Reuse and Customization for Code Generators: Synergy by Transformations and Templates

Robert Eikermann, Katrin Hölldobler, Alexander Roth, and Bernhard Rumpe

Software Engineering, RWTH Aachen University, Ahornstrasse 55, Aachen, Germany
{eikermann, hoelldobler, roth, rumpe}@se-rwth.de
http://www.se-rwth.de

1 PRINCIPLES OF CODE GENERATION

Model-driven development (MDD) relies on code generation as an essential ingredient to systematically generate source code from an abstract representation of a software system. To generate source code, each concept of the input language has to be mapped to concepts of the target language [6, 35]. The most used approaches that have been proposed to perform this mapping are template-based [36, 4] and transformation-based approaches [20, 6, 10, 9].

However, there is much more to say about good code generators. It is therefore worthwhile to first look at the important principles for a good code generator:

- *Code generation facilitate the reuse of models and the generator.*
  Ideally, a code generator is independent of any model and thus can be reused for many different models. Reuse, however, also works, when one model is mapped to different target platforms and operating systems or evolving hardware and software technology stacks.
- *Within a project, code generation must be re-doable at each time and by any developer.*
  As a consequence, generated code cannot be manipulated by hand. However, handwritten code and generated code interact with each other and fine-grained interaction patterns are necessary.
  Another consequence is to not version control generated code, because redoing the generation in parallel may lead to conflicts (even if its only the generation time stamp).
- *Generated and handwritten code are strictly separated in different artifacts.*
  Some build processes (e.g. Apache Maven [39]) even separate these artifacts into different hierarchies. The generation-gap problem can be overcome with a variety of mechanisms, some of which are assisted by object-orientation, such as subclass building. An interesting approach is for example the generation of classes that are embedded in a "sandwich", namely a handwritten super-interface to allow function extensions and a handwritten subclass for overriding [18].

The need of embedding all code into classes complicates the interaction between handwritten and generated code, but higher order functional abstractions, partial classes and similar approaches further improve a conceptual integration while keeping the code separate.

- *Code generators needs to be flexible to adapt to different, and evolving technology stacks.*
  Pure development of a code generator is always more work than directly implementing the system by hand. Code generators become interesting, when they are reusable. That includes a given code generator can easily be adapted to a new or evolving technology stack. Flexibility of the code generator includes the possibility to adapt the generation result without having to know all details of the generator. Templates for example are a good technology to adapt a generator, without going into its code.
  Flexibility is potentially also necessary within the project, where some parts of the model are mapped to different target stacks than other parts. Sometimes the target code needs to be enhanced by additional functionality, etc.
- *Maintainability of the generator is important.*
  Maintaining a generator is necessary, when the technology stack of generator changes, or when the generation has to be adapted. Thus maintainability is to some extent covered by flexibility of the generator. If however, the designated flexibility doesn't suffice anymore, it is important to have access to the generators source, e.g. from an open-source project, and the generator architecture is well structured and the code readable.
- *The generation process needs to be reliable and potential error messages understandable.*
  It is necessary, that the generation process doesn't terminate with internal errors – at least the user needs to get an understandable explanation and should be able to act accordingly.
  On the contrary, it's also necessary that erroneous models are detected early in the generation process and communicated to the developer. It is mandatory, that only correct code is generated. This includes that the generated code is compilable, but also behaves well.
- *The generated code needs to be reliable.*
  The generated code should be pretty robust against incorrect forms of usage. Depending on the form of use, it should on the other hand either be very robust (e.g. in airplanes) or early and quickly give errors and exceptions, when detecting a misuse (e.g. when the generated code is itself used in a batch tool).
  Reliability also includes, the code uniformly behaves in the same way, or that generated data structures are by construction correct and their data cannot be corrupted. The DEx generator e.g. has taken deep efforts to ensure correctness of generated classes and association codes at all (interesting) times [28].
- *A generator should be compositional.*
  Composition comes in several flavors: (1) Several different models are fed into one generator, that internally is composed of several sub-generators,

contributing to the same artifacts. (2) Several individual generators take the same model and produce artifacts that can be composed (in the product). (3) Several individual generators take individual models, but again produce composable artifacts.
  This task is difficult to achieve, because it either enforces that generators exchange information about the targets they create, or the developers of generators have an agreement, how the interfaces of the generated artifacts interact. Sometimes, both is necessary.
- *A generator should be smart.*
  One could even say, that a generator could be intelligent. That would mean that some of the domain or technology knowledge is neither part of the model, nor part of the hand written code that the generator generates against. E.g. if a class diagram contains certain classes, such as *Person*, the generator knows what kinds of attributes and functions should be embedded, where to store the objects in the data base, etc.
  A smart generator could allow much more concise and abstract models. That would also mean that the purely mechanical view on generators (mentioned above an in [6, 35]) would not be completely valid anymore.
  It could be that it's not the generator itself, but its customization through templates, predefined models and code that are woven into the generated code, which make a generator smart. However, from the users point of view it's the generator that bears smartness.
- *A generator should assist agile development.*
  In principle a project agility becomes more agile, when we can use abstract models instead of more detailed hand written code [29, 31].
  However, agility also very much lives from early and immediate feedback. If the generation process takes to long, the number of build script executions goes down considerably, the feedback gets later and thus error finding more complex. Furthermore, developers get bored and start doing other things, which leads to an interruptions of the developer "flow".
  As a consequence, a generator must be quick when generating and incremental, that means should detect, what really needs to be re-generated.

We can also observe, that round-trip engineering, i.e., generating code from models and retrieving models from code either only works for structurally equivalent models, such as class diagrams and object-oriented programs. The other alternative only was to embed the original model as an (unreadable) comment and retrieve the model not from the code, but from the comment. While having merits in some situations, both approaches are not really that helpful in normal projects.

1.1 AN EXAMPLE FOR CODE GENERATION

After having discussed so many general principles, we have to admit that these principles to some extent conflict. For example reliability of the generated code very much depends on how much flexibility of generator is actually used. The

more and deeper adaptations are made to the generator, the riskier it is that a generator doesn't fulfill all required goals anymore.

Even though, we don't have answers for all needs, we would like to demonstrate in the following, how a combination of a template- and transformation-based approach improves the situation with respects to several of the goals. These ideas go back to several projects, including the MontiCore language workbench [21, 22] itself, where this combination is widely used. Especially for *data-centric applications* [25] practice has shown that using transformation- or template-based code generation in isolation restricts flexibility when generating source code for the presentation layer and the application layer (cf. [26]) and has disadvantages in realizing code generator modularity (cf. [41]).

