Automatability of Coupled Evolution of Metamodels and Models in Practice

Markus Herrmannsdoerfer[1], Sebastian Benz[2], and Elmar Juergens[1]

[1] Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{herrmama, juergens}@in.tum.de
[2] BMW Car IT GmbH
Petuelring 116, 80809 München, Germany
sebastian.benz@bmw-carit.de

Abstract. Model-based software development promises to increase productivity by offering modeling languages tailored to a problem domain. Such modeling languages are often defined by a metamodel. In consequence of changing requirements and technological progress, these modeling languages and thus their metamodels are subject to change. Manually migrating models to a new version of their metamodel is tedious, error-prone and heavily hampers cost-efficient model-based development practice. Automating model migration in response to metamodel adaptation promises to substantially reduce effort. Unfortunately, little is known about the types of changes occurring during metamodel adaptation in practice and, consequently, to which degree reconciling model migration can be automated. We analyzed the changes that occurred during the evolution history of two industrial metamodels and classified them according to their level of potential automation. Based on the results, we present a list of requirements for effective tool support for coupled evolution of metamodels and models in practice.

1 Introduction

Due to their high degree of specialization, domain-specific languages (DSLs) are a promising approach to decrease software development costs by increasing development productivity. Consequently, a variety of metamodel-based approaches for DSL construction, such as Generative Programming [1], Model-Driven Architecture [2] and Software Factories [3], have been proposed in recent years. In response, DSLs are receiving increased attention in industry. BMW Car IT for instance applies DSLs for the specification of user interfaces [4] and test case generation [5]. With the integration of DSLs into industrial development practice, their maintenance is gaining importance. Although significant work in both research and practice has been invested into tool support for the initial development of DSLs, issues related to their maintenance are still largely neglected.

One important issue in language maintenance is the need to migrate existing models in response to the adaptation of their metamodel. As all software

artifacts, DSLs are subject to change. Although often neglected, even general-purpose languages are evolving because of changing requirements and technological progress [6]. Due to their close proximity to their problem domain, DSLs are even more prone to change, since many problem domains undergo continuous evolution. However, when a DSL's metamodel changes, dependent artifacts such as models, editors and interpreters may no longer obey the adapted metamodel. Among the required reconciliation efforts, migration of models is probably most challenging, since their number typically outweighs the other artifacts by far. Manually migrating models to a new version of their corresponding metamodel is costly, tedious and error-prone. In order to make cost-effective model-based software development feasible in practice, methods and tools are required to support efficient migration of models in response to changing metamodels.

We consider the evolution of a DSL as a *coupled evolution*[3] of its metamodel and models. Coupled evolution is a problem encountered in several areas of computer science [7], e.g. schema evolution, grammar evolution and format evolution. For each area, different approaches to reduce the associated effort – each with specific advantages and weaknesses – have been proposed. Recently, several approaches [8–10] have been transferred to the problem of metamodel evolution. Unfortunately, the requirements for effective tool support for coupled evolution of metamodels and models in practice are far from clear. On the one hand, transferability of existing approaches for e. g. schemata or grammars to the technical space of metamodels is difficult to assess: success of schema evolution approaches has been governed, amongst other things, by their ability to perform data migration in an online fashion; grammar evolution approaches have to take care to preserve the class of context free grammars that the employed parsing technology can handle – success or failure of these approaches in their respective domain can thus not immediately be transferred to metamodeling. On the other hand, and even more importantly, it is largely unknown to which degree changes occurring during language maintenance in practice can be automated. Can suitable tool support substantially reduce migration effort or are changes so domain specific, that they defy generic solutions? Given the increasing importance of language maintenance in practice, we consider this lack of understanding a precarious situation.

To provide a better understanding of language evolution in practice, and of how much associated effort can be reduced by adequate tool support, this paper presents a study of the histories of two industrial metamodels. To the best of our knowledge, this is the first work to examine the automatability of coupled evolution of metamodels and models in practice. The main contributions are:

- a classification of metamodel adaptation and corresponding model migration operations according to their potential for automation,
- an empirical study of the evolution of two industrial metamodels with respect to this classification, and
- a substantiated list of requirements for effective tool support in practice.

[3] Throughout the paper, we use the term *coupled evolution* instead of the term *co-evolution*, as we feel it better conveys the notion of coupling.

Outline. The remainder is structured as follows: In Section 2, we provide an overview of existing approaches to coupled evolution of specifications and instances. The proposed classification of combined metamodel adaptation and model migration is presented in Section 3. In Section 4, we outline the setup and results of the study we performed on the histories of two industrial metamodels. In Section 5, we discuss the results and derive requirements for efficient tool support. We conclude and give directions for future work in Section 6.

