The SDMLib solution to the FIXML case for TTC2014

Christoph Eickhoff[1], Tobias George[1], Stefan Lindel[1], Albert Zündorf[1]

Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
cei|tge|slin|zuendorf@cs.uni-kassel.de

Abstract. This paper describes the SDMLib solution to the FIXML case for the TTC2014 [9]. SDMLib already provides Java code generation for class models / class diagrams. In addition, SDMLib provides a mechanism for learning class models from generic example object structures. Thus, for the FIXML case we just added an XML reader that reads an example file and creates a generic object structure reflecting its content.

1 Introduction

Our team at Kassel University found this case particularly interesting as we give a course on CASE tool construction where one assignment to the students is to learn a class diagram / class model from an XML file containing object descriptions but without an explicit XML schema. Thus, the case looked quite familiar to us.

In addition, our team has developed a software development process called *Story Driven Modeling* [3, 1]. Story Driven Modeling starts with textual scenarios that describe example situations and how they evolve through the execution of a certain user action. In a second step, the textual scenarios are extended with informal object diagrams modeling how the desired program might represent the described situations as object structure at runtime. Initially, the informal object diagrams may omit object types. The types are added in another design step that formalizes the object diagrams until a class diagram can be derived, automatically. This initial class diagram may be extended several times in order to support additional scenarios and in order to e.g. add support for certain design patterns like composite pattern or visitor pattern or strategies. Then, an implementation of the modeled classes may be generated using e.g. Fujaba [2] or UMLLab [8] or SDMLib [7].

To support Story Driven Modeling, SDMLib provides *Generic Object Diagrams* [5]. Generic Object Diagrams are able to represent untyped object structures, they allow to add type information at runtime and SDMLib is able to learn a class diagram from Generic Object Diagrams and to generate a Java implementation from it. This is discussed in section 2.

To address the FIXML case, we just used the standard Java XML parser and wrote a small transformation that translate the read XML data into a Generic Object Diagram. Then, we used the SDMLib mechanism to learn a class diagram and to generate a Java implementation, cf. section 3.

2 SDMLib support for Story Driven Modeling

Figure 1 shows the classes of the generic object model provided by SDMLib. The corresponding Java implementation allows users to create generic object structures e.g. in a JUnit test as shown in listing 1.1. Similarly, a graphical editor or some dedicated textual object model description language may be used.

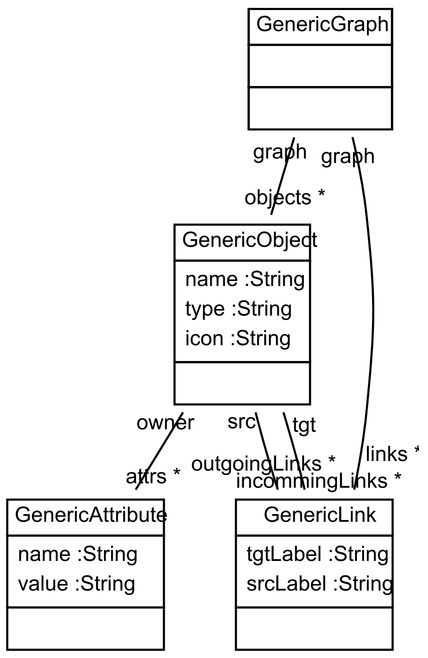

Fig. 1. Generic Object Diagram Class Model

```
...
GenericGraph graph = new GenericGraph();

GenericObject building = graph.createObjects()
.with("name", "WA73")
.withName("WilliAllee")
.withType("Building");

GenericObject wa13 = graph.createObjects()
.with("name", "WA13")
.with("level", "1")
.withName("seFloor")
.withType("Floor");

graph.createLinks()
.withSrc(building)
.withTgt(wa13)
.withTgtLabel("has");

GenericObject wa03 = graph.createObjects()
```

```
.with("name", "WA03")
.with("level", "0")
.with("guest","Ulrich")
.withName("digitalFloor")
.withType("Floor");

graph.createLinks()
.withSrc(building)
.withTgt(wa03)
.withTgtLabel("has");
...
```