Our approach targets object-oriented output languages and is subdivided into three steps. First, after parsing the input model, an initial step applies a sequence of transformations including a model-to-model transformation that translates the input model to an *intermediate representation (IR)*, which abstracts from object-oriented programming languages and allows to describe structural aspects of the generated code. Second, different transformations can be applied to the IR to enrich the representation, e.g., by additional classes, methods, etc. that should be generated. Flexibility of this code generation approach is achieved by allowing to switch between both approaches, i.e., transform the IR or attach templates to IR elements. Third, the transformed IR including the attached templates is used as input for a template engine, which uses a default set of templates for a target language to generate code based on the IR. The default templates are used whenever there is no template attached to an element of the IR. Integrating transformation- and template-based code generation and employing an IR allows for target language independent transformations with additional target language specific templates. The contribution of this paper are as follows:

- Flexible integration of transformation- and template-based code generation
- Proposal for a object-oriented domain-specific language for an IR
- Customization approach for code generation via template attachments

Hence, we first give an example in Section 2 to demonstrate the challenge in MDD of data-centric applications. Afterwards, we form an understanding of a code generator and template- and transformation-based code generation in Section 3. Then, we present and discuss our approach for code generation using templates and transformation in combination (Section 4). Next, we present a use case, where we applied our approach in Section 5. Finally, an overview of current research in this field is given in Section 6 and the paper is concluded in Section 7.

2 MOTIVATING EXAMPLE

As a motivating example, we consider the MDD of data-centric applications. Each data-centric application offers management functionality for structured and consistent information described by a model [28]. On the left-hand side

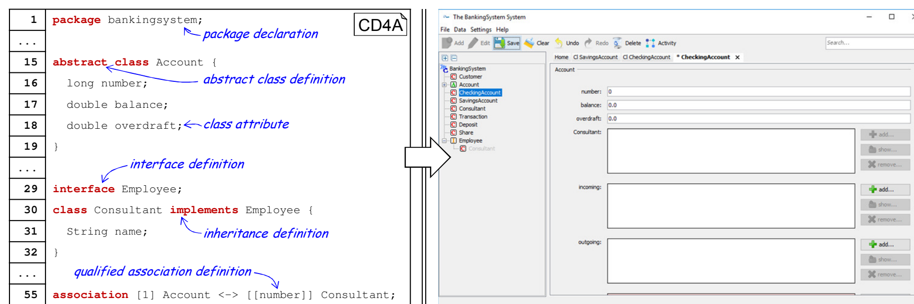

Fig. 1. Generation of Data-Centric Applications from Analysis Models.

in Figure 1, the model is an instance of the class diagram for analysis (CD4A) domain-specific language (DSL) to describe class diagrams created during the analysis phase (*analysis model* [31]). This model is systematically transformed to the executable data-centric application, which offers SCRUD (search, create, read, update, and delete) functionality shown on the right-hand side (Figure 1).

Due to varying user requirements, a data centric application has to be customizable. In this example, the developer has to change the user interface generated for each class in the input model. Since no additional models are used and adapting the generated source code in not practical, because it requires a suitable approach for handwritten code integration (cf. [17, 18]) and customization of each generated artifact, the developer aims to design a customizable code generator.

In addition, the developer aims to reduce the development time of future code generators by reusing parts of the data-centric code generator, e.g., the part to generate the source code representing the data structure (i.e., Plain-Old-Java-Objects). However, a code generator my use a different input and output model (i.e., conforming to a different DSL). Hence, to make parts of the code generator reusable, the developer plans to realize this part of the code generator independent of the input and target language.

Besides customization and reuse concerns, the developer needs to choose a code generation approach. However, generating data-centric applications has multiple challenges. First, generating an application core requires the code generation to traverse the representation of the model and generation of mainly changing code. Second, generation of a graphical user interface requires the generation of mainly static source code and from dedicated parts of the input models, e.g., classes. Hence, the developer wants to mix both code generation approaches to avoid unmaintainable templates and complex transformations.

3 TRANSFORMATION- AND TEMPLATE-BASED CODE GENERATION

In model-driven development a software system that produces an implementation from a higher-level description of a (part of a) software is regarded as a generator [6]. A code generator is a special kind of a generator that creates an implementation in a programming language from a set of input artifacts, which are, typically, models. Such code generators that always terminate and generate at least one output artifact are build on top of existing compilers for programming languages and consists of a front-end (*language processing*) and a back-end (*code generation*) [28].

Language processing is concerned with parsing the models, checking language constraints and creating an internal representation (*abstract syntax tree* and *symbol table*). Code generation systematically transforms the internal representation to concrete code, which is stored in generated artifacts. The most dominating technical realizations of code generation are *template-based* and *transformation-based* code generation. Subsequently, both approaches are explained in more detail.

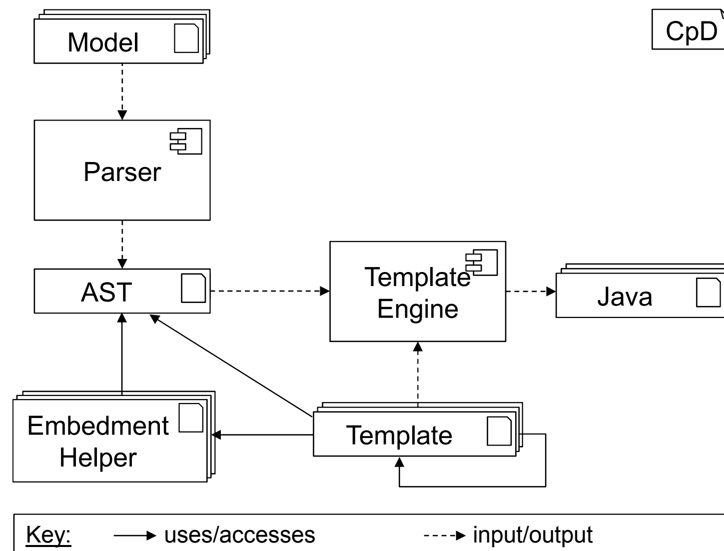

Fig. 2. Overview of a template-based generation process.

3.1 Template-based Code Generation

The prerequisite for template-based code generation is a template language and a corresponding template engine. A template forms the main artifact of interest

and consists of plain *target language code*, which is the source code that is generated, and additional template language instructions. Each template is processed by a template engine, which evaluates the template language instructions and prints the plain target language code and the evaluation result into an output file. As shown in the overview in Figure 2, after parsing the input model, the abstract syntax tree (AST) (and possibly a symbol table) and a set of templates is passed to the template engine, which systematically traverses the AST and calls a template for each particular AST element type. Additionally, embedment helpers can be used to outsource complex computations from templates. After the template is evaluated, the resulting plain source code, e.g., Java, is written into an output file.