2 Related Work

When a specification changes, potentially all existing instances have to be reconciled in order to conform to the updated version of the specification. Since this problem of *coupled evolution* affects all specification formalisms (e. g. database or document schemata, types or grammars) alike, numerous approaches for *coupled transformation* [7] of a specification and its instances have been proposed. Apart from the target specification formalism, existing approaches mainly differ in their degree of automation and expressiveness, i. e. the kinds of coupled transformations they support. In this section, we outline different classes of approaches to coupled evolution, namely schema, grammar, format and metamodel evolution, focusing on their coupled evolution capabilities rather than on idiosyncrasies of their target specification formalism.

Schema evolution – the migration of database instance data to an updated version of the database schema – has been a field of study for several decades, yielding a substantial body of research [11]. For the ORION database system, Banerjee et al. propose a fixed set of change primitives that perform coupled evolution of the schema and data [12]. While highly automated, their approach is limited to local schema restructuring. To allow for non-local changes, Ferrandina et al. propose separate languages for schema and instance data migration for the O_2 database system [13]. While more expressive, their approach does not allow for reuse of coupled transformation knowledge. In order to reuse recurring complex coupled evolutions, SERF, as proposed by Claypool et al., offers a mechanism to define arbitrary new high-level primitives [14], providing both automation, reuse of coupled transformation knowledge and expressiveness.

Grammar evolution – the migration of textual programs to changes of their underlying grammar – has been studied in the context of grammar engineering [15]. Lämmel proposes a comprehensive suite of grammar transformation operations for the incremental adaptation of context free grammars [16]. The proposed operations are based on sound, formal preservation properties that allow reasoning about the relationship between grammars before and after transformation, thus helping engineers to maintain consistency of their grammar. However, the proposed operations are not coupled since they do not take the migration of words into account. Building on Lämmel's work, Pizka and Juergens propose a tool for the evolutionary development of textual languages called Lever, which is also able to automate the migration of words [17]. Primitive grammar and word evolution operations can be invoked from within a general-purpose language to

perform all kinds of coupled transformation. Similar to SERF, Lever provides a mechanism to define arbitrary new high-level primitives.

Format evolution denotes the migration of a class of documents to changes to their document schema. Lämmel and Lohmann suggest operators for format transformation, from which migrating transformations for documents are induced [18]. The suggested operators are based on Lämmels work on grammar adaptation. Furthermore, Su et al. propose a complete, minimal and sound set of evolution primitives for formats and documents and show that they preserve validity and well-formedness of both formats and documents [19]. Even though both approaches are able to automate document migration for a fixed set of format changes, they are not able to handle arbitrary, complex migrations.

Metamodel evolution denotes the migration of models in response to changes to their metamodel. In order to reduce the effort for model migration, Sprinkle proposes a visual, graph-transformation based language for the specification of model migration [20]. Gruschko et al. envision to automatically derive a model migration from the difference between two metamodel versions [9, 10]. Wachsmuth adopts ideas from grammar engineering and proposes a classification of metamodel changes based on instance preservation properties [8]. In order to automate model migration, the author plans to provide a predefined set of high-level transformations which represent the defined classes and are able to adapt the metamodel as well as to migrate models. Due to lack of reports on experience of their application, little is known on how effectively these approaches can be applied to coupled evolution of metamodels and models in practice.

In a nutshell, apart from differences between their target specification formalisms, existing approaches to coupled evolution mainly differ in the provided level of automation, expressiveness, reuse of coupled transformation knowledge and well defined preservation properties. To our best knowledge, little is known on the combination of capabilities that best supports the requirements faced during development and maintenance of metamodels and models in practice.

3 Classification

In this section, we introduce a classification that allows us to determine how far coupled evolution can be automated. Usually the metamodel is adapted manually and models are migrated at different levels of automation. The first level of automation is to encode a transformation that is able to automatically migrate a single model. A higher level of automation is achieved, if a single transformation can be used to migrate all existing models of a metamodel. When manually specifying such transformations, one discovers that they contain recurring patterns. Thus, the third level of automation corresponds to the application of generic transformations embodying such recurring patterns that automate both metamodel adaptation and model migration. In order to define the levels of automation, we introduce the notion of a coupled change. A *coupled change* is defined as a combination of an adaptation of the metamodel and the reconciling migration of the models conforming to that metamodel. Coupled changes

do not comprise metamodel changes that do not require a migration of models, e. g. additive changes. In the following, we introduce the classes in combination with representative examples, working our way up from lower to higher levels of potential automation.

3.1 Running Example

We use a simple modeling language for hierarchical state machines to illustrate our classification. Figure 1 depicts the metamodel and a corresponding model in both concrete and abstract syntax.

