Addressing Industrial Needs with the Fulib Modeling Library

Albert Zündorf[1], Adrian Kunz[1] and Christoph Eickhoff[1]
[1]*Kassel University, Germany*
{zuendorf, a.kunz, christoph.eickhoff}@uni-kassel.de

Keywords: Models, Model Transformations, Modeling Tools.

Abstract: Fulib is a new lightweight modeling tool providing code generation and model transformations. Code generated by Fulib does not need a Fulib runtime library. Fulib has been designed to be integrated into agile software development processes. Fulib collaborates with versioning tools like Git. These features address practical problems with modeling tools that frequently prevent their usage in industry.

1 Introduction

The International Conference on Model Transformation (ICMT) October 2019 in Eindhoven had a panel discussion on “Is there a future for Model Transformation Languages?”. Basically, they answered this question with “No!” and terminated the ICMT conference series. The main argument was that model transformations were not able to gain industrial relevance. The main reasons given by the panel were: 1) proprietary libraries, 2) licence issues, 3) vendor lock-in to (immature) university prototypes, 4) poor integration with iterative software development approaches. There is also the paper of Paige and Varro (Paige and Varró, 2012) that discusses similar insights.

Following that panel and reading (Paige and Varró, 2012), we recognized: “Well, these are exactly the issues that forced us to stop working on Fujaba (Nickel et al., 2000) and SDMLib (Zündorf et al., 2013) and to come up with Fulib”. Actually, we have started to work on Fulib in August 2018 and the first official release was in October 2019. Thus, this paper outlines how our new lightweight modeling FUjaba Library *Fulib* addresses the typical industrial concerns with modeling tools.

2 No Fulib Runtime Library

Within the model transformation community, the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008) has become the de-facto standard for the implementation of models. Besides being a common standard for many modern modeling tools, EMF does a great job in generating model implementations and providing model management functionality like serialization to XML, generic model viewers and editors, etc.

However, if your model e.g. has a class *Order* or *Product*, then the generated Java classes will inherit from the basic library class (interface) *EObject*. If your model uses to-many associations, these will be implemented using the library class *EList*. Thus, your model code relies heavily on an EMF runtime library. In order to use the generated model code within your final product you have to include the EMF runtime library and thus you have to deal with the according licence. Generally, the EMF tooling and especially the EMF runtime library are pretty mature and ready for industrial use and the Eclipse Public Licence is pretty general. Still this triggers the industrial concerns raised by the ICMT panel discussed in our introduction.

To avoid these industrial concerns, the Java classes generated by Fulib do not require any runtime library. Fulib does not use a common superclass or interface like *EObject* and our associations are implemented with standard Java container classes (default *ArrayList*). Additional model functionalities like bidirectional associations are generated into the Java classes. Instead of runtime libraries, Fulib relies on generated code and on generic reflection functionality, cf. Section 6. Thus, you may use Fulib to generate (parts of) your model and when you ship your product, you do not ship any Fulib libraries or parts and there are no Fulib licences involved.

```java
public class GenModel implements ClassModelDecorator {
    @Override
    public void decorate(ClassModelManager mm) {
        Clazz shop = mm.haveClass("Shop", c -> {
          c.attribute("name", STRING);
        });
        Clazz customer = mm.haveClass("Customer", c -> {
          c.attribute("customerId", STRING);
          c.attribute("name", STRING);
          c.attribute("address", STRING);
        });
        Clazz product = mm.haveClass("Product", c -> {
          c.attribute("productId", STRING);
          c.attribute("description", STRING);
          c.attribute("price", DOUBLE);
        });
        Clazz order = mm.haveClass("Order", c -> {
          c.attribute("orderId", STRING);
          c.attribute("date", STRING);
        });
        mm.associate(shop, "customers", MANY, customer, "shop", ONE);
        mm.associate(shop, "products", MANY, product, "shop", ONE);
        mm.associate(customer, "orders", MANY, order, "customer", ONE);
        mm.associate(order, "products", MANY, product, "orders", MANY);
    }
}
```

Listing 1: Fulib class model e.g. in *src/gen/java/theModelPackage/GenModel.java*

```
plugins {
 id 'java'
 id 'org.Fulib.FulibGradle'
     version '0.2.0'
}
```