This approach is simple and easy to use as it allows to directly write target language code with additional template instructions whenever a value needs to be computed from the given AST of the input model, e.g., the name of the AST element. While this provides a comfortable way for code generator developers to write generators, it has several disadvantages: (a) templates become complex, hardly readable, and, in consequence, hardly maintainable; (b) an additional infrastructure is necessary to handle complexity of templates (see embedment helpers); and (c) no static checking of the generated source code before writing into a file is possible because no internal representation exists.

3.2 Transformation-based Code Generation

In transformation-based code generation, transformations are the central artifact. In this approach, an input model is parsed and an arbitrary number of transformations is applied to the AST of the input model to create an output model [20], i.e., an abstract syntax tree of the target language (cf. Figure 3). Note that transformations can also be applied to the output model AST. Finally, the output model AST is systematically transferred into a textual representation, which conforms to the syntax of the output language, by using a pretty printer.

Transformations in this process are a sequence of endogenous transformations modifying the input model and conform to the source modeling language [7] followed by an exogenous transformation translating the input model to an IR-AST. Within the generation process a structured representation is present that can be checked for errors and processed by further transformations.

This approach has the advantage that calculations on and traversal of the input model can be replaced by pattern matching. When domain-specific transformations [1, 30, 34] are employed, the concrete syntax of the output language can be used to describe the output [20]. In addition, the AST representation allows for syntactic checks and extensibility of the generation by applying further transformations before transferring the result into a code. This benefit implies (a) the need for a grammar or meta model of the output language that precisely defines its structure. Furthermore, (b) transformations my become to complex and unmaintainable demanding for transformation development guidelines and additional frameworks to compose transformations. Large plain target language

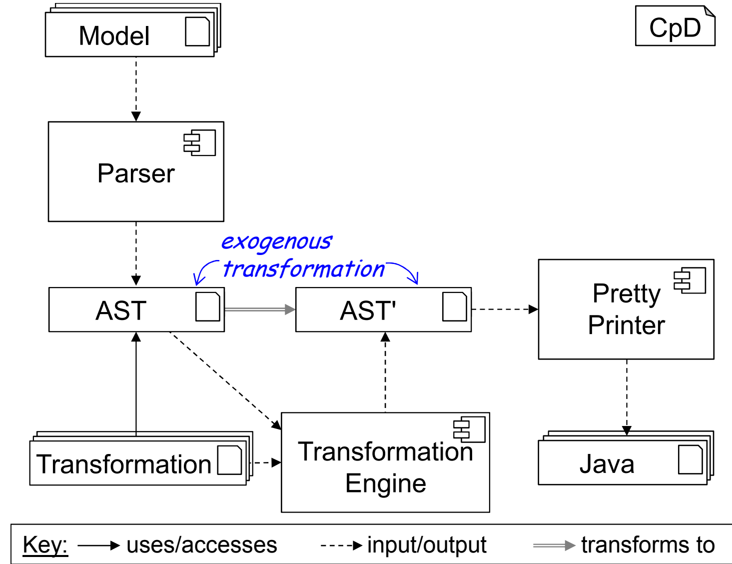

Fig. 3. Overview of transformation-based generation.

code fragments for single model element can more easily be created with templates [26].

4 SYNERGETIC TRANSFORMATION- AND TEMPLATE-BASED CODE GENERATION

To support reusable, customizable, and flexible code generator design, we propose an approach based on the conjunction of transformation- and template-based code generation. As both approaches have their advantages and disadvantages (cf. Section 3), we have combined both in a way that retains their advantages and minimizes their disadvantages.

An overview of a generation process is shown in Figure 4. The integration of transformation- and template-based code generation results in a partitioning of the overall code generation process into three steps (numbers in the figure).

Preprocessing. First, after the parser has created the abstract syntax tree of the input model ((1) in Figure 4), it is transforms into an abstract syntax tree of an IR, which may either be completely built from scratch or may be based on an existing AST that is adapted. In this step, the input model AST may be manipulated by transformations before it is eventually transformed to the IR.

Transforming. In the second step, the IR-AST is consecutively transformed by a sequence of transformations to enrich the intermediate AST with elements that will be generated ((2) in Figure 4). Each transformation adds, removes, or

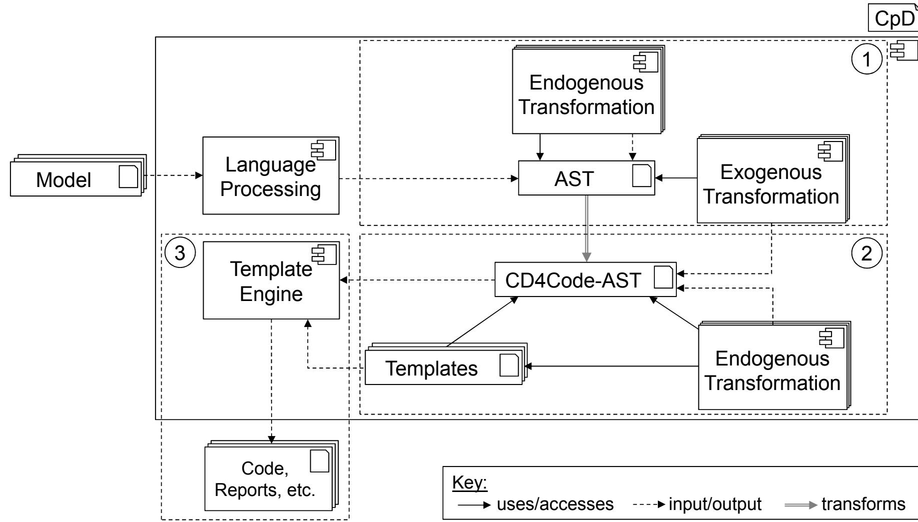

Fig. 4. Integration of template- and transformation-based code generation consists of three consecutive steps: *pretransformation, intermediate transformation,* and *template-based code generation.*

updates elements in the AST and may also require the input model AST, which is created after parsing, to perform its manipulations. Since the IR is independent of the output language, additional output language specific templates can be attached to elements of the intermediate AST.

Generating. Finally, in the last step (3 in Figure 4) a template engine is used to generate code by passing the transformed AST including the attached templates and a set of default templates for the output language and the AST of the IR.

