A Method for Aspect-oriented Meta-Model Evolution

Reiner Jung
Software Engineering Group, Kiel University
reiner.jung@email.uni-kiel.de

Robert Heinrich
Software Design and Quality, Karlsruhe Institut of Technology
heinrich@kit.edu

Eric Schmieders
Paluno - The Ruhr Institute for Software Technology
eric.schmieders@paluno.uni-due.de

Misha Strittmatter
Software Design and Quality, Karlsruhe Institut of Technology
strittmatter@kit.edu

Wilhelm Hasselbring
Software Engineering Group, Kiel University
hasselbring@email.uni-kiel.de

ABSTRACT

Long-living systems face many modifications and extensions over time due to changing technology and requirements. This causes changes in the models reflecting the systems, and subsequently in the underlying meta-models, as their structure and semantics are adapted to adhere these changes. Modifying meta-models requires adaptations in all tools realizing their semantics. This is a costly endeavor, especially for complex meta-models.

To solve this problem we propose a method to construct and refactor meta-models to be concise and focused on a small set of concerns. The method results in simpler meta-model modification scenarios and fewer modifications, as new concerns and aspects are encapsulated in separate meta-models. Furthermore, we define design patterns based on the different roles meta-models play in software. Thus, we keep large and complex modeling projects manageable due to the improved adaptability of their meta-model basis.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages

Keywords

Design Pattern, Aspect Modeling, Evolution, Meta-Model Extension

1. INTRODUCTION

Long-living software systems face diverse changes over time induced by new or altered requirements, and changes to environment and technology, such as service-oriented or cloud paradigms [9]. These changes require an adaptation of the models representing the system and their underlying meta-models due to domain alterations. Furthermore, the boundary between design-time and run-time disappears as models are used in all phases of the MAPE loop [7], requiring models to support new features, such as traceability. The Palladio [1] architecture simulator, for example, has been originally developed to predict performance of software systems, based on a component-based architecture meta-model, which is called the Palladio Component Model (PCM). Later, the PCM has been extended to support the analysis of reliability [4], to support business process modeling and simulation [10], and even an extensions for software process modeling and analysis have been proposed [11].

Such extensions cover specific views and concerns of a software system and require alterations to the meta-models providing the means to specify such systems. However, alterations of meta-models require the adaptation of editors, simulators, transformations, and all other dependent software components used in the development and operation of software systems, which become more costly with increasing size of meta-models and their semantics. To reduce evolution costs and to foster meta-model re-use, meta-models must be designed to be extendable in a non-invasive manner. Furthermore, extensions should not require new meta-meta-models as this makes already existing tooling useless. In this paper, we propose an aspect-oriented meta-modeling method that fulfills these criteria.

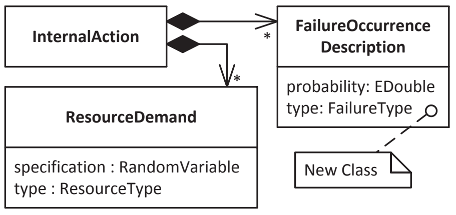

Figure 1: Example excerpt from the PCM

We demonstrate the benefits of our method in comparison to related work using the following example: In PCM, services which are provided by components, are specified by flowchart-like submodels called *Service Effect Specifications* (SEFF), which comprise different kinds of actions, amongst them InternalActions. An InternalAction contains ResourceDemands which are important to predict the performance of the system. While extending the PCM with reliability properties, amongst other changes, a new containment reference was added from the InternalAction to the introduced FailureOccurrenceDescription class (see Figure 1), to be able to express the types of failures and their respective failure probability of an InternalAction. However, such an intrusive extension approach has drawbacks. As further content is added to meta-models they may become complex and cluttered with features, which are not always needed.

In Section 2, we discuss how related approaches deal with this example. In Section 3, we describe how the extension could be made non-intrusively using our method. Four use cases of the proposed method are discussed in Section 4. We conclude the paper in Section 5 and discuss future work.

2. RELATED WORK

Meta-model extension with the Eclipse Modeling Framework (EMF) has been addressed by different approaches, based on EMF features, object oriented pattern, and additional frameworks. In this section we discuss them briefly.

2.1 EMF Meta-Model Extension

EMF has a meta-model extension mechanism, where a meta-model can load another meta-model and then define new classes or reference and subclass classes from the loaded meta-model. To add properties to a class, a subclass is created with the additional properties.

Our example can be implemented with this method, by creating a new meta-model, load the PCM, and subclass InternalAction in the new meta-model. Then add FailureOccurrenceDescription to the new meta-model and provide the proper containment references between both classes.