Listing 1.1. Creating a Generic Object Model via Java API

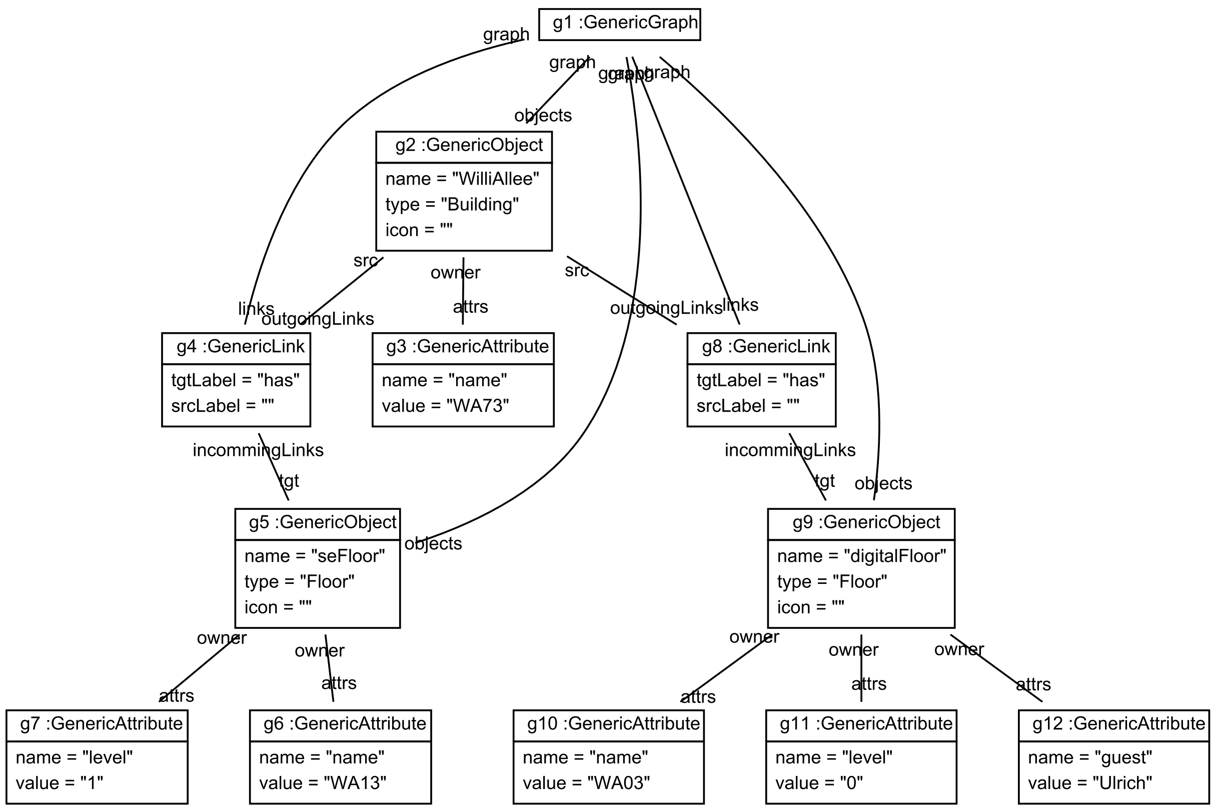

Fig. 2. Example Informal Generic Object Model

Using GraphViz [4], SDMLib is able to render object models as object diagrams, cf. figure 2. As our generic object model use explicit objects for attributes and links, it is quite big even for small examples. Thus, SDMLib provides another visualization, where attribute and link objects are rendered as usual attributes and links, cf. figure 3.

