ACM/IEEE 16th International Conference on
Model Driven Engineering Languages and Systems

September 29, 2013 – Miami, Florida (USA)

XM 2013 – Extreme Modeling Workshop Proceedings

Juan de Lara, Davide Di Ruscio, Alfonso Pierantonio (Eds.)

Published on September 23, 2013 v1.0

Language Support for Megamodel Renarration

Ralf Lämmel[1] and Vadim Zaytsev[2]

[1] Software Languages Team, Universität Koblenz-Landau, Germany
[2] Software Analysis & Transformation Team, CWI, Amsterdam, The Netherlands

Abstract. Megamodels may be difficult to understand because they reside at a high level of abstraction and they are graph-like structures that do not immediately provide means of order and decomposition as needed for successive examination and comprehension. To improve megamodel comprehension, we introduce modeling features for the recreation, in fact, renarration of megamodels. Our approach relies on certain operators for extending, instantiating, and otherwise modifying megamodels. We illustrate the approach in the context of megamodeling for Object/XML mapping (also known as XML data binding).

Keywords: megamodeling, linguistic architecture, renarration, software language engineering, XML data binding

1 Introduction

Models (of all kinds) may be difficult to understand when they reside at a high level of abstraction and when they are not structured in a way to serve successive examination and comprehension. In this paper,[3] we are specifically concerned with the modeling domain of the *linguistic architecture* of software systems [5] and a corresponding form of *megamodels* [3]. These are highly abstract models about software systems in terms of the involved languages, technologies, concepts, and artifacts. We aim to improve understanding of such models by means of renarration such that a megamodel is described (in fact, recreated) by a 'story' as opposed to a monolithic, highly abstract graph.

Contribution of this paper We enrich the megamodeling language MegaL [5] with language support for renarration such that megamodels can be developed in an incremental manner, subject to appropriate operators such as 'addition', 'restriction', or 'instantiation', also subject to an appropriate notion of megamodel *deltas*. In previous work [16], we have introduced the notion of renarration of megamodels in an informal manner as the process of converting a collection of facts into a story, also inspired by natural language engineering [15], computer-assisted reporting [13] and database journalism [8]. In this paper, we take the next step: we enrich megamodeling with proper language support for renarration.

[3] The paper's website: http://softlang.uni-koblenz.de/megal-renarration

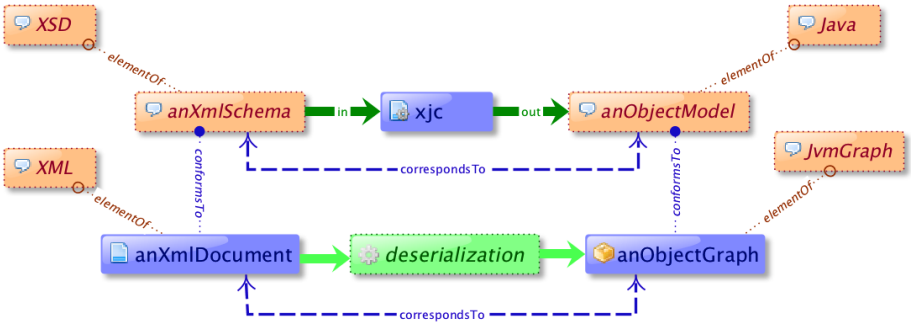

Fig. 1. A megamodel for Object/XML mapping (also known as XML data binding)

Roadmap §2 provides background on megamodeling and motivates the need for renarration. §3 recalls the MegaL language. §4 describes the specific approach to renarration. §5 provides a catalogue of operators that are used to express steps of renarration. §6 validates the approach in the context of megamodeling for Object/XML mapping. §7 discusses related work. §8 concludes the paper.

2 On the need for megamodel renarration

"A megamodel is a model of which [...] some elements represent and/or refer to models or metamodels" [3]—we use this definition by interpreting the notion of (meta)models in a broad sense to include programs, documents, schemas, grammars, etc. Megamodeling is often applied in the context of model-driven engineering while we apply it in the broader software engineering and software development context.