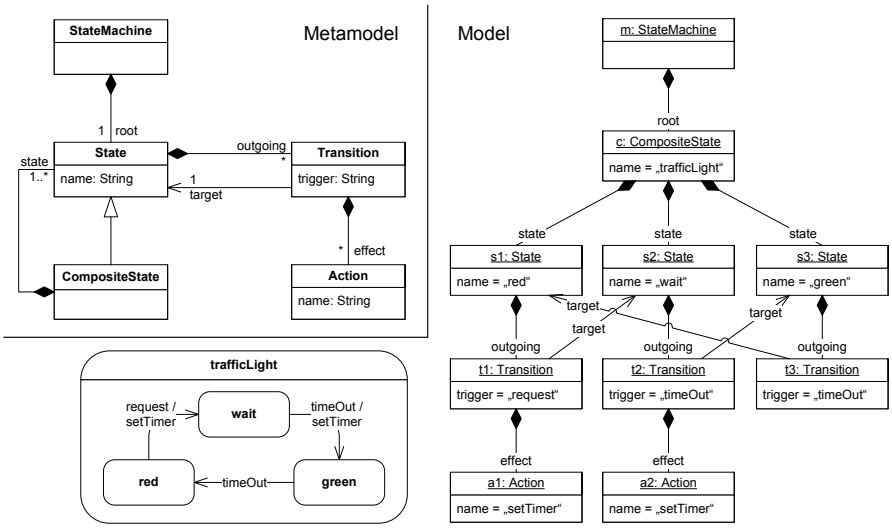


Fig. 1. State machine example

The root element of a state machine model is of type StateMachine and contains the root state. A State may be decomposed into sub states through its subclass CompositeState. A Transition has a target state and relates to its source state through the outgoing composition. Upon activation of a transition by its trigger, a sequence of Actions is performed as effect. Strings are used for state and action names and to denote triggers. The model describes the simplified behavior of a controller for a pedestrian traffic light and uses all the constructs defined by the metamodel. When the traffic light is red and a pedestrian requests a green phase, the controller transitions to wait and activates a timer (setTimer). When its timeOut occurs, the controller transitions to green and activates the timer again. When its timeOut occurs, the controller returns to state red.

3.2 Model-Specific Coupled Change

A coupled change is called *model-specific* if the migrating transformation is specific to a single model and thus cannot be reused to migrate different models of the same metamodel. This happens when the specification of a migration requires information which varies from model to model. Figure 2 depicts both metamodel and model after an example of a model-specific coupled change[4]. In the metamodel, the class Action is refined in order to group actions into specialized kinds. At the same time, the attribute name is deleted, by means of which actions were assigned a meaning before. As the class Action is itself made abstract, model elements of this type need to be migrated to a refined type in order to reestablish conformance and preserve information. As the metamodel was not precise enough to restrict the syntax of the action name, different names might be used in different models to denote the same kind of action. Therefore, model-specific information is required to be able to completely specify the migration.

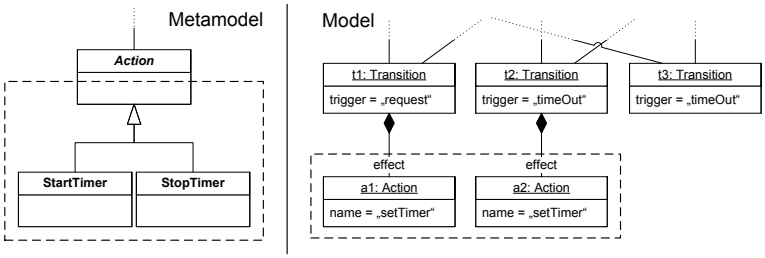


Fig. 2. Refinement of *Action*

3.3 Model-Independent, Metamodel-Specific Coupled Change

When a coupled change is not model-specific and all models of a metamodel can be automatically migrated, it is called *model-independent*. If the change is specific to the metamodel's domain, it is called *metamodel-specific*. In that case, its reuse across different metamodels makes no sense. Figure 3 depicts the impact of a metamodel-specific coupled change, which introduces a separate class for events. In the metamodel, the trigger attribute is factored out into the class Event. The state machine then administers all events and the transitions only refer to them by the trigger association. Since the attribute trigger is removed at the same time, models are no longer conforming to the modified metamodel. To readapt the model to the metamodel, an instance of Event has to be created for each distinct trigger name and to be assigned to the parent state machine. Therefore, only one event with name timeOut is created in the example and both transitions refer to it. As the migration is rather specific and therefore not likely to recur very often, it makes no sense to reuse this coupled change across metamodels.

[4] For better overview, modified elements are highlighted by a dashed box.

