Template- and modelbased code generation for MDA-Tools

Leif Geiger
SE, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
leif.geiger@uni-kassel.de

Christian Schneider
SE, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
cschneid@uni-kassel.de

Carsten Reckord
SE, Universität Kassel
Wilhelmshöher Allee 73
34121 Kassel
creckord@uni-kassel.de

ABSTRACT

The Model Driven Architecture (MDA) proposes model transformations to obtain an executable model from a platform independent model. Unless one uses an interpreter the common executable model of an application is specified in some programming language. To obtain such an implementation of a model automatically is the task of *code generation* in MDA-Tools. In this paper we present a modelbased approach to this task. It uses explicitly modelled intermediate data and makes use of code templates for the final transformation into pieces of text.

1. INTRODUCTION

CASE-Tools which implement operational semantics do commonly provide either an interpreter or a code generation component to make use of this semantics. In this paper we discuss a concept for such code generation component. The general task of code generation is to transform an abstract syntax graph (ASG) into one or more programming language files. These are compiled (if applicable) and executed to operationalize the specification in the CASE-Tool afterwards.

Our approach to code generation in Fujaba [2] uses Velocity Templates [1] to generate source code in the final step. See Figure 7 for example template code.

To choose the templates to be applied and to supply the template instantiation with parameters an intermediate layer of tokens was introduced. These tokens are created by analysing the ASG elements, for which code should be generated. This enables sorting, optimizations and extensions to work with explicit object structures without altering the ASG of the specification.

1.1 Example

As a simple example we want to show a part of the code generation for a simple graph transformation rule throughout this paper. Generating code for Fujaba's graph transformation rules is one of the core requirements that must be fulfilled by a code generation for Fujaba. The mapping from graph transformations to java code in general is described in [7].

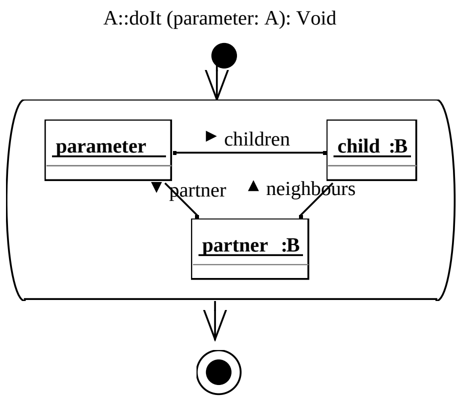

Figure 1: Fujaba rule diagram as an example

The example can be described as follows (cf. Figure 1): The object `parameter` is passed to the rule as method parameter, the `child` object can be found over a link called `children`. Additionally an object named `partner` exists. This can be found by navigating along the link `neighbours` from object `child`. Alternatively it can be found over the `partner` link starting at `parameter`. In this case the object `child` can be found by navigating along the `neighbours` from object `partner`. The rule does not change the object graph (graph-theoretically spoken: RHS equals LHS).

2. CONCEPT

The code generation was split into several subtasks which will be described in detail in the following subsections 2.1 to 2.4. A brief overview is given by Figure 2.

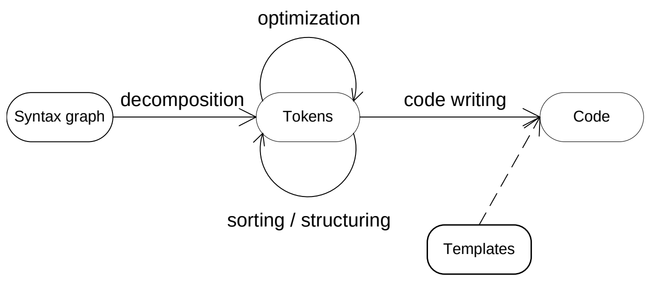

Figure 2: Subtasks of the code generation with initial and resulting data

2.1 Creating Tokens (decomposition)

The first task is to create atomic operation tokens for each syntax-graph element. The same kind of syntax-graph elements can cause different tokens to be created, because of their different attribute values or context. Additionally a single syntax-graph element usually results in multiple tokens. Each token represents an code fragment that should be generated.

The result of the token creation task is a set of tokens which are usually referring each other in several ways (so forming a graph of tokens).