That is, we use megamodels to model the *linguistic architecture* of software systems [5]. By linguistic architecture of software systems or technologies, we mean their architecture expressed in terms of the involved software languages, software technologies, software concepts, software artifacts, and the explicit relationships between all these conceptual and actual entities. In our recent work, we have shown the utility of megamodels for understanding the linguistic architecture of diverse software (language) engineering scenarios [5,16].

Consider Figure 1 for an illustrative megamodel rendered in the visual syntax MegaL/yEd [5]. The nodes represent entities (languages, schemas, tools, etc.). The edges represent relationships ('elementOf', 'conformsTo', 'correspondsTo', etc.). The megamodel sketches basic aspects of Object/XML mapping according to the JAXB technology for XML data binding in the Java platform. Specifically, there is the aspect of deriving an object model (i.e., Java classes) from an XML schema (see the upper data flow in the figure) and the aspect of de-serializing an XML document to an object graph in the JVM (see the lower data flow in the figure).

One impediment to megamodel comprehension is the abstraction level of megamodels. In particular, the role and the origin of the entities as well the meaning of the relationships may not be evident. In recent work [5], we have

proposed a remedy for this problem. Our proposal involves linking megamodel entities and relationships to proper artifacts or extra resources for conceptual entities.

This paper focuses on another impediment to megamodel comprehension: megamodels are essentially just graph-like structures that do not immediately provide means of order and decomposition as needed for successive examination and comprehension. Consider the figure again. The following kinds of questions naturally arise. Where to start 'reading' in the figure? Are there any subgraphs that can be understood independently? Do any of the entities arise as instantiations of more general entities that may be worth mentioning to facilitate understanding?

The latter impediment to comprehension is not unique to megamodeling, of course. Various modeling or specification languages are prone to the same problem. Various remedies exist, e.g., based on modularization, abstraction, refinement, annotation, and slicing. In this paper, we put to work renarration which is indeed inspired by existing ideas on refinement, modularization, and slicing.

In general, renarration is the process of creating different stories while reusing the same facts (cf. narration[4]). In literature, for example, renarration is a technique to create a story by the narrator based on fixed plot elements; the story itself can be adapted to the audience and other circumstances—we refer to [2] for more background information. In megamodeling, renarration is a process of creating stories for the recreation of a megamodel. Recreation may cater for the audience's technical background and interest, time available and yet other factors. In our experience, the process of recreating a megamodel is needed to make megamodels meaningful to humans. Recreation may be interactive, e.g., by renarrating megamodels on the whiteboard, encouraging questions from the audience, and responding to these questions in the continuation of the story. This paper provides language support for the process of renarration.

3 Megamodeling with MegaL

Figure 1 provided a first illustration of the MegaL [5] language for megamodeling. In the rest of the paper, we use the textual MegaL syntax, i.e., MegaL/TXT. A megamodel is a collection of declarations of the following kinds.

Entity declarations A name is introduced for a conceptual entity, an actual entity, or a parameter thereof; an entity type (e.g., *Language* or *File*) is assigned. For instance:

```
Java : Language // "Java" as a language entity
JavaGrammar : Artifact // the "JavaGrammar" as an artifact entity
BNF : Language // "BNF" as a language entity
?aLanguage : Language // parameter "aLanguage" for a language entity
?aProgram : File // parameter "aProgram" for a file entity
```

[4] According to Merriam-Webster: *narration*: the act or process of telling a story or describing what happens http://www.merriam-webster.com/dictionary/narration Visited 14 September 2013.

We speak of a conceptual entity, if it exits in our mind, as in the case of a language. We speak of an actual entity (or simply an artifact), if it is manifest in some way: it exists on the file system (e.g., a language description) or as data structure at runtime (e.g., a parse tree).

