Towards Evolution Scenarios of Integrated Software Artifacts

Johannes Meier, Andreas Winter
Carl von Ossietzky University, Oldenburg, Germany
{meier,winter}@se.uni-oldenburg.de

Abstract

In software development different artifacts like source code, requirements, test cases and diagrams are involved, which are interrelated on content level, but separated on technical level. To link and synchronize such artifacts with each other, they have to be integrated. After integrating them into one comprehensive model, this paper discusses several evolution scenarios which arise in such integrated software artifacts.

1 Motivation

In software development lots of different artifacts like source code, requirements, test cases, architecture diagrams and documentation are used. Although they are often stored in different files and used by different tools and stakeholders, they are interrelated strongly on content level. As ongoing example in this paper, Java source code is complemented with textual requirements, and UML class diagrams: Each requirement is linked with parts of the source code which fulfill this requirement. UML class diagrams describe data structures which are realized in Java.

To link and synchronize different artifacts automatically, they need to be integrated. As described in more detail in Section 2, the models and conforming metamodels of each artifact will be integrated into one comprehensive model with a conforming metamodel which integrates all information of all artifacts.

After this initial integration, models and metamodels of all artifacts evolve over time with impact on all other artifacts: Renaming methods in UML class diagrams has to be synchronized into the Java source code automatically. Releasing new versions of Java or UML requires manual changes in the integration itself. This paper presents and discusses different evolution scenarios which arise in integrated software artifacts.

2 Metamodel Integration

To automatically link and synchronize artifacts with each other which are separated on technical, but interrelated on content level, these artifacts has to be integrated somehow. [1] proposes to create Single Underlying Models (SUM) which contain the information of all artifacts in an integrated way. Each SUM conforms to a Single Underlying MetaModel (SUMM) and does not contain duplicated information [1]. Additional relations between artifacts are realized by adding new associations in the SUMM. Synchronization effort for duplicated information is saved, because the SUM does not contain duplicated information anymore.

An approach to create such *SUM(M)*s in bottom-up way is presented in [2] and used in this paper: Each artifact is used as *data source* in form of its model and a conforming metamodel. All (Meta-)Models are integrated by step-wise applying special operators on them. These operators are coupled, as they combine a small change in the metamodel with changes in the model to keep it conform [3], and bidirectional, as they are executable in forward and backward direction. Figure 2 shows a simplified integration of Java source code, textual requirements, and UML class diagrams. The operators `IntegrateMetamodel` before ❶ and ❸ are used to load the single (meta-)models into the current (meta-)model. To link requirements and source code with each other, an operator `AddRelation` ❶→❷ is used to add a new association between the meta-classes which represent single requirements and parts of the source code. In backward direction, `DeleteRelation` deletes the new association in ❷ to get the previous version ❶. After executing all operators, both the SUM and the conforming SUMM are created as initial integration of all artifacts.

Additionally, new *views and conforming viewpoints* can be defined on top of the SUM(M) by configuring another chain of operators. As example, Figure 2 implies a new viewpoint `ReqJavaTable` listing all requirements with their linked source code in form of a table. This is another benefit of the SUM, because all integrated information can be reused in new views.

3 Evolution Scenarios

After the initial integration of all artifacts, as described in Section 2, seven different evolution scenarios can appear, depicted as $\boxed{1}\ldots\boxed{7}$ in Figure 1. Since

Artifact	Location	Model	Metamodel
SUM(M)	external	$\boxed{1}$	$\boxed{5}$
Data Source	external	$\boxed{2}$	$\boxed{6}$
View(point)	external	$\boxed{3}$	$\boxed{7}$
Intermediate	internal	$\boxed{4}$	–

Figure 1: Classification of Evolution Scenarios

SUM(M), data sources, and new view (points) are visible to and usable by external tools and user, the evolution scenarios of these artifacts are classified as *external*. *Internal* scenarios describe changes in the *intermediate* models and metamodels during the integration, drawn as ❶. . . ❻ in Figure 2.

In this approach, each software artifact has an explicit metamodel (schema) and a conforming model (instance) [4]. Therefore, the evolution scenarios are

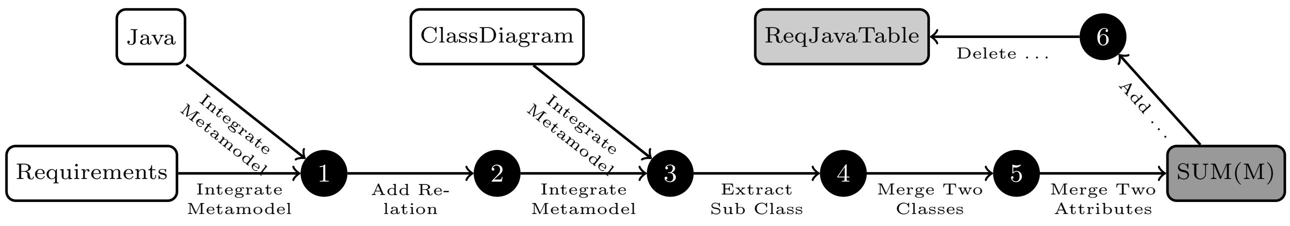

Figure 2: Simplified operator chain to integrate Java source code, textual requirements, and UML class diagrams

distinguished into the evolution of the model ($\boxed{1}$...$\boxed{4}$) and of the metamodel ($\boxed{5}$...$\boxed{7}$). In general, the model evolution scenarios occur often by working with the artifacts and are automated by executing the configured operators. The metamodel evolution scenarios occur rarer, e.g. when a new tool version or a new Java version arise, and are realized only once by manually changing the configured operator chain.