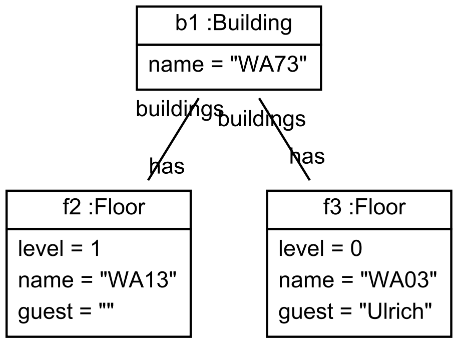

Fig. 3. Example Generic Object Model Rendered as Formal Object Diagram

Listing 1.2 shows the SDMLib algorithm for learning a class model from a generic object structure. First, line 5 loops through all generic objects and line 7 queries the class model for a class with a name corresponding to the type of the current generic object. Method `getOrCreateClazz` creates a new class, if the object type shows up the first time. Then, line 10 loops through the generic attributes attached to the current generic object. For each attribute method `getOrCreateAttribute` retrieves an attribute declaration in the current class, cf. line 11.

```
 1 public ClassModel learnFromGenericObjects(String packageName, GenericGraph root){
 2    this.setPackageName(packageName);
 3
 4    // derive classes from object types
 5    for (GenericObject currentObject : root.getObjects()) {
 6       if (currentObject.getType() != null) {
 7          Clazz currentClazz = this.getOrCreateClazz(currentObject.getType());
 8
 9          // add attribute declarations
10          for (GenericAttribute attr : currentObject.getAttrs()) {
11             Attribute attrDecl = currentClazz.getOrCreateAttribute(attr.getName());
12
13             learnAttrType(attr, attrDecl);
14          }
15       }
16    }
17
18    LinkedHashSet<String> alreadyUsedLabels = new LinkedHashSet<String>();
19
20     // now derive assocs from links
21     for (GenericLink currentLink : root.getLinks()) {
22        String sourceType = currentLink.getSrc().getType();
23        if (sourceType == null) continue; //<============
24
25        String targetType = currentLink.getTgt().getType();
26        if (targetType == null) continue; //<============
27
28        String sourceLabel = currentLink.getSrcLabel();
29        if (sourceLabel == null) {
30           sourceLabel = StrUtil.downFirstChar(sourceType) + "s";
31        }
32
33        String targetLabel = currentLink.getTgtLabel();
34        if (targetLabel == null) {
35           targetLabel = StrUtil.downFirstChar(sourceType) + "s";
36        }
37
38        Association currentAssoc = getOrCreateAssoc(sourceType, sourceLabel,
39                                                    targetType, targetLabel);
40
41        if (alreadyUsedLabels.contains(
42                  currentLink.getSrc().hashCode() + ":" + targetLabel)) {
43           currentAssoc.getTarget().setCard(R.MANY);
44        }
45
46        if (alreadyUsedLabels.contains(
47                  currentLink.getTgt().hashCode() + ":" + sourceLabel)) {
48           currentAssoc.getSource().setCard(R.MANY);
49        }
50
51        alreadyUsedLabels.add(currentLink.getSrc().hashCode()+":"+targetLabel);
52        alreadyUsedLabels.add(currentLink.getTgt().hashCode()+":"+sourceLabel);
53     }
54     return this;
55  }
```

Listing 1.2. Learning a Class Model from a Generic Object Model

Learning the type of an attribute is done by method `learnAttrType` called in line 13 of listing 1.2. The body of method `learnAttrType` is shown in listing 1.3. Basically, we retrieve the value of the current generic attribute, cf. line 3. For our generic object model, attribute values are just strings. To learn more specific attribute types, we just try to parse the value string to an `int` or a `double` or a `java.util.Date` value. On success, we store the detected type in variable `attrType`. On different attribute values belonging to the same attribute declaration, this parsing step may compute different results. For example one generic object may have a `num` attribute with value `42` while the next generic object may have a `num` attribute with value `23.5`.

The first case computes to attribute type `int` while the second case computes to `double`. To resolve this, line 34 compares the type computed for the current value with the current type of the attribute declaration. If the new type is *more general* than the old type, we switch to the new type. In the general case, we stick to type `String` that is able to encode any attribute value. Note, we have added the learning of attribute types to SDMLib especially to meet the FIXML case. As the FIXML files use various `Date` formats we tried to cover all of them but finally gave up and decided to google for some general date reader. This needs to be completed. There are also other attribute types (e.g. boolean) still to be covered. So far, our attribute type learning algorithm delivers good results. It computes reasonable attribute types from the example data. However, large real world data may contain invalid attribute values, e.g. a value like "???" for an int attribute of some object. In this case our algorithm would change the attribute type to "String". Thus, a single invalid attribute value out of thousands of number values may change the attribute type. Probably, such spurious incorrect attribute values should not result in a change of an attribute type that fits for the vast majority of values. This needs further investigation.

```
 1 private void learnAttrType(GenericAttribute attr, Attribute attrDecl)
 2 {
 3    String valueString = attr.getValue();
 4
 5    String attrType = "String";
 6    try {
 7       Integer.parseInt(valueString);
 8       attrType = "int";
 9    } catch (NumberFormatException e)
10    {
11       try {
12          Double.parseDouble(valueString);
13          attrType = "double";
14       } catch (NumberFormatException e1)
15       {
16          try {
17             DateFormat.getDateInstance().parse(valueString);
18             attrType = "java.util.Date";
19          }  catch (ParseException e2)
20          {
21             try {
22                SimpleDateFormat simpleDateFormat =
23                   new SimpleDateFormat("yyyy-mm-dd'T'hh:mm:ss");
24                simpleDateFormat.parse(valueString);
25                attrType = "java.util.Date";
26             } catch (ParseException e3)
27             { ... }
28          }
29       }
30    }
31
32    String typeOrder = "Object␣int␣double␣java.util.Date␣String";
```

```

   if (typeOrder.indexOf(attrDecl.getType()) < typeOrder.indexOf(attrType))
   {
      attrDecl.setType(attrType);
   }
}
```