Listing 2: Including the Fulib Gradle Plugin into build.gradle

3 Fulib Lightweight Tooling

A major problem with many modeling tools is, that you have to learn some complex diagram tool, e.g. MagicDraw (mag, 2020) or Microsoft Visio or Enterprise Architect (ent, 2020) or UML-Lab (UML, 2020) or others. Recently, many textual modeling languages have popped up e.g. based on xCore (xCo, 2020) or (Rumpe and Hölldobler, 2017). These textual modeling languages still require you to use some complex tool and to learn some new language. To avoid this, Fulib encodes your class model in Java using a small API, cf. Listing 1. Fulib then renders your class model as UML diagram for you, cf. Figure 1. To be honest, to use Fulib, you still have to learn the Fulib API. However, you may stick with your favorite Integrated Development Environment (IDE) and your IDE will provide you with comfortable editing support like completion and compile time checks.

To use Fulib you may download the Fulib libraries and add them to your classpath. However, we also provide a Gradle plugin that does this job for you (a Maven plugin is current work), cf. Listing 2.

The *FulibGradle* plugin provides a *generateScenarioSource* task. This task looks for *ClassModelDecorator*s within a dedicated *src/gen/java/...* subdirectory and compiles and runs these classes. This generates source code within your *src/main/java/...* directory. The *generateScenarioSource* task runs before your usual compile and test tasks.

4 Fulib Integration with Iterative Software Development

As discussed, our *FulibGradle* plugins keeps your model and your source code in sync, on each gradle build. This addresses the next main problem with modeling tools in industrial projects: their poor inte-

```
 1 public class ModelswardStoryTest {
 2     @Test
 3     public void testPayPal() {
 4         Shop uksShop = new Shop().setName("UKSShop");
 5         Customer alice = new Customer().setCustomerId("alice1")
 6                 .setName("Alice").setAddress("Wonderland␣1").setShop(uksShop);
 7         Customer bob = new Customer().setCustomerId("bob2").setName("Bob")
 8                 .setAddress("Wonderland␣1").setShop(uksShop);
 9         Product tshirt = new Product().setProductId("t42").setPrice(13.37)
10                 .setDescription("Great␣UKS␣Tshirt").setShop(uksShop);
11         Product mug = new Product().setProductId("m43").setPrice(4.20)
12                 .setDescription("UKS␣Coffee␣Mug").setShop(uksShop);
13         Order o44 = new Order().setOrderId("o44").setCustomer(alice)
14                 .setDate("2020.09.02").withProducts(tshirt, mug);
15
16         uksShop.process(o44);
17
18         FulibTools.objectDiagrams()
19                 .dumpSVG("paper/paypalObjects.svg", uksShop);
20         assertThat(o44.getState(), is("payed"));
21     }
22 }
```

Listing 3: Test Driven Development

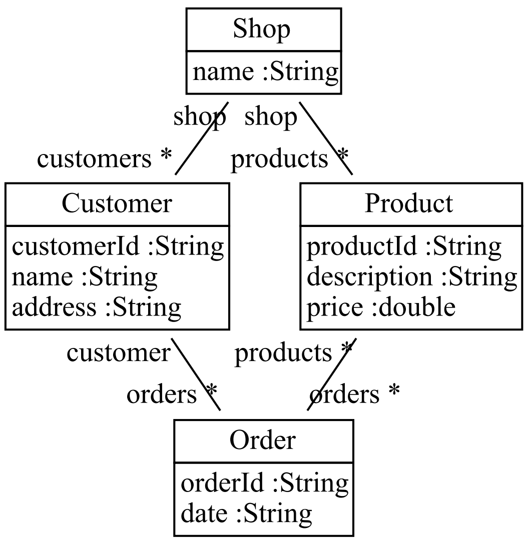

Figure 1: Example Class Diagram

gration into iterative software development processes. Frequently, modelling tools are used only in initial project phases and as soon as you start to add your business logic, code generation is detached and your code develops independent from the model.

If you use an agile or iterative software development approach you may follow the ideas of Test Driven Design. Therefore, you start writing a test that sets up a scenario for a certain functionality and then invokes the corresponding operation and validates that the desired effects have been achieved. We call such a test a scenario test, cf. (Zündorf et al., 1999). Such scenario tests may also be related to use cases.