Relationship declarations Two declared entities (or parameters thereof) are related by a binary relationship (e.g., 'elementOf' or 'conformsTo'). For instance:

```
aProgram elementOf Java // a program of the Java language
JavaGrammar elementOf BNF // the Java grammar is a BNF−style grammar
JavaGrammar defines Java // the Java grammar defines the Java language
aProgram conformsTo JavaGrammar // a program conforming to the Java grammar
```

Entity-type declarations There is a number of predefined, fundamental entity types, as exercised in the earlier examples, but new entity types can be defined by specialization. For instance:

```
OopLanguage < Language // an entity type for OO programming languages
FpLanguage < Language // an entity type for functional programming languages
```

Relationship-type declarations Likewise, there is a number of predefined, fundamental relationship types, as exercised in the illustrations above, but new relationship types can be defined on predefined as well as explicitly declared entity types. We do not further discuss such expressiveness in this paper.

The declarations simply describe a graph as illustrated in Figure 1. The order of all declarations of a megamodel is semantically irrelevant. The lack of any intrinsic notion of order (as in an imperative setting) or decomposition (as in substitution or function composition in functional programming) feeds into the comprehension challenge to be addressed by renarration. We mention in passing that megamodels have an interesting evaluation semantics. That is, declared relationships may be checked by applying some programmatic relationship-specific check on resources linked to declared entities.

4 Megamodel renarration

We add language support for renarration to the megamodeling language MegaL. We commit to a specific view on renarration such that megamodel *deltas* are used in the recreation of a megamodel through a sequence of steps with each step being effectively characterized by ingredients as follows:

- An informative *label* of the step, also serving as an 'id' for reference.
- The actual *delta* in terms of added and removed declarations (such as entity and relationship declarations). Added declarations are prefixed by '+'; removed declarations are prefixed by '−'. Deltas must preserve well-formedness of megamodels. In particular:
  - Entities are declared uniquely.
  - All entities referenced by relationship declarations are declared.
  - Relationships are applied to entities of suitable types.

Consider the following megamodel (in fact, megamodeling pattern) of a file and a language being related such that the former (in terms of its content) is an element of the latter.

```
[Label="File with language", Operator="Addition"]
+ ?aLanguage : Language // some language
+ ?aFile : File // some file
+ aFile elementOf aLanguage // associate language with file
```

In a next step, let us instantiate the language parameter to actually commit to the specific language *Java*. Thus:

```
[Label="A Java file", Operator="Instantiation"]
+ Java : Language // pick a specific language
+ aFile elementOf Java // associate the file with Java
- ?aLanguage : Language // removal of language parameter
- aFile elementOf aLanguage // removal of reference to language parameter
```

Fig. 2. An illustrative renarration

– An *operator* to describe the intent of the step. Each operator implies specific constraints on the delta, as discussed below.

The steps are interleaved with *informal explanations*.

See Figure 2 for a trivial, illustrative renarration. The first step introduces some entities and relates them. Nothing is removed; thus, the use of the operator 'Addition'. The second step instantiates the megamodel to a more concrete situation. The more general declarations are removed according to the delta and more specific declarations are added; thus, the use of the operator 'Instantiation'. Arguably, the instantiation could be characterized more concisely than by listing the delta, but we like to emphasize the utility of deltas for at least explaining the intended semantics of the renarration operators.

5 Renarration operators

The illustrative renarration of Figure 2 has started to reveal some operators: *addition* and *instantiation*. In this section, we provide a catalogue of operators. In the next section, the operators will be illustrated by a larger renarration.

– *Addition*: declarations are exclusively added; there are no removals. Use this operator to enhance a megamodel through added entities and to constrain a megamodel through added relationships.
– *Removal*: the opposite of *Addition*.
– *Restriction*: net total of addition and removal is such that entities may be restricted to be of more specific types. Also, the set operand of 'elementOf' and the super-set operand of 'subsetOf' relationships may be restricted.
– *Generalization*: the opposite of *Restriction*.

