Challenges in evolving Metamodels

Misha Strittmatter, Robert Heinrich
Karlsruhe Institute of Technology (KIT)
{strittmatter | heinrich}@kit.edu

Abstract

Like every other software artifact, metamodels are subject to change even in later phases of the software life cycle. In this problem description paper, we first classify metamodel changes. We then elaborate on the challenges of metamodel evolution. The main challenges are the tight coupling of code to metamodels and the pervasiveness of metamodel dependencies. As this is a problem description paper, we will only present a brief overview of possible solutions.

1 Introduction

Metamodels started out as a way to define models of software (e.g. the UML metamodel). With the rise of the Eclipse Modeling Framework (EMF) [12], the usage of metamodels spread and was applied to other purposes. Now, software engineers use metamodels as a basis for analysis and simulation. They are further used for code generation, serve as an abstraction of the storage layer, are used simply as a data container and as a means of persisting data.

There are several characteristics, where the maintenance of metamodels differs from maintenance of code or software systems in general. When metamodels are used in software systems, they tend to be central artifacts. I.e., many other software artifacts depend on them or are even generated from the metamodel. Further, metamodels have no interfaces through which they could be used. Therefore, dependent code is coupled tightly to the metamodel. While there are extensibility mechanisms for code (plug-ins, patterns, inheritance ...), extension as an evolution technique is not very established with metamodels. Further, the way to modularize code (usually by functionality and responsibility) is very different from structuring metamodels, where many modularization options exist.

This paper is concerned with Essential Meta-Object Facility (EMOF) [11] conforming metamodels (e.g. instances of EMF's Ecore meta-metamodel). However, many aspects can be transferred to metamodels, which are conceptually similar. This paper covers metamodels of arbitrary purpose, as long as the purpose implies code being dependent on the metamodel. This includes code generation, analysis, simulation and editor-based graphical notations.

This paper is structured as follows: the remainder of the introduction will outline the state of the art in metamodel evolution and subsequently explain the foundations of metamodeling. Section 2 gives a brief overview of metamodel changes and their types of causes. Section 3 states the challenges of metamodel maintenance. Section 4 presents our experiences with the evolution of a specific metamodel: the Palladio Component Model (PCM) [1]. Section 5 points out possible ways to mitigate the problems of metamodel maintenance. Section 6 concludes the paper.

Regarding state of the art, much work deals with the evolution of metamodels [2, 3] and the co-evolution of models and metamodels [6, 4, 10]. However, the impact on the surrounding software system is often out of scope. Iovino et al. [7] present an approach to assess the impact of metamodel changes in Model-driven Engineering by using mega models. In an attempt to automate the evolution of the dependent software artifacts as much as possible, Di Ruscio et al. [5] propose a rule-based evolution support tool.

A metamodel defines and constrains the set of its instances (i.e. models). In the sense of EMOF, a metamodel consists of metaclasses, which in turn contain relations and attributes. If the elements of a model conform to the definitions in the metamodel, the model is an instance of the metamodel. EMOF-conforming metamodels are similar to UML class diagrams. The differences are that they have to be complete and have to form a containment tree.

The relations that can be defined in a metamodel connect two metaclasses. Attributes of metaclasses have only primitive types. Metaclasses are able to inherit from each other. A special case of a relation is the containment relation. Each element of a model, except the root, has to be contained in another element. A metamodel can be subdivided into packages. A metamodel may also reference metaclasses of another metamodels. Constraints can be defined (e.g. using the Object Constraint Language (OCL)).

Usages of metamodels are manifold. From the definition of a metamodel, model code can be generated. During runtime, this code can be instantiated to build or load models (i.e. the metamodel's instances). Rudimentary editors can also be completely generated. For more advanced graphical or textual editors, libraries and frameworks can be used (e.g. GMF, Graphiti [16] or Sirius [17]). Model transformation engines can be

used to alter models or transform them into another form. E.g. in model-driven software development, the code of a product is partially generated.

2 Overview of Metamodel Changes

There are several types of causes for metamodel changes. If new (functional) requirements arise after the metamodel has been specified, content may have to be added. This is also the case, if requirements did not actually change, but have not been met. Fixing an error or implementing the change of a requirement may also necessitate metamodel changes.

