Towards Roundtrip Engineering - A template-based Reverse Engineering approach

Manuel Bork, Leif Geiger, Christian Schneider, and Albert Zündorf

Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, D-34121 Kassel, Germany

Abstract. Model driven development suggests to make models the main artifact in software development. To get executable models in most cases code generation to a "traditional" programming language like e.g. Java is used. To obtain customizable code generation template-based approaches are applied, commonly. So, to adapt the generated code to platform specific needs templates are modified by the user. After code generation, in real world application the generated code is often changed e.g. by refactorings. To keep the code and the model synchronous reverse engineering is needed. Many approaches use a Java parser and a mapping from the Java parse tree to the UML model for this task. This causes maintenance issues since every change to a template potentially results in a change to this parse tree - model mapping. To tackle this maintenance problem our solution does not use a common language parser but uses the templates as a grammar to parse the generated code, instead. This way changes to the templates are automatically taken into account in the reverse engineering step. Our approach has been implemented and tested in the Fujaba CASE tool as a part of the model and template-based code generator CodeGen2 [11].

1 Introduction

Transforming models into various kinds of text languages is common practice, nowadays. These textual languages are used to provide an executable mapping for different kinds of models. Therefore such a model-to-text translation, to a programming or description language, is subject to optimization, adaption and maintenance work.

To obtain the required flexibility in the transformation engine, text templates are a common means to facilitate the final transformation from model-to-text, cf. e.g. [1, 2]. These templates can easily be tuned, adapted and maintained. Even users can edit templates to change the transformation results according to their needs.

Transforming models to text is not the only direction that is needed. There are still developers who edit source code directly, there are processes requiring people to change text artifacts, and sometimes pieces of generated text may have lost there corresponding models. Thus transformation from text to models are a requirement as well.

Common techniques for transforming text to models utilize a parser for the specific text language and operate on the parse tree afterwards. This approach is limited by the flexibility of the language parser (problems like syntax errors in the text document) and it easily causes a maintenance problem if the templates used for to-text-transformation are changed. In contrast to the template editing for the forward engineering, the reverse engineering - parsing of text - cannot be configured easily. To overcome these weaknesses our text-to-model transformation engine exploits the very same templates used for the model-to-text direction for reverse engineering.

The work presented here has mainly been done in the context of the bachelor and master thesis of Manuel Bork [6, 7].

2 Related Work

Template based text generation has established itself as a standard technique for web pages e.g. based on PHP or Java server pages. Consequently, the same technique is frequently used for model-to-text code generation for example in eclipse by Java emitter templates [2] or in Fujaba [10] by velocity [1].

For text-to-model or reverse engineering one commonly uses parsing technologies. This means, based on e.g. some Java grammar and e.g. a compiler compiler like javacc/jjtree [3, 4] one builds an abstract syntax tree and this tree is then transformed with a model-to-model transformation into the original model. In the Fujaba project we have been following this approach for several years, too [14, 17, 19, 16, 18, 20]. Especially, the thesis of Thomas Klein [13] created a first reverse engineering component with reasonable capabilities for Fujaba. However, due to several changes to our code generation strategies, e.g. for association implementation, we frequently had to update this reverse engineering component and after only one year, the functionality was lost. Another approach [20] tried to overcome the maintenance problems with fuzzy reasoning. They relaxed the exactness of the pattern matching process in order to deal with minor code variations. This of course had the drawback of false positives and false negatives.

Other reverse engineering approaches use fact extractors. Facts are tuples of entities (i.e. classes, variables, methods) and relations (i.e. inheritance, function calls, instantiations). A fact extractor for Java is introduced in [12].

In order to reverse engineer dynamic models other approaches are used. Briand et al. generate UML sequence diagrams by protocolling execution traces at runtime [8]. Rountev et al. however analyse directly the control flow in the source code [21].

With the Columbus framework [9] it is possible to reverse engineer even large C/C++ projects. It generates class diagrams, an abstract syntax tree and call graphs. Furthermore, it supports design pattern recognition. CPP2XMI [15] is based on Columbus and extracts UML class, sequence and activity diagrams.