A typical scenario test will set up some example object model and then invoke some business logic operation and than validate the results. Fulib has no generic model editor to set up an example object model. Actually, we have a simple textual scenario language you could use (cf. www.fulib.org). This textual scenario language may e.g. be used to discuss examples with customers or users during requirements engineering. However, this paper focuses on the design and implementation phase. Thus during design and implementation, you may just code the creation of example object models based on the class model implementation generated by Fulib, cf. Listing 3.

Building a model via Java code leverages code completion and editing support from your integrated development environment. Note also that our set-methods return the underlying object and thus they may be used in a ‘fluid’ code style.

To visualize the created object model you may use the FulibTools library, e.g. Line 19 of Listing 3 dumps an object diagram of our model, cf. Figure 2. As you use this within your tests only, it will not ship with your product, thus still no licence issues. We use these

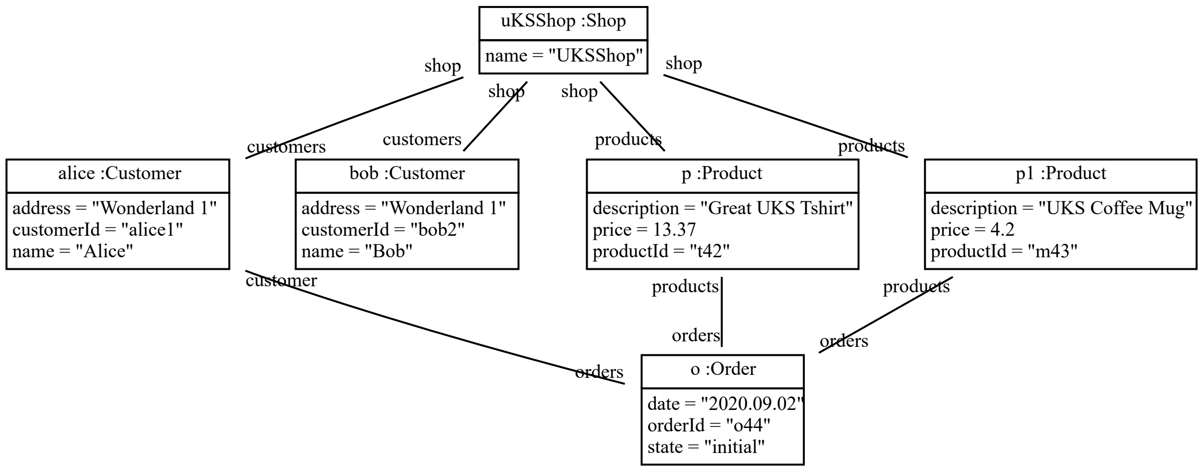

Figure 2: Example Object Diagram

object diagram dumps a lot during debugging, too.

Note, our test already contains a business logic method (*Shop.process(order)*), cf. Line 16 of Listing 3 and see Listing 4. Following the test driven development idea, we first wrote the method call in Line 16 of Listing 3 and the assert statement in Line20. Then we used auto repair features and code completion of our IDE to create method *Shop.process(order)* and the *state* attribute within our *Order* class. Then we used gradle to compile and run our test. This gradle build did another code generation. During this code generation, the Fulib code generator recognized the manual extensions of our model classes and merged them with the generated parts, seamlessly.

While you could / should add things like the *state* attribute to the model, sometimes you do not consider such helper elements as relevant for the model or you are focusing on your algorithms and do not want to bother with the modeling level. Or you just follow the test driven development approach and want to start with the usage of a feature before adding it to your system (or model). Actually, we might add the *state* attribute to our model after introducing it to our code via IDE. Then, the Fulib code generator will identify the manually added implementation and replace it with the (similar) generated implementation. To achieve this, Fulib uses a Java parser to identify attribute and method declarations and then these declarations are related to model elements via their names. After all, code and model would by in sync and the manually added attribute becomes part of the model, too.