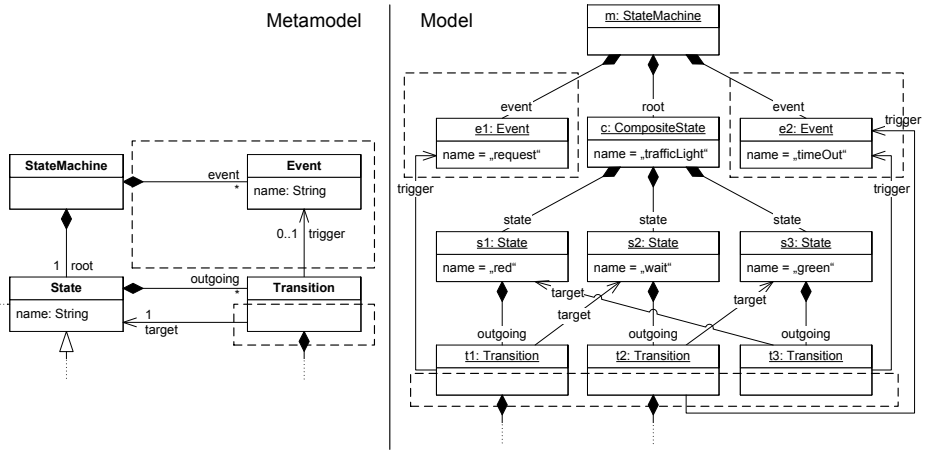


Fig. 3. Introduction of *Event*

3.4 Metamodel-Independent Coupled Change

A coupled change is called *metamodel-independent* if both metamodel adaptation and model migration do not depend on the metamodel's domain and can be expressed in a generic manner. If they are likely to recur in the evolution of different metamodels, it makes sense to generalize them into an operation that that can be reused to evolve other metamodels. Figure 4 depicts the impact of a metamodel-independent coupled change, which is a first step to introduce the concept of concurrent regions to the modeling language. In the metamodel, the class Region is introduced as a container of the directly contained sub states within a CompositeState. To compensate the change in a model, the migration creates a Region as child of each CompositeState and moves all directly contained sub states to the newly created Region. A possible generalization of this coupled change extracts a collection of features of one class into a new class which is accessible from the old class via a single-valued composition to the new class.

3.5 Summary

Figure 5 depicts an overview of the classification, and is augmented by a separate class for *metamodel-only* changes which do not require a migration of the models. For each class of coupled changes, the figure indicates to which level they are specific: The higher the level on which a coupled change depends, the more can be reused and therefore automated. A model-specific coupled change can only be used for a subset of the models of a metamodel. A metamodel-specific coupled change provides automation for all models of a metamodel. A metamodel-independent coupled change can be even applied to all metamodels and their models.

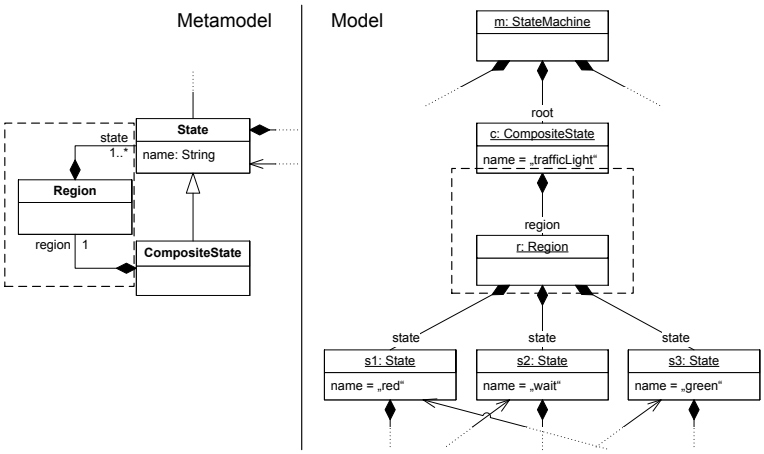


Fig. 4. Introduction of *Region*

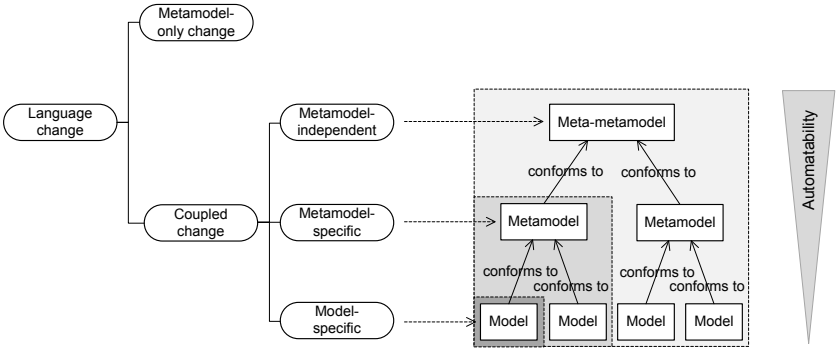


Fig. 5. Classification of coupled changes

4 Study

In order to asses the potential for automation in practice, we applied the classification to the histories of two industrial metamodels. In this section, we present the execution and the results of this study.