4.1 Intermediate Representation

As the IR forms the core for the transformations and the abstract representation of the generated code, it should provide an abstraction of the output language but should also be as input language independent as possible. As most software systems are implemented using object-oriented programming languages, we decided to choose a representation that provides an abstraction of the most common object-oriented programming concepts. As a consequence, this approach is limited to generate source code that use the object-oriented programming paradigm [11].

A natural choice to describe an IR is a DSL representing UML class diagrams [37], which provide most of the concepts found in object-oriented languages. Thus, our IR is a restricted version of UML class diagrams called *class diagram for design (CD4D)*. It only contains the most relevant parts including classes, interfaces, abstract classes and enumerations. A class may extend another class

and implement interfaces. In addition, associations with navigation directions and cardinalities are supported. An association as well as each of its role ends may have names and can be ordered or qualified. Classes may have attributes with associated types. From our choice of the IR, a second limitation to our approach emerges: the IR can only describe structural aspects of the generated code. Behavioral aspects have to be added using additional language specific templates.

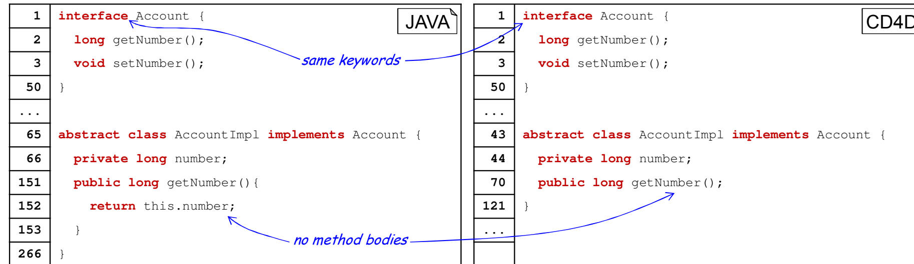

Fig. 5. Example of the IR (right-hand-side) for Java code (left-hand-side).

An example that can be described with the IR is shown in Figure 5. Note that we use concrete syntax rather than the abstract syntax for presentational purpose. The example shows an excerpt of the Java source code (left-hand-side) that is generated for the CD4A model shown on the left-hand side in Figure 1. It shows the interface `Account` (ll.1-50) and the implementation `AccountImpl` (ll.62-266). On the right-hand side, the textual notation of the CD4D DSL describing the object-oriented structure of the Java source code on the left-hand side. The CD4D model uses the same keywords as the Java source code to denote the corresponding UML class diagram concepts (inheritance, classes, interfaces, etc.). It also contains visibility, e.g., `private` in l.44 and `public` in l.10 on the right-hand-side. However, no method bodies are present in the IR (l.70 right-hand side).

The IR serves as the abstract representation of the source code that is to be generated. Model-to-model transformations successively enriched the IR with relevant information regarding the generated code, e.g. additional technical classes or methods. However, to make use of this language independent representation, the input model needs to be mapped to the IR.

4.2 Model-To-Model Transformations

The first two steps, i.e., the transformational part, of our approach illustrated in Figure 4, rely on model-to-model transformations [8]. Two types of transformation can be distinguished within our approach, *endogenous* and *exogenous*

transformation. After the model has been parsed and, thus, is available in its AST representation our approach allows to manipulate the model *endogenously*, i.e., apply transformations that change the model but do not change the language it belongs to. This initial transformations are helpful to reduce the high-level concepts used in the model, i.e., normalizing [24] the input model, such that the model only uses the core concepts of the source language. By first normalizing the input model the subsequent transformations are simplified as only the core concepts of the source language needs to be considered. By allowing normalization first our approach is robust regarding newly introduced syntactic sugar as this can be handled by adding further normalizing transformations.

After the input model is normalized an *exogenous* transformation translates the input model to the IR. This transformation maps concepts of the input language to concepts of the IR. Even though depicted as a single transformation there is no need to construct the complete IR in one huge step. Our approach allows to further transform the IR endogenously. Thus, this step just needs to do minimal mapping of the input model to the internal representation.

Finally, the transformational part of our approach allows to stepwise extend the created IR and attach templates to elements of the internal representation as explained in the next section. Within this transformation the "final" IR is constructed that serves as the basis for the template-based part of our code generation. To ease the construction of the "final" IR each transformation step is allowed to add, remove and update model elements as well as attach one or several templates to specific elements and can even consider a former state of the input model, e.g. the AST after the model has been parsed. The code generator developer is free to add as much information as she likes to the IR using transformations and can even have recourse to results of arbitrary former transformation steps. The final IR AST used for the template-based code generation does not necessarily has to be too detailed about the generated structure, because the templates can add missing details. However, by switching to template-based code generation as the back-end of our approach there is no need for a pretty printer for the output language. Furthermore, switching to templates provides a simple way to merge the information stored in the IR, e.g. merge different parts of a class or method generated by different templates without relying on output language concepts such as partial classes that are present in C# but missing in Java.

4.3 Template Attachment

Transformations on the IR allow to manipulate generic object-oriented programming concepts, e.g. classes, attributes, and are target language independent. However, in case target language specific code is needed a purely transformation-based approach requires an abstract representation of the target language. To avoid this and address customizability concerns (cf. Section 2), we allow to attach templates to IR elements.

For instance, assume a transformation added a new method and the method body should be defined. Rather than creating a new method body by using an

abstract representation and attaching it to the IR (which would make the IR target language dependent), we attach a template for the new method. This template contains the method implementation and may use the IR to access values, e.g., the method name and parameters. To efficiently manage the template attachments, we use an infrastructure that maps a template to a particular AST element. It is, therefore, possible to attach multiple templates to one AST element as shown in Figure 4 or even to attach one template to multiple AST elements.

Besides attaching one or multiple templates, it is also possible to define the order in which templates of a AST element are executed in the code generation phase. The order of template execution is evaluated for each individual IR-AST node. To modify template execution, the following operations are provided:

- **Replace Operation:** to replace a particular template from the template extension.
- **Add-Before Operation:** to add a template at the beginning of the template extension.
- **Add-After Operation:** to append a template at the end of the template extension.

If multiple templates are marked to be executed first, last, or before a template, the template registered first is executed. The operations for template attachments may result in conflicting operations. For example, consider two replace operations for the same template; or a cyclic replacement. Hence, conflicts are handled as shown in Figure 6.