Changes to metamodels can be classified into two categories: *modifications* and *extensions*. A modification changes (rename, deletion, move, property change) the content of a metamodel, while an extension adds new content (i.e. metaclasses, relations or attributes). A modification can either be implemented *intrusively* in the metamodel or as a *branch*. Sometimes, intrusive modifications are unavoidable, especially if errors have to be fixed in the metamodel. The benefit of intrusive modification is, that the shared metamodel is kept identical for all dependent software. However, the change also affects all dependent software artifacts, which have to be adapted to be again compatible to the metamodel. In contrast, creating a branch and applying the modification only to that branch has the benefit that only the software has to be adapted that is of interest to that modification. The development of the branch is decoupled from the development cycle of the main branch. Changes in the main branch's metamodel do not impose any modifications of the software which works on the separate branch. However, this has the huge disadvantage, that the branch and software that uses it gradually get more and more incompatible to the main branch.

An extension can also be implemented *intrusively*, in a *branch* or *externally* in a new metamodel (a metamodel extension). External extensions have the advantage that the original metamodel is not altered. They are therefore the preferred approach. Technically, external extensions are always possible. However, in some cases, the original metamodel is not designed to support a particular extension (e.g. it requires extension points which are not present). In such cases, the extension cannot be implemented in a conceptually clean way but only as a workaround. The original metamodel could be changed to solve this problem.

The effort caused by metamodel changes increases the later the changes are carried out. E.g. it is easy to change a metamodel while it is being designed or initially implemented. It gets more and more costly to change it after it has been implemented and further software is developed on top of it. Thus, delaying refactorings has dormant consequences. If changes are not carried out, new functionality cannot be supported and bugs cannot be fixed which leads to increase in technical (metamodel) debt.

3 Challenges in Metamodel Maintenance and Evolution

By the nature of metamodels, software that is dependent on it is *tightly coupled* with it. From the outside, every metaclass of a metamodel can be referenced and every concrete metaclass can be instantiated. This means that in principle, each intrusive modification of a metamodel has implication onto external code. The more code depends on the metamodel, the higher the impact of the change.

The challenge of tight coupling is intensified by the fact, that in metamodel-centric systems, many modules or programs are dependent on the metamodel. When changing the metamodel intrusively, all these programs have to be adapted. Depending on the type of program, this can be done with relatively little effort, if the programs logic is oriented heavily on the structure of the metamodel (e.g. editors, validators). However, if an external functionality is implemented (e.g. a model is interpreted), the change impact can be grave. External extensions do not break dependent code. They simply are not supported until the needed functionality is implemented.

As with each software artifact, historically growing metamodels *degrade structurally* over time (due to intrusive extensions and modifications). It is possible to foresee and plan for future extensions to provide extension points. However, not all possible extension scenarios are known beforehand and some modifications cannot be avoided. Degradation of structure hinders all: intrusive extensions and modifications, branches as well as external extensions.

A further problem that plays into the degradation of structure over time is the *loss of knowledge*. If a metamodel is modified, the developer who carries out the change should have sufficient knowledge of the metamodel or else the change could be implemented incorrectly. If he has no prior knowledge, he has to spend time to learn and understand the metamodel. As developer teams change, rationale knowledge of the metamodel's design is lost. Decisions that have been intentionally made at one point may later seem counter intuitive to someone else. If the modeling is then changed or a workaround implemented, the initial good intention is lost and the stringency of the metamodel impaired. Loss of knowledge impedes all evolution types.

A hindering factor towards evolving metamodels are modifications made to generated code (model code, editor code ...). For changes of the metamodel to take effect, it is necessary to regenerate the code from the metamodel. In general, manual changes to the generated code are lost, as soon as the code is re-

generated from the metamodel. These changes then have to be reapplied to the generated code. The more changes have been made to the generated code, the more of a burden it becomes to regenerate and reapply the changes. This can go as far that the process of regeneration is delayed until a certain amount of changes to the metamodel has accumulated. It is even possible that changing the metamodel is avoided at all. As a workaround, it is possible to automate the reapplication of the changes. However, this reapplication is dependent on the metamodel structure. If the structure changes, the reapplication mechanism has to be co-evolved. Problems with regeneration encumber intrusive extensions, modifications and branches.