4.1 Goals

The study was performed to answer which fraction of language changes

- requires no model migration?
- is model-specific and thus defies automation of migration?
- is metamodel-independent and thus generalizable across metamodels?

4.2 Setup

Two industrial metamodels from BMW Car IT were chosen as input. Both metamodels were developed and maintained by several persons. FLUID (FLexible User Interface Development) is a framework for rapid prototyping of human machine interfaces in the automotive domain [4]. A metamodel defines a modeling language that enables the abstract specification of a human machine interface. An executable prototype of the human machine interface can be generated from a model written in that language. TAF-Gen (Test Automation Framework - Generator) is a framework to automatically generate test cases for human machine interfaces in the automotive domain [5]. The metamodel defines a statechart variant, a structural screen model and a test case language.

The histories of the metamodels were only available in the form of snapshots. A snapshot depicts the state of a metamodel at a particular point in time. As a consequence, further information had to be inferred to obtain the coupled changes leading from one metamodel version to the next. In order to achieve this, we performed the following steps[5]:

1. *Extraction of metamodel versions* (1 person week): Each available version of the metamodel was obtained from the revision control system used in the development of the metamodel[6].
2. *Comparison of subsequent metamodel versions* (2,5 person weeks): Since both revision control systems used are snapshot-based, they provide no information about the sequence of changes which led from one version to the following. Therefore, successive metamodel versions had to be compared in order to obtain the changes in a difference model. The difference model consists of a collection of primitive changes from one metamodel version to the next and has been determined with the help of tool support[7].
3. *Detection of coupled changes* (3 person weeks): Some primitive changes only make sense when regarded in combination with others. When an attribute is for example removed from a class and an attribute with the same name and type is added to its super class, then the two changes have to be interpreted as a compound change in order to conserve the values of the attribute in a model. Therefore, primitive changes were combined based on the information how corresponding model elements were migrated. The coupled changes between metamodel versions were documented in a table.
4. *Classification of coupled changes* (1 person week): The classification was applied to each coupled change.

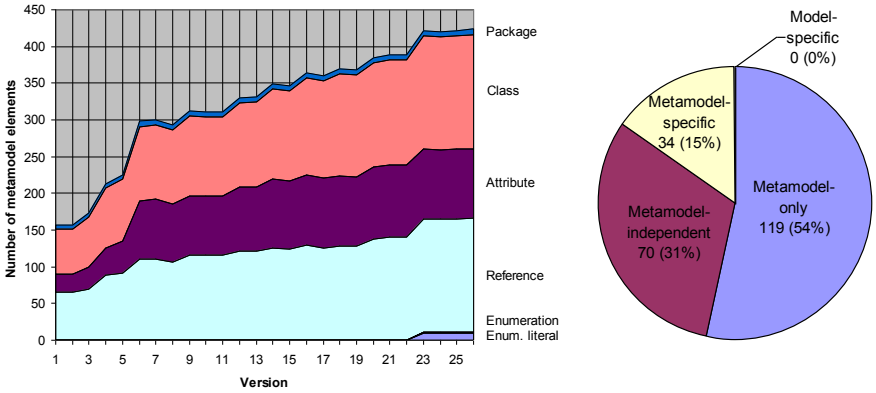
[5] In order to get an impression of the extent of the study, the approximate effort is mentioned in parenthesis for each step.

[6] The metamodels were specified by the Ecore metamodeling language from the Eclipse Modeling Framework (EMF, `http://www.eclipse.org/modeling/emf/`).

[7] Two prototypes now contributed to the EMF Compare tool (`http://www.eclipse.org/modeling/emft/?project=compare`) were applied.

4.3 Results

In this section, we present the results of the study in a compiled form. The full results are shown in Table 1 in the appendix[8]. In order to get a better impression of the evolution, Figure 6(a) illustrates the number of metamodel elements for each version of the FLUID metamodel. The area is further partitioned into the different kinds of metamodel elements. The figure clearly shows the transition from an initial development phase to a maintenance phase, where the grow in number of metamodel elements slows down. Figure 6(b) shows which fraction of all language changes falls into each class. 54% of the language changes can be classified as metamodel-only. No coupled change can be classified as model-specific, 15% as metamodel-specific and 31% as metamodel-independent. Note that each language change was counted as one, even though some changes were more complex than others.