Spoken in terms of our example a token of type `CheckBoundOperation` is created to check if the object `parameter` is bound. For each link a token of type `CheckLinkOperation` is created for each direction the link can be traversed. Figure 3 shows the two resulting `CheckLinkOperations` for the `child` link (the direction is defined by the `subject` link). Tokens for the other two links are created accordingly (not shown). As the `bound` attribute (not displayed in Figure 3) of the `child` object is false, the generated code must search for the object. Therefore a token of type `SearchOperation` is created for each link leading to the object. In addition to the `SearchOperation` along the `child` link shown in Figure 3, another `SearchOperation` for the `child` object is created for the `neighbours` link and two more for the two links connected to the `partner` object. There is no `SearchOperation` for the `parameter` object as it is bound.

Most of the operations require one or more of the involved objects to be matched before they can be performed. These prerequisite objects are specified with `needs` links from the tokens. For example, `CheckLinkOperation t3` needs Objects `o1` and `o2` to check the link between them.

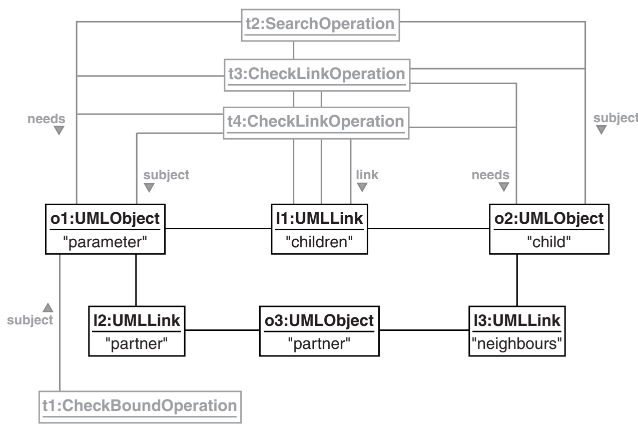

Figure 3: Exemplary tokens (grey) for the transformation rule seen in Figure 1

2.2 Sorting and Structuring

In most cases translating the generated token graph into code directly is difficult. It is easier to first sort and structure the tokens to get a token graph that better reflects possible operational dependencies among tokens. The transformations necessary to structure the tokens depend on the kind of syntax graph the tokens are derived from. Tokens from class diagrams usually need little to no further structuring. Behavioral diagrams on the other hand usually require the token graph to be structured and brought into a hierarchical form that is later mirrored in the hierarchical block structure of the generated code.

We will focus on the structuring of Fujaba's rule diagrams, but similar transformations can be used to structure the token graphs of other behavioral diagrams. A detailed explanation of rule diagrams used in Fujaba can be found in [7]. To generate code for a rule diagram, a *search plan* - an operational form of the diagram - has to be found, that defines how to match the LHS of the rule and how to perform the graph transformation. Since in general many valid search plans exist for a rule diagram, it is also important to find an efficient one, which will be discussed in section 2.3.

In [7] Zündorf describes a basic method to find a search plan and create code directly from the rule diagram. We will now present a method to find a search plan through transformation of the token graph and will then use the found search plan for the template based code generation.

One problem in finding a valid search plan is to decide which tokens are to be used, because usually not all of the generated tokens are needed in a search plan. For example, the objects `child` and `partner` in the example of Figure 1 can both be reached by a `SearchOperation` directly from `parameter`. In that case a `SearchOperation` between `child` and `partner` is not needed and instead a `CheckLinkOperation` can be performed for the `neighbours` link.

The other problem is to sort the used tokens correctly such that all prerequisites of an operation are already matched when the operation is to be performed. In our example an alternative search plan could reach the `partner` object by first finding the `child` from `parameter` and then matching `partner` from `child` over the `neighbours` edge.

Our search plan is a tree structure with ordered child lists, which will be interpreted in a depth-first manner in the rest of the code generation process. Tokens depending on other tokens due to prerequisite objects are located in the subtree below the tokens they depend on, ensuring that all prerequisites are matched when using depth-first traversal.