As manual code extension is frequently required, there are many sophisticated techniques that allow developers to protect their code modifications and extensions from being overridden by some code generator, e.g. special code regions marked with pseudo comments or adaption of generated code via sub and super classes (e.g. in (Rumpe and Hölldobler, 2017) or generation gap pattern by Fowler (Fowler, 2010)). To make this even simpler and more seamless, Fulib identifies which code elements (attributes / methods, etc.) stem from or correspond to a model element (Ful, 2020b). Such code fragments will be overwritten on each project build. Manual changes to such code fragments will be lost. (You may protect such elements with a "*// no Fulib*" comment.) Any code fragment that has been added manually, e.g. a helper (*state*) attribute or business logic method (*process()*) remains untouched and is preserved during each project build. Thus, if developers extend the generated code with attributes or methods, they do not need to bother with the Fulib model or code generation.

In other approaches, the extra effort for code preservation frequently causes developers to detach the model from the code i.e. remove the automatic code generation from the build process or just never run the code generation again. With Fulib, the code generation is such seamless that you may keep the Fulib Gradle plugin included.

When breaking down our example *Shop.process(order)* method, cf. Listing 4, we recognized that we need an *email* address for our customers. This time we just extend the class model code from Listing 1 and run gradle build. This adds the *email* attribute to our model and to the generated code, still seamlessly integrated with the manual extensions.

A great functionality of integrated development environments are refactorings, e.g. renaming a class or an attribute together with all its usages. Most code generators have difficulties when you do e.g. such a renaming on generated code. As an example let us rename our *Shop* class into a *Store*. Using your IDE this will also update your test code and other usages

```
public class Shop {
   public void process(Order order) {
      if (order.getState().equals("initial")) {
         requestPayment(order);
      }
      else if (order.getState().equals("payed")) {
         deliver(order);
      }
   }

   private void requestPayment(Order order) {
      ...
   }

   private void deliver(Order order) {
      ...
   }
   ...
}
```

Listing 4: Business Logic Development

of the old *Shop* class.

If we run a gradle build after renaming *Shop* to *Store*, the Fulib code generator will regenerate the *Shop* class. This will also regenerate the association between *Shop* and e.g. *Product*. This results in numerous compile errors e.g. between the regenerated method *Product.setShop(Shop)* and the old refactored method *Product.setShop(Store)*[1] that both access the *Product.shop* attribute which the Fulib code generator has changed back to type *Shop*.

Luckily, the Fulib code generator is able to repair this mess, easily. We just rename *Shop* to *Store* in our class model code (Listing 1) and run the gradle build again. The Fulib code generator keeps a copy of the class model that has been used during the last code generation. On each new code generation, the Fulib code generator compares the old class model and the new class model. Thus Fulib recognizes that the *Shop* class is gone and that the corresponding associations are now referencing the new *Store* class. Thus, Fulib removes all code generated for the old *Shop* and the old associations. Then, the new elements are generated. During the generation of new elements, Fulib merges the new code with existing code (from the refactoring). Thus, once we have completed the renaming of *Shop* to *Store* in our model, Fulib repairs all code and the code compiles and works, again. A disciplined developer might do the renaming in code and model directly and no problem occurs. If you forget about the model, you will recognize the problem on the next gradle build, repair the model, and you are good, too.

In the best of all worlds, such a renaming would work on code and model, simultaneously. As far as we know, UML-Lab (UML, 2020) is the only modeling tool that supports this. To achieve this, your modeling tool needs to be tightly integrated with your IDE (which provides the code refactoring functionality including manually created code that is not addressed by your modeling tool.) However, Fulib is just a lightweight tool that is integrated into your development workflow via gradle and that works with any IDE. Thus, you have to do both renamings, manually. At least Fulib deals with the problem that you may forget to adapt the model, directly. This works not only for the renaming of classes but also for the renaming of attributes and associations. We have used Fulib in a Bachelor course on object oriented modeling in winter term 2019/2020 at Kassel University with about 80 students with great success and no student removed the FulibGradle plugin from his or her project. We use it a lot in our research projects, again with very good experiences. Code generation and manual code extension works great with Fulib.

