Repleo: a Syntax-Safe Template Engine

Jeroen Arnoldus
Instituut voor Informatica
Hogeschool van Amsterdam
Weesperzijde 190, NL-1097 DZ
Amsterdam, The Netherlands
b.j.arnoldus@hva.nl

Jeanot Bijpost
Mattic B.V.
Chatelainestraat 35, NL-1336 SC
Almere, The Netherlands
j.w.bijpost@mattic.com

Mark van den Brand *
Department of Mathematics and
Computer Science
Eindhoven University of Technology
Den Dolech 2, NL-5612 AZ
Eindhoven, The Netherlands
m.g.j.v.d.brand@tue.nl

Abstract

Templates are a very common solution to generate code. They are used for different tasks like rendering webpages, creating Java Beans and so on. Most template systems have no notion of the object language and just generate text. The drawback of this approach is the possibility to generate syntactical incorrect code. This can lead to all kinds of annoying errors.

In this paper we present an approach for a syntax safe template engine. Syntax safety guarantees that the generated code can be correctly parsed. To ensure this we use the object language grammar to evaluate the template.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.3.4 [*Programming Languages*]: Processors

General Terms Languages

Keywords Repleo, Code Generators, Templates, Syntax Safety, ASF+SDF, Meta Programming, Concrete Object Syntax

1. Introduction

The way we program evolves. Over the years applications get more complex. To handle this complexity we program at a higher level of abstraction and encapsulate algorithms in objects. A next step in abstraction is model driven generation, such as Model Driven Architecture (MDA) [18]. The goal of MDA is to specify a model independent of the platform and maximize the automation to convert this model into working applications. These applications have the characteristics that they consist of various components, implemented in the most suited programming language. For instance SQL for the database, Java for the business rules and JSP for the web interface.

In this article we will focus on the technology to transform a model into source code. The OMG MDA standardization committee works on a model-to-text [5] proposal. They propose a generator based on templates. Templates are text documents (object language) punched with holes (placeholders with meta language). We believe this WYSIWYG[1] approach is user friendly, but considering the object language as text should be avoided because it can lead to syntax errors [17].

Some existing meta-programming approaches [19] [14] improve this by offering syntax safe or even type safe code generation. Syntax safety guarantees that the generated code can be correctly parsed, in other words the code does *not* contain context-free syntax errors. Type safety guarantees that the code does not contain non-context-free errors like type conflicts, double declarations of variables, etcetera. Switching to other object languages is not trivial with these approaches.

We present Repleo, a generic system to build template evaluators, parameterized with an (off-the-shelf) object language grammar [12]. This offers the opportunity to improve error checking by checking the syntax of a template and guarantees that the generated code is syntactically correct. The use of such an instantiated template evaluator is equivalent to a text based template evaluator, with respect to writing templates.

Writing templates is a complex and error prone task. We developed an IDE to support the writing of templates, which provides syntax directed features for templates such as syntax highlighting. These features are based on the parameterized parser of the template evaluator.

Our system is flexible with respect to changing to another object language. It is a main requirement for using the system in an (MDA) environment where a lot of different target languages must be supported. We have a parameterization mechanism to instantiate the evaluator for different object languages.

1.1 Roadmap

This paper is organized as follows: Section 2 describes syntax safe templates. In section 3 we present the template meta language. The implementation of the template engine is discussed in section 4. An illustration of possible applications will be given in section 5. Related work and conclusions are presented in section 6 and 7.

2. Syntax Safe Templates

A template based generation process uses a model and a set of templates. An evaluator is used to generate code given these two ingredients, see Figure 1. The model is an artifact describing the application at a high level of abstraction. It will probably be developed

* This work has been carried out as part of the Falcon project under the auspices of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik Program.

GPCE'07, October 1–3, 2007, Salzburg, Austria.

[1] What You See Is What You Get: Templates representing concrete object code.