Due to our knowledge, there is no other approach that exploits code generation templates for parsing directly. One may argue that template files form some kind of language grammar. However this template file based grammar will

most likely not fulfill the constraints of an LALR1 or LL1 grammar, cf. [5]. Thus one has to use more general techniques for context free grammars as e.g. the Cocke-Younger-Kasami (CYK) algorithm. However, our templates contain local variables, expressions, and control structures which again complicate matters.

3 Code generation

A template based code generation is a prerequisite of our new reverse engineering approach. This section gives an overview of the code generation software used for our prototype. The code generation process is split into three tasks, cf. Figure 1.

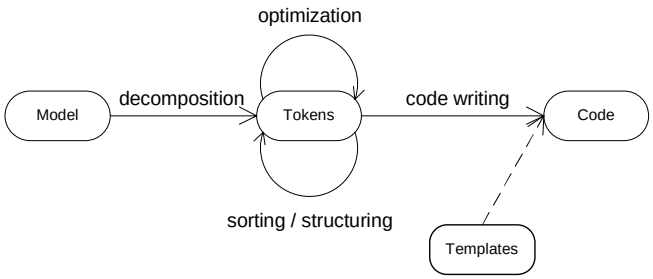

Fig. 1. Subtasks of the code generation with initial and resulting data

Our approach first transforms the original model into an intermediate token structure. This intermediate token structure defines a visiting order for model elements. This initial transformation step also handles a lot of conditional cases and alternatives, e.g. this step may choose a specific strategy for the implementation of associations. Note, one model element may create multiple tokens for different purposes, e.g. a model class may create a token structure for a Java interface class and another one for an implementation class. The result of the token creation task is a tree of tokens where the tokens may refer each other in several ways (thus forming a graph of tokens).

The generated token graph represents our intermediate language. The code generation for Fujaba's graph rewrite rules performs additional structuring, sorting and optimizing on this intermediate language. Tokens from class diagrams usually need little to no further structuring.

Note, in reverse engineering, the token structure is extended by temporary string attributes that hold references to model elements. These string references are then resolved to real model references in a separate step.

In code generation, the final step generates code for the resulting token graph. Therefore, the underlying token tree is visited in postorder. Every visited token is passed to a chain of code writers. Usually, the responsible code writer opens a

specific template file and passes the token and additional information as context to the template engine. This additional information includes the code generated for all children of the token in the tokens hierarchy.

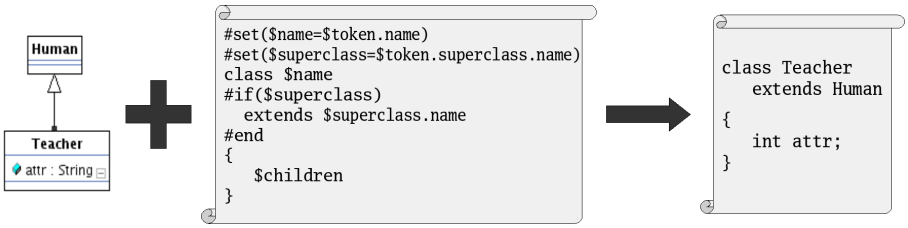

Fig. 2. Code generation example

In our implementation, we use the velocity template engine [1]. Figure 2 shows an example model and a simplified version of the template used for class tokens. Within the class template, the class token is stored in variable `$token`. From this variable, the template may access the original model via the token structure. For example, the first two lines of the template declare two local variables: `$name` and `$superclass`. The `$name` variable is read from the `name` attribute of the passed `$token` object. The `$superclass` variable gets its value via the access chain `$token.superclass.name`. Velocity allows attribute access or even method calls on every object passed to the template engine. That makes the templates highly customizable since every model element can be queried during code generation. In line 3 of the template finally code is generated: The constant string "class" followed by the current value of variable `$name` is produced as output. Velocity also allows control flow such as looping or branching. The code fragment in line 5 is only added to the output if the statement in line 4 evaluates to `true`. This means it is only added if the variable `$superclass` is not empty. After the opening bracket from line 7 the code for all subtokens (like methods or fields) is added to the class' code. That code is passed to the engine in the `$children` variable. The code for the class is finished by the closing bracket from line 9.