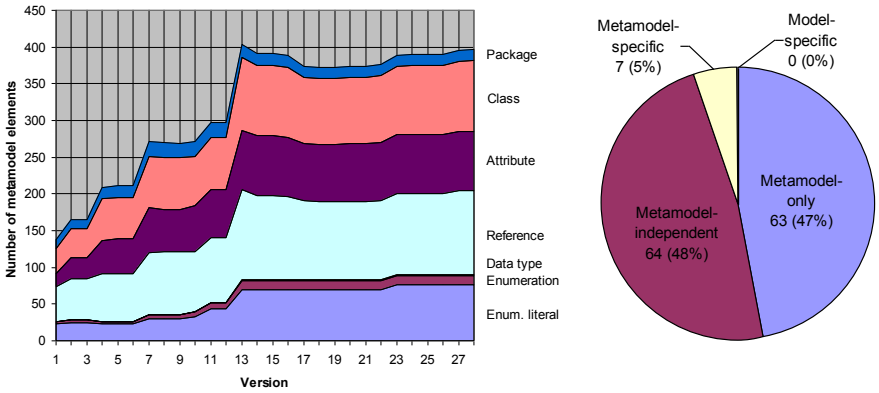


(a) Evolution in numbers (b) Classification

Fig. 6. Evolution of FLUID metamodel

Figure 7(a) depicts the number of metamodel elements for each version of the TAF-Gen metamodel. In this case the passage from the initial development phase to the maintenance phase is more distinctive and happens around version 13. Figure 7(b) illustrates the fragmentation of all encountered language changes into the four classes. 47% of the language changes have not required a migration at all. As in the history of FLUID, no coupled change is model-specific. The fraction of metamodel-independent coupled changes is even higher than in case of FLUID, amounting to 48% compared to 5% classified as metamodel-specific.

[8] Due to a non-disclosure agreement, we cannot provide more detailed information. Informal description of the language changes are made available through our website `http://wwwbroy.in.tum.de/~herrmama/cope`.



(a) Evolution in numbers (b) Classification

Fig. 7. Evolution of TAF-Gen metamodel

4.4 Discussion

The study showed that in practice the history of a metamodel can be split into mostly small language changes. We found out that most metamodel changes required a migration of existing models. Furthermore, snapshots from different metamodel versions are not sufficient to derive the model migration. As we have not found any model specific coupled changes, we would have been able to specify transformations to automate migration of all models. However, model-specific coupled changes cannot be entirely excluded and we plan to further investigate them. More than two thirds of all coupled changes were classified as metamodel-independent which provides a large potential for further automation. We also found a small number of metamodel-specific coupled changes and thus the coupled evolution was a combination of both metamodel-specific and -independent coupled changes.

4.5 Threats to Validity

The result of the study suggests a high degree of potential automation for model migration. However, threats need to be mentioned which can affect the validity of the result. They are presented according to the steps of the execution to which they apply together with the measures taken to mitigate them:

1. It was assumed that committing the metamodel to the revision control system indicates a new version of the metamodel. Therefore, only the primitive changes from one commit to the next were considered to be combined. However, metamodels were sometimes committed in a premature version and hence complex changes which span several commits of the metamodel threaten the validity. Even though enacted development guidelines at BMW

Car IT forbid to commit artifacts in a premature version, primitive changes of other metamodel versions were also taken into account when a migration could not be determined otherwise.

2. A prerequisite to determine the differences is the calculation of a matching between the elements of one metamodel version and those of the next. However, in the absence of unique and unchangeable element identifiers, the comparison cannot always be performed unambiguously [21]. Furthermore, the difference model leaves out changes which have been overwritten by others in the course of the evolution from one version to the next. In order to mitigate this threat, the correctness of the primitive changes was validated in close cooperation with the metamodel developers.
3. Unfortunately, models were not available for all versions of the corresponding metamodels. This poses a threat to the correct formation of coupled changes, since primitive changes were combined based on the associated migration. In order to mitigate this risk, the metamodel developers were exhaustively questioned about the correctness of the derived migration.
4. The differentiation between metamodel-specific or -independent coupled changes is not 100% sharp. Even though a generalization may be constructed for the most sophisticated changes, it is unlikely that it can be reused on any other metamodel. In order to mitigate this risk, we chose a conservative strategy: When we were not sure whether reuse makes sense, we classified such a coupled change rather metamodel-specific than metamodel-independent.

5 Requirements for Automated Coupled Evolution

Based on the results of the analysis, we discuss several requirements that an approach must fulfill in order to profit from the automation potential in practice.

Reuse of migration knowledge. In order to profit from the high number of metamodel-independent coupled changes found in the study, a practical approach needs to provide a mechanism to specify metamodel adaptation and corresponding model migration independent of a specific metamodel.

Expressive, custom migrations. As there was a non-negligible number of metamodel-specific coupled changes, the approach must be flexible enough to allow for the definition of custom metamodel adaptation and model migration. Since metamodel-specific changes can be arbitrarily complex, the formalism must be expressive enough to cover all evolution scenarios.