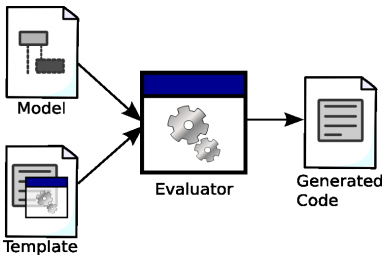

Figure 1. Template based generator

using a graphical notation like UML [6]. The model is used by the evaluator to replace the placeholders in a template. Those placeholders contain expressions to obtain data from the model. To illustrate the concept of templates a Java template is shown in Listing 1. This template is evaluated using the model given in Listing 2 to generate the Java class presented in Listing 3.

```
1  class <%record/name%> {
2
3   <%record/cons/vis%> <%record/name%>(){}
4
5   <%foreach record/field do%>
6    private <%type%> <%name%>;
7
8    <%type%> <%"get" || name%>(){
9     <%if log then%>
10    System.out.println("get"
11    +<%"\""|| name ||"\""%>+"() is called.");
12    <% fi %>
13    return <%name%>;
14   }
15  <%od%>
16 }
```

Listing 1. Java Template

```
1  record(
2    number(104),
3    name("Customer"),
4    cons(
5       vis("public")
6    ),
7    field (
8      name("firstName"),
9      type("String"),
10     log(true)
11   ),
12   field(
13     name("lastName"),
14     type("String"),
15     log(false)
16   )
17 )
```

Listing 2. Data structure example

```
1  class Customer {
2    public Customer(){}
3    private String firstName;
4    String getfirstName(){
5      System.out.println("get"
6         +"firstName"+"() is called.");
7      return firstName;
8    }
9    private String lastName;
10   String getlastName(){
11     return lastName;
12   }
13 }
```

Listing 3. Result of generation

The importance of syntax safe templates is illustrated by Table 1. This table summarizes errors that could be made in a template as shown in Listing 1 and in a model as shown in Listing 2. There are three possible causes of syntax errors in a template based generation system. First, the object code of a template contains syntax errors and the generated code inherits those errors (error A, B, C). Second, the data obtained from the model to substitute a placeholder does not syntactically fit into the object code of the template (error D, E, F). Finally, the meta code contains errors (error G, H).

In our system we use a grammar of the object language combined with a meta language grammar. Such a grammar allows us to parse the object and meta code of templates simultaneous, thereby addressing error A, B, and C. Error D, E, and F are detected by parsing the data obtained from the model before inserting it in the object code. The meta code is checked during evaluation which detects error G and H. The result is a syntax safe template evaluator.

The syntax safe template evaluator provides an early detection of syntactic errors resulting in a less error prone generation system and easier debugging. A text based evaluator with a parser at the back-end is not sufficient to ensure the syntax safety of the output. Consider error C of Table 1. This error will only arise in the generated code when the expression of the condition of line 9 of Listing 1 yields true. Our approach detects this kind of syntax errors at the moment the template is parsed, so we can ensure a condition can not introduce syntax errors. A syntax error is detected statically and not by the use of the generated code.

3. Template Meta Language

In the previous section we have shown a template. Templates consists of object language code with placeholders. These placeholders contain instructions for the evaluator. We use a simple tree navigation language similar to XPath [4]. The exact form of the meta language is not essential for our requirements. It could be any other language as long it has a grammar.

We have chosen for a simple meta language because templates are by nature complex due to the mix of object language and meta language. Since readability of templates is mainly influenced by the amount and complexity of the meta code, we use a meta language which only supports basic instructions such as queries, a basic set of expressions and instructions for the generation process like substitution, iteration and conditional (enumerated in Table 2).

This restricted meta language has some consequences for our input model. A restricted meta language can not be used for complex queries or calculations. Therefor the results of such calculations must be present in the model. However most models are specified at a high level of abstraction, independent of the object code. As a result a restricted meta language might require an enriched model. To separate concerns we derive this model during a separate model transformation phase. These transformations can be compared with

	Substitute line	by	creating error
A	1	`clss <%record/name%> {`	class misspelled.
B	1	`class <%foreach record/attribute do%> <%name%> <%od%> {`	lists after class not allowed.
C	10	`1System.out.println("get"`	1System is not a valid Java identifier.
D	1	`class <%record/number%> {`	Identifier is not allowed to be a number.
E	Model, 3	`name("Shop-Client")`	Identifier contains a dash.
F	Model, 5	`vis("abstract" )`	constructor modifier abstract not allowed.
G	1	`class <%record/naam%> {`	path record/naam is not a valid path in model.
H	1	`class <%1 || record/name%> {`	type conflict in meta code `int || string`.

Table 1. Possible errors in template and model

Syntax	Description
`<%Expr%>`	Substitute placeholder by result of `Expr`.

`<%foreach Expr do%>`
`  subtemplate`
`<%od%>`
	For every element in the list returned by `Expr` evaluate the subtemplate.

`<%if Expr then%>`
`  subtemplate`
`<%fi%>`
	If `Expr` is true then include subtemplate.
	Note: only allowed if result may be empty in object language.