Note, although the Velocity template engine provides control structures itself, we do most of the complex computation during the construction and optimization of the intermediate token structure. This reduces the complexity of our templates dramatically and makes them reasonably simple.

4 Reverse engineering

It is common practice to use a parser which is based on the grammar of the target language to reverse engineer a piece of source code. But this common approach has several disadvantages: First, a separate parser for each language, which can be generated with the code generator, is needed. In contrast to that, it is sufficient to write a set of new templates for a template based code generator

to support a new target language (with similar structure). Second, the parse tree - the result from parsing a piece of source code with a common language parser - is very fine grained. It is quite tedious to map this parse tree information to an application meta model (e.g. UML). To accomplish this task model elements must be associated with more or less complex patterns, found in the parse tree. Afterwards a pattern recognition mechanism is used to identify pattern instances in the parse result and these matches are translated into model elements. As of the nature of the parse results these pattern need to be adapted according to the possible generated or hand written implementation style, found in the parsed source code.

In the template-based approach, presented in this paper, this mapping is still needed. Though, the result from the template-based parsing has a higher granularity and higher abstraction level than the parse tree from a common language parser. Thus the part of the pattern matching algorithms which is under major maintenance due to template- or code-style-changes is instead already covered in the template-based parsing.

Our text-to-model approach uses the very same templates that were used for code generation before. So, while adapting the templates for whatever reason one adapts the reverse engineering mechanism at the same time. This tackles the major part of the maintenance problem. Section 3 introduced our template based code generator. In a nutshell the application's model is transformed into an intermediate token tree and then passed to the template engine. The template engine generates code, specified in the the according template, afterwards. In Figure 2 an example for this procedure was given. For reverse engineering we invert this procedure. Figure 3 exemplifies the reversed procedure.

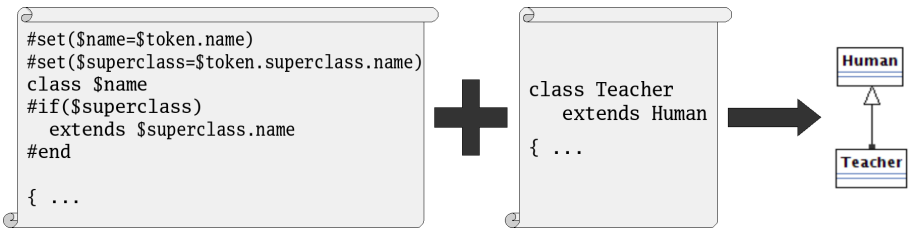

Fig. 3. Template and source code are parsed to reconstruct the model.

Our approach works as follows. Given a piece of source code, the code generator states which template is used first. Then, while traversing template and source code the template's variables are assigned with textual fragments of the source code so that the given source code would be generated again. These assignments are utilized later on to reconstruct the intermediate token layer and finally the model itself. As there are several possible solutions for this first step some reasoning is performed next. Thereby boolean conditions that consist of multiple variables are split up and contradictory solutions are removed. Both

tasks, template-based parsing and reasoning, are repeated for all included templates. After that the intermediate token layer is reconstructed generically. Then textual references are resolved to real objects and mapped to the model.

In the following we present each subtask in detail.

4.1 Template-based parsing

The goal of this subtask is to discover whether a given template has been used to generate the source code given and how the templates variables were assigned.

The left side of Figure 3 illustrates an excerpt of a template for a Java class. The right side shows two lines of source code. In order to reverse engineer this source code the values assigned to the template variables are inferred. So first of all the template itself must be parsed. Therefore the velocity template engine contains a template parser. This parser returns a parse tree that is traversed in the following step. To find the values in the example (i.e. for the classname `$name` and the superclass' name `$superclass`) the parser traverses the template and tries to match the terminals[1] of the template with text fragments from the source code, first. Thus, in the example, the parser starts at the beginning of the template and reads the terminal `class`. It finds the same text in the source code at the starting position, too. As this is a successful match, parsing process continues. After that the variable `$name` is processed. As the parser does not know, yet, which value is assigned to the variable, it just stores the variable name as pending. The next processed template part is a branching statement. As the parser does not know yet how to evaluate the branching condition (`$superclass`), it checks both possible paths through the template: a) assuming the condition to be true and b) assuming it to be false. So in case a) the parser tries to find the terminal `extends` in the source code. It is matched in the middle of the source code. It is now possible to assign a value to the pending variable: `$name` gets the assignment `Teacher`. After that the succeeding variable `$superclass` and the rest of the template is processed likewise.

