Generating Safe Template Languages

Florian Heidenreich Jendrik Johannes Mirko Seifert Christian Wende Marcel Böhme

Lehrstuhl Softwaretechnologie
Fakultät Informatik
Technische Universität Dresden, Germany
{florian.heidenreich,jendrik.johannes,mirko.seifert,c.wende,marcel.boehme}@tu-dresden.de

Abstract

Template languages are widely used within generative programming, because they provide intuitive means to generate software artefacts expressed in a specific object language. However, most template languages perform template instantiation on the level of string literals, which allows neither syntax checks nor semantics analysis. To make sure that generated artefacts always conform to the object language, we propose to perform static analysis at template design time. In addition, the increasing popularity of domain-specific languages (DSLs) demands an approach that allows to reuse both the concepts of template languages and the corresponding tools.

In this paper we address the issues mentioned above by presenting how existing languages can be automatically extended with generic template concepts (e.g., placeholders, loops, conditions) to obtain safe template languages. These languages provide means for syntax checking and static semantic analysis w.r.t. the object language at template design time. We discuss the prerequisites for this extension, analyse the types of correctness properties that can be assured at template design time, and exemplify the key benefits of this approach on a textual DSL and Java.

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

General Terms Languages

Keywords generative programming, template language, safe authoring, language extension

1. Introduction

Generating programs and other artefacts (e.g., documentation or web pages) is considered a viable approach to reuse recurring text fragments in variable settings. By using a parametrisable generator, the time consuming manual creation of boilerplate can be avoided. Among the various ways to built such a generator, template languages have become very popular. Based on the concrete syntax of the object language (i.e., the language to be generated) and a basic set of imperative operators (e.g., loops and conditions), templates can be used to perform arbitrary generation tasks. In comparison to other approaches, such as generating artefacts based on their abstract syntax, templates are easier to read and maintain.

However, a closer look reveals some serious problems faced by template developers. First, template languages can be either specific to a particular language or language agnostic. In the former case reusing the template language and tools is sacrificed for the benefit of language specific support (e.g., syntax and semantics checks, highlighting etc.). In contrast, the latter type of languages can be reused for arbitrary object languages, but often at the cost of sacrificing sophisticated development support. Second, both kinds of template languages described before are often executed by translating them to meta programs with string literals. Template instantiation is thus based on plain string concatenation. Tracing the impact of input parameters to a particular template instance is consequently only possible at a very low level. A type safe composition of the template contents and the parameters, as available when abstract syntax is used for generation, is not possible.

Previous works [Arnoldus et al. 2007] indicate that the advantages of meta programs based on abstract syntax (i.e., syntax safety) and the readability of templates using concrete syntax can be combined. We carry this idea on by presenting an approach for automatic derivation of template languages from object language specifications and exemplify this using concrete applications.

Based on a metamodel that describes the abstract structure of a language and a syntax specification which maps the abstract structure to a textual representation, we show how to add the concepts of template languages to arbitrary object languages. To do so, we first discuss design principles for languages to allow extensibility without invasive modification. Then we show how both tool support and language-specific checks (syntactical and semantical) can be transferred to a template language that is an extension of an object language. The interaction between the template concepts and the object language is studied to allow type checking of templates.

The contributions of this paper are as follows. We extend previous work on syntactically safe template languages [Arnoldus et al. 2007] to textual *modelling* languages. A detailed study of metamodel design enables language developers to arrange for extensibility. Our representation of templates, template instances and parameters as models allows to perform more checks at template design time. We discuss how to enhance static semantic analysis at template design time by incorporating the impact of template concepts on name and type analysis in a modular fashion. Additionally, the queries on the parameter model are checked statically to ensure their correctness. To automatically extend arbitrary model-based languages, we present an algorithm which allows the generation of template languages for any textual metamodel-based language. Finally, we show the benefits of reusing the template interpreter.