`<%if Expr then%>`
`  subtemplate1`
`<%else%>`
`  subtemplate2`
`<%fi%>`
	If `Expr` is true then include subtemplate1 else include subtemplate2.

Table 2. Evaluator instructions of the template meta language

the model transformations in the MDA approach [18]. Further discussion of this topic is beyond the scope of this article.

4. Repleo Implementation

In order to validate our ideas we designed and implemented a prototype of a syntax safe template evaluator. The implementation is split in two parts. The first part consists of a parser based on the combination of the grammars of the object language and the meta language to derive a parse tree.

The second part is the evaluator and is responsible for substituting the placeholders. An important requirement is that the evaluator does not introduce syntactically incorrect nodes in the parse tree while substituting these placeholders.

4.1 Syntax Definition Formalism

A grammar is used to check the syntactical correctness of an input sentence and to construct a parse tree of this sentence. We use the Syntax Definition Formalism (SDF) [7] for defining grammars. SDF supports, besides the traditional BNF-like symbols, also symbols like character classes and associative lists to define the repetition of symbols. The main differences between SDF and BNF-like formalisms are modularity and the way production rules are written. The production rules differ because of the swapped left- and right-hand side, for example `"a" -> A` instead of `A ::= 'a'`. SDF definitions can be divided in modules. Modularity allows us to combine grammars, which is very suited for meta programming. It will be explained in Section 4.2,

SDF can be used to specify grammars for programming languages, like Java and C. Listing 4 shows an SDF module for a small artificial boolean expression language to illustrate relevant SDF features.

```
1  module BooleanList
2
3  imports Whitespace
4  context-free start-symbols      Booleans
5  context-free syntax
6     { Boolean "," }*             -> Booleans
7     Boolean "&" Boolean          -> Boolean
8     BoolCon                      -> Boolean
9     "true"                       -> BoolCon
10    "false"                      -> BoolCon
```

Listing 4. A boolean expression grammar.

The grammar of Listing 4 can be used to parse a term like `true, true & false, false`. The start symbol `Booleans` is defined as a list of at least zero `Boolean` nonterminals separated by a comma and is an example of an associative list[2]. The parse tree for this term is shown in Figure 2. For the ease of presentation we have chosen for this small specification because the Java grammar is too big.

4.2 Grammar Modules

SDF supports a modular structure for describing grammars. Grammar chunks can be embedded in modules and imported by other modules. The modular structure provides a powerful mechanism to re-use parts of grammar definitions.

The `BooleanList` module of Listing 4 imports the module `Whitespace` that defines the layout (spaces, tabs, and new lines). The `BooleanList` module itself can be imported by other modules.

[2] SDF supports separated lists; $\{Boolean", "\}*$ is an example.

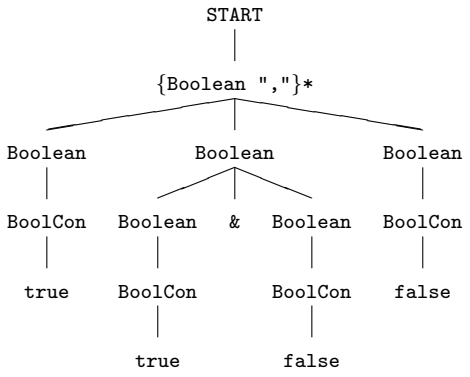

Figure 2. Parse tree for the term `true, true & false, false`.

4.3 Combining Object Language and Meta Language

The modularity of SDF provides a powerful way to define template grammars. Template grammars are based on three subgrammars: the object language grammar, the meta language grammar and a grammar defining the combination of both.

The modularity allows us to specify these grammars separately. The advantage of this concept is the ease of using off-the-shelf object language grammars [12]. When a grammar is not present, it is possible to specify the complete grammar, but this can be a non-trivial job. However, minimal syntax correctness can be guaranteed by defining a restricted grammar consisting of only the minimal lexical requirements [15].

Combining the object language and meta language is as easy as importing both grammars in *one* module and describing their connection. In order to connect the object and meta language, the nonterminals of the object language must be related to the nonterminals of the meta language. The implementation will be given in the rest of this section.

4.3.1 Defining Substitution Placeholders

In order to parse a template with a substitution placeholder it is necessary to embed the placeholder in the object language. For example the substitution placeholder `<%elem%>` in the template `<%elem%> , true & false , false` can only be parsed when it is recognized as a `Boolean` [3]. This can be achieved by embedding the substitution placeholder in the BooleanList grammar as an alternative for `Boolean`.