By this manner the parser finds the following assignments for the variables from the template as a first possible solution: `$name` with `Teacher` and `$superclass` with `Human`. But one branch is still left to check. So the parser skips the complete branching statement in the other case (assuming `$superclass` to be false or empty) and tries to match the terminal {, which is found at the end of the source code snippet. So the parser can assign the previously read source code fragment `Teacher extends Human` to the variable `$name`. Obviously this assignment does not make much sense, but nevertheless it is a valid result concerning the parsing process. So this first subtask results in two possible solutions.

Trying all possible ways through the template is a very time consuming task. Additionally many possible assignments emerge, if a terminal occurs several times within a source code. For example, there are many opening braces in a piece of Java source code: This causes the same number of possible assignments for the first template variable, in the given example, as the number of opening

[1] text fragments which are not substituted by the template engine

braces in the source code. We address this problem by specifying constraints for the allowed values of variables. These constraints span from very simple constraints (i.e. "does not contain white spaces") to very expressive constraints (i.e. a complex regular expression). Commonly, most variables of a template cover only one single line of generated source code. Thus specifying a single line constraint as a default constraint for all variables of a template, helps decreasing the parsing time dramatically. In the example we would specify a constraint for both variables `$name` and `$superclass`, which denotes that the assigned value must not contain any whitespaces. This would discard the second solution.

In our implementation these constraints can be specified directly in the template. We introduced a simple comment syntax for this purpose. Figure 4 shows a constraint specification for the previous example.

```
## hints[ $name ] := SingleWordConstraint
## hints[ $superclass ] := RegularExpressionConstraint( "^[^\s]+$" )
```

Fig. 4. Example of a specification of constraints for variables of a template.

The constraint definition starts with the keyword `hints` followed by the name of the variable to constrain in brackets. An assignment operator follows and finally the a constraint name with parameters concludes the statement. With this syntax it is also possible to define multiple constraints for one variable by using `+=` as operator. If parameters are passed to the constraint, they are specified in braces after the name of the constraint surrounded by double quotes. In Figure 4 the second line shows such a case: A constraint specifying a regular expression that does not allow any whitespace characters within a string. The `SingleWordConstraint` in the upper line is a more convenient method for this same purpose.

4.2 Reasoning

After the subtask of parsing there are often multiple possible solutions how the template's variables can be assigned with fragments of the source code. Several of these solutions might be contradictory if e.g. a variable is one time evaluated to `true` and another time evaluated to `false` in the same template application. The reasoning subtask addresses this issue by removing contradictory parse results. To retrieve all possible information complex expressions are used for reasoning, too. E.g. branching conditions that consist of multiple variables concatenated by an `AND` operator are split using constraint solving techniques. Additionally the reasoning subtask is responsible for the removal of local variables that do not originate from the context. The reasoning is discussed in this section.

Branching conditions often consist of multiple variables. We use a constraint solver to split up these compositions and infer the boolean value of previously not assigned variables wherever possible. The upper part of Figure 5 shows a

template with two conditional statements. The parser result states that the first constant string "some text" was found but the second one "some other text" was not. So, the first condition was evaluated to `true` and the second one to `false`. This knowledge (shown in the lower left-hand site of Figure 5) is passed to the constraint solver. The right-hand site of the figure below shows the results returned by the constraint solver. The variable `$c` has to be `true` to fulfill the constraints. For variable `$b` no information can be inferred. So, a new assignment `$c = true` is added to the set of assignments.

```
#if( ($a && $b ) || $c ) some text #end
#if( $a ) some other text #end
```

```
($a && $b ) || $c == true      ⟹    $c := true
$a == false                          $b := undef
```

Fig. 5. Above: Extract of a template. Left side: Conditions from the template and their boolean value, assigned to them by the parser. Right side: Values inferred by the reasoning mechanism.