Modularity. In order to be able to specify the different kinds of coupled changes independently of each other, a practical approach must be modular. Modularity implies that the specification of a coupled change is not affected by the presence of any other coupled change.

History. Since models may be distributed and therefore not all models may be available during metamodel adaptation, a history is required that comprises the information to migrate the models at a later instant.

Existing approaches to automate model migration only satisfy the stated requirements to a certain degree: Sprinkle's visual language [20] does not provide a construct for the reuse of migration knowledge. Gruschko's approach [9, 10] leaves open how it achieves modularity and how it deals with complex custom migrations. And it remains unclear whether Wachsmuth's high level primitives [8] are able to perform all kinds of migration scenarios. In order to fully profit from the automatablity of coupled evolution in practice, an approach is needed that fulfills all the presented requirements.

6 Conclusion

We presented a study of the evolution of two real world metamodels. Our study confirmed that metamodels evolve in practice and that most metamodel changes require a migration of existing models. The study's main goal was to determine the potential for reduction of language evolution efforts through appropriate tool support. To this end, we categorized metamodel changes according to their degree of metamodel specificity. When a change is metamodel-specific, the corresponding model migration is as well. Otherwise, the model migration can be reused to migrate models that obey to different metamodels. Our results show that there is a large potential for the reuse of coupled evolution operations, because more than two thirds of all coupled changes were not metamodel-specific. If metamodel adaptation and model migration are encapsulated into a coupled operation, it is possible to reuse the operation for the evolution of different metamodels and their models. Such reuse of already tested coupled evolution operations can reduce maintenance effort and error likelihood. Nevertheless, a third of the coupled changes were specific to the metamodel's domain and therefore required a custom model migration. A metamodel hence evolves in a sequence of *metamodel-specific* and *-independent* changes. Therefore, an approach for automated model migration must support the reuse of coupled changes as well as the definition of new metamodel-specific changes. We hope that the results of this study can provide helpful input to guide the development of effective tool-support for coupled evolution of metamodels and models in practice.

References

1. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. Addison-Wesley, New York, NY, USA (2000)
2. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston, MA, USA (2003)
3. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
4. Hildisch, A., Steurer, J., Stolle, R.: HMI generation for plug-in services from semantic descriptions. In: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems (SEAS), Washington, DC, USA, IEEE Computer Society (2007)

5. Benz, S.: Combining test case generation for component and integration testing. In: Proceedings of the 3rd International Workshop on Advances in Model-based Testing (A-MOST), New York, NY, USA, ACM (2007) 23–33
6. Favre, J.M.: Languages evolve too! changing the software time scale. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE), Washington, DC, USA, IEEE Computer Society (2005) 33–44
7. Lämmel, R.: Coupled Software Transformations (Extended Abstract). In: First International Workshop on Software Evolution Transformations. (2004)
8. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: ECOOP 2007. Volume 4609 of LNCS., Springer Berlin / Heidelberg (2007) 600–624
9. Becker, S., Gruschko, B., Goldschmidt, T., Koziolek, H.: A Process Model and Classification Scheme for Semi-Automatic Meta-Model Evolution. In: Proc. 1st Workshop MDD, SOA und IT-Management (MSI), GI, GiTO-Verlag (2007) 35–46
10. Gruschko, B., Kolovos, D., Paige, R.: Towards synchronizing models with evolving metamodels. In: Proceedings of the International Workshop on Model-Driven Software Evolution. (2007)
11. Rahm, E., Bernstein, P.A.: An online bibliography on schema evolution. SIGMOD Rec. **35**(4) (2006) 30–31
12. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. Volume 16., New York, NY, USA, ACM (1987) 311–322
13. Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., Madec, J.: Schema and database evolution in the O2 object database system. In: Proceedings of the 21th International Conference on Very Large Data Bases (VLDB), San Francisco, CA, USA, Morgan Kaufmann (1995) 170–181
14. Claypool, K.T., Jin, J., Rundensteiner, E.A.: SERF: schema evolution through an extensible, re-usable and flexible framework. In: Proceedings of the seventh international conference on Information and knowledge management (CIKM), New York, NY, USA, ACM (1998) 314–321
15. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3) (2005) 331–380
16. Lämmel, R.: Grammar testing. In: Fundamental Approaches to Software Engineering. (2001) 201–216
17. Pizka, M., Juergens, E.: Automating language evolution. In: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE), Washington, DC, USA, IEEE Computer Society (2007) 305–315
18. Lämmel, R., Lohmann, W.: Format Evolution. In: Proceedings of the 7th International Conference on Reverse Engineering for Information Systems (RETIS). Volume 155., OCG (2001) 113–134
19. Su, H., Kramer, D., Chen, L., Claypool, K., Rundensteiner, E.A.: XEM: Managing the Evolution of XML Documents. In: Proceedings of the 11th International Workshop on research Issues in Data Engineering (RIDE), Washington, DC, USA, IEEE Computer Society (2001) 103
20. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. Journal of Visual Languages and Computing **15** (2004) 291–307
21. Robbes, R., Lanza, M.: A change-based approach to software evolution. Electron. Notes Theor. Comput. Sci. **166** (2007) 93–109