In order to add a substitution placeholder to the BooleanList grammar we introduce a generic module, as shown in Listing 5. This module defines a substitution placeholder typed by any arbitrary nonterminal X and adds this placeholder as an alternative for this nonterminal X. This allows us to re-use this module for any object language. Line 5 of Listing 5 shows the definition of the substitution placeholder. It consists of hedges[4] and the nonterminal `Expr`. The `Expr` nonterminal is defined in the module MetaExpressions containing the grammar for the basic operations of the meta language and generic instructions for obtaining data from models. The parameter `X` of the nonterminal `PlaceHolder[[X]]` is used for the explicit typing. The connection of object and meta language is defined on line 6. This production rule adds an alternative for recognizing the placeholder of type `X` to the nonterminal `X`.

[3] `elem` is a query pointing to the content of the `elem` node in the model.

[4] Hedges mark the transition between object and meta language.

```
1  module PlaceHolder[X]
2
3  imports MetaExpressions
4  context-free syntax
5    "<%"Expr"%>"          -> PlaceHolder[[X]]
6    PlaceHolder[[X]]      -> X
```

Listing 5. Generic substitution placeholder module.

The combination module for the BooleanList template grammar is presented in Listing 6. This module imports the object language module and the placeholder module where parameter `X` is substituted by the nonterminal `Boolean` [5].

```
1  module Combination
2
3  imports BooleanList
4  imports PlaceHolder[Boolean]
```

Listing 6. Combination module for parsing BooleanList template.

The parse tree of a BooleanList template is displayed in Figure 3. The nonterminal `PlaceHolder[[Boolean]]` marks the subtree representing the substitution placeholder. Embedding placeholders in an object language can result in syntactical ambiguities when parsing a template. In Section 4.7 we will discuss this problem in more detail.

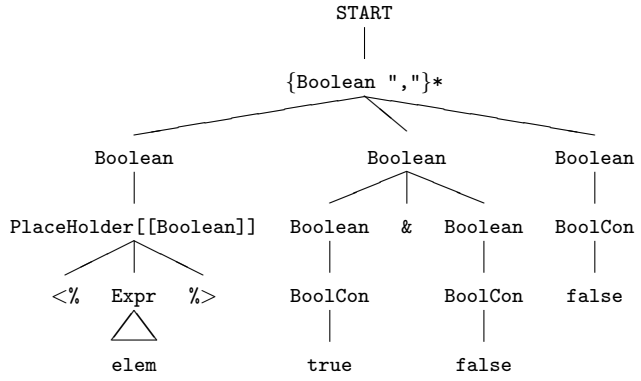

Figure 3. Parse tree of the template `<%elem%> , true & false, false`.

4.3.2 Java Example

The grammar shown in Listing 7 is defined to parse the Java template of Listing 1. The parameterization of the placeholders allows to define a very compact combination module.

The selection process of the nonterminals for parameterization of the placeholders must be done manually. Automatic parameterization of placeholders with every object language symbol can lead to unexpected behavior. It is hard to automatically predict which symbols must be selected for the parameterization of placeholders. A similar problem is discussed in [23]. Although they generate their connection rules, they consider it as useful to have full control over the selection of the symbols.

[5] Details on the parameterization mechanism of SDF can be found in [2].

```
1  module JavaTemplate
2
3  imports Java
4  imports PlaceHolder[Identifier]
5  imports PlaceHolder[Type]
6  imports PlaceHolder[Modifier]
7  imports PlaceHolderList[ClassBodyDeclaration]
8  imports PlaceHolderIfThenList[BlockStm]
```

Listing 7. Combination module for Java template.

Special attention should be given to line 7 and 8 of Listing 7. Those lines introduces new imported modules. The `PlaceHolderList` module defines the grammar of the iterator placeholder in order to parse the `foreach` construct of line 5 to 15 in the Java template of Listing 1. A substitution placeholder is not sufficient to generate multiple items, in this case body declarations for a class. The iterator processes a list obtained from the model and instantiates the subtemplate for each element of the list. Listing 8 shows the grammar of the generic iteration placeholder.

```
1  module PlaceHolderList[X]
2
3  imports MetaExpressions
4  context-free syntax
5    "<%foreach" Expr "do%>"  X* "<%od%>"
6                  -> PlaceHolderList[[X]]
7    PlaceHolderList[[X]]  ->  X*
```

Listing 8. Generic iteration placeholder module.