After the constraint solving is done, a check is performed that ensures that all assignments are consistent. First of all a variable that is only accessed read-only in the template (meaning there is no direct assignment to an value) must have the same source code fragment assigned each time it is used in the complete template. If a variable is assigned to a value (or another variable, or some calculated value), we distinguish if it is assigned only once before it is accessed or even afterwards. In the first case the variable is treated as if it was accessed read-only. In the second case the variable is marked as *mutable* and cannot be used for the reasoning subtask. If there are contradictory assignments the complete solution can be dismissed. This way the reasoning reduces the number of solutions - ideally only one solution will remain.

The last step of the reasoning is the removal of local variables. As many template languages, Velocity offers the possibility to specify local variables. Local variables are often used in templates e.g. to calculate a boolean value only once, if it is needed several times. While the parser only assigns these local variables with values, they should not be used to reconstruct the model, because the reconstruction of the model should not depend on implementation details of the templates. So, the reasoning mechanism has to remove the assignments of local variables. After the removal of the local variables, only the (textual) values of the attributes read from the intermediate layer are left. Figure 6 shows an example of this step.

While parsing the allocation of a local variable was found in the template. Then the parser was able to assign the local variable `$myLocal` with the logical value `true`. From the definition of the local variable `$myLocal` the reasoner is able to infer, that if `$myLocal` is `true`, both `$token.foo` and `$token.bar`

```
#set($myLocal = $token.foo && $token.bar)        ⟹   $token.foo := true
$myLocal == true                                      $token.bar := true
```

Fig. 6. Left side: Example of an allocation of a local variable within a template and an assignment found by the reasoner. Right side: Values inferred by the reasoning mechanism.

must be `true`, too. So by reversing all allocation statements (`#set` directives in Velocity), local variable assignments are removed and the attribute values of the intermediate layer are inferred.

At the end of the reasoning subtask ideally one solution remains. But if the combination of template and source code is ambiguous, there may still remain several solutions. A simple example of this case is if there are two succeeding conditions with identical bodies but different condition statements. Then two valid solutions remain even after reasoning and it is not possible to say which solution has been used for code generation. Such problems should be avoided by the template designer whenever possible. Anyhow, if several solution are found, our implementation needs user interaction to choose the right solution.

Parsing and reasoning are not necessarily subsequent tasks. In fact it makes much sense to combine both tasks for performance reasons. Since the reasoning excludes possible solutions, excluding those as soon as possible can really speeds up the parsing process. E.g. if the parser has assigned one variable and finds another value for the same variable later on, the current solution can be skipped and the parser does not have to match neither the rest of the current template nor nested templates.

4.3 Creating tokens

After the subtask of reasoning there are lists of assignments for each pair of template and source code. These assignments are key-value assigning attributes of the intermediate layer of tokens to text extracted from the source code. Though it is possible to generate the model directly from the information included in these assignment, we create an intermediate layer first. This intermediate layer can be generated generically. The advantage of this approach is that the updating/creating of the model can now be described as a mapping between two models.

So, this subtask addresses the issue of creating an intermediate layer of tokens. Each token represents a single code fragment, for example an attribute of a class or a method declaration. So, there exists a token type for every template. As described in section 3, the token layer is linked as a tree: Source code generated for each parent token contains the source code that is generated for its child tokens. This structural information is obtained from CodeGen2.

The reconstruction of the token layer is done as follows: For each solution (consisting of template, source code, and assignments) a token object is created and then linked with previously reconstructed tokens according to the structural

information given by CodeGen2. Afterwards the textual assignments are mapped to attributes of the created token. Figure 7 shows an example of this procedure.

```
$token.name := "example"
$token.visibility := "public"
$token.type := "int"
```

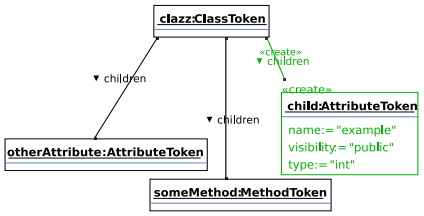

Fig. 7. Left side: Textual assignments found for a template of an attribute. Right side: Object diagram representing the token layer.

The left side of figure 7 shows the assignments found for a template representing an attribute. So a token of type `AttributeToken` is created and linked in the tree of previously created tokens. Then the fields of the attribute token are accessed via reflection: The field `name` gets the value `"example"`, `visibility` the value `"public"` and `type` gets `int`. In this way the complete layer of token is created.