Cyclic add and replace operations, i.e., (a) and (b) in Figure 6, are resolved by avoiding a transitive resolution of these operations and only resolve to depth one, i.e., only one operation. For example, in (a) the first operation (execution order is from top to bottom) is the replacement of template A with template B. As a result, only the first operation is executed. Another example is shown in (b) (execution order is from left to right and top to bottom), where only the replacement of template A with template B is executed. Note that if a template is replaced, all before and after templates of the replacing template are added.

However, operations on template attachments are executed sequentially, thus run-time errors may occur. For example, when two *replace operations* sequentially replace the same template, i.e., (c) in Figure 6, the latter replace operation will fail, because the template has previously been replaced. In aspect-oriented programming this is considered as the *"fragile point cut"* problem [32]. An approach to detect such errors is to process and analyze the sequence of template attachment operations a priori to detect conflicts. However, this will involve processing the complete code generator's source code. Alternatively, another approach is to detect such conflicts at run-time of the code generator and inform the developer but do not stop code generation.

The strong bond between templates and the IR AST demands for handling template attachments, when the IR AST is transformed. For example, assume a template `tmpl` is attached to the class `C` in the IR, then another transformation

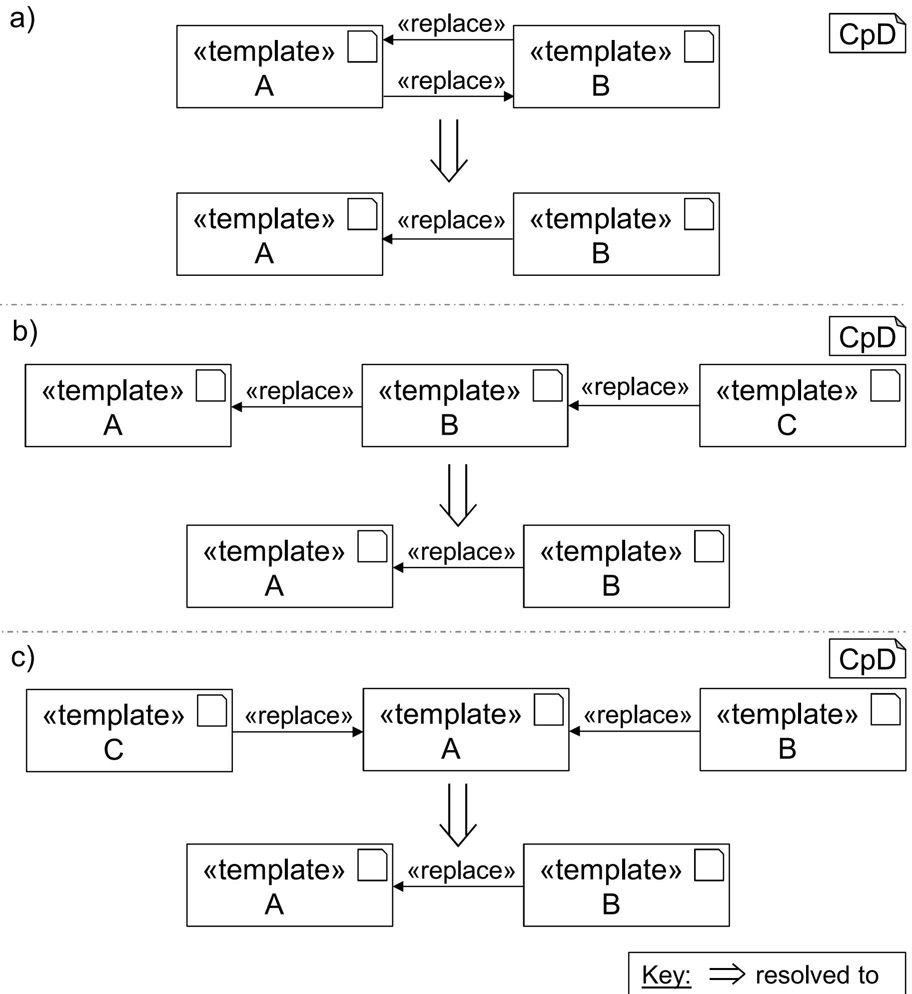

Fig. 6. Conflict resolution is done by only executing the first replace operation (a,c) in a non-transitive manner (b).

changes `C` and attaches another template `tmpl2`. With one exception - AST element deletion -, the result of this will be that the AST element will have both templates attached `tmpl` and `tmpl2`. In the case, when the AST element is deleted, we also delete the attachment.

4.4 Generating Code

After all transformations have successfully been applied and templates have been attached to AST elements, the resulting IR needs to be mapped to concrete code.

The essential requirements for such a mapping is a clear understanding of how to map each element of the IR to the output language source code and a set of default templates for the output language that realizes this mapping.

As the main goal of the IR was to abstract from specific output language concepts and provide only relevant object-oriented programming concepts, mapping the IR to an object-oriented language is straight forward. This means classes, interfaces, enumerations, and attributes can easily be mapped to object-oriented programming languages. Association and composition are the only two concepts with varying semantics, since different approaches have been proposed on how to perform this mapping, e.g. [15, 14]. However, the most simplest approach is to add a variable storing the associations links to the source class of the association and add corresponding mutators and accessors for each association direction. This simple mapping is realizable by every object-oriented programming language.

Having a mapping of the IR AST to an output language, a set of templates that realizes this mapping is needed. Indeed, this has to be done for each output language that is to be generated. However, it allows for exchangability of the output language, as the IR AST structure is not changed. However, it needs to be considered that attached templates are written in the output language and need to be exchanged as well.

As shown in Figure 4, the template engine is called with the transformed IR AST and the default set of templates. The template engine traverses the input AST and for each visited AST node the attached templates are called. If no template is registered for an AST element, then a default template for this type of AST element is used from the default template set. The output is either written to a file or returned as a result to be embedded in another output.

4.5 DISCUSSION

Since this approach merges templates and transformation it needs to be discussed in which cases to use templates and in which cases to use transformations. Certainly, this cannot be answered in general. However, in current literature [26] it is proposed to use templates for code that does not require much computation and is primarily static. For instance, graphical user interface code calls methods from libraries. In contrast, transformations should be used whenever static checking is required for certain elements. For example, by adding a class to the IR, certain properties can be checked. Moreover, changes that affect the overall code generation, e.g., name of classes, and, thus, require changes that regard different output files should be realized as transformations.