Listing 1.3. Learning the type of an attribute

Next, the loop in line 21 of listing 1.2 is used to learn associations from generic links. For each link we retrieve the types of the connected objects and the role labels for the link ends. Then, method `getOrCreateAssoc` searches the class model for a matching association or creates one, otherwise. Note, this step is sensible to the direction of links, two similar links with swapped source and target roles might result in two associations with swapped roles instead of a single one. This is easy to fix but results in a more complicated learning algorithm and is thus omitted for lack of space.

Finally, we have to deal with association cardinalities. The most general approach is to use to-many cardinality for all association roles. To-many associations are able to store to-one relations, too, and thus to-many association would work in all cases. However, in many cases a to-one cardinality would suffice and might be more natural to the user. Thus, SDMLib starts with a to-one cardinality for all new associations and roles and we change the role cardinality as soon as there is an object with two similar links attached to it. In listing 1.2 we use variable `alreadyUsedLabels` to keep track of object and label combinations already used. For each link, lines 51 and 52 store a string identifying an object on one side and the label on the other side of the link. In later iterations, line 41 and 46 check whether this combination of object and label have been used before. If this is the case, we change the cardinality of the corresponding association role to to-many. So far, this approach delivers reasonable results. However, in some cases we derive to-one associations that actually need to be to-many associations. This happens if the current example is too small and does not show that there may be multiple links of a certain type. So far, we only guarantee that the derived class model is able to represent all example object structures we have used to learn it. Note, the learning of association cardinalities has been added to SDMLib especially for the FIXML case.

Our class model learning approach does not yet factor out common attributes or roles to common super classes. For example, if there is an object of type `Man` with attributes like `name` and `dayOfBirth` and there is another object of type `Woman` with similar attributes, we might want to introduce a class `Person` with attribute declarations for name and `dayOfBirth` and let the classes `Man` and `Woman` inherit from it. This is not yet supported by our approach. However, once a class model has been learned, we use graphviz to render it to the user as a class diagram, cf. figure 4. In addition, the user has the possiblity to refactor the learned class diagram e.g. via the SDMLib API.

From the resulting class model, SDMLib generates a Java implementation with

– one plain Java class per model class,
– a private attribute with public get and set methods for each attribute in the class model,
– a private attribute with public access methods for each association role (the access methods of the two roles that build an association call each other to achieve referential integrity of the pairs of pointers that represent a link, to-many associations use special container to hold multiple pointers).
– For each model class like `Building` we generate a `BuildingSet` class. These classes are used for to-many roles. In addition, these set classes provide the same methods as the original model classes, e.g. `FloorSet::getName()`. In a set class, methods like `getName()` are applied to each contained element, the results are collected and then returned. Thus, for a variable `mainBuilding` of type `Building` the call

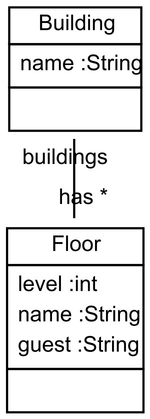

Fig. 4. Class Model learned from Generic Object Model

`mainBuilding.getHas()` delivers the set of floors of that building and `mainBuilding.getHas().getName()` delivers a list of names of these floors.

- We also generate model specific classes like `BuildingPO` that are used to represent pattern objects in model transformations. For more details see [6].
- Finally, we generate factory classes that facilitate the creation of model objects and that provide a reflective access layer for the the model. This means, you may ask these factories for the names of all attributes and association roles of a model class and you may read and write attribute values using their names as simple strings. This reflective layer is also used to provide generic serialization mechanisms to load and store model object structures from / in JSON or XML format.