	Metamodel evolution	Model migration	Class	FLUID	TAF-GEN	Overall
pack.	add package	no migration required	MMO	0	10	10
	move package	compensate move	MMI	0	8	8
	remove empty package	no migration required	MMO	0	4	4
enum.	add enumeration	no migration required	MMO	0	4	4
	change order of enumeration	no migration required	MMO	0	1	1
	rename enumeration literal	compensate rename	MMI	0	4	4
class	add class	no migration required	MMO	0	3	3
	add sub class	no migration required	MMO	45	2	47
	change super class	remove data of lost features	MMI	0	1	1
	move class	compensate move	MMI	5	18	23
	remove class	remove objects	MMI	1	0	1
	remove sub class	remove objects	MMI	1	0	1
	rename class	compensate rename	MMI	5	5	10
	set class abstract	migrate objects to sub classes	MMS	1	0	1
	specialize super class (optional features)	no migration required	MMO	3	1	4
attribute	add optional attribute	no migration required	MMO	15	5	20
	add required attribute	set default value	MMI	5	1	6
	change type of attribute	convert values	MMI	2	1	3
	decrease upper bound of attribute	remove superfluous values	MMS	1	0	1
	drop attribute as key of class	no migration required	MMO	6	0	6
	extend multiplicity of attribute	no migration required	MMO	5	1	6
	make attribute an key of class	guarantee uniqueness	MMS	3	0	3
	move optional attribute to super class	no migration required	MMO	1	0	1
	move required attribute to super class	set default value	MMI	1	0	1
	remove attribute	remove attribute values	MMI	11	1	12
	rename attribute	compensate rename	MMI	3	1	4
association	add optional association	no migration required	MMO	5	4	9
	add optional composition	no migration required	MMO	3	3	6
	add required composition	create objects	MMI	3	0	3
	change type of association	remove links	MMI	5	0	5
	change type of composition	migrate objects	MMS	1	0	1
	decrease upper bound of association	remove superfluous links	MMS	1	1	2
	drop persistence of association	remove links	MMI	0	1	1
	establish persistence of association	no migration required	MMO	0	1	1
	extend multiplicity of association	no migration required	MMO	14	2	16
	generalize type of association	no migration required	MMO	6	4	10
	increase lower bound of association	no migration required	MMO	2	0	2
	make assocation bidirectional	no migration required	MMO	0	3	3
	move association by association	move links accordingly	MMI	0	1	1
	move association to sub classes	no migration required	MMO	0	2	2
	move optional association to super class	no migration required	MMO	1	1	2
	remove composition	remove part objects	MMI	3	0	3
	remove cross association	remove links	MMI	1	4	5
	remove opposite association	no migration required	MMO	0	2	2
	rename association	compensate rename	MMI	7	3	10
	transform association to composition	no migration required	MMO	1	2	3
	transform composition to association	assign otherwise	MMS	22	3	25
composite	add proxy	add proxy object	MMI	0	1	1
	extract part class	add part objects	MMI	6	2	8
	extract super class	no migration required	MMO	3	1	4
	inline part class	remove object of inlined class	MMI	0	1	1
	integrate association into another	move data accordingly	MMI	4	0	4
	integrate sub class	migrate objects to super class	MMI	0	2	2
	integrate super class	no migration required	MMI	0	1	1
	merge sub classes	migrate objects to common class	MMI	1	0	1
	remove proxy	remove proxy objects	MMI	1	0	1
	replace association by key	set identifier based on links and remove links	MMI	2	0	2
	replace key by association	add links based on identifier	MMI	2	0	2
	replace inheritance by composition	add part object	MMI	0	1	1
	use part class	add part objects	MMI	1	1	2
	use super class	no migration required	MMO	1	0	1
compl.	add optional part structure	no migration required	MMO	8	7	15
	remove part structure	remove data	MMI	0	6	6
	revolution	complex migration	MMS	5	3	8
				223	**134**	**357**
		Metamodel-only	MMO	**119**	**63**	**182**
		Metamodel-independent	MMI	**70**	**64**	**134**
		Metamodel-specific	MMS	**34**	**7**	**41**
		Model-specific	MS	**0**	**0**	**0**

Table 1. Number of occurred language changes and their classification