A disadvantage of the simplified IR is the lack of output language specific concepts, e.g. Java annotations. While for most cases it is sufficient to neglect them (e.g. `@Override`), they are of essential importance when generating e.g. J2EE applications. However, the IR can be extended with additional stereotypes. Each stereotype can then be used to define output language specific annotations. Besides stereotypes, other output language specific concepts that cannot be represented using the IR. In this case, either the IR has to be extended or already

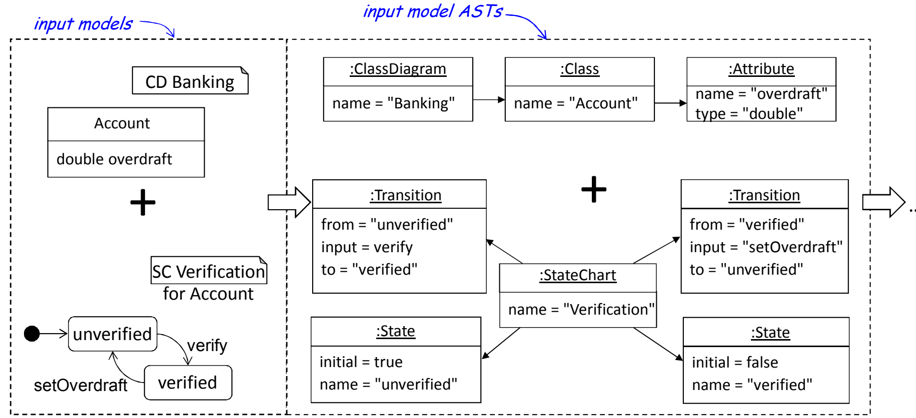

Fig. 7. Graphical notation of input models and corresponding abstract syntax (continued in Figure 8).

existing concepts of the IR have to be reused. Perhaps, output language specific templates can be used to resolve this issue.

As code generators are software systems on their own, they have to address issues such as maintainability and complexity as well. In our approach we do not particularly address these issues. Instead, we provide an approach to structure the code generation as a whole. We try to tackle these challenges by providing signatures for templates, i.e., templates have signatures such as methods and can simply be invoked with different arguments. In addition, guidelines have to be designed in order to prevent transformations and templates to become too complex.

Another challenge that has to be addressed results from the operations on template attachments, which can introduce syntax errors in the generated source code. A typical cause for such syntax errors is the use of incomplete target language statements in templates that combined with other templates produce syntactically correct source code but if these templates are executed in the wrong order or are replaced, the resulting source code may contain syntax errors. A restriction of templates to only syntactically correct target language statements has been proposed in [40, 41] to address this issue. However, this approach is not of practical use because it restricts the flexibility of the proposed code generation approach. Hence, the proposed approach is not restrictive and demands for a methodological approach to handle this challenge.

5 USE CASE

To demonstrate the proposed approach, we provide an example that extends the motivating example from Section 2 with Statecharts. In Figure 7, two different representations of the input models are shown. On the left-hand side is the

graphical notation and on right-hand side is the condensed parsed form as an AST. The class diagram provides the static part of a data-centric application, i.e., the class `Account` with one single attribute `overdraft` (condensed model from Section 2). This attribute stores the maximum possible amount of overdraft for a specific account.

The presented application in Figure 1 provides functionality to load and store data. However, this data is not validated. We use the flexibility of the presented approach to extend this with behavioral aspects using Statecharts. It is important to verify all changes on the overdraft attribute, thus we use a Statechart to keep track if the current overdraft is verified (by an internal process of the bank).

The Statechart `Verification` is bound to the class `Account` using the `for Account` tag. It defines two states for the corresponding class `unverfied` and `verified`, meaning each instance of `Account` gets it's own associated state. The state `unverified` is the initial state. Each new account needs verification of the overdraft. Two stimuli are defined which handle the transitions between states during lifetime of an `Account`. The internal verification process is encapsulated in the stimulus `verify`. The process is designed in such a way, that it always ends with a verified account, but may include adjustment of overdraft. Every time the overdraft changes the state switches to `unverified`.

Both input models are parsed (separately) into independent ASTs as shown in Figure 7. Parsing of the models and creation of the AST is easily possible with the MontiCore Language Workbench [22]. The AST of the `Banking` class diagram consists of the class `Account` and has the attribute `overdraft` of type `double` attached. The Statechart AST consists of the two states `verified` and `unverfied` and two connecting transitions `verify` and `setOverdraft`. Since no initial transformations have to be performed (but could be if e.g. the Statechart would have hierarchical states), the next step is to build up the IR from both ASTs, as shown in Figure 8 (cf. Section 4).

FreeMarker

```
1 public class ${ast.getName()}{
2  <#list ast.attributes as attr>
3   ${include("PrivateAttributeDecl")}
4  </#list>
5  //...
6  <#list ast.methods as method>
7   ${include("EmptyMethodBody")}
8  </#list>
9 }
```

Listing 1. Default Template to generate a Java Class.

As we are aiming at an object-oriented target programming language, we use the AST of the *CD4D* DSL (cf. Section 4.1). The root element `CDDefinition`

(left-hand-side in Figure 8) bundles all classes and associations. The input class `Account` is directly transformed in the corresponding class element `CDClass` from *CD4D*. Attached is the `overdraft` attribute that was modeled in the input. Additionally, the Statechart AST is transformed into three classes implementing the Statechart pattern [12] and an association. The class `VerificationState` is the abstract super class for the two states, which are implemented by the classes `Unverified` and `Verified`. The Statechart `Verification` is bound to the `Account` class and thus an association connecting each `Account`. Its actual state is the class `VerificationState`, which is attached to the `CDDefinition` root element. Note that the *SC* tag in the IR-AST denotes the elements created from the Statechart input model, whereas the *CD* tag denotes the elements created form the class diagram input model.

In a next step, the IR is transformed by endogenous transformations to lift the stimuli into the `Account` class by creating a method for each stimulus: `verify` and `setOverdraft` (right-hand-side in Figure 8). Via template attachment, the corresponding method body is attached as shown for the `verify` method. For the method `setOverdraft`, we attached (a) setter functionality from the input class diagram and (b) the Statechart transition handling. Further transformations on the IR can add persistence functionality, builders for all classes, or a GUI.

Finally, in the last step, the IR needs to be mapped to Java code. A default set of templates is required to perform this mapping. Having such a default set, code is generated by traversing the IR-AST and calling a template, which is either attached or defined as a default template for this particular type of AST elements. An example of a default template to generate a Java class from the IR is shown in Listing 1. It shows that classes of the IR can directly be mapped to Java classes. It also defines that for all attributes, which will become global variables in the Java source code, the `PrivateAttributeDecl` template is called if no attachment is defined. For methods, the `EmptyMethodBody` template is used. With respect to the above example, for the `verify` method, this default template is neglected and the attached template is called.