3 Solving the FIXML case with SDMLib

Section 2 shows the SDMLib support for learning class models from example object structures. To apply these techniques to the FIXML case, we just developed an XML reader that turns the example input data into generic object structures. Then, the SDMLib techniques are used to learn a class model and to generate Java code for it.

To read the XML data we use standard `javax.xml.parsers.DocumentBuilder`. This creates a `org.w3c.dom.Document` which is a tree of `Node` objects. This tree is then visited by method `visitXMLEntities` shown in listing 1.4.

```java
public static void visitXMLEntities(Node xmlRoot,
                        GenericGraph graph, GenericObject parent)
{
   for (int i = 0; i < xmlRoot.getChildNodes().getLength(); i++)
   {
      Node entity = xmlRoot.getChildNodes().item(i);

      if (entity.getNodeType() == Node.TEXT_NODE)
      {
         continue;
      }

      // handle attributes encoded between tags
      if (entity.getChildNodes().getLength() == 1
            && entity.getAttributes().getLength() == 0)
      {
         Node singleChild = entity.getFirstChild();

         if (singleChild.getNodeType() == Node.TEXT_NODE)
         {
            parent.createAttrs()
                  .withName(toValidJavaVarId(entity.getNodeName()))
                  .withValue(singleChild.getNodeValue());

            continue;
         }
      }

      GenericObject genericObject = graph.createObjects()
            .withType(toValidJavaTypeId(entity.getNodeName()));

      if (parent != null)
      {
         // create link to new kid
         parent.createOutgoingLinks()
               .withSrcLabel(toValidJavaVarId(parent.getType()))
               .withTgtLabel(toValidJavaVarId(entity.getNodeName()))
               .withTgt(genericObject);
      }

      // handle attributes
      for (int j = 0; j < xmlRoot.getAttributes().getLength(); j++)
      {
         Node attr = xmlRoot.getAttributes().item(j);
         String key = attr.getNodeName();
46              String value = attr.getNodeValue();
47
48              genericObject.createAttrs()
49                      .withName(toValidJavaVarId(key))
50                      .withValue(value);
51          }
52
53          visitXMLEntities(entity, graph, genericObject);
54
55      }
56  }
```

Listing 1.4. Reading XML Examples into Generic Object Structures

Method `visitXMLEntities` is called recursively on each node of the XML document tree and in addition to the current node `xmlRoot` we pass the generic `graph` under construction and the generic object `parent` that corresponds to the current xml node. Line 4 loops through all children of the current node. Some of the child nodes are simple text nodes containing only white spaces. We skip these in line 10.

Lines 14 through 27 handle attributes that are coded as single text content in XML nodes with no XML attributes, e.g. `<id>42</id>`.

Per default, line 29 creates a generic object for each XML node. The XML tag / node name becomes the object type. Following international coding conventions, we decided to use upper case letters for the beginning of type names and lower case letters for the the beginning of attribute names. Unfortunately, the example XML contains attributes with name `Long`. Our naming conventions turned this into `long` which creates a name clash with the java type `long`. Generally, XML tags may contain characters not valid for Java identifiers and XML tags may clash with Java reserved words. Thus, we use methods `toValidJavaVarId` and `toValidJavaTypeId` to handle such problems and to establish our naming conventions.

Line 35 connects the new generic object to its parent. As target label we use the XML tag / type of the child node (with lower case initial letter) and as source label the XML tag / type of the parent node.

The loop of line 42 handles attributes of XML nodes as e.g. `<man name="Bob">`. XML attributes become generic attributes with name and value retrieved from the XML node. Note, the attribute values retrieved from XML are always `String`s. We handle this during the learning of class models as discussed in section 2.

4 Conclusions

The FIXML case was made for us. SDMLib provides a lot of functionality for generic object structures, learning class models, and generating Java code. As an example, figure 5 shows the class model learned from the XML data of the FIXML case 1.

Our solution to the FIXML case first learns one separate class model for each example XML file. Then, we use all example files to learn one common class model that covers all cases. As the class model learning algorithm for each generic object, attribute, and link first looks whether it already has an appropriate declaration, you can also start with a class model learned from other cases and add more examples one after the other.

