 2003.

Specification of a legacy tool by means of a dependency graph to improve its reusability

Paola Vallejo, Mickaël Kerboeuf, and Jean-Philippe Babau

University of Brest (France), Lab-STICC, MOCS Team
{vallejoco,kerboeuf,babau}@univ-brest.fr

Abstract. This position paper, investigates a way to improve the reusability of legacy tools in specific contexts (defined by specific metamodels). The approach is based on a dedicated language for co-evolution, called Modif. Its associated process involves two model migrations: 1) The Migration, allows to put data under the scope of a legacy tool. 2) The Reverse Migrationallows to put the legacy tool's output back into the original specific context. We generalize the approach by introducing the notion of dependency graph. It specifies the relations between the legacy tool's input and the legacy tool's output. The graph is then used to address some complexities of the Reverse Migration. The improvement is illustrated by the reuse of a flattener tool defined on a specific metamodel of FSM (finite state machines).

1 Introduction

For a DSML (Domain-Specific Modeling Language), the reuse of legacy tools reduces the cost of producing its entire tool support. In [5], the reuse is applied to metamodel transformations. The specific functions for a DSML, are frequently provided by a legacy tool conforms to a variant of the DSML metamodel. It raises two questions: how the DSML model can be adapted to be conform to the legacy tool's metamodel? And how the output of the legacy tool can be adapted in return to the DSML context? *Modif* [1] addresses those two questions as shown in Figure 1.

The process compound of *Migration*, *Tool* and *Reverse Migration* operations correspond to the *Tool reuse*. The legacy tool metamodel *MM*, its conforming model *M1*, the legacy tool metamodel *MM'* and the legacy tool *Tool* are those we aim at reusing. Then, from *M1*, the objective is to obtain *M2*, *M3* and *M4* automatically.

Adaptation performs *Refactoring* operations at the metamodel level and *Migration* at the model level [2, 7]. The *Reverse Migration* is achieved by the Modif's *Contextualization* step. A common operation performed during *Migration* is deletion of unnecessary information (slicing [6]). Then, the *key* mechanism allows to recover the instances that have been deleted during *Migration*. This approach presents some limits when the legacy tool creates of aggregates different instances.

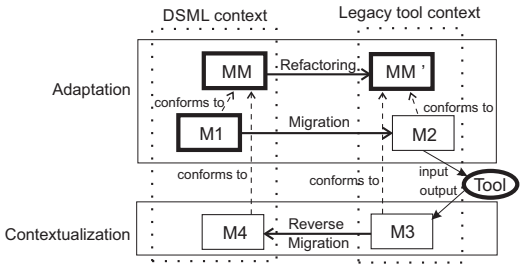


Fig. 1. Modif's legacy tool reuse's process

This paper is organized as follows. The next section presents the background of this work and some motivations. Then, we present the proposition to assist the user in the process of putting back the legacy tool's output into his DSML context, by means of a dependency graph. We finally conclude the paper and give some perspectives.

2 Background and Motivation

To obtain the model *M2*, Modif proposes a set of co-evolution operators (update, delete) like classically proposed by [3]. In the context of tool reuse, the most adapted operations are *rename* and *delete*.

Modif [4] proposes a *key* mechanism. A *key* is an attribute associated to each instance of *M1* that uniquely identifies it. The *key* mechanism allows to keep a relationship between instances of *M1* and those still exist in *M2* and *M3* after *Migration* and *Tool* application. Then, *M4* is build by adding *M3* instances and instances that have been deleted during *Adaptation*. This is possible by applying the concept of relational natural join of relational databases. To illustrate the approach and its limits, we present a case study of simple FSM:

- *MM* defines the concepts of state, transition, action (associated to states) and event (associated to transition). A state can contain other states inside it (hierarchical finite state machine);
- *MM'* is the metamodel of input data expected by a flattener legacy tool, it is similar to *MM*, except that it does not contain actions;
- *M1* (Figure 2) is a state-machine model conforms to *MM*;
- *M2* (Figure 2) is an adaptation of *M1*: actions are deleted;
- *Tool* removes hierarchy by producing atomic states (aggregation of super-states and states);
- *M3* (Figure 2) depicts the legacy tool's output.
- *M4* (Figure 2) illustrates the result of the *Reverse Migration* by using the *key* mechanism.