This use case shows two of the main benefits of the approach: (a) generating Java code from the IR is straight forward, and (b) default templates are small and maintainable (the generator consists of 8 templates only to generate arbitrary Java classes).

6 RELATED WORK

Code generation as an integration of transformations and templates, is based on several approaches that have been proposed in order to improve code generation. Additionally, there are approaches used in reverse engineering that involve transformations and code generation such as [3]. However, their main focus is on model extraction instead of a flexible combination of transformation- and template-based code generation.

An approach to integrate template-based code generation into graphical model transformations has been proposed in [16]. It allows to graphically model

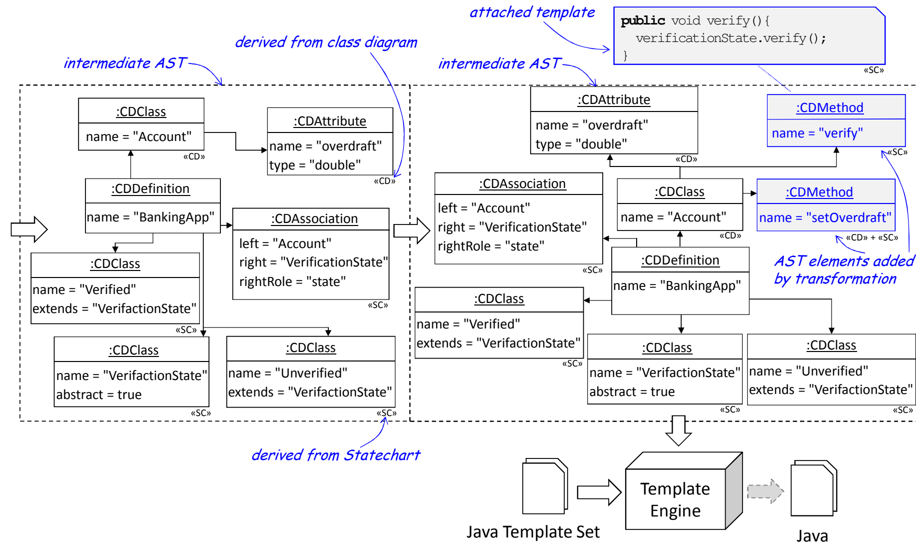

Fig. 8. Constructing an IR from the input models (left-hand-side) and attaching templates to methods (right-hand-side).

endogenous transformations and use a template language to generate strings to define method bodies. A similar approach has been presented in [2]. As opposed to these approaches, our approach is broader as it allows for endogenous and exogenous transformations by employing an IR. Moreover, our approach regards templates as the primary artifact to generate concrete source code. Templates may not only be used to generate method bodies but also complete target language constructs such as classes with methods.

A fully transformation-based approach that systematically transforms the input model into concrete source code has been proposed in [20]. It is based on a meta-model of the target language that is extended with additional concepts to allow merging the generated code, e.g. partial classes. By using transformations the target language meta-model is consecutively transformed and the result is pretty printed to an target file. To explain practical use of code generation as model transformations, the approach has been evaluated for generating code from sequence diagrams [23]. Another transformation-based approach for a Java-based IR has been proposed by [10]. Their IR is enriched with EJB specific concepts to generate Java EJB applications. After applying all mode-to-model transformations, a model-to-text transformation pretty prints the target into a file. In contrast, our approach does not require a meta-model of the target language but uses an intermediate representation instead to be target language independent. Moreover, all templates attached to the AST are not regarded by the transformations, which allows templates to neglect the resulting code.

In [13] an approach has been presented that transforms the input model to a token tree first and, afterwards, to a search tree to, finally, use templates to generate code. In contrast to our proposed approach, this approach does not make use of one intermediate representation but instead for each model a different token and search tree is created. Moreover, only one template is attached to the root element of the tree. It then handles all children. Our approach allows to attach multiple templates to every AST element of the IR.

Another approach that separates transformations and template-based code generation has been described by the openArchitectureWare system [19], which is now part of the Xtext project [38, 5]. It proposes a workflow that involves a transformation step prior to the code generation step. The basic idea of our approach is similar but we extend it to allow a set of default templates, an intermediate representation, and allow template attachments to instances of AST elements rather then types of AST elements.

The Clearwater code generation approach uses an XML-based IR [33]. Although the IR is not restricted to an input or a target language, because it accepts arbitrary new tags, this IR hampers well-formedness checking due to its XML basis.

In [27], an IR for template-based code generation to separate the input model and the target source code from the code generation process and achieve flexibility has been proposed. However, the IR is restricted to class diagrams as input and mixes AST and additional symbol related information.

Additionally, a meta-model transformation-based approach, which uses the target language's meta-model as an IR, has been proposed in [20]. A similar approach based on a Java IR with additional EJB extensions to generate Java EJB applications is described in [10]. Both meta-model-based approaches use target language meta-models, which makes them target language dependent and reduces developers flexibility because it contains every detail of the generated code.

7 CONCLUSION

Code generation in model-driven development targets generation of source code from abstract models. Even though approaches exist for performing this transformation, they mainly focus on code generation and neglect aspects as flexibility, reusability, and maintainability. In the introduction, we have therefore discussed the main principles necessary for a good code generator.

Afterwards we have presented an approach that integrates template- and transformation-based code generation to address those principles. As a result, a user of this approach is able to flexibly choose how much of the generation process is done by transformations and at which point templates are better suited for the remaining generation process. For this purpose, we separated the code generation process into three phases. First, as a preparation the abstract syntax tree can be normalized and once this is done, be mapped to an intermediate representation (IR). It represents an abstraction from object-oriented programming languages

and forms a lightweight version of UML class diagrams. With this abstraction the whole code generation is output language independent. In the second phase, the IR can be refined and output language specific templates can be added to add detailed output dependent information. In the last phase, we employ a template engine and a set of templates for an output language to generate concrete source code. The template engine traverses the IR AST and calls the attached templates. If no template is attached, a default template defined for the particular AST type is used.

It is, however, difficult to decide without much empirical evidence, whether this form of combined code generation will be appropriate to get more every-day developers and especially agile developers towards using abstract models and code generators, instead of handcoding. However, in our projects, a number of them are industry relevant, we experienced that this form of code generation is really of help and improves the overall productivity. However, we are somewhat biased and thus don't count as empirical evidence. It would be nice, if others can comment on these principles and the presented approach, to get more knowledge about what does work well and what needs to be changed.