In Scenario $\boxed{1}$, the integrated model containing all information (the SUM) is changed. This scenario is automated by executing the complete operator chain in backward direction from the SUM to all data sources and new views, which are updated accordingly. This scenario exploits the bidirectionality of all operators and is important for reengineering tasks, because the SUM allows to analyze and improve all artifacts together at the same time.

In Scenario $\boxed{2}$, one data source is changed, e.g. by changing the Java source code through an IDE. This scenario is automated by executing the operators between this data source and the SUM in forward direction to update the SUM accordingly. After that, the resulting change in the SUM is propagated to all data sources and views as in Scenario $\boxed{1}$. This scenario is important to support tools which work only with that data source and to update legacy data sources.

In Scenario $\boxed{3}$, a new view is changed, if it supports changes in it (in contrast to read-only views). This scenario is automated in the same way as Scenario $\boxed{2}$ and supports maintaining integrated information, e.g. linking requirements and Java source code.

Scenario $\boxed{4}$ appears internally within the integration structure by executing operators during the other scenarios $\boxed{1}$, $\boxed{2}$, and $\boxed{3}$. This scenario is automated by applying a configured operator (e.g. `AddRelation`) to the current model ❶. After its execution, the model ❷ is available. Since the metamodels of ❶ and ❷ are the same for each execution, no further evolution appears (no Scenario $\boxed{8}$ in Figure 1).

Scenario $\boxed{5}$ describes changes in the SUMM, because of bugs, refactorings, or additional information. This scenario is realized by manually adding more operators into the current chain to describe $\boxed{\text{SUMM}}\leftrightarrow\boxed{\text{SUMM'}}$.

Scenario $\boxed{6}$ describes changes in the metamodel of data sources, e.g. because of new versions for Java or UML. The manual realization of this scenario depends on the size and impact of the metamodel change: Only additional meta-classes and -associations requires no additional effort, while refactorings in the metamodel require changes of the operator chain ❶...❺. In the worst case of a completely changed metamodel, the complete integration has to be done again.

Scenario $\boxed{7}$ arises when the new viewpoint should be changed because of bugs, refactorings, or additional information. Depending on the amount of changes, only some more operators describe $\boxed{\text{Viewpoint}}\longleftrightarrow\boxed{\text{Viewpoint'}}$. In the worst case, all operators between this viewpoint and its starting node has to be changed manually (Figure 2: only ❻).

4 Application

For the described metamodel integration in Section 2, a supporting framework is under development in Java using ECore to describe metamodels and reusing some coupled operators from Eclipse EDapt [3]. A first chain of bidirectional operators was configured to integrate Java, textual requirements and class diagrams. The resulting SUM(M) allows to automatically evolve the integrated model $\boxed{1}$, while the automated handling of changes in data sources $\boxed{2}$ and new views $\boxed{3}$ is under development. The internal evolution of intermediate models $\boxed{4}$ is already automated by the operators. The evolution of the metamodels ($\boxed{5}$ to $\boxed{7}$) is handled by manually extending the configured chain of operators as described in Section 3.

5 Conclusion

This paper discussed the evolution of interrelated software artifacts. As solution, all the models and conforming metamodels of these artifacts are integrated by using bidirectional coupled operators. This integration of artifacts works bottom-up and therefore enables the evolution of existing, interrelated legacy artifacts. Evolution appears in scenarios regarding the evolution of the artifacts itself in form of their models with automation as well as the evolution of schemata of the artifacts in form of their metamodels as manual step. As result, legacy artifacts are kept synchronized with each other, while reengineering tasks benefit from the integration of all legacy artifacts.

References

[1] C. Atkinson, D. Stoll, and P. Bostan, “Supporting View-Based Development through Orthographic Software Modeling,” *Evaluation of Novel Approaches to Software Engineering (ENASE)*, pp. 71–86, 2009.

[2] J. Meier and A. Winter, “Traceability enabled by Metamodel Integration,” *Softwaretechnik-Trends*, vol. 38, no. 2, 2018.

[3] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, “An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models,” *Software Language Engineering*, LNCS, 2011.

[4] D. Jin, J. Cordy, and T. Dean, “Where’s the schema? A taxonomy of patterns for software exchange,” *Int. Workshop on Program Comprehension.* IEEE, 2002.