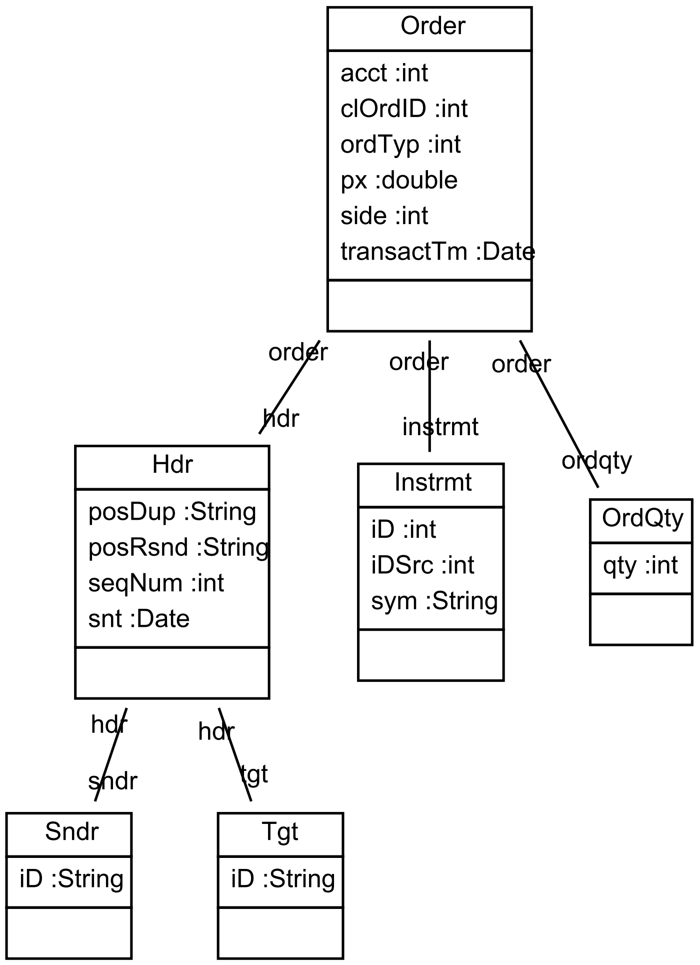

Fig. 5. Class Diagram Learned for Case 1

Once we have learned a class model and we have generated its Java implementation, SDMLib also allows to convert generic object structures into model specific object structures. This is done using the reflective access layer generated for the model. Once the generic object structure has been transformed into a model specific object structure, you may program model specific algorithms based on the generated Java implementation leveraging static type checking and compile time consistency checks. Then you may load XML files, convert them to model specific objects and run your algorithm. Your algorithm may also utilize the the set based model layer generated by SDMLib or even the model transformation layer. As simple example for the set based layer you may write `currentOrder.getOrdqty().getQty().sum()`. This looks up the set of `OrdQty` objects attached to the current order. Note, in figure 5 association `ordqty` has cardinality to-one. However, this is changed by later examples to to-many. Next our code looks up the `qty` attribute of these objects.

Generally, attribute values are collected in list to allow multiple occurences of the same value. In our case the order quantities are collected in a list of `int` values. For lists of numbers, SDMLib provides some special operations like `min`, `max`, and `sum`. The latter computes the sum of the numbers of the list.

Thus, SDMLib does not only generate a Java implementation for the example XML files. It also provides a mechanism to load XML files to a model specific object structure and SDMLib allows to run complex algorithms and model transformations on that data.

Although we only generate Java and the generated Java does not exactly follow the style proposed by the FIXML case study, we hope that our class model learning approach is a valuable contribution to TTC2014.

References

1. I. Diethelm, L. Geiger, and A. Zündorf. Systematic story driven modeling. Technical Report, Universität Kassel, 2002.
2. T. Fischer, J. Niere, L. Torunski, and A. Zndorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.
3. U. Norbisrath, R. Jubeh, and A. Zndorf. *Story Driven Modeling.* CreateSpace Independent Publishing Platform, 2013.
4. A. Research. Graphviz - graph visualization software, 2008.
5. SDMLib Generic Object Diagrams. https://rawgithub.com/azuendorf/SDMLib/master/SDMLib.net/doc/index.html, 2014.
6. SDMLib Model Navigation and Model Transformations Example. https://rawgit.com/azuendorf/SDMLib/master/SDMLib.net/doc/StudyRightObjectModelNavigationAndQueries.html, 2014.
7. Story Driven Modeling Library. http://sdmlib.org/, 2014.
8. UML LAB from Yatta Solutions. http://www.uml-lab.com/de/uml-lab/, 2014.
9. FIXML Case for the TTC 2014. https://github.com/transformationtoolcontest/ttc2014-fixml, 2014.