Transforming the token graph into this search tree is fairly simple:

1. Add a new temporary root node to the LHS graph of the rule diagram and connect all bound objects of the LHS to it. Find a spanning tree starting at this node (using the links as edges).
2. For all edges in the spanning tree the `SearchOperation` towards the child is added to the search tree. Its parent is the `SearchOperation` that finds its sole prerequisite object or the root node if the prerequisite object is bound (cf. Figure 3: `t2` finds `o2` and has `o1` as its prerequisite).
3. For all links not in the spanning tree a `CheckLinkOperation` will be added in the next step. Discard

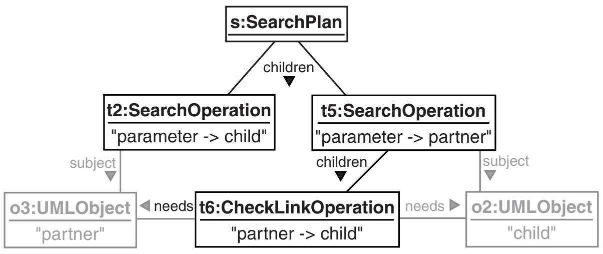

Figure 4: Search plan with partially added Check-Operation

all other `Search-` and `CheckLinkOperations` (cf. Figure 3: Only one of the three operations `t2-t4` for the `children` link will be in the final search plan).

4. Successively add all `CheckOperations` to the search tree as follows:
   (a) Add the `CheckOperation` as child to a `SearchOperation` that matches one of its prerequisites
   (b) Find a `SearchOperation` for another of its prerequisites. Find the first common ancestor of the two `SearchOperations` and move the subtree with the `CheckOperation` from the common ancestor to the new `SearchOperation`. This is possible because siblings in the tree are independent of each other.
   (c) Repeat for all prerequisites.
5. Add the tokens for the RHS as children of the root node, adhering to the order of object and link destruction, creation and collaborations

The search tree to match all unbound objects in our example consists of two `SearchOperations` matching the `child` and `partner` object. Additionally a `CheckLinkOperation` for the remaining link not used for the search is required. Figure 4 shows the search tree with the `CheckLinkOperation` for the `neighbours` link added below the first of the two `SearchOperations` for its prerequisites as described in step 4.1.

Now the tree has to be modified to get the `CheckLinkOperation` below the other `SearchOperation`, too, as described in step 4.2. Therefore the subtree starting at `SearchOperation t5` and containing the `CheckLinkOperation` is moved below `SearchOperation t5`. Since the `CheckLinkOperation` has no further prerequisites it is now correctly added to the search plan. The resulting, final search plan is shown in Figure 5.

To support easy extensibility, the creation of the token tree is realized with a handler chain similar to the chain of responsibility pattern. The search plan is successively built by the handlers in the chain. The first handler receives the set of available tokens and the empty root node of the tree to be built. Each following handler receives the remaining unused tokens and the tree from the previous handler, restructures or incorporating new tokens into the tree and passes it on. This way, handling of new tokens can be added fairly easy, even though in most cases this will not be necessary because

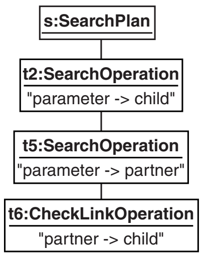

Figure 5: Final Search plan for our example

generic handlers exist that can deal with most tokens based solely on their prerequisites, which is usually sufficient.

2.3 Optimization

An important quality feature of the generated code is its runtime efficiency. Therefore we want to find, among the valid search plans, the one that results in the best runtime cost.

The most optimization potential can be leveraged from the selection of the `SearchOperations` used in the initial search tree. Obviously following a to-one link is cheaper than checking multiple objects via a to-many link. Given a cost model for the tokens, a good solution can therefore be found easily by finding a minimal spanning tree to build the initial search tree.

Additionally, fast operations (like link checks) should be performed as early as their prerequisites allow to find invalid matches early and thus avoid further expensive searches. Therefore, when moving a subtree as described in the tree generation process above, its tokens should afterwards be propagated towards the common ancestor as far as their prerequisites allow or until only cheaper tokens are on their path to the common ancestor.