Bibliography

[1] Baar, T., Whittle, J.: On the Usage of Concrete Syntax in Model Transformation Rules. In: 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics. Springer-Verlag (2007)

[2] Balogh, A., Varró, D.: Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In: ACM Symposium on Applied Computing. ACM (2006)

[3] Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: A model driven reverse engineering framework. Information and Software Technology 56(8) (2014)

[4] Chared, Z., Tyszberowicz, S.S.: Projective Template-Based Code Generation. In: CAiSE'13 Forum at the 25th International Conference on Advanced Information Systems Engineering. vol. 998. CEURS-WS.org (2013)

[5] ekkes corner: Mobile && iot. `https://ekkescorner.wordpress.com/2009/07/16/galileo-openarchitectureware-moved-to-eclipse-modeling-projects-oaw5/` (October 2015)

[6] Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

[7] Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA (2003)

[8] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3) (2006)

[9] Di Ruscio, D., Eramo, R., Pierantonio, A.: Model Transformations. In: Formal Methods for Model-Driven Engineering, LNCS, vol. 7320. Springer Berlin Heidelberg (2012)

[10] El Beggar, O., Bousetta, B., Gadi, T.: Automatic code generation by model transformation from sequence diagram of system's internal behavior. International Journal of Computer and Information Technology 1(02) (2012)

[11] Eliens, A.: Principles of Object-Oriented Software Development. Addison-Wesley Longman Publishing Co. Inc. (1994)

[12] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)

[13] Geiger, L., Schneider, C., Reckord, C.: Template- and modelbased code generation for MDA-tools. Tech. rep. (2005)

[14] Génova, G., Del Castillo, C.R., Llorens, J.: Mapping UML Associations into Java Code. Journal of Object Technology 2(5) (2003)

[15] Gessenharter, D.: Implementing UML Associations in Java: A Slim Code Pattern for a Complex Modeling Concept. In: Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL '09). ACM (2009)

[16] Girschick, M.: Integrating Template-Based Code Generation into Graphical Model Transformation. In: Modellierung 2008. Berlin (2008)

[17] Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., Navarro Pérez, A., Plotnikov, D., Reiss, D., Roth, A., Rumpe, B., Schindler, M., Wortmann, A.: A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. CoRR abs/1509.04498 (2015)

[18] Greifenberg, T., Hölldobler, K., Kolassa, C., Look, M., Mir Seyed Nazari, P., Müller, K., Navarro Pérez, A., Plotnikov, D., Reiss, D., Roth, A., Rumpe, B., Schindler, M., Wortmann, A.: Model-Driven Engineering and Software Development: Third International Conference, chap. Integration of Handwritten and Generated Object-Oriented Code. Springer International Publishing (2015)

[19] Haase, A., Völter, M., Efftinge, S., Kolb, B.: Introduction to openarchitectureware 4.1. 2. In: MDD Tool Implementers Forum at TOOLS Europe. `http://www.dsmforum.org/events/mdd-tif07/oAW.pdf` (2007)

[20] Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: a case study in transformation modularity. Software & Systems Modeling 9(3) (2010)

[21] Krahn, H., Rumpe, B., Völkel, S.: Monticore: Modular development of textual domain specific languages. In: Proceedings of Tools Europe (2008)

[22] Krahn, H., Rumpe, B., Völkel, S.: Monticore: a framework for compositional development of domain specific languages. In: International Journal on Software Tools for Technology Transfer (STTT). vol. 12, pp. 353 – 372 (2010)

[23] Kundu, D., Samanta, D., Mall, R.: Automatic code generation from unified modelling language sequence diagrams. Software, IET 7(1) (2013)

[24] Mens, T., Czarnecki, K., Gorp, P.V.: A Taxonomy of Model Transformations. In: Language Engineering for Model-Driven Software Development. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI) (2005)

[25] Mir Seyed Nazari, P., Roth, A., Rumpe, B.: Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In: Proceedings of the Workshop on Domain-Specific Modeling. ACM (2015)

[26] Mohan, R., Kulkarni, V.: Model Driven Development of Graphical User Interfaces for Enterprise Business Applications - Experience, Lessons Learnt and a Way Forward. In: Model Driven Engineering Languages and Systems, LNCS, vol. 5795. Springer Berlin Heidelberg (2009)

[27] Reiß, D.: Modellgetriebene generative Entwicklung von Web-Informationssystemen. Ph.D. thesis, RWTH Aachen University, Aachen (2015)

[28] Roth, A., Rumpe, B.: Towards Product Lining Model-Driven Development Code Generators. In: 3rd International Conference on Model-Driven Engineering and Software Development. Springer International Publishing (2015)

[29] Rumpe, B.: Modeling with UML. Springer (2016)

[30] Rumpe, B., Weisemöller, I.: A Domain Specific Transformation Language. In: Workshop on Models and Evolution. vol. 11 (2011)

[31] Rumpe, B.: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer International (2017)

[32] Störzer, M., Koppen, C.: PCDiff: Attacking the Fragile Pointcut Problem. In: European Interactive Workshop on Aspects in Software (2004)

[33] Swint, G.S., Pu, C., Jung, G., Yan, W., Koh, Y., Wu, Q., Consel, C., Sahai, A., Moriyama, K.: Clearwater: Extensible, Flexible, Modular Code Generation. In: 20th IEEE/ACM international Conference on Automated software engineering. ACM (2005)

[34] Visser, E.: Meta-programming with Concrete Object Syntax. In: Generative Programming and Component Engineering, LNCS, vol. 2487. Springer Berlin Heidelberg (2002)

[35] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L.C.L., Visser, E., Wachsmuth, G.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)

[36] Wachsmuth, G.: A Formal Way from Text to Code Templates. In: Fundamental Approaches to Software Engineering, LNCS, vol. 5503. Springer Berlin Heidelberg (2009)
[37] www: OMG UML Specification. `http://www.omg.org/spec/UML/2.5/` (October 2015)
[38] www: openarchitectureware. `https://web.archive.org/web/20140225123932/http://www.openarchitectureware.org/index.php` (October 2015)
[39] www: Apache maven project. `https://maven.apache.org/` (August 2017)
[40] Zschaler, S., Rashid, A.: Symmetric Language-Aware Aspects for Modular Code Generators. Tech. Rep. TR-11-01, King's College, Department of Informatics (2011)
[41] Zschaler, S., Rashid, A.: Towards modular code generators using symmetric language-aware aspects. In: Proceedings of the 1st International Workshop on Free Composition. pp. 6:1–6:5. FREECO '11, ACM, New York, NY, USA (2011)