However, this method has drawbacks. First, multiple extensions cannot be used simultaneously when developed independently. Second, if the method is iterated to realize multiple extensions, they form a chain of meta-models making it impossible to use an arbitrary subset of them. And third, tools for the original meta-model are unable to load models based on extended meta-models, requiring tool adaptation, even if the additions are irrelevant for its operation.

2.2 Decorator Pattern

The decorator design pattern [6] can also be used as an extension mechanism in EMF. It enables addition of information to a class without the aforementioned drawbacks. However, the pattern requires to foresee future class extensions and it introduces additional classes resulting in a more complex meta-model. Software utilizing such meta-models must deal with stacked decorators, which results in more complex code, and some editor frameworks, like GMF or Spray, are unable to handle the decorator pattern.

A realization of the illustrative example shows this complexity: The decorator implementation is realized through three classes AbstractInternalAction replacing InternalAction in the class hierarchy, ConcreteInternalAction representing the properties of InternalAction, and AbstractDecorator realizing the model extensions through a reference to AbstractInternalAction. In a new meta-model FailureOccurrenceDescription is specified together with a subclass of AbstractDecorator containing the reference to FailureOccurrenceDescription.

2.3 EMF Profiles

EMF is a framework to define meta-models and construct models. In contrast to UML, it lacks the facility of profiles, a situation the EMF profile approach [16] aims to change. In contrast to UML profiles, which allow only one stereotype per class, EMF profiles support multiple stereotypes even of the same type. This enables them to be used as a meta-model extension mechanism.

When applying EMF profiles to our example, specifies FailureOccurenceDescription as stereotype for InternalAction keeping the original meta-model unaffected. However, EMF profiles rely on an extended meta-meta-model which cannot be used with genmodels to create Java classes, in contrast to plain Ecore meta-models. It can only be accessed through an API similar to dynamic EMF which is significantly slower and harder to debug, as types are not handled by the Java compiler. Furthermore, this excludes widely used modeling tools and frameworks, like the Xtext DSL framework, the auto-layout and diagram framework KLighD [17], and the Xtend transformation language.

3. META-MODELING METHOD

Meta-models undergo many modifications and extensions. We therefore propose patterns to construct, maintain and extend meta-models, which work together with existing EMF tooling and follow a set of design principles [16]. Central to our method is the differentiation in aspect and base meta-models. Aspect meta-models allow for extending base meta-models in a non-intrusive way. Thus, our method fosters re-use of aspect and base meta-models. In contrast to EMF profiles or the decorator pattern approach, our method does not require a new framework or meta-meta-model hindering its use together with existing techniques and tooling. Furthermore, we identified different types of meta-models which are determined by their roles in software systems. These contextual roles together with the distinction of aspect and base meta-model form the basis for a non-intrusive meta-model design and evolution method.

3.1 Fundamental Properties of Meta-Models

Meta-models represent an abstract notation designed to describe concepts and knowledge of a specific domain, view or aspect of a system, like architecture description, performance annotations, business processes, or medical history.

In EMF, meta-models comprise, inter alia, classes, attributes and references. Classes contain attributes and references, based on primitive types and classes, respectively. References are distinguished in normal or aggregation references, and containment references. The latter describe explicitly the containment hierarchy of the meta-model.

The containment references form a directed graph where the classes are nodes and references are edges. A meta-model which allows to form self-contained models is one where all classes can be reached by a path over containment references from a root node. We call the aggregation of all paths an containment graph. The meta-models in this paper need to fulfill these criteria.

Beside syntactical properties, meta-models have semantics which are not always defined explicitly. In general, a meta-model can be divided in parts defining structure, expressions and common primitives, which is very similar to computer languages. The semantic of the structural part can therefore be specified with type system methods, while expressions can be described with operational semantic rules. For EMF based meta-models, tooling is already available to specify the semantics of a meta-model [2].

3.2 Aspect and Base Meta-Model Relationship

Core and cross-cutting concerns are represented in aspect-oriented modeling (AOM) through base and aspect models [15]. In some AOM approaches base and aspect model use the same meta-model to ease model weaving. However, in context of domain-specific languages and the complexity of long-living systems, base and aspect models conform to different meta-models expressing different concerns. This allows to focus meta-models on the concerns they describe instead of their technical realization. Furthermore, aspect meta-models used as an extension method, are created after the base meta-model and introduce therefore new classes and terms not present in the base meta-model.

In our example, the existing base meta-model comprises InternalAction and ResourceDemand, while the new aspect meta-model expresses failure properties through FailureOccuranceDescription. The relationship between FailureOccuranceDescription and InternalAction can be realized in two different ways. First, in case of an aspect meta-model designed to be re-used with different base meta-models, a base model query and reference meta-model must be added to the aspect meta-model (cf. [12]). Second, in case of extending a specific meta-model, a much simpler pattern can be used, where the FailureOccuranceDescription owns a reference pointing to an InternalAction realizing the annotation.