Finally, with the exception of tokens from the RHS of the graph, siblings in the tree are independent from each other. Therefore, subtrees with a low runtime cost relative to the number of tokens in the subtree can be moved to the front of the ordered child lists, again allowing for earlier detection of invalid matches at a lower total cost.

All optimization steps can be easily realized as handlers in the handler chain. The cost model for the tokens is realized as a separate chain of responsibility that can be accessed by all the handlers. For link operations an additional model for the average payload of the referenced link is maintained, separating access costs for the different link types (sorted, ordered, hashed etc.) from the typical number of objects reachable by the link.

In the current implementation the cost and payload models give a static cost estimation, only. They can however be easily extended to e.g. take statistical information gathered from execution on typical data into account.

2.4 Code writing

After having optimized the set of tokens, we are finally able to generate code for them The class responsible for this is

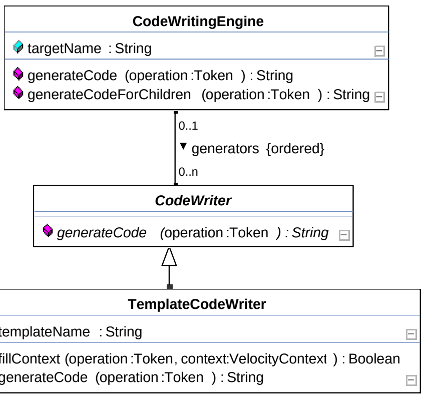
CodeWritingEngine
targetName : String
generateCode (operation :Token) : String
generateCodeForChildren (operation :Token) : String
0..1
generators {ordered}
0..n
CodeWriter
generateCode (operation :Token) : String
TemplateCodeWriter
templateName : String
fillContext (operation :Token, context:VelocityContext) : Boolean
generateCode (operation :Token) : String

Figure 6: Class diagram for code writing

called `CodeWritingEngine`. It has a list of `CodeWriters` which implements the chain of responsibility design pattern to allow extension, cf. figure 6.

The token tree is visited inorder. Every visited token for which we want to generate code is then passed to the chain, so that the code writer responsible can generate code for the token. This is usually done by an instance of class `TemplateCodeWriter`. This code writer opens the velocity template with the name specified by its `templateName` attribute and passes the token and additionally needed information as context to the velocity template engine. This additional information also includes the code generated for all the children of the token in the tokens hierarchy. Then the velocity engine is used to generate the code.

If e.g. a token of type `ObjectAssignmentOperation` is visited, it is passed to the chain. The object of type `ObjectLifecycleCodeWriter` is responsible for such tokens, so it will initialize the template which is shown in Figure 7. The `ObjectLifecycleCodeWriter` will look up the `UMLObject` which is refered by the token and pass it as `object` parameter when executing the template. In the template in lines 1 to 3 some local variables are set (the name and type of the object and whether or not it is optional. In lines 4-6 the `$tmpName` variable is set depending on whether a type cast is needed or not. In line 9 the object is finally bound. The following lines preform a type check if a type cast is needed.

3. MODELBASED TESTING

Testing the correctness of generated code is generally a hard task. To test code generation, one would start with an arbitrary syntax element in some context and generate code for it. Just comparing the generated code with the expected one would not give a good test criteria: if the code is indented a different way or somehow refactored (different code but same semantics), a test failure would nonetheless be reported. We made the experience that in this case the developer tends to believe that the new code is correct and just overwrites the expected code with the code generated by his new code generation. This way the test would of course execute successful again but possible bugs would have been ignored.

It would be more useful if one could test whether or not the generated code has the expected behaviour. For code

```
 1 #set( $name = $object.ObjectName )
 2 #set( $type = $object.ObjectType )
 3 #set( $optional = $object.isOptional() )
 4 #if ( $typeCast )
 5   #set( $tmpName = fujaba__TmpObject )
 6 #else
 7   #set( $tmpName = $name )
 8 #end
 9 $tmpName = $source ;
10 #if ( $typeCast )
11    ensure correct type and really bound
12   #if ( $optional )
13     if ( $tmpName )
14     {
15        $name = ($type) $tmpName ;
16     }
17   #else
18     JavaSDM.ensure ( $tmpName instanceof $type ) ;
19     $name = ($type) $tmpName ;
20   #end
21 #end
```

Figure 7: Example template that generates Java code for binding an object

which is compiled afterwards, like e.g. our java code, a first hint whether or not the code may be correct is given by the compiler. If the compiler quits with errors, the code is not correct. But obviously this is not a sufficent test criteria.

