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).

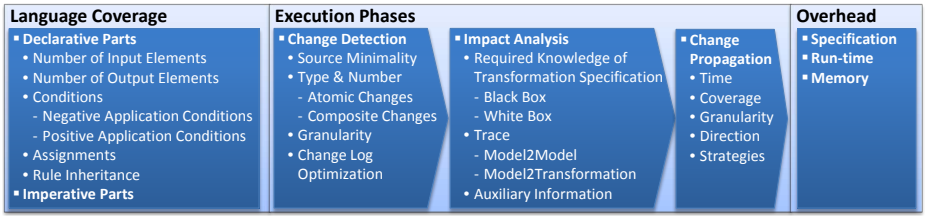

Fig. 1. Evaluation Criteria for Incremental Model Transformation Approaches

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)