While our minimalistic example only comprises a single class in the aspect meta-model, conceptually aspect meta-models can be of any size. They can be used to encapsulate and separate cross-cutting and other concerns, like deployment, behavior and configuration. The distinctive feature between base and aspect meta-model is the direction of the references, which must originate in the aspect meta-model and end in the base meta-model. These two roles of meta-models are contextual, meaning a base meta-model in one relationship could be the aspect meta-model in another.

The previously expressed containment graph requirement of meta-models and the directionality property for aspect references, can be used as a separation criterion for legacy meta-models. However, the criterion only shows potential candidates. Therefore, the concerns represented by the meta-model parts must be consulted to select valid candidates. After a selection is made, the identified meta-models are often in violation of the directionality property, which can be corrected by reversing references like in our example.

A special case for our method is the use of models at runtime in transformations or interpreters. Due to the non-intrusive nature of the method, looking up aspect information for a specific node is time consuming. However, this can be circumvented with an additional mapping meta-model providing references in the inverse direction (see Section 4).

3.3 Contextual Meta-Model Patterns

In the previous section, we differentiated between base and aspect meta-models. However, meta-models can also be distinguished by their tasks or use cases. First, navigation and traceability are functions required in many modeling contexts. Second, meta-models are used to describe derived information. And third, meta-models are used to model expressions, data structures, and state.

Model navigation is generally realized by explicitly defined references in a meta-model, and as stated before, by inverse references implied by the containment hierarchy, as realized in EMF [18]. We, therefore, encourage meta-model designers to remove explicit inverse references as they are unnecessary and make meta-model comprehension more difficult.

On a more abstract level, references have different semantics: A reference in an aspect meta-model represents a join point, while the reference from InternalAction to ResouceDemand is a containment reference implying that both classes together form one entity. A *traceability reference* expresses the fact that one node is derived from another node or subgraph. Traceability information is used in transformations to guide model weaving, or to relate analysis results based on derived models back to the original model.

Frequently, meta-models are complex and not always suited directly for the analysis. Therefore, specialized models are derived, which require meta-models specific to the task, e.g., queuing models. After the analysis traceability references are used to relate the results back to the original model, otherwise the result is meaningless. These references are either part of the meta-model for the derived model (requires access to the meta-model), or alternatively are represented by a utility meta-model realizing the mapping either through a hash map, relating derived model nodes to original model nodes, or a more complex mapping based on subgraphs.

The third group of meta-models is related to the execution or interpretation of models, which is a reoccurring task in long-living systems, like software behavior forecasts or measurement evaluation. They require data structures, expressions, and states which have different characteristics. Data meta-models do not possess references to meta-models of other types, as they are only used to store data. Execution meta-models comprise notational features determined by underlying paradigms, used to specify behavior. They require access to a type system (cf. [13]) and to data structures. The third meta-model type is used to preserve the state of an interpreter, comprising references to the data and execution meta-model. It provides the base to combine data and execution meta-model, and can be seen as the equivalent of program counter and registers in hardware.

4. USE CASES

In context of long living software systems, we identified four key use cases: editors, transformations, simulations or evaluations, and runtime models (cf. [16]), which highlight different characteristics meta-models must fulfill.

4.1 Editors

EMF based editors can be constructed with various frameworks, like Xtext[1], or KLighD [17]. In context of our illustrative example, we have an editor supporting PCM as base meta-model unaware of a new aspect meta-model. However, it is still usable, as the base meta-model is unchanged. This is especially helpful when we cannot modify the editor.

In editors, where we can modify the code base, an extension mechanism for aspect handling can be integrated, e.g., OSGi-based plug-ins. Node editing in the base model could then be checked for breaking references and handled accordingly, either by prohibiting the modification or by updating the reference in the aspect model. Furthermore, the corresponding aspect editor could be opened. A behavior already supported by DSL editors based on Xtext or KLighD.

4.2 Transformations

Transformations have manifold properties relating to their context, the involved meta-models, and the level of abstraction [3]. A transformation relies heavily on navigation to query the source model and determine the correct insertion point of new nodes in the target model. In context of AOM, we assume separate transformations for aspect and base meta-model. An aspect model contains references to an external base model. While the origin of these references

[1]http://www.eclipse.org/Xtext/

lie inside the scope of the aspect transformation, the destination is located in the base model. To transform the references correctly, the aspect transformation requires model traces relating base source model nodes to their target counterparts. This information can either be collected during a run of the base transformation, or calculated by a function.