Our idea is then to run the generated code and test the results. We do this at model level using bootstrapping. To test our java code generation, we use the following approach:

- Structural code, like class definitions, method and attribute declaration, is tested by hand written JUnit tests. Code generation for class definitions for instance is tested using the java compiler, for the most part, which is invoked by a unit test.
- Code generation for method calls within activities is tested by hand written tests as well. In this simple case this is done by comparing the code with the expected one.
- Additional syntax elements of Fujaba's rule diagrams are tested using the modelbased testing approach described in the following paragraph.

The idea of the modelbased testing approach is to model JUnit tests in Fujaba. For these tests code is generated using the code generation to be tested. The tests are then executed using the JUnit framework. The tests should check the behaviour of the generated code by using just the axioms already tested by the hand written tests described above. In more detail, this is done the following way:

- A test class extending the `TestCase` class, provided by the JUnit framework, is modeled in Fujaba.
- Within this class, a unit test, which checks whether or not constraints are interpreted the correct way, can be modeled. This test makes use of method calls, only, which are already tested.

- Assuming that the code generation for constraints does work, what means that the previous test executes successfully, new tests can be modeled which make use of constraints. Such test are tests for the activity diagram parts (sequences, loops, branches).
- On top of this, tests can be modeled, which check additional constructs (creation of objects and links, checking of links, destruction of objects and links...).
- Code for the test modeled in Fujaba is generated using the new code generation and the JUnit tests are executed.

Figure 8 shows the method body for the test method which checks the code generation for to-one link checks. In the first activity two objects are created. The next activity should (if code generation works) check that there is a link between these two objects. If a link is found by the generated code, this is obviously not the desired behaviour (as there is no link between these objects) and a JUnit failure is reported. Otherwise such a link is created and checked for again. If this executes successfully, the generated code has the desired behaviour. The test finishes successfully. That means, if certain parts of the code generation (creation of objects and links as well as sequences of activities and branching) do work, the test in figure 8 checks whether or not code generation for to-one link checking works.

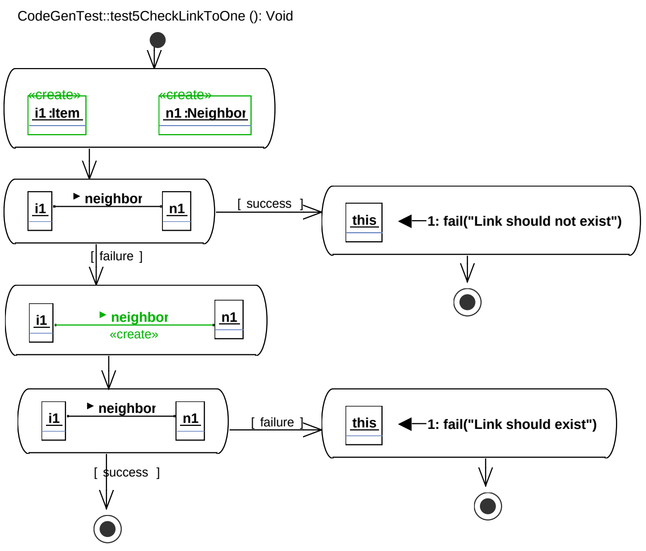

Figure 8: Test method for to-one link checks

4. BOOTSTRAPPING

As Fujaba offers a full-featured model transforamtion language, it would be a good proove of concept if we model Fujaba with Fujaba itself. Such process is called bootstrapping.

As all parts described above are modeled in Fujaba, bootstrapping Fujaba is finally possible at least for Fujaba's code generation. Till now, not all features of Fujaba are implemented within the new code generation (e.g. support for multi links is still missing). So the bootstrapping process is not yet complete. As soon as we have added these missing features, it would be possible to generate code for the specification in Fujaba using a code generation which was generated by Fujaba itself. This way, it should be possible for a code generation to generate its own code. This bootstrapping is planed for near future using the following process:

If a complete version of the code generation is available:

- Generate code for the new version using the previous code generation.
- Execute the JUnit tests as described in chapter 3.
- Generate code for the new version using itself.
- Execute the JUnit tests against this code generation.
- Generate code from the specification again to ensure, that the generated code equals the one generated before.