[1] Yes, we should refactor the *shop - products* association to a *store - products* association. This refactoring may be done in the same way as the renaming of *Shop* to *Store*: first run the refactoring in your IDE and then do the renaming in your model, too.

```
...
CustomerTable customerTable = new CustomerTable(alice);
OrderTable orderTable = customerTable.expandOrders("Order");
orderTable.filter( order -> order.getState().equals("initial"));
ProductTable productTable = orderTable.expandProducts("Product");
doubleTable priceTable = productTable.expandPrice("Price");
System.out.println(productTable);
System.out.println("Total:␣" + priceTable.sum());
/* prints:
| Customer       | Order           | Product          | Price          |
| --- | --- | --- | --- |
| alice1 Alice Wonderland 1 | o44 2020.09.02 | t42 Great UKS Tshirt | 13.37 |
| alice1 Alice Wonderland 1 | o44 2020.09.02 | m43 UKS Coffee Mug | 4.2 |
Total: 17.57
*/
productTable.toSet().forEach(product -> uksShop.retrieve(product));
...
```

Listing 5: Fulib generated query API

5 Fulib Git Integration

Another frequent reason for decoupling the model from your code is when the model is not under Git or version control together with your code. EMF class models are usually stored in ECore files using XML format. XML based files have serious problems when e.g. a Git version control tool is used and merge conflicts occur. Fulib uses a Java based API to create a class model and to generate the implementation code, cf. Listing 1. This class model description (code) is compiled and executed by the *FulibGradle* plugin during the code generation phase and code is generated within the *src/main/java/...* directory of your project. As developers may extend the generated code manually, we recommend to put the whole *src* directory (including the generated code) under Git version control. As the class model description (code) is now versioned together with all other code of your project, you may use all Git features for collaborative software development on your model and your code together. You may e.g. use multiple feature branches to develop different features independently. For each feature you may extend and adapt the model description code and the other code parts of your system and then you merge the feature into your main development branch. Your model description code will merge similar to all other code.

6 Model Management

Industrial projects will most likely have a predefined architecture that e.g. employs a specific platform for model persistence. This might be an object relational mapping middleware or a noSQL database or some serialization framework like Jackson. As (Paige and Varró, 2012) states it is important to interoperate with such frameworks. Fulib achieves this interoperability by generating plain old Java code that follows the Java Beans conventions that are the basis for most persistence frameworks. In addition, it is possible to add arbitrary framework specific annotations to classes, attributes, and methods as frequently required by many frameworks.

Although industrial projects will most likely use their own persistence means, Fulib provides a general Yaml serializer *FulibYaml*. We use FulibYaml first of all to serialize our class models during code generation, within a file called *ClassModel.yaml* located within the Java source code directory of your model. Having a copy of the class model that has been used to generate the current Java implementation allows us to compute the changes done to the class model as well as to the Java code. This allows us to do some kind of three way merge of model changes and code changes. Per default, *ClassModel.yaml* is added to the project's Git repository together with the Java code of the project. FulibYaml keeps the elements of the *ClassModel.yaml* file as stable as possible such that merge conflicts occur as seldom as possible. In our modeling course and in our research projects that works great so far.

We use FulibYaml also for serialization within stu-

```
...
PathTable table = new PathTable("Customer", alice);
table.expand("Customer", Customer.PROPERTY_orders, "Order");
table.filter("Order", order -> ((Order)order).getState().equals("initial"));
table = table.expand("Order", Order.PROPERTY_products, "Product");
table.expand("Product", Product.PROPERTY_price, "Price");
System.out.println(table);
System.out.println("Total:␣" + table.sum("Price"));
/* prints:
| Customer        | Order           | Product          | Price          |
| --- | --- | --- | --- |
| alice1 Alice Wonderland 1 | o44 2020.09.02 | t42 Great UKS Tshirt | 13.37 |
| alice1 Alice Wonderland 1 | o44 2020.09.02 | m43 UKS Coffee Mug | 4.2 |
Total: 17.57
*/
table.toSet("Product").forEach(product -> uksShop.retrieve((Product) product));
...
```

Listing 6: FulibTables runtime library query API

dent projects during teaching. When a student or an industrial project extends the class model manually we want to be able to still use FulibYaml to serialize it. Therefore, FulibYaml uses reflection mechanisms and just relies on common Java Beans conventions. Thus, FulibYaml may also be used on manual and on other model implementations. FulibYaml even works for EMF models. This is easy for serialization. For deserialization we added some extra mechanisms that lookup the EMF factory that is necessary to create objects.