A further challenge is caused by remnant generated code. When regenerating, only existing classes are regenerated (they overwrite existing code). However, if a metaclass is deleted or renamed from the metamodel, the deleted class or the class with the old name is not deleted in the model code. When not aware of this, external code is still able to compile, but will not incorporate the metamodel changes and use outdated classes. Resulting errors are masked by the outdated code and thus are not easily identified. Remnant code interferes with intrusive modifications and branches.

Another challenge in metamodel design as in maintenance is finding the right compromise between a clean and clear metamodel and on the other side incorporating auxiliary content for tooling. A metamodel that only contains the necessary information to model a certain subject matter is precise, easy to understand and to evolve. However, the complexity and the efficiency of tooling which works on the metamodel can be improved by including utility content in the metamodel (e.g. additional attributes or relations for eased navigation). This should not be overdone, as there is the risk to encumber the metamodel. Furthermore, if a metamodel becomes too tool specific, there is the risk to impede its usability for specific tooling or its reusability in other contexts. Implementing the right degree of tooling information is a challenge in intrusive modification, branches and the initial design. In some cases, however, it can be factored out into external extensions.

Another trade-off which the metamodel developer has to tackle, even in maintenance, is implementing the right degree of extensibility and generality. Some extension scenarios have to be provided with extension points beforehand to be able to implement them in a conceptually clean way. A metamodel for a very specific purpose may be very precise. However, as requirements change, such a precise metamodel may turn out to be inflexible and not well suited for extension. On the other side, too many predefined extension points increase the complexity of the metamodel. Making a metamodel too general makes it too abstract, which impedes its use- and reusability. The extensibility trade-off is a challenge for all evolution types.

4 Palladio Component Model

In our research group, we face the challenges of evolution and maintenance [14, 13] of the *Palladio Component Model* (PCM) [1]. We present this case as a concrete example. However, most insights in this paper are applicable to metamodel-centric software systems in general. The PCM is a modeling language (defined by a metamodel) to describe component architectures of software systems. It is the centerpiece of the Palladio Approach [1]. The PCM is modeled in Ecore. In the versioning system, its files can be traced back to 2006. There are editors for the PCM as a graphical notation. Several simulations and analyses exist, which operate on PCM models. Most of them perform performance prediction, as this is the PCM's original focus. There are also some which are concerned with security analyses, reliability, maintainability etc. There are extensions to the PCM, which enrich it by further information like: requirements, patterns, design decisions. There are editors for PCM models, transformations to and from other languages as well as model extractors.

The concerns that were central to the original purpose of the PCM were incorporated directly in the metamodel. However, as the focus broadens, these concerns are irrelevant for certain new purposes. The package structure evolved historically. So its internal structure is not optimal. Concerns are spread over packages and packages contain multiple concerns. The dependencies between packages have grown wild. Intrusive extensions were implemented. Branches exist, where separate developments were continued.

The main challenge that we face in evolving the PCM is the amount of software that is dependent on it. Changing the metamodel implies changes in many dependent components. This is amplified by the tight coupling to the metamodel, as there is no extra abstraction layer to decouple code from the metamodel. As the PCM has reached a certain age, the loss of knowledge, especially about design decision rationale, is also affecting us.

5 Solutions and Best Practices in Metamodel Evolution

As it is not the focus of this problem description paper, we will only give an overview of possible ways to overcome the challenges of metamodel evolution. Some of these approaches, techniques and best practices also work preventatively. They reduce the need to modify a metamodel.

Of course, a (1) *good initial design* and a proper (2) *assessment of the requirements* can prevent the need for evolution to a degree. Foreseen extensibility should be regarded in the design. Implementing

new content as (3) *external extensions* (e.g. aspect-oriented extension [8]) mitigates many of the drawbacks from intrusive extension. However, not all extensions can be foreseen. The right amount of (4) *modularity* helps in giving extensions a clean base where they can build upon. A well designed (5) *structure* of packages and extensions is important to maintain long-living metamodels properly. It is important to decide the order of modularization criteria consciously. An explicit (6) *reference structure* helps in making the structure explicit (e.g. for quality-aware ADLs [15]). A conscious decision for a (7) *migration strategy* is important (slow iterative vs. big bang). The implications of the big bang migration method can be somewhat mitigated with a transformation between the old and new metamodels. Clear (8) *conventions* (especially about extension), good knowledge of metamodeling (9) *best practices* and rigorous (10) *documentation* (especially tracing of design decisions and requirements) are important. Monitoring (11) *metrics* (similar to regression testing in continuous integration) can help in detecting detrimental modifications. (12) *Explicit treatment of dependencies* between packages helps in keeping dependencies correct and precise. If a new dependency is introduced between two packages that are not yet dependent in that direction, it should be reviewed for bad smells and correctness. Dependency cycles between packages (and extensions) should be avoided.