At the end of this task all textual assignments are transferred into the intermediate layer of tokens.

4.4 Updating of the model in the CASE tool

The last step is now to create/update the model in the application itself. Therefor the intermediate layer of tokens is traversed and the model is adapted accordingly. This cannot be done generically, because the UML model is a complex graph and not a plain tree like the token layer. So, there is a strategy for each token type that updates the model. Even in case of reverse engineering, when the existing model is empty, it is not sufficient to let the strategies simply create all model elements. In fact it is necessary to search the model element first, because it could have been created by another strategy before. Figure 8 shows an example of this situation.

In this example two classes with one connecting association have been parsed and the intermediate token layer was created. The model mapping mechanism starts with the package token and continues with either of both class tokens, for example with the one generated for the class `Teacher`. So a class `Teacher` is created in the model, if it does not yet exist. The next token in the token tree is the role token `toN`. In order to create the complete association in the model the association's target role and the corresponding class need to be created, too. It is known that the role's class must have the name `Course`, because this is the Type of the field in class `Teacher`. So `Course` is created in the model, then both roles and finally the association itself. These steps are done by the strategy for role tokens. Later on, the other class has to be mapped to the model. Because the

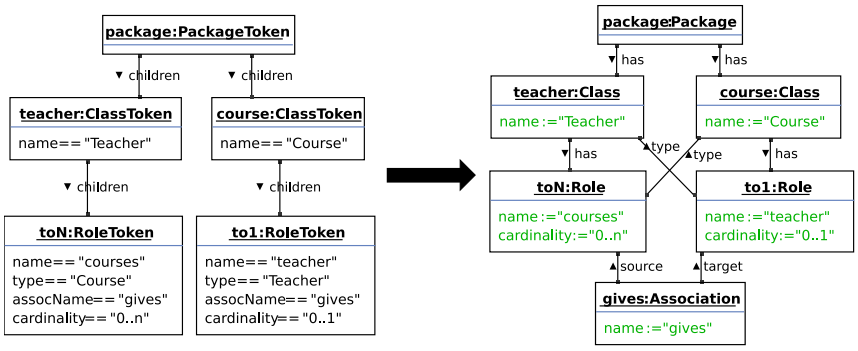

Fig. 8. Object diagram of token layer (left side) and corresponding model.

class `Course` already exists, it is not created but updated. The class' visibility and some other properties are set that were not known while creating the class. The same is done for the other role-token which is processed next: As there already exists a role connected to an association with the name `gives` it must only be updated. After this task the model represents the parsed source code.

Note that association detection normally is a complex task because one must determine if two roles belong to the same association. As we generate the association's name into the source code (as comment of the roles) we can easily identify both roles belonging to one association.

4.5 Dealing with nested templates

In section 4.1 we introduced the parsing as the first step of reverse engineering. There we assumed that the template to use is predetermined. But in fact there are often several templates to be considered. Furthermore the code generator nests multiple templates to generate one single source code file. As described in section 3 the code generator traverses the token tree inorder and starts generating code for the leaves of that tree. The code generated for all child token is passed when generating code for their parent token which usually includes the child code in the parent code. In our implementation the parent templates include the children's code in the template variable `$children`.

So, the `$children` variable needs special treatment when reverse engineering a piece of source. It is not sufficient if one single template matches[2] a piece of source code, but potentially embedded templates have to fit, too. It is possible that a parent template matches but the children dont. So if the parser assigned some value to the variable `$children`, one or more child templates must fit at least once.

Figure 9 shows an example for nested templates.

The start template is predetermined by the code generator. So the parser starts parsing the complete Java class with the template for Java classes and

[2] Matching in the sense of sections 4.1 and 4.2.