– *ZoomIn*: net total of addition and removal is such that relationships are decomposed to reveal more detail. Consider, for example, the relationship type mapsTo, which is used to expressed that one entity is (was) transformed into another entity. When zooming in, a relationship x mapsTo y could be expanded so as to reveal the function that contributes the pair $\langle x, y \rangle$.
– *ZoomOut*: the opposite of *ZoomIn*.
– *Instantiation*: parameters are consistently replaced by actual entities. We may describe such instantiation directly by a mapping from parameters to entities as opposed to a verbose delta. (A delta is clearly obtainable from such a mapping.)
– *Parameterization*: the opposite of *Instantiation*.
– *Connection*: convert an entity parameter into a *dependent entity*, which is one that is effectively determined by relationships as opposed to being yet available for actual instantiation. Such a dependency often occurs as the result of adding other parameters, e.g., a parameter for the definition of a language. We prefix dependent entity declarations by '!' whereas '?' is used for parameters, as explained earlier.
– *Disconnection*: the opposite of *Connection*.
– *Backtracking*: return to an earlier megamodel, as specified by a label. This may be useful in a story, when a certain complication should only be temporarily considered and subsequent steps should relate again to a simpler intermediate state.

6 An illustrative renarration

We are going to renarrate a megamodel for Object/XML mapping. We begin with the introduction of the XML schema which is the starting point for generating a corresponding object model:

```
[Label="XML schema", Operator="Addition"]
+ XSD : Language // the language of XML schemas
+ ?anXmlSchema : File // an XML schema
+ anXmlSchema elementOf XSD // an XML schema, indeed
```

On the OO side of things, we assume a Java-based object model:

```
[Label="Object model", Operator="Addition"]
+ Java : Language // the Java language
+ ?anObjectModel : File+ // an object model organized in one or more files
+ anObjectModel elementOf Java // a Java−based object model
```

The entities *anXmlSchema* and *anObjectModel* are parameters (see the '?' prefix) in that they would only be fixed once we consider a specific software system. We assume that schema and object model are related to each other in the sense that the former is mapped to ('transformed into') the latter; these two data models also correspond to each other [5].

```
[Label="Schema first", Operator="Addition"]
+ anXmlSchema mapsTo anObjectModel // the schema maps to the object model
+ anXmlSchema correspondsTo anObjectModel // the artifacts are "equivalent"
```

The ‘mapsTo’ relationship is helpful for initial understanding, but more details are needed eventually. Let us reveal the fact that a ‘type-level mapping’ would be needed to derive classes from the schema; we view this as ‘zooming in’: one relationship is replaced in favor of more detailed declarations:

```
[Label="Type−level mapping", Operator="ZoomIn"]
+ ?aTypeMapping : XSD -> Java // a mapping from schemas to object models
+ aTypeMapping(anXmlSchema) |-> anObjectModel // apply function
- anXmlSchema mapsTo anObjectModel // remove too vague mapping relationship
```

It is not very precise, neither is it suggestive to say that type-level mapping results in arbitrary Java code. Instead, we should express that a specific Java *subset* for simple object models (in fact, POJOs for data representation without behavioral concerns) is targeted. Thus, we restrict the derived object model as being an element of a suitable subset of Java, to which we refer here as *OxJava*:

```
[Label="O/X subset", Operator="Restriction"]
+ OxJava : Language // the O/X−specific subset of Java
+ OxJava subsetOf Java // establishing subset relationship, indeed
+ anObjectModel elementOf OxJava // add less liberal constraint on object model
- anObjectModel elementOf Java // remove too liberal constraint on object model
```

We have covered the basics of the type level of Object/XML mapping. Let us look at the instance level which involves XML documents and object graphs (trees) related through (de-)serialization. Let us assume an XML input document for de-serialization which conforms to the XML schema previously introduced:

```
[Label="XML document", Operator="Addition"]
+ XML : Language // the XML language
+ ?anXmlDocument : File // an XML document
+ anXmlDocument elementOf XML // an XML document, indeed
+ anXmlDocument conformsTo anXmlSchema // document conforms to schema
```

The result of de-serialization is an object graph that is part of the runtime state. We assume a language for Java’s JVM-based object graphs. The object graph conforms to the object graph previously introduced:

```
[Label="Object graph", Operator="Addition"]
+ JvmGraph : Language // the language of JVM graphs
+ ?anObjectGraph : State // an object graph
+ anObjectGraph elementOf JvmGraph // a JVM−based object graph
+ anObjectGraph conformsTo anObjectModel // graph conforms to object model
```

De-serialization maps the XML document to the object graph:

```
[Label="Instance−level mapping", Operator="Addition"]
+ ?aDeserializer : XML -> JvmGraph // deserialize XML to JVM graphs
+ aDeserializer(anXmlDocument) |-> anObjectGraph // map via deserializer
```