4.3 Simulation and Evaluation

Interpreters, simulators, and model checkers are used to analyze or predict software behavior. From a modeling point of view, they require terms for evaluation, data, and state which are processed by an algorithm implementing the semantics of the tool. In context of base and aspect models, a simulator for a base meta-model is not aware of nodes which are annotated by external information. While it is useful for aspects not relevant to the simulation, it limits the expressiveness of the simulation. Therefore, we devise an extension mechanism. First, a utility model describes inverse references for the aspect references to improve the lookup time for aspect annotations. This map can be computed at start time. Second, an aspect interpreter is invoked within the present context of the base model simulator processing the aspect. The extension of the base simulator can be realized technically in different ways, like injection, load-time weaving or with OSGi-plugins.

4.4 Runtime Models

Run-time models provide abstract views on systems and their contexts during run-time [8] derived from design-time models and run-time monitoring data [5, 12]. Autonomic managers utilize these views to self-adapt a system to reach the systems goals [14]. While analysis tools which operate on design-time models [1] exist, approaches, such as [19], propose tailored views for relevant aspects of the system to avoid complex and overloaded views. In context of our method, this is achieved with derived models in conjunction with built-in or a separate traceability model.

5. CONCLUSION

Our lightweight method guides the construction and evolution of meta-models in the context of long-living software. In contrast to other methods and approaches we do not clutter meta-models with helper constructs, nor do we require new frameworks incompatible with the tooling landscape.

Our next steps are to create a detailed description of various meta-models types and their implications on tooling, and the evaluation of our method in the PCM modernization effort to provide a solid core PCM with a wide palette of extensions covering a multitude of knowledge domains.

6. ACKNOWLEDGMENTS

This work was partially supported by the DFG (German Research Foundation) under the Priority Program SPP1593: Design For Future – Managed Software Evolution (grant HA 2038/4-1, RE 1674/6-1, PO 607/3-1) and the Helmholtz Association of German Research Centers.

7. REFERENCES

[1] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *J. of Systems and Software*, 82:3–22, 2009.

[2] L. Bettini. Implementing java-like languages in xtext with xsemantics. In S. Y. Shin and J. C. Maldonado, editors, *SAC*, pages 1559–1564. ACM, 2013.

[3] M. Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, July 2010.

[4] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner. Architecture-based reliability prediction with the Palladio component model. *IEEE Transactions on Software Engineering*, 38(6):1319–1339, 2012.

[5] K. I. Eder, N. M. Villegas, F. Trollmann, P. Pelliccione, H. A. Müller, D. Schneider, L. Grunske, B. Rumpe, M. Litoiu, and A. Perini10. Assurance using models at runtime for self-adaptive software systems. In *State-of-the-Art Survey on Models at Runtime*. Springer, Berlin, LNCS, 2013.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[7] C. Ghezzi. The fading boundary between development time and run time. In *Web Services (ECOWS), 2011 9th IEEE European Conf. on*, page 11, Sept 2011.

[8] H. Giese and T. Vogel. *Model-driven engineering of adaptation engines for self-adaptive software*. Universität Potsdam, 2013.

[9] W. Hasselbring, R. Heinrich, R. Jung, A. Metzger, K. Pohl, R. Reussner, and E. Schmieders. iobserve: Integrated observation and modeling techniques to support adaptation and evolution of software systems. Research report, Kiel University, Kiel, Germany, October 2013.

[10] R. Heinrich, J. Henss, and B. Paech. Extending Palladio by business process simulation concepts. In *KPDAYS*, pages 19–27, 2012.

[11] O. Hummel and R. Heinrich. Towards automated software project planning - extending Palladio for the simulation of software processes. In *KPDAYS*, pages 20–29, 2013.

[12] R. Jung, R. Heinrich, and E. Schmieders. Model-driven instrumentation with Kieker and Palladio to forecast dynamic applications. In *KPDAYS*, pages 99–108, 2013.

[13] R. Jung, C. Schneider, and W. Hasselbring. Type systems for domain-specific languages. In S. Wagner and H. Lichter, editors, *Software Engineering 2013 Workshopband*, volume 215 of *Lecture Notes in Informatics*, pages 139–154, Bonn, February 2013. Gesellschaft für Informatik e.V.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[15] J. Kienzle, W. A. Abed, and J. Klein. Aspect-oriented multi-view modeling. In *AOSD*, pages 87–98, 2009.

[16] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. EMF Profiles: A lightweight extension approach for emf models. *JOT*, 11(1):1–29, 2012.

[17] C. Schneider, M. Spönemann, and R. von Hanxleden. Just model! – Putting automatic synthesis of node-link-diagrams into practice. In *Proceedings of VL/HCC*, San Jose, CA, USA, 15–19 September 2013.

[18] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[19] T. Vogel, S. Neumann, S. Hildebrandt, H. Giese, and B. Becker. Incremental model synchronization for efficient run-time monitoring. In *Models in Software Engineering*, number 6002 in LNCS, pages 124–139. Springer Berlin Heidelberg, Jan. 2010.