Figure 2 illustrates the way in which the states evolve and the keys are propagated. Only Ki by characterizing a state concept is shown. The actions associated

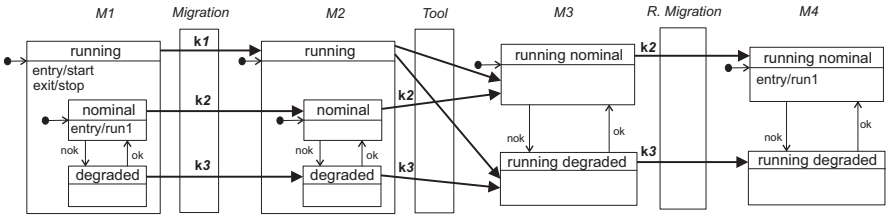


Fig. 2. States' evolution in a tool's reuse process

to a super-state cannot be recovered automatically. And, if *Tool* performs creation of new instances instead of updating the existing ones, it is not possible to recover any deleted action.

3 Proposition

We present a proposition to enhance *Modif* and its *keys* mechanism, by introducing the notion of *dependency graph*. A dependency graph is considered as an additional *specification* of the legacy tool. It specifies the dependencies between each instance of the legacy tool's output and the set of instances of the legacy tool's input. The set is compounded of the instances that are involved in the creation or modification of the legacy tool's input instance. In this paper, the dependency graph is obtained by instrumenting the legacy tool. We log each concept of the legacy tool's output and the set of concepts of the legacy tool's input that participates in its creation or modification.

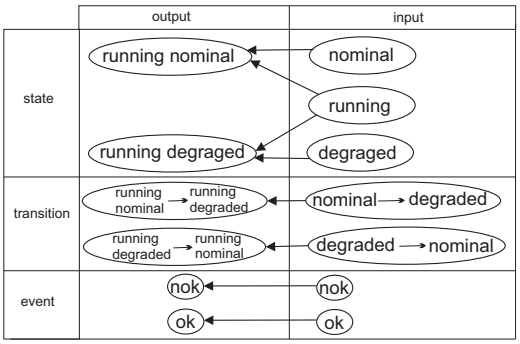


Fig. 3. Flattener dependency graph

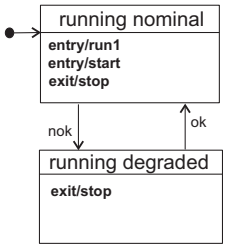


Fig. 4. Contextualization

For the case study of FSM, the *Reverse Migration* is applied by using the *key* mechanism, in order to recover deleted actions. Moreover we also use the

information given by the dependency graph to re-connect more actions. The dependency graph of this example is shown in Figure 3. Now, the challenge is about how to use the additional information to keep the recovered instances and to reconnect them.

We propose the following *by default* behavior for each recovered action: **R1)** If its related state in *M1* still exists in *M3*; the action is automatically connected to it. **R2)** If the state no longer exists in *M3*, but another states of *M1* are related to it and they still exist; then, the action is connected to all of them. **R3)** If the state no longer exists in *M3* neither the state related to it; then, the action is not connected to any state. The behavior is presented by means of auto generated code. If the designer does not agree this *by default* behavior, he can adapt it by integrating his requirements. An example of the additional behavior defined by an user is: **D1)** If the action is an *entry* one, it is reconnected with initial states: Adaptation of R1. **D2)** If the action is an *exit* one, it is reconnected to all states: same as R2.

The model *M4* obtained by following this behavior is presented in Figure 4. All actions are recovered and reconnected. *run1* stills related to *runing nominal*. *start* is connected to *nominal*, now *running nominal* because it is an initial state. *stop* is connected to all states. The approach using the dependency graph allows to keep at *Reverse Migration*, the DSML instances deleted during the *Migration*.

4 Conclusion and future works

In this paper, we present an approach to facilitate the legacy tool's reuse process. In particular, it improves the *Reverse Migration* for legacy tool's reuse by means of a dependency graph. The dependency graph provides an additional specification of the legacy tool to be reused. It enables to recover DSML instances deleted before using the legacy tool and to reintegrate them to its original DSML context. We are now working on the formalization of the*Reverese Migration*.

References

1. J.-P. Babau and M. Kerboeuf. Domain Specific Language Modeling Facilities. In *proceedings of the 5^{th} MoDELS workshop on Models and Evolution*, 2011.
2. K. Garcés, F. Jouault, P. Cointe, and J. Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of ECMDA-FA*, 2009.
3. M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE*, 2010.
4. M. Kerboeuf and J.-P. Babau. A DSML for reversible transformations. In *proceedings of the 11^{th} OOPSLA workshop on Domain-Specific Modeling*, 2011.
5. D. Mendez, A. Etien, A. Muller, and R. Casallas. Towards Transformation Migration After Metamodel Evolution. In *Model and Evolution Wokshop*, 2010.
6. S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model Pruning. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, Oct 2009.
7. G. Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proceedings of ECOOP*, 2007.