At this point, the mappings both at type and the instance levels (i.e., *aTypeMapping* and *aDeserializer*) are conceptual entities (in fact, functions) without a trace of their emergence. We should manifest them in relation to the underlying mapping technology. We begin with the type level.

```
[Label="Code generator", Operator="Addition"]
```

```
+ ?anOxTechnology : Technology // a technology such as JAXB
+ ?anOxGenerator : Technology // the generation part
+ anOxGenerator partOf anOxTechnology // a part, indeed
```

By relating generator and type mapping, we stop viewing the (conceptual entity for the) mapping as a proper parameter; rather it becomes a dependent entity.

```
[Label="Dependent type−level mapping", Operator="Connection"]
+ anOxGenerator defines aTypeMapping // mapping defined by generator
+ !aTypeMapping : XSD -> Java // this is a dependent entity now
- ?aTypeMapping : XSD -> Java // Ditto
```

Likewise, de-serialization is the conceptual counterpart for code that actually constructs and runs a de-serializer with the help of a designated library, which is another part of the mapping technology:

```
[Label="O/X library", Operator="Addition"]
+ ?anOxLibrary : Technology // the O/X library
+ anOxLibrary partOf anOxTechnology // an O/X part
+ ?aFragment : Fragment // souce code issuing de−serialization
+ aFragment elementOf Java // source code is Java code
+ aFragment refersTo anOxLibrary // use of O/X library
```

Again, we eliminate the parameter for the de-serializer:

```
[Label="Dependent instance−level mapping", Operator="Connection"]
+ aFragment defines aDeserializer // fragment "constructs" de−serializer
+ !aDeserializer : XML -> JvmGraph // this is a dependent entity now
- ?aDeserializer : XML -> JvmGraph // Ditto
```

Let us instantiate the mapping technology and its components to commit to the de-facto platform standard: JAXB [9]. We aim at the following replacements of parameters by concrete technology names:

```
[Label="JAXB", Operator="Instantiation"]
anOxTechnology => JAXB // instantiate parameter ... as ...
anOxGenerator => JAXB.xjc // ditto
anOxLibrary => JAXB.javax.xml.bind // ditto
```

Thus, we use qualified names for the component technologies of JAXB, thereby reducing the stress on the global namespace. We omit the the lower level meaning of the instantiation in terms of a delta.

Let us now generalize rather than instantiate. To this end, we first backtrack to an earlier state—the one before we instantiated for JAXB:

```
[Label="Dependent instance−level mapping", Operator="Backtracking"]
```

Now we can generalize further by making the language a parameter of the model. (Again, we show the concise mapping of actual entities to parameters as opposed to the delta for all the affected declarations.)

```
[Label="Beyond Java", Operator="Parameterization"]
Java => anOopLanguage // replace ... by parameter ...
OxJava => anOxLanguage // ditto
```

Arguably, we should use more specific entity types to better characterize some of the parameters of the model. For instance, the intention of the language

parameter to be an OOP language is only hinted at with the parameter's name; we could also designate and reference a suitable entity type:

```
[Label="Taxonomy", Operator="Restriction"]
+ OopLanguage < Language // declare entity type for OOP languages
+ ?anOopLanguage : OopLanguage // limit entity type of language
+ ?anOxLanguage : OopLanguage // limit entity type of language
- ?anOopLanguage : Language // remove underspecified declaration
- ?anOxLanguage : Language // remove underspecified declaration
```

7 Related work

In the presentation of actual megamodels, e.g., in [5,6,7,12,14], arguably, elements of renarration appear, due to the authors' natural efforts to modularize their models, to relate them, and to develop and describe them in piecemeal fashion. Renarration as an explicit *presentation* technique in software engineering was introduced in previous work [16]. Renarration as an explicit *modeling* technique is the contribution of the present paper.