We want FulibYaml to deal with iterative software development and with model evolution as flexible as possible. Therefore, FulibYaml does not use a model specific schema for its Yaml files. If one extends the class model e.g. with a new attribute, old Yaml files will just not contain a value for the new attribute but still load without problems. The new attribute just does not get a value. Similarly, if we remove a certain property from our class model, that property will be ignored on loading old Yaml data. It is also possible to integrate explicit data migration steps into the loading process.

Well, you may do similar things with many other serialization frameworks, too. We just wanted to emphasize that support for iterative software development and a flexible integration into the software development process is important for the serialization mechanism, too. This should be made as easy as possible.

7 Model Queries and Transformations

In addition to code generation for class models, support for model queries and transformations is another major contribution of modeling tools and frameworks. Again, 1) proprietary libraries, 2) licence issues, 3) vendor lock-in to (immature) university prototypes, and 4) poor integration with iterative software development are major concerns. In addition, a proprietary query and transformation language adds to the complexity of the overall development process.

Fulib provides different approaches in order to deal with these problems. First, on demand, the *Fulib.tablesGenerator* generates model specific query operations. Listing 5 shows the usage of such generated tables based on the object structure created in Listing 3.

For each model class like *Customer*, Fulib generates a corresponding table class, in this case *CustomerTable*. These table classes provide *expand* operations for each attached association e.g. *expandOrders*, cf. Line 3 of Listing 5. Operation *expandOrders* creates a new wrapper object of type *OrderTable* that refers to the original underlying table data. Operation *expandOrders* takes each row of the original table (in our case one row with one column with content *alice1*) and computes the set of all objects reachable via an *orders* link (in our case e.g. order *o44*). For each combination of source and target object a new row is added to the underlying table data. (The old rows are removed.) The filter operation in line 4 of Listing 5 iterates through all rows of the underlying table and ap-

```
...
PatternBuilder pb = FulibTables.patternBuilder();
PatternObject customer = pb.buildPatternObject("customer");
PatternObject order = pb.buildPatternObject("order");
PatternObject state = pb.buildPatternObject("state");
PatternObject product = pb.buildPatternObject("product");
PatternObject price = pb.buildPatternObject("price");
pb.buildPatternLink(customer, Order.PROPERTY_customer,
      Customer.PROPERTY_orders, order);
pb.buildPatternLink(order, null, "state", state);
pb.buildAttributeConstraint(state, str -> str.equals("initial"));
pb.buildPatternLink(order, Product.PROPERTY_orders,
      Order.PROPERTY_products, product);
pb.buildPatternLink(product, "price", price);
PatternMatcher matcher = FulibTables.matcher(pb.getPattern());
ObjectTable table = matcher.match("customer", alice);
System.out.println(table);
/*
| customer | order | state | product | price |
| --- | --- | --- | --- | --- |
| alice1 Alice Wonderland 1 | o44 2020.09.02 | initial | t42 Great UKS Tshirt | 13.37 |
| alice1 Alice Wonderland 1 | o44 2020.09.02 | initial | m43 UKS Coffee Mug | 4.2 |
*/
```

Listing 7: FulibTables query pattern

plies the passed lambda expression on the element of the current column (in our case on the order element). In our example this evaluates to true and we keep our single row. Now line 5 expands the current order(s) via its (their) *products* link. This results in two rows consisting of a copy of the old row extended by one of the two ordered products. Line 6 adds a *price* column for each *product*. Line 7 prints the resulting table. This has been added to Listing 5 as comment in lines 10 to 13. Line 8 calls *sum* on the *price* column of our table and prints the result, cf. Line 14.

If you want to do not only a model query but also a model transformation you may e.g run a lambda expression on each element of a certain column, cf. line 16 of Listing 5. Alternatively, you might iterate through all rows of the underlying table. This allows you to access e.g. customer, order, and product together.

The code generated for these table operations is again self contained and does not need any runtime library. The code only relies on the typical get operations. Thus, the generated table code works not only with model classes generated by Fulib, but also with model classes from other frameworks (e.g. EMF) or manually crafted model classes.