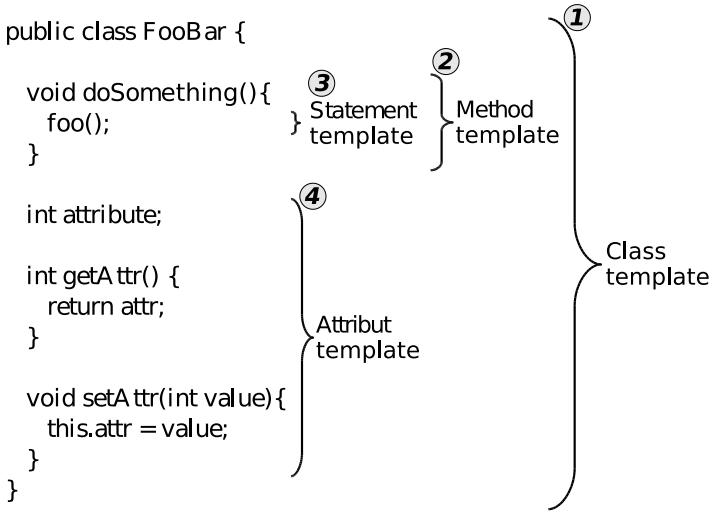

Fig. 9. A Java class and corresponding templates. The numbers identify the order in which the parser tries the nested templates. Note that the parser attempts to match all suitable templates in each step.

assigns the class' body to the template's variable `$children`. Then multiple templates have to be considered: As a Java class can contain methods and attributes, the parser attempts to match the body with both templates. As in the second step the method template matches - which includes other templates, too - the parser attempts to parse the methods body before continuing parsing the rest of the class' body. In the example the template that matches the rest of the class' body is the template for attributes.

In general the order in which the parser attempts to match templates to source code does not matter. But if the parser tries to match the subsequent templates before he matches the child templates successfully (in the example steps three and four would be exchanged), the parsing time increases. As a method template only consists of a method declaration, opening braces, the `$children` variable and closing braces, this template is able to match less or more source code than one complete method. For example starting at the method declaration and ending at some closing braces in the middle of the method. As this does not make much sense the complete solution should be discarded as soon as possible and a longer match should be tried. But if the parser instead tries to match subsequent templates parsing time increases. So our parser attempts to match child templates before matching subsequent templates.

5 Conclusion

This paper presents an approach for reverse engineering of code generated by a template based code generator. Our approach uses the templates as a "grammar"

to parse the given code. Thus, we have build some kind of compiler compiler. However, since our template based "grammars" do not conform to LL1 or LALR1 grammar restrictions, our approach has to fight several performance issues. With the help of some template extensions, we have achieved a reasonable performance for our examples. This still needs improvement.

The main advantage of our approach is that the reverse engineering mechanism is language independent since it does not rely on a specific language parser. Additionally, the very frequent changes that are made to the templates by our developers are instantly taken into account by the reverse engineering component. Only if our meta model or the structure of our intermediate token layer changes, we have to adapt the model access parts of our templates. However, compared to the templates, our meta model and the token layer are quite stable. Thus, we have reduced the maintenance problem of keeping forward and reverse engineering synchronous, dramatically.

Note, if a template is modified, our parsing approach is adapted, instantly. Thus it is instantly able to recognize code generated with the new templates. However, it may now fail to recognize code generated with the earlier version of the template. Thus, in future work we will keep the earlier versions of our templates and use those as fallbacks if the new version fails.

Using our new approach, we observed that the template based parsing approach facilitates the recognition of quite complex Java code structures, dramatically. For example, our code generation implements a to-n association with the help of a container attribute and about 11 access methods. Using the old pattern matching on the Java parse tree it was quite tedious to identify all these parts of an association implementation, correctly, cf. [17]. Within our association template all these access methods are listed in a row. Thus, the association template provides exactly the pattern required to recognize all the access methods during parsing. However, this has the drawback, that the template expects the access methods in the given order and with exactly the given implementation. If e.g. some IDE reorders the methods, our recognition will fail. In our experiences this did not turn out to happen (to us) in practice.

We found that using the templates for parsing can result in a very slow parser since the possibilities of template applications can easily explode. Since the templates form some kind of a context free grammar, ideally, the generated parser should achieve a worst-case runtime complexity of $O(n^3)$ as the CYK algorithm. While this is already pretty inefficient, the CYK algorithm requires normalized grammar rules much simpler than our template structures and our parser faces the additional task of resolving template control structures and variable assignments. To overcome the performance problems, we add constraints to the templates that reduce the state space. An open problem is to what extent such constraints may be inferred directly from the meta-model. Additionally, to speed up the parsing, our reasoning step excludes paths from the state space, as soon as possible.