The iteration placeholder is also parameterized by a nonterminal. This parameterization is necessary for the connection of the placeholder to the object language nonterminal. Beside the connection the parameterization is also used to define the start symbol of the subtemplate. It enforces that an iteration placeholder for a list of nonterminals `X` can only contain a subtemplate consisting of a list of nonterminals `X`.

The second new placeholder, introduced in line 8 of the `JavaTemplate` module is the conditional placeholder. This kind of placeholders is used to include a subtemplate based on a condition based on data obtained from the model. In order to guarantee the syntactical correctness, the subtemplate(s) of a conditional may not introduce syntactical incorrect code. The concept to guarantee the syntactical correct subtemplate is similar to the iterator approach. Consider the grammar of the conditional if-then placeholder for lists in Listing 9. It is parameterized with nonterminal `X`. The start symbol of the subtemplate of the conditional must be of type `X`.

```
1  module PlaceHolderIfThenList[X]
2
3  imports MetaExpressions
4  context-free syntax
5    "<%if" Expr "then%>"  X* "<%fi%>"
6                  -> PlaceHolderList[[X]]
7    PlaceHolderList[[X]]  ->  X*
```

Listing 9. Generic if then placeholder module for lists.

4.4 SQL Example

The next example shows a SQL select statement template. Consider Listing 10, this template also uses the model of Listing 2. The result of the evaluation is shown in Listing 11.

```
1  SELECT
2     <%foreach record/field do%>
3       <%name%>
4     <%od%>
5  FROM <%record/name%>;
```

Listing 10. SQL Template

```
1  SELECT
2     firstName, lastName
3  FROM Customer;
```

Listing 11. Result of evaluation of SQL Template

The combination module to parse this template is shown in Listing 12. It introduces a new placeholder at line 5. This placeholder is for associative lists with a separator. The separator is defined in the second argument of the parameterization.

```
1  module SqlTemplate
2
3  imports Sql
4  imports PlaceHolder[Identifier]
5  imports PlaceHolderSepList[SelectSubList ","]
```

Listing 12. Combination module for SQL template.

4.5 Evaluation

In the previous section we described the approach to guarantee the syntax correctness of the template. In this section we will discuss the evaluator. In order to generate syntactic correct code the evaluator takes a parsed template and a model.

The first step in evaluating a template is locating the placeholders in the parse tree. This is done by a top down tree traversal mechanism. The traversal stops when a node is recognized as a placeholder. When a placeholder is located the evaluator interprets the meta code of this placeholder and replaces it if nothing fails.

In this section we will illustrate the mechanism by discussing the evaluation of the substitution and iterator placeholders.

4.5.1 Substitution Evaluation

The most elementary feature of a template is the substitution placeholder. A substitution placeholder is replaced by the result of the evaluation of the expression inside the placeholder.

Consider line 1 of the Java template of Listing 1. It contains the substitution placeholder `<%record/name%>`. The expression is a query to obtain data from the model. When evaluating this template the following steps are executed:

1. Locate placeholder (`<%record/name%>`).
2. Evaluate the expression (lookup the value stored at `record/name` in the model).
3. Parse the result obtained by evaluating the expression (`parse-identifier("Customer")`).
4. Replace the placeholder by the result of the parser (`<%record/name%> := Customer`) if step 2 and 3 succeed.

The replacement of the placeholder is only allowed if step 2 and step 3 are successfully evaluated. In case of an error the evaluator generates an error message and does not replace the placeholder.

The error messages of step 2 are related to the evaluation of the expression. The following errors can occur:

- The path of the query does not exist in the model (e.g. `record/naam`).
- The expression contains a type conflict (e.g., `12||"foo"`).

These errors are checked while interpreting the meta language constructs.

Step 3 is the most important step to guarantee the syntactical correctness of the resulting code. The result of the expression is parsed. An error message is generated when parsing fails otherwise the placeholder is replaced. The start symbol for the parser is inferred from the explicit typing of the placeholder with the nonterminal.

4.5.2 Iteration Evaluation

The foreach-statement is used to generate repetitive language elements in the object code. This process is based on instantiating a subtemplate for every element in a list obtained from the model. A query is used to obtain the list from the model.

Consider the foreach-statement at line 5 to 11 of the Java template of Listing 1. The subtemplate of this foreach is represented in line 6 to 10 and contains Java class body declarations (i.e. a field declaration and a method declaration).

The evaluator will iterate over the list obtained from the model and will evaluate the subtemplate for every element in the list. The iterator evaluation algorithm is based on the substitution placeholder algorithm except for step 3. Step 3 is replaced by a loop instantiating the subtemplate for every element in the list. The instantiation process is recursive, so a subtemplate can contain object code with placeholders.

To reduce the length of the queries the foreach instruction uses a context switch when evaluating the subtemplate. This allows to use the query `type` and `name` instead of `record/field/type` and `record/field/name` in the subtemplate, shown in line 6 of Listing 1.

4.6 Implementation Framework

We have shown the basic concepts of the implementation of the template grammar and evaluator. A prototype of the evaluator is implemented in the term rewriting system ASF+SDF [2]. We have chosen for ASF+SDF because of the combination of a parser and strongly typed first-order functional language.

SDF is used in combination with the SGLR parser [22]. The SGLR parsing algorithm allows us to define an SDF definition containing multiple languages and to use this grammar to parse a term in a single parsing process. We do not need different parsers for the different languages inside a template.

ASF (Algebraic Specification Formalism) can be seen as a strongly typed first-order functional programming language. It provides traversal functionality to visit nodes in a tree and match on certain nonterminals [20]. Beside the traversals ASF allows to define equations which can be executed as rewrite rules. The strong typing of the equations of ASF guarantees the syntax safety of our evaluator.

4.7 Syntactic Ambiguities

Introducing placeholders in an object language can lead to undesired behavior. Sometimes a placeholder in a template can be valid for different defined placeholder types. When parsing a template this leads to ambiguities. The nonterminals to parameterize the placeholders must be selected carefully to prevent those ambiguities, although ambiguities could not always be avoided.

Consider line 3 of the example Java template of Listing 1. This line contains two placeholders. It could be interpreted as a constructor declaration or a method declaration. This conflict is caused by the first placeholder, which could be parsed as a `Type` or `Modifier` illustrated in Figure 4. The final result when evaluating this line depends on the content of the model.

We allow ambiguous templates and we use a disambiguation filter based on term rewriting [21] to solve the ambiguities. The template is parsed and instead of solving the ambiguities during parsing, the ambiguities are stored in the parse tree. Those ambiguities are represented by a node containing a list of possible alternatives.

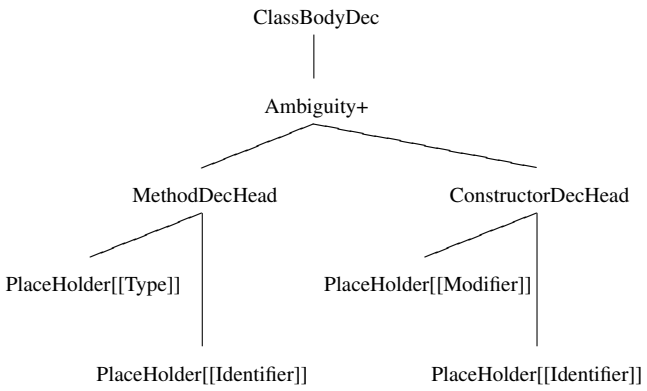

Figure 4. Simplified parse tree of the ambiguity Listing 1 line 3.

The evaluator is capable to detect these ambiguity nodes. In order to deal with ambiguities the evaluator tries to evaluate the alternatives of the ambiguities in the same model context. We stop evaluating the alternatives when the first successfully evaluated alternative is found. We can omit from evaluating every alternative to improve the performance of the system. Evaluating the different alternatives could result in different structures of the parse trees, but the leaves of those trees contain the same lexicals. The unparsed text of the different trees does not differ. This assumption and the fact that we unparse the parse tree allows us to stop evaluating the ambiguity alternatives after a successful evaluation is found. The evaluator generates an error when it is not possible to evaluate any of the alternatives successfully .

5. Applications

Our main goal is to provide a syntax safe template engine for source code generation in a MDA context. Repleo is not limited to this domain. In this section we discuss in which domains our system could be applied.

5.1 Information Systems

From the MDA perspective this is an interesting domain. A lot of information system frameworks exists to support developers and to reduce boiler-plate code. At the moment the Spring framework [10] is a popular base architecture for Java based information systems.

Although those frameworks provide a lot of abstraction, there exists repetitive definitions. This repetition does not only exist in the java source code files, but also in XML property files and JSP files.

At the moment we generate Java Beans, SQL queries and XML files for the Spring framework. A class diagram is used as model.

5.2 ApiGen

ApiGen [11] is an application to generate a Java or C API for creating, manipulating and querying tree-like data structures based on ATerms [1]. The base for an API is an SDF syntax definition to define the structure of the tree. The current implementation of ApiGen is based on a Java program consisting of `println()` statements.

ApiGen consists of two stages. First, it reads an Abstract Data Type (ADT) specification, generated from an SDF definition, containing the specification of the tree structure. This ADT format

is concise. ApiGen enriches the ADT with inferred information, which could be seen as separated model transformations as discussed in section 3. The second stage is generating the code.

In order to proof our concept we extracted the object code from the generator source and implemented it in Java templates. The `println()` statements could easily be mapped into templates. We also used an ADT enrich stage. This allows us to reduce the amount of calculations in the templates. The meta code of the templates consists mainly of queries and string concatenations.

We have experienced a number of benefits over the `println()` generator. First, the structure of the templates is very similar to the output. The syntax safety makes it easier to write templates, because it directly generates an error if we make a syntax mistake in the template. Second, the generator is smaller and easier to maintain, because we have separated the evaluator logic from the representation of the object code.

6. Related Work

There are many approaches and concepts to generate code from a model. We give an overview of some work similar to our approach.

6.1 Text Template Based Generators

A popular mechanism to generate code are text-based template evaluators. They are well accepted and frequently used for different tasks, like HTML generation for webpages and code generation. The benefits are separation of concerns (logic and object code) and reusability of templates.

A lot of template evaluators exists. Some examples are ERb [8] and StringTemplate [16]. They all have in common that they are text-based. Ignoring the object language makes them flexible, easy to parse and fast in evaluating, but also unsafe. We were inspired by the WYSIWYG approach of these systems. In contrast with our system, misspellings and other syntactical errors in the object code of the templates and the generated code are not detected by the text based template engines.

6.2 Concrete Object Language Meta Programming

MetaML [19], domain specific embedded compilers [14] and MetaBorg [3] present approaches to generate programs, where a grammar is used for the object language.

MetaML is a homogeneous meta programming approach to generate ML with ML in the context of multistage programming. ML is a general-purpose functional programming language. Multistage programming is a technique to produce a specialized instantiated solution from a generic solution. For example unfolding a recursive function for x^y, when y is known before runtime, to $x * x^{(y-1)}$ etcetera until $y = 0$. MetaML ensures the generated code is syntactically correct and type safe.

Domain specific embedded compilers [14] is a technology to express a domain specific language, such as SQL, in a high order typed language like Haskell. This solution can ensure the SQL statements are syntax and type correct. The type safety is obtained by introducing phantom types for SQL expressions. Phantom typing is a technique to create annotations containing type information for the nonterminals in the parse tree of the SQL expression. It is a nice idea for typing languages embedded in some other (host) language. This system is useful when the embedded language is used by the host language to invoke another component, such as a database, or when this language is used to expand to host language, such as a UI definition, by an external generator.

MetaBorg [3] shows another approach for embedding domain specific languages in a host language. They show a technique to embed a DSL for UI definitions (SWUL) in Java source code and expand this definition in Java code when evaluating the input program. The type safety check is performed by an extended Java typechecker on the input program. The extension provides a mechanism to check the DSL, the connection between the DSL and Java and finally the Java code. The transformation of SWUL to Java is expressed in Stratego/XT [24]. There is no guarantee the code will not get ill-formed during this transformation, because the transformation may be not syntax safe.

Repleo is a heterogeneous meta programming system with a parameterizable object language. More important, none of these approaches use an external model to instruct their generation process. Furthermore, domain specific embedded compilers and MetaBorg are not template evaluators.

6.3 Safegen

SafeGen [9] is a recent approach to achieve type safe templates. It uses an automatic theorem prover to prove the well-formedness of the generated code for all possible inputs.

This approach heavily depends on the assumptions that the input is a valid Java program and the knowledge of the Java type system. The template programmer can define placeholders (cursors) to obtain data from the Java input program. Those placeholders must contain constraints based on their use in the template. For example a placeholder in the extends section of a class in a template must guarantee the extended class is not final. A prover is used to check the constraints, this ensures that the template can not generate ill-formed code.

This system depends on the notion that the input and output program is Java. This fact makes the environment not capable to generate code from an abstract high-level model in another representation than the object language, because it depends heavily on reflection. Although they could give more and better guarantees about the generator, we believe switching to another object language and model representation is hard. Our approach is more flexible in respect to these important requirements for a MDA environment.

6.4 API Based Generators

Another kind of approach for generating code is to manually build a structural representation, like an abstract or concrete syntax tree, of the object code.

6.4.1 API

An API based on the grammar of a language can provide functionality to build an abstract syntax tree or concrete syntax tree of the object language. Such an API consists of methods, classes or functions to instantiate the different components of a language, for example a class Class, class Method, class FieldDeclaration. The structure is hierarchical, so a Class can contain Methods and FieldDeclarations. Examples of these systems are Jenerator [25] providing an environment for building generators for Java or an API based on a grammar generated by ApiGen [11], as described in section 5.2.

6.4.2 Generator

A generator based on an API instantiates an abstract or concrete syntax tree when generating code. This tree can be used to guarantee the syntactical correctness. The hierarchical structure of the tree must match the grammar of the object language and the constructors of the classes instantiating the lexical nodes must validate the lexical requirements.

The drawback of this approach is the complexity of defining a generator. The generator programmer must manually instantiate the nodes of the abstract or concrete syntax tree and therefor needs detailed knowledge of the object language grammar. This approach also implies the representation of the object code is not concrete and so hard to read and understand for a human.

6.5 Future Work

At the moment we are able to guarantee the syntax correctness of the templates. An open question is to guarantee type correctness of the templates and/or the generated code. There are two problems. First, the context, like imported libraries or code generated in the future, of the template is not known. Second, the exact content of the model is not known before evaluating the template.

Given that the context and model are not known, it is not possible to fully type check the generated code during the evaluation of the template. Probably a type inference system could provide some guarantees. However it is questionable whether it is worth to invest a lot of effort to implement such a system for every object language, while type correctness can not be ensured completely.

Another problem which can occur in the generated code is the clash between a variable already defined in the template and a generated variable based on data obtained from the model. It could be interesting to investigate whether syntactic hygiene [13] can be used to reduce these possible conflicts between the model and the template.

7. Conclusion

In this paper we have presented a syntax safe template based generation system. Syntax safe templates provide a mechanism to detect syntactical errors earlier in the process and closer to the source of the error, instead of dealing with errors at compile time.

The parameterization concept of the evaluator allows us to instantiate template evaluators for different languages. This especially confirms to the requirements for the MDA approach to generate code for various languages. We already have an SDF, Java, SQL and XML engine.

The approach presented in this paper provides syntax safety of templates and can be used instead of text based template engines. Our approach allows the WYSIWYG development of templates in the same way as for text templates. The amount of syntactical checking finally depends on the granularity of the underlying grammar.

Finally, having the parse tree of both object language and meta language creates the possibility to implement advanced IDE features. Templates are not well supported by IDE's due their multilingual nature. We already have syntax highlighting, but other source code based features for both languages simultaneously could be possible, such as auto-completion.

Acknowledgments

We thank Jacob Brunekreef and Jurgen Vinju for having proofread and commented on earlier versions of this paper. We also want to thank Martin Bravenboer for a fruitful discussion on the syntactic ambiguity problem.

References

[1] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.

[2] M.G.J. van den Brand and P. Klint. ASF+SDF Meta-Environment User Manual.

[3] M. Bravenboer, R. de Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *GTTSE '05: Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering*, 2005.

[4] J. Clark, S. DeRose, et al. XML Path Language (XPath) Version 1.0. *W3C Recommendation*, 16:1999, 1999.

[5] Object Management Group. MOF model to text transformation language rfp. OMG Document ad/2004-04-07 (2004).

[6] Object Management Group. Unified modeling language: Superstructure-version 2.0-final adopted specification. 2005.

[7] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[8] J. Herrington. *Code Generation in Action*. Manning Publications, 2003.

[9] S.S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 2005.

[10] R. Johnson and J. Hoeller. *Expert One-on-One J2EE Development without EJB*. John Wiley & Sons, 2004.

[11] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. *Journal of Logic and Algebraic Programming*, 59(1-2):35–61, 2004.

[12] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

[13] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM Press.

[14] D. Leijen and E. Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122, New York, NY, USA, 1999. ACM Press.

[15] L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, oct 2001.

[16] T.J. Parr. Enforcing strict model-view separation in template engines. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM Press.

[17] T. Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.

[18] R. Soley and the OMG Staff Strategy Group. Model-driven architecture. November 2000.

[19] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. Technical report, 1999.

[20] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[21] J. J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, University of Amsterdam, 2005.

[22] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, Universiteit van Amsterdam, juli 1997.

[23] E. Visser. Meta-programming with concrete object syntax. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 299–315, London, UK, 2002. Springer-Verlag.

[24] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[25] M. Völter and A. Gartner. Jenerator - generative programming for java. OOPSLA, October 2001.