However, code generation for this model specific table code relies on the existence of a Fulib class model (as created e.g. in Listing 1). If you have created your model (code) in a different way, you could reverse engineer the class model for Fulib and then use only the table generator. If you do not want to generate the table code, Fulib also provides a generic *PathTable* within a small runtime library (i.e. *FulibTables*).

Listing 6 shows the same query as Listing 5, but this time using our generic *PathTable*. Thus, in Listing 6 we need to provide the names of table columns and associations and attributes as string parameters. This has three drawbacks: first, we loose static type checking, you may accidentally ask a *Customer* object to expand via link "xy" which is not possible as the class *Customer* does not have such an association. Correlated with missing type information is the lack of auto completion support in your IDE. Finally, *PathTable*s rely on Java reflection. This is a little bit slower than the generated solution. Again *PathTable* works with Fulib models, EMF models, manually crafted models, etc.

In addition to *PathTable*s our *FulibTables* runtime library also provides more general model queries via object patterns, cf. Listing 7. These patterns are inspired by graph transformations, cf. (Zündorf et al., 2017). Note, patterns are still compatible to all models that stick to the Java Beans conventions. Thus, you may e.g. use our patterns on EMF models or on manually crafted models.

In (Eickhoff et al., 2019) we also provide EMFeR a generic model checking tool based on model queries and model transformations. EMFeR works perfectly well with Fulib queries and transformations, too.

8 Summary

The Fulib modeling library tries to address typical concerns and requirements of industrial software developers. You get a lot of Fulib functionality without the need of a runtime library. Code generation is integrated into your Gradle build. Fulib supports iterative software development as good as we can. Fulib deals with Git version management reasonably well. Model specific query functionality may be generated which again avoids runtime libraries and licence issues. Generic query functionality is provided within a small runtime library (MIT licence). The generic query functionality is agnostic to different model implementations. It works not only for Fulib generated code but also for code generated by other tools or for manually crafted code. We have evaluated the Fulib library within a modeling course with about 80 participants at Kassel University in winter term 2019/2020 and within several research projects at our department. We are now ready to approach industrial partners to get their feedback. We have already demonstrated Fulib within a talk to local industry at the Java User Group Hessia, Germany, in August 2020 (Ful, 2020a).

REFERENCES

(2020). Enterprise architect. https://www.sparxsystems.de/uml/neweditions/. Accessed: 2020-09-02.

(2020a). Fulib - macht java code aus klassendiagrammen. https://www.youtube.com/watch?v=eamdeXCxysk. Accessed: 2020-09-02.

(2020b). Fulib class org.fulib.generator. https://github.com/fujaba/fulib. Accessed: 2020-02-12.

(2020). Magic draw. https://www.nomagic.com/products/magicdraw. Accessed: 2020-09-02.

(2020). Uml-lab, yatta solutions gmbh. https://www.uml-lab.com/de/uml-lab/. Accessed: 2020-09-02.

(2020). xcore. https://wiki.eclipse.org/Xcore. Accessed: 2020-09-02.

Eickhoff, C., Lange, M., Raesch, S.-L., and Zündorf, A. (2019). Emfer: Model checking for object oriented (emf) models. In *MODELSWARD*, pages 511–518.

Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

Nickel, U., Niere, J., and Zündorf, A. (2000). The fujaba environment. In *Proceedings of the 22nd international conference on Software engineering*, pages 742–745.

Paige, R. F. and Varró, D. (2012). Lessons learned from building model-driven development tools. *Software & Systems Modeling*, 11(4):527–539.

Rumpe, B. and Hölldobler, K. (2017). Monticore 5 language workbench. edition 2017.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: eclipse modeling framework*. Pearson Education.

Zündorf, A., Gebauer, D., and Reichmann, C. (2017). Table graphs. In *International Conference on Graph Transformation*, pages 221–230. Springer.

Zündorf, A., George, T., Lindel, S., and Norbisrath, U. (2013). Story driven modeling libary (sdmlib): an inline dsl for modeling and model transformations, the petrinet-statechart case. *Sixth Transformation Tool Contest (TTC 2013), ser. EPTCS*.

Zündorf, A., Schürr, A., and Winter, A. J. (1999). *Story driven modeling*. Univ.-Gesamthochsch. Paderborn, Fachbereich Mathematik-Informatik.