The problem of (13) *regeneration* can be tackled in multiple ways. Some but not all frameworks allow separation of generated and manual code. This is the case with the Ecore generator and Spray [18], a generator for the Graphiti editor framework [16]. When regenerating, only the generated code is overwritten. The Sirius editor framework [17] performs its GMF generation transparently for the user. User defined free-form functionality can be added using generic extension points. The availability of these solutions, however, is limited to specific frameworks. To harness them for code generation in general, they have to be implemented in the corresponding generator.

There are further possible solutions that are, however, subject to research and are currently not usable for Ecore. (14) *Visibility* of abstract metaclasses could be restricted to prohibit illegitimate inheritance. Proper support of a (15) *extension mechanism* will improve reuse. If matured, (16) *synchronization* approaches for models and code [9] could be used to keep metamodels and generated code consistent without losing manual code on metamodel changes.

6 Conclusion

In this problem description paper, we have given a brief overview and classification of metamodel changes. We presented the challenges of metamodel maintenance and evolution, as well as briefly outlined possible solutions.

In conclusion, the potential effort caused by metamodel evolution is grand. How to best prevent or handle it is still an open question. Tackling these problems as early as possible is important, as the required effort increases over time. Proper assessment of requirements can alleviate some problems. Extensibility should be regarded in the design of metamodels. For future research, promising approaches are modular metamodels, domain specific reference structures, metamodel interfaces and visibility constraints within and between metamodels.

Acknowledgments

This work was partially funded by the Helmholtz Association of German Research Centers. We'd like to thank Michael Langhammer, Philipp Merkle and Kiana Rostami for their input. We also thank Marco Konersmann, Emre Taspolatoglu and the peer reviewers for their feedback.

References

[1] Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. to appear.

[2] Burger and Gruschko. A Change Metamodel for the Evolution of MOF-Based Metamodels. *Modellierung 2010*, 285–300. GI.

[3] Burger and Toshovski. Difference-based Conformance Checking for Ecore Metamodels. *Modellierung 2014*, 97–104. GI.

[4] Cicchetti et al. Automating co-evolution in model-driven engineering. *EDOC 2008*, 222–231. IEEE.

[5] Di Ruscio et al. What is needed for managing co-evolution in MDE? *IWMCP 2011*, 30–38. ACM.

[6] Favre. Meta-model and model co-evolution within the 3D software space. *ELISA 2003*, 98–109.

[7] Iovino et al. On the impact significance of metamodel evolution in MDE. *JOT 2012*, 11(3):3–1.

[8] Jung et al. A method for aspect-oriented meta-model evolution. *VAO 2014*, 19–22. ACM.

[9] Kramer et al. View-centric engineering with synchronized heterogeneous models. *VAO 2013*, 5:1–5:6. ACM.

[10] Narayanan et al. Automatic domain model migration to manage metamodel evolution. *MODELS 2009*, 706–711. Springer.

[11] Object Management Group. MOF 2.4.2 Core Specification, 2014.

[12] Steinberg et al. *EMF: eclipse modeling framework*. Pearson, 2008.

[13] Strittmatter and Langhammer. Identifying semantically cohesive modules within the palladio metamodel. *SSP 2014*, 160–176. UB Stuttgart.

[14] Strittmatter et al. Towards a modular palladio component model. *SSP 2013*, 49–58. CEUR.

[15] Strittmatter et al. A modular reference structure for component-based architecture description languages. *ModComp 2015*, 36–41. CEUR.

[16] Graphiti homepage. https://eclipse.org/graphiti/.

[17] Sirius homepage. https://eclipse.org/sirius/.

[18] Spray homepage. http://eclipselabs.org/p/spray/.