5. RELATED WORK

Zündorf describes in [6] how the graph transformations of PROGRES [3] can be split into operations in an operation graph. Then he discusses how to find a search plan (a sequence of search operations) in the operation graph. The search plan is optimized using a given cost model. The decomposition described in chapter 2.1 as well as the optimization in chapter 2.3 uses similar techniques.

In [5] Varró et al. describe a method to find cost efficient search plans from statistical data gathered on typical instance models at design time. Then they propose an adaptive approach that generates multiple search plans and selects the best one at runtime based on statistical evaluation of the current instance model. This approach could be easily incorporated into our current approach since the cost model is well-prepared for more elaborate analysis and the statistical data could easily be gathered by preparing the velocity templates accordingly.

In [7] the transformation from Fujaba's rule diagrams to java code is described. The proposed java code is the basis for our templates discussed in chapter 2.4. A short algorithm for code generation is also stated. Our approach uses a more elaborated algorithm since the algorithm in [7] does not create an intermediate model and only applies few optimization strategies.

The MoTMoT approach [4] also uses a template-based approach to generate code from transformations specified in Fujaba's model transformation language. But unlike Fujaba, MoTMoT does not offer an editor to create story diagrams, but provides a UML 1.4 profile which uses annotated UML class diagrams and annotated UML activity diagrams to model rule diagrams. This way, story diagrams can be drawn with every UML 1.4 compliant editor, like Together, MagicDraw or Poseidon. However, the MoTMoT approach also lacks an intermediate model and elaborated optimization techniques.

6. CONCLUSIONS AND FUTURE WORK

The model-based approach to code generation described in this paper has shown to be very flexible, easy to implement and simple to use. We managed to avoid dependencies to the target textual language in the generator model. All target language elements are expressed in the templates. Only the basic language paradigm (imperative) and some structural information (class, method, declaration hierarchy) is implicitly contained in the implementation.

We expect, introducing new imperative output languages will be possible very quickly. However, this causes creation of multiple similar template files. This tends to increase maintenance cost as behavioral changes in a template must be reflected for all generated langauges. In opposition to that the amount of template code is very low for a single language, compared to the code that was neccessary in the previous Fujaba code generation (more than factor 3).

As the complete code generation model (without templates) is modelled with Fujaba itself this approach paves the way to bootstrapping Fujaba - generating Fujaba with Fujaba. But as well as completing the code generation to support all syntax elements of Fujaba, bootstrapping is still future work.

From the optimizations described in section 2.3, only the minimal spanning tree approach is currently in use. The other methods remain to be implemented. Another area of future work is the optimization based on statistical execution data.

We expect, that the currently implemented transformations, that are used to generate and alter the intermediate data, can be inverted quite easily (except for omitted tokens). This makes us confident that reverse engineering of the generated code to obtain the original model again should be achieved with low cost. Singly the inversion of templates still requires some research work.

7. REFERENCES

[1] Velocity Homepage. http://jakarta.apache.org/velocity/, 1999.

[2] Fujaba Group. The Fujaba Toolsuite. http://www.fujaba.de/, 1999.

[3] Progres Group. PROGRES: Programmed Graph Rewriting System. http://www-i3.informatik.rwth-aachen.de/research/projects/progres/, 2004.

[4] H. Schippers, P. V. Gorp, and D. Janssens. Leveraging UML Profiles to generate Plugins from Visual Model Transformations. In *Software Evolution through Transformations 2003 (SETra03)*. ICGT, Rome (Italy), October 2004.

[5] G. Varró, K. Friedl, and D. Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In *International Workshop on Graph and Model Transformation (GraMoT)*. GPCE 2005, Tallin (Estonia), September 2005.

[6] A. Zündorf. Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Verwendung, PhD Thesis (in German), 1995.

[7] A. Zündorf. Rigorous Object Oriented Software Development, Habilitation Thesis, 2001.