It may seem that the required language support is straightforward, if not trivial. For instance, one may compare delta-based megamodel recreation with language support for model construction (creation), e.g., in the context of executable UML, as supported by action languages [11]. However, the renarration operators are associated with diverse constraints, as hinted at in their description, which brings them closer to notions such as refactoring or refinement or, more generally, model evolution. Deltas are used widely in model evolution; see, for example, [4]. In this context, it is also common to associate low-level deltas, as observed form the change history, with high-level intents in the sense of model-evolution operators.

A more advanced approach to the renarration of megamodels may receive inspiration from, for example, model management in MDE with its management operators (e.g., for composition [1]) and grammar convergence [10] with its rich underlying operator suite of (in this case) grammar modifications.

The field of natural language engineering contains many problems such as deriving a syuzhet from a fabula, a plot from genre elements, or a story from a plot. Recent solutions to these problems are advanced, formal and automated [15], and can be reused for software language engineering to facilitate semi-automatic or genetic inference of megamodel renarrations based on given constraints.

8 Concluding remarks

We have introduced language support for renarrating megamodels. With a relatively simple language design, we have made it possible to recreate (renarrate) megamodels in an incremental manner, while expressing intents by means of designated operators along the way.

In future work, we plan to provide a precise semantics of the operators. Further, by applying renarration to a number of different megamodeling scenarios, we also hope to converge on the set of operators needed in practice. Deltas,

as such, are fully expressive to represent any sort of recreation, but the suite of operators needs to be carefully maintained to support enough intentions for convenient use and useful checks on the steps. Yet another interesting area of future work is the animation of renarrations for a visual megamodeling language; we use the visual approach already informally on the whiteboard. Finally, the improvement of megamodel comprehension through renarration should be empirically validated.

References

1. A. Anwar, T. Dkaki, S. Ebersold, B. Coulette, and M. Nassar. A Formal Approach to Model Composition Applied to VUML. In *Proc. of ICECCS 2011*, pages 188–197. IEEE, 2011.
2. M. Baker and A. Chesterman. Ethics of Renarration. *Cultus*, 1(1):10–33, 2008. Mona Baker is interviewed by Andrew Chesterman.
3. J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. *OOPSLA & GPCE, Workshop on best MDSD practices*, 2004.
4. A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.
5. J.-M. Favre, R. Lämmel, and A. Varanovich. Modeling the Linguistic Architecture of Software Products. In *Proc. of MODELS 2012*, volume 7590 of *LNCS*, pages 151–167. Springer, 2012.
6. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *ENTCS*, 127(3), 2004.
7. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing Architecture Frameworks Through Megamodelling Techniques. In *Proc. of ASE 2010*, pages 305–308. ACM, 2010.
8. A. Holovaty. A Fundamental Way Newspaper Sites Need to Change, Sept. 2006. http://www.holovaty.com/writing/fundamental-change/.
9. JCP JSR 31. JAXB 2.0/2.1 — Java Architecture for XML Binding, 2008. http://jaxb.dev.java.net/.
10. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proc. of IFM 2009*, volume 5423 of *LNCS*, pages 246–260. Springer, 2009.
11. C.-L. Lazar, I. Lazar, B. Pârv, S. Motogna, and I. G. Czibula. Using a fUML Action Language to Construct UML Models. In *Proc. of SYNASC 2009*, pages 93–101. IEEE Computer Society, 2009.
12. B. Meyers and H. Vangheluwe. A Framework for Evolution of Modelling Languages. *Science of Computer Programming*, 76(12):1223–1246, 2011.
13. L. C. Miller. *Power Journalism: Computer-Assisted Reporting*. Harcourt Brace College Publishers, 1997.
14. J.-S. Sottet, G. Calvary, J.-M. Favre, and J. Coutaz. Megamodeling and Metamodel-Driven Engineering for Plastic User Interfaces: MEGA-UI. In *Human-Centered Software Engineering*, pages 173–200. Springer, 2009.
15. K. Wang, V. Q. Bui, and H. A. Abbass. Evolving Stories: Tree Adjoining Grammar Guided Genetic Programming for Complex Plot Generation. In *Proc. of SEAL 2010*, pages 135–145. Springer, 2010.
16. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *Proc. of MPM 2012*, pages 61–66. ACM, 2012. http://dx.doi.org/10.1145/2508443.2508454.