We have implemented our approach as a part of the code generation of the Fujaba Tool Suite [10, 6, 7]. We are now able to reliably reverse engineer every code

generated by our code generation. In current work, we address manual changes to the source code. If such changes obey the coding rules or our templates, our reverse engineering works fine. However, a simple `System.out.println` in an attribute access method may suffice to disable the recognition of the corresponding template. Similarly, manual declarations of attributes or manual implementations of associations will most likely not be recognized by our usual template based parser. To address such manual code, we have added so called "legacy templates" to our reverse engineering component. These legacy templates cover all basic Java elements. We use these legacy templates as fallbacks if the usual code generation templates fail. The result of such a legacy template recognition is usually rather low level, e.g. an attribute of some container type instead of a to-n association to some user defined class. However, we are able to reverse engineer every possible Java source code.

References

1. Velocity Homepage. http://velocity.apache.org/, 2006.
2. Java Emitter Templates Tutorial. http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html/, 2008.
3. JavaCC Homepage. https://javacc.dev.java.net/, 2008.
4. JJTree Reference Documentation. https://javacc.dev.java.net/doc/JJTree.html, 2008.
5. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
6. M. Bork. Reverse Engineering generierten Quelltexts durch Analyse von Velocity Templates. Master's thesis, Kassel, Germany, 2007. Diploma I Thesis.
7. M. Bork. Reverse Engineering von Legacy Code: Optimierung des template-basierten Reverse Engineerings zu einem transparenten und flexiblen Erkennungsmechanismus. Master's thesis, Kassel, Germany, 2007.
8. L. Briand, Y. Labiche, and Y. Miao. Towards the Reverse Engineering of UML Sequence Diagrams. *wcre*, 0:57, 2003.
9. R. Ferenc, F. Magyar, A. Beszedes, A. Kiss, and M. Tarkiainen. Columbus - Reverse Engineering Tool and Schema for C++. *icsm*, 00:0172, 2002.
10. Fujaba Group. The Fujaba Toolsuite. http://www.fujaba.de/, 1999.
11. L. Geiger, C. Schneider, and C. Record. Template- and modelbased code generation for MDA-Tools, 2005.
12. M. Kaastra and C. Kapser. Toward a semantically complete java fact extractor. Department of Computer Science, University of Waterloo, April 2003.
13. T. Klein. Rekonstruktion von uml aktivitats- und kollaborationsdiagrammen aus java quelltexten. Master's thesis, Paderborn University, 1999.
14. T. Klen, U. A. Nickel, J. Niere, and A. Zündorf. From uml to java and back again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.
15. E. Korshunova, M. Petkovic, M. G. J. van den Brand, and M. R. Mousavi. CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code. In *WCRE*, pages 297–298. IEEE Computer Society, 2006.
16. U. A. Nickel and J. Niere. Modelling and simulation of a material flow system. In *Proc. of Workshop 'Modellierung' (Mod), Bad Lippspringe, Germany*. Gesellschaft für Informatik, 2001.

17. U. A. Nickel, J. Niere, J. P. Wadsack, and A. Zündorf. Roundtrip engineering with fujaba. In J. Ebert, B. Kullbach, and F. Lehner, editors, *Proc of* 2^{nd} *Workshop on Software-Reengineering (WSR), Bad Honnef, Germany*. Fachberichte Informatik, Universität Koblenz-Landau, August 2000.
18. J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press, May 2002.
19. J. Niere, J. P. Wadsack, and L. Wendehals. Design pattern recovery based on source code analysis with fuzzy logic. Technical Report tr-ri-01-222, University of Paderborn, Paderborn, Germany, March 2001.
20. J. Niere, L. Wendehals, and A. Zündorf. An interactive and scalable approach to design pattern recovery. Technical Report tr-ri-03-236, University of Paderborn, Paderborn, Germany, January 2003.
21. A. Rountev, O. Volgin, and M. Reddoch. Control flow analysis for reverse engineering of sequence diagrams. Technical Report OSU-CISRC-3/04-TR12, Ohio State University, Mar. 2004.