The remainder of this paper is structured as follows. We introduce a simple toy language in Sect. 2 that will be used to explain our approach. In Sect. 3 we discuss how languages can be designed or

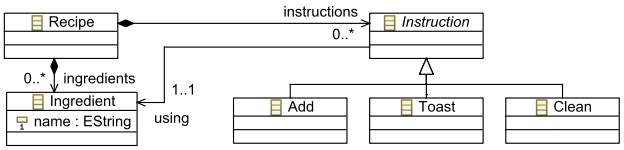

Figure 1. Metamodel for sandwich recipes

refactored to enable unforeseen extensions. Based on this knowledge, we explain in Sect. 4 how a concrete extension (i.e., introducing template concepts) can be performed. Sect. 5 contains details about the application to a more complex language, i.e., Java. There we discuss how a language that was not explicitly designed for extensibility can still benefit from our approach. We compare our work to related approaches in Sect. 6 and conclude with Sect. 7.

2. Introductory Example

To illustrate our approach, we will use a simple example language from the fast food domain. The language was designed to incorporate all aspects that are important to the language extension mechanism which will be explored in this paper. Later on, we will apply the same procedure to Java as an example of a more complex language.

According to the metamodel shown in Fig. 1 our language provides concepts to specify recipes for sandwiches. A recipe consists of several ingredients where each one has a name. The ingredients can then be used in the recipe to give instructions what to do with the ingredients. For the sake of simplicity we only allow to `Add`, `Clean` and `Toast` ingredients. The language contains references with different cardinalities (`ingredients` and `using`). The latter is a cross reference, which will serve to illustrate the static semantic analysis for templates later on.

The textual syntax for the sandwich language was defined using EMFText [Heidenreich et al. 2009]. The text syntax specification is shown in Listing 1. For each concrete metaclass, a textual syntax representation is specified using an EBNF-like rule. An example instance in textual syntax is given in Fig. 2.

Suppose we own a sandwich shop and our customers have individual requests which we want to take into account when making sandwiches. That is, we use customer profiles to produce different kinds of sandwich recipes. Our customers might be vegetarians or request additional ingredients, for example, extra cheese. To produce these different kinds of sandwiches, a custom template version of the sandwich language would be nice to have.

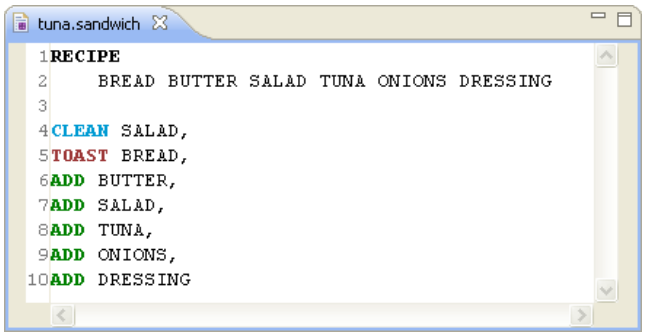

Figure 2. An example sandwich in textual syntax

```
1 Recipe      ::= "RECIPE" ingredients*
2                 instructions ("," instructions)*;
3 Ingredient  ::= name[];
4 Clean       ::= "CLEAN" using[]?;
5 Add         ::= "ADD"   using[]?;
6 Toast       ::= "TOAST" using[]?;
```

Listing 1. Text syntax for sandwich recipes

3. Designing Languages for Extensibility

To turn the sandwich *object* language into a sandwich *template* language, we require the language to be extensible for this purpose. Since we aim for an object-language-independent approach, this section defines the extensibility properties in general but exemplifies them on the sandwich language. More precisely, the extensibility properties are defined on the basis of the artefacts that describe the language to make the extensibility applicable in practice. These artefacts are (according to [Kleppe 2009]): a metamodel to describe the language's abstract syntax, a concrete syntax specification to describe the language's (in our case textual) syntax and a description of the language semantics that operates on sentences of the language (i.e., instances of the metamodel). In the following, we introduce, for each of the three artefacts, rules that have to be fulfilled to support the required extensibility. Furthermore, we introduce an algorithm that can automatically refactor existing artefacts to adhere to the rules where possible.

3.1 Extensible Metamodels

We assume the metamodels defined according to the EMOF [OMG 2006] standard. EMOF is an object-oriented metamodelling language that supports multiple inheritance. Our results can be transferred to any metamodelling language with the same properties.

Since EMOF is object-oriented, the main reuse mechanism of the language is *subclassing*. Since the advent of object-oriented programming, subclassing (or object inheritance) has been discussed controversially in literature—for example, in [Bracha and Lindstrom 1992, Taivalsaari 1996]. Subclassing combines the concepts of *subtyping* and *inheritance*, which makes it easier to implement and to use in certain cases, but has drawbacks in particular when it comes to unforeseen extensibility. While subtyping expresses that subtypes of the same supertype are exchangeable in references defined between the types, inheritance describes that concrete features (i.e., attributes, references and operations) defined by an (abstract) superclass have to be implemented by all concrete subclasses.

In the following, we argue that it is necessary to separate subtyping and inheritance to ensure metamodel extensibility. The subclass concept, as present in EMOF, is however still applicable when used according to restrictions we define below.

Consider the metaclass `Instruction` in Fig. 1. It defines both a type (for the `instructions` reference of `Recipe`) and a feature (the `using` reference to `Ingredient`). Any potential subclass that one introduces as a metamodel extension will subtype `Ingredient` but also inherit the `using` reference. Assume that we want to extend the metamodel with the additional `Instruction` subtype `WrapSandwich`. `WrapSandwich` should be an `Instruction` (subtyping), but should not inherit the `using` reference (feature inheritance) because it does not require an `Ingredient`. Such an extension is not possible with the subclassing currently used in our metamodel.

The separation of subtyping and inheritance however can be emulated by a pattern we call *reference abstraction*. The pattern can be described in an algorithm which can either be followed at language design time or automatically applied on an existing metamodel. The algorithm is a generalisation of the algorithm we

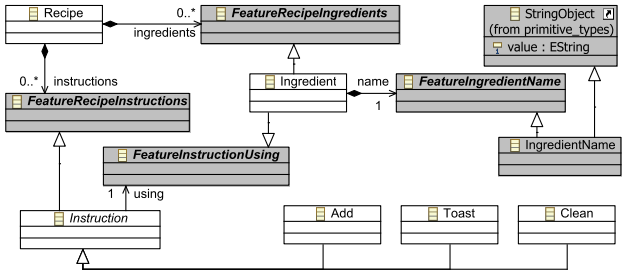

Figure 3. Refactored metamodel for sandwich recipes

presented in [Heidenreich et al. 2008]. Its application is harmless in the sense that it does not change the language itself—it only refines the metamodel to enable extensibility by introducing additional superclasses—and thus does not break existing instances of the metamodel. Consequently, the algorithm can safely be applied on existing metamodels. A general overview of the algorithm is given below:

For each feature's type that has at least one superclass or defines at least one feature itself do:

1. introduce a new abstract metaclass with the name `Feature<ClassName><FeatureName>`[1]
2. change the type of the feature to the new metaclass
3. make the former type of the feature a subclass of the new metaclass

Since subclassing of primitive types (e.g., `String`) is not possible, features with a primitive type require an additional preprocessing step that replaces the primitive type with a metaclass. We call this *primitive type wrapping*. For this, a library of metaclasses that contains wrappers for all primitive types (e.g., `StringObject` with an attribute `value` of type `String`) is provided. Figure 3 shows the complete refactored metamodel for the sandwich language. The new classes introduced by the refactoring are depicted in grey shade.

The refactoring effectively separates the typing of each feature from the feature inheritance hierarchy. That is, each feature obtains an individual pure type since the metaclass through which the type is defined does not define any features itself (step 1 in the algorithm). The type is still connected to the original type hierarchy through the newly introduced inheritance relation (step 3 in the algorithm). This way, the change does not affect the language defined by the metamodel, but opens it for extension since any new metaclass can be plugged in as an additional type for an arbitrary reference by subtyping without feature inheritance.

3.2 Extensible Text Syntax

We assume the language's text syntax to be defined based on the metamodel in a grammar-like fashion as defined in [Kleppe 2009] and implemented in EMFText [Heidenreich et al. 2009]. Such a text syntax is already modularised with respect to the metamodel. It defines one grammar rule for each concrete metaclass—that is for each metaclass that can be instantiated using the syntax. Thus, text syntax extensions can be performed by adding new rules for new concrete metaclasses. The restriction for this new syntax is that it must not introduce ambiguities w.r.t. the existing syntax.

Since a text syntax does not directly refer to any abstract metaclass and the metamodel refactoring only introduces new abstract metaclasses, the impact of the refactoring on the text syntax is minimal. The only issue is the primitive type wrapping that requires adjustments to the syntax specification which can however be performed automatically together with the wrapping in the metamodel. This is illustrated on the syntax specification for `Ingredient.name` (respectively `IngredientName`) in Listings 1 (Line 3) and 2 (Lines 3–4).

[1] Note: This basic version of the algorithm does not handle name clashes.

```
1 Recipe     ::= "RECIPE" ingredients*
2                instructions ("," instructions)*;
3 Ingredient ::= name;
4 IngredientName ::= value[];
5 Clean      ::= "CLEAN" using[]?;
6 Add        ::= "ADD"   using[]?;
7 Toast      ::= "TOAST" using[]?;
```

Listing 2. Refactored text syntax for sandwich recipes

Note that it is never necessary or possible to split a syntax rule to achieve the desired extensibility. In contrast to purely grammar-based languages, the syntax rules here are defined on top of a metamodel. The metamodel defines the concepts of the language, their granularity and how they are composed. Thus, each syntax rule defines the syntax for exactly one language concept (i.e., one metaclass) which can not be further decomposed.

3.3 Modular Static Semantics

So far we only considered syntactic language extension. However, the correctness of the context-sensitive properties of generated code is also of great importance. This demands a modular and extensible representation of language semantics in correspondence to the modularised syntax. Name and type analysis is of particular importance since it is the prerequisite of more sophisticated semantic checks to find duplicate declarations, not initialised variables, or unreachable statements. A modular way to describe name and type analysis for Java using attribute grammars is presented for JastAddJ [Ekman and Hedin 2006]. Attributes are used to specify the propagation of information in abstract syntax trees. EMFText emulates this behaviour for EMOF metamodels in *semantic resolvers* modularly implemented for each cross-reference between metaclasses. The standard resolvers provide a generic implementation for name and type analysis, that works for simple languages which come without specific scopes—such as the sandwich language. To address the sophisticated scoping rules of a language like Java the generic analysis can be refined accordingly by working with nested namespaces.

During language extension one needs to specify the influence of the extension on the name and type analysis of the object language. This can be done by providing specific resolvers for the new concrete classes introduced that reflect their semantic impact on name and type analysis.

While the semantic analysis is implemented in Java for the examples used in this paper, it could also be defined in other formalisms which are interpretable. Such semantics can be used by connecting the corresponding interpreters to the Eclipse/EMF platform as shown, for example, in [Sadilek and Wachsmuth 2009].

4. Adding Template Concepts to Languages

Having examined the generic principles for language extensibility, we now consider the specific addition of concepts found in template languages. To do so, we shortly recapitulate the elements of template languages and then discuss how the textual syntax and the static semantics can be extended along with a metamodel.

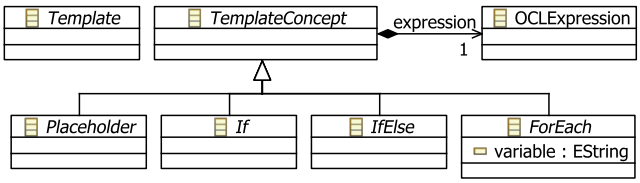

Figure 4. Metamodel for template concepts

4.1 Template Concepts

Conceptually, a template defines an artefact that provides built-in support for variability. In other words, the template can be instantiated to produce different variants. These variants are typically called *template instances*. The selection of a variant is controlled by *template parameters*. These parameters are evaluated when a template is instantiated and based on concrete values the variant is selected.

Among the various template languages that were developed over the last decades, the ones providing simple imperative constructs are very popular especially among practitioners. The common concepts that can be found in such imperative template languages are *placeholders*, *conditions* and *loops*.

Placeholders (`PH` construct) can be used to insert elements originating from parameters upon template instantiation. These elements can be either primitive ones (e.g., strings or numbers) or complex ones (e.g., structures passed as parameters). The latter is obviously only possible if the object and the parameter language share common concepts.

Conditions allow template designers to embed parts of the template only if certain boolean constraints on the parameters are met. Conditions can either consist of one branch (`IF` construct) or two branches (`IFELSE` construct). In the former case, the branch is embedded if the specified condition is true. The latter has a second branch that is embedded if the opposite is true.

Loops (`FOR` construct) can be used to iterate over collections in the parameter model. The body of the loop is then repeatedly inserted into the template instance. Usually, the current element of the iteration is available inside the loop, for example, by accessing a variable.

To extend a language with template functionality the template concepts need to be introduced into the language. Figure 4 shows all concepts as EMOF metamodel. All types are abstract, because they will always be subclassed. In the following two sections, we describe how such an extension by subclassing can be performed.

4.2 Extending the Metamodel

After refactoring existing metamodels as described in Sect. 3.1, one can start to add the concepts of template languages. Here, the question is how the different concepts (`PH`, `IF`, `IFELSE`, `FOR`) can be added in a meaningful way to a metamodel. To answer this question, one must consider the semantics of the template constructs. Upon template instantiation, the different constructs are replaced by elements of the parameter model or static elements of the template itself. To obtain a valid instance of the object language, both the types and the cardinalities of inserted elements must be correct. Thus, the metamodel should be extended such that expansion of template constructs always inserts correct types and a valid number of them.

Preserving Types For each reference (with a corresponding type) we introduce specific subtypes of the template constructs. These subtypes inherit both from the original type of the reference and the abstract template construct. The inheritance relation to the original type establishes the exchangeability of the new template construct with the original (static) element. The inheritance relation to the abstract template construct allows to check and interpret the template later on.

Cardinality	`IF` (0..1)	`PH`, `IFELSE` (1..1)	`FOR` (0..∗)
0..1	x	x	
1..1		x	
0..∗	x	x	x

Table 1. Compatible template concepts for reference cardinalities

Each new subtype of `If` and `ForEach` is equipped with a reference `body`. The subtypes of `IfElse` contain two references (`thenBody` and `elseBody`). All these new references have the original type. This construction allows exactly the elements that were valid for the original reference inside of conditions and loops.

For placeholders, preserving the type is slightly different, because these are replaced with elements from the input model. The obvious consequence is that only types that are shared by both the object and the input language can be used here. Usually the intersection of the types of the two languages encloses only primitive types. However, inserting complex types with a placeholder is also possible as long as its type is available in both the object and the input language.

Preserving Cardinalities To decide which concept is appropriate for which reference we consider the cardinality of the reference. Only constructs that are compatible (i.e., that produce the same number of elements) are used. For example, loops can only be introduced for references that have an unlimited upper bound. The conditional `IF` statement can be introduced for references having a lower bound of 0, because the condition of the `IF` might not be fulfilled upon template instantiation potentially leading to no insertion of an element. Conditionals with two branches (`IFELSE`) can be used for references where the cardinality range encloses 1, because they produce an element in each branch. The same applies to the `PH` concept, because placeholders are always replaced by a single value, i.e., they can only be inserted where one element is permitted by the cardinality of the reference.

A special case is the introduction of conditional statements (both `IF` and `IFELSE`) for unlimited references (0..∗). As both constructs add either zero or one elements, which is permitted by the cardinality, they can be used here too. Table 1 summarises the relation between cardinalities of references and valid template concepts.

In this paper we will restrict ourselves to the kinds of cardinalities shown in Table 1. References with cardinalities having a lower bound greater than one or an upper bound other than one or infinity will simply not be extended with template concepts. This keeps the template language safe, but restricts the metamodel design. In Sect. 5.2.4 we will sketch possible solutions to handle references having other cardinalities as well.

The result of this extension for our example language is shown in Fig. 5. For each type of reference the appropriate template concepts were introduced. By multiple inheritance the concrete concept extends both the abstract concept (defined in the template concept metamodel—Fig. 4) and the type of the respective reference. For example, the class `IfFeatureRecipeIngredients` extends `If` and `FeatureRecipeIngredients`.

4.