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 for the linguistic architecture of software systems can become difficult to understand because they reside at a high level of abstraction and they are graph-like structures without much intrinsic modularity or prescribed order for mental navigation. To facilitate megamodel comprehension, we extend megamodeling by modeling features for renarration such that megamodels can be developed in an incremental manner with the help of appropriate renarration operators, also subject to a simple form of deltas on megamodels. We validate 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

1 Introduction

"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, grammars, etc. In our recent work, we have shown the utility of megamodels in understanding the linguistic architecture of software systems and software (language) engineering scenarios and software technologies; see, for example, [5,12].

Megamodels may be difficult to understand because they reside at a high level of abstraction and the role of entities as well the meaning of relationships may not be evident. Previously [5], we have addressed megamodel comprehension by linking megamodel entities and relationships to proper artifacts or reified conceptual entities. In this paper, we are concerned with another substantial impediment: megamodels are hard to access because they are essentially just graph-like data structures without much intrinsic modularity or prescribed order for mental navigation. In previous work [12], we have introduced the notion of renarration of megamodels in an informal manner, also inspired by natural language processing and database journalism. In this paper, we take the next step: we enrich megamodeling with proper language support for renarration.

Consider Figure 1 for an illustration. 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 and the aspect of de-serializing

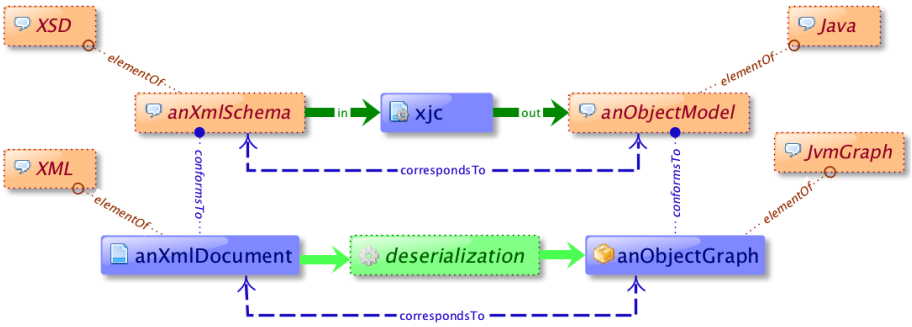

Fig. 1. A megamodel for Object/XML mapping (also known as XML data binding)

an XML document to an object graph (as represented in the JVM). In our experience, such models must be renarrated to become meaningful to others. A complete megamodel is useful as a specification for validation/testing, but the process of a megamodel's creation is more important for comprehension.

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 deltas for megamodels.

Roadmap. §2 describes the specific approach to renarration. §3 validates the approach in the context of megamodeling for Object/XML mapping. §4 summarizes related work. §5 concludes the paper.

2 Renarrating megamodels

We enrich the base megamodeling language MegaL[3] [5] by language support for renarration. We briefly recall MegaL. Then, we describe a specific form of renarration, which is centered around megamodel deltas. Finally, we list operators than can be used to describe the intents of renarration steps.

2.1 Megamodels

A megamodel collects declarations of the following kinds.

Entity declaration. A name is introduced for a conceptual or manifested entity or a parameter thereof; an entity type is assigned. Here are some illustrative examples:

```
Java : Language // Java as a language entity
JavaGrammar : Artifact // the Java grammar 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
```

[3] https://github.com/avaranovich/megal/

Relationship declaration. Two previously declared entities (or parameters thereof) are related by a binary relationship. Here are some illustrative examples:

```
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 declaration. There is a number of predefined, fundamental entity types, as exercised in the illustrations above, 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 declaration. 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 leverage such expressiveness in this paper.

2.2 Renarration

We refer to [2] for general background on renarration. In this paper, we commit to a specific view on renarration as breaking down into a sequence of steps with each step being effectively characterized by some ingredients:

- 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 '−'.
- An *operator* to describe the intent of the step, thereby also defining constraints on the delta.

The steps are interleaved with informal explanations. Due to space constraints, no grammar of renarrations is specified here. See Figure 2 for a trivial, illustrative renarration. Regardless of constraints associated with specific operators, deltas must preserve well-formedness of megamodels. In particular: (i) Entities are declared uniquely. (ii) All entities referenced by relationship declarations are declared. (iii) The types of the entities in relationship declarations are permitted for the relationship type at hand.

2.3 Operators

The illustrative renarration of Figure 2 has started to reveal some operators: *addition* and *instantiation*. The more complex example of §3 leverages several additional operators. Here is a short characterization of a catalogue of operators:

- *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.

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. This is the initial step of the renarration and thus, the delta only involves added declarations as opposed to any removed declarations:

```
[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 element Java // associate the file with Java
- ?aLanguage : Language // removal of language parameter
- aFile elementOf aLanguage // removal of reference to language parameter
```

That is, the parameter declaration and the use of the parameter are removed, whereas *Java* is declared and used instead. (Arguably, such instantiation could also be characterized more concisely by just stating that the parameter shall be replaced by a specific entity.)

Fig. 2. An illustrative renarration

- *Removal*: the opposite of *Addition*.
- *Restriction*: net total of addition and removal is such that entities may be of more specific types and 'elementOf' as well as 'subsetOf' relationships may also be restricted in a similar manner; see label 'O/X subset' in §3.
- *Generalization*: the opposite of *Restriction*.
- *ZoomIn*: net total of addition and removal is such that relationships are decomposed to reveal more detail. For instance, a relationship x mapsTo y could be expanded so as to reveal the function that contributes the pair $\langle x, y \rangle$; see label 'Type-level mapping' in §3.
- *ZoomOut*: the opposite of *ZoomIn*.
- *Instantiation*: parameters are consistently replaced by actual entities. We describe such instantiation directly by a mapping from parameters to entities as opposed to a verbose delta.
- *Parameterization*: the opposite of *Instantiation*.
- *Connection*: convert an entity parameter into a dependent entity, which is one that is effectively determined by relationships. We prefix dependent entity declarations by '!' (whereas '?' is used for parameters, as explained earlier); see label 'Dependent type-level mapping' in §3.
- *Disconnection*: the opposite of *Connection*.
- *Backtracking*: return to an earlier megamodel, as specified by a label.

3 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
+ 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 // schema and object model are similar
```

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 (see '−') is replaced in favor of more detailed declarations (see '+'):

```
[Label="Type−level mapping", Operator="ZoomIn"]
+ ?aTypeMapping : XSD -> Java // a mapping from schemas to object models
+ aTypeMapping(anXmlSchema) |-> anObjectModel // map, indeed
- 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. Instead, we should express that a specific Java *subset* for simple object models (in fact, POJOs for data representation without behaviorial 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; that is an element of an assumed language for Java or JVM object graphs; that also 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 documents 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
+ anOxGenerator defines aTypeMapping // a mapping defined by a generator
```

With the generator in place, we should no longer view the (conceptual entity for the) mapping as a proper parameter; rather it becomes a dependent entity.

```
[Label="Dependent type-level mapping", Operator="Connection"]
+ !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 libary, 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
+ aFragment defines aDeserializer // provision of the de-serializer
```

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

```
[Label="Dependent instance-level mapping", Operator="Connection"]
+ !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 [8]. 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
```

4 Related work

In the presentation of actual megamodels, e.g., in [5,6,7,10,11], arguably, elements of renarration appear, due to the authors' natural efforts to modularize their models, to relate them, and to develop them in piecemeal fashion. Renarration as an explicit *presentation* technique in software engineering was introduced in previous work [12]. Renarration as an explicit *modeling* technique is the contribution of the present paper. General renarration [2] prospects in software engineering and computer science remain to be investigated.

Due to space constraints, we cannot properly discuss the broader area of related work on refinement, refactoring, composition, and other kinds of transformation of specifications, programs, and models. 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 [9] with its rich underlying operator suite of (in this case) grammar modifications.

5 Concluding remarks

We have introduced language support for renarrating megamodels. With a relatively simple language design, we have made it possible to develop (renarrate) megamodels in an incremental manner, while applying different operators along the way. Deltas describe the renarration steps at a low level of abstraction. As illustrated with instantiation and parameterization, one can also aim at higher-level versions of the operators akin to refactoring or evolution operators used in other areas of modeling [4]. 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.

References

1. A. Anwar, T. Dkaki, S. Ebersold, B. Coulette, and M. Nassar. A Formal Approach to Model Composition Applied to VUML. In *Proceedings 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 *Proceedings of MODELS 2012*, LNCS. Springer, 2012. 17 pages. To appear.
6. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. *Electronic Notes in Theoretical Computer Science, Proceedings of the SETra Workshop*, 127(3), 2004.
7. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing Architecture Frameworks Through Megamodelling Techniques. In *Proceedings of ASE'10*, pages 305–308, New York, 2010. ACM.
8. JCP JSR 31. JAXB 2.0/2.1 — Java Architecture for XML Binding, 2008. http://jaxb.dev.java.net/.
9. R. Lämmel and V. Zaytsev. An Introduction to Grammar Convergence. In *Proceedings of IFM 2009*, volume 5423 of *LNCS*, pages 246–260. Springer, 2009.
10. B. Meyers and H. Vangheluwe. A Framework for Evolution of Modelling Languages. *Science of Computer Programming*, 76(12):1223 – 1246, 2011.
11. 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.
12. V. Zaytsev. Renarrating Linguistic Architecture: A Case Study. In *Proceedings of MPM 2012*. ACM DL, 2012. Available via http://avalon.aut.bme.hu/mpm12/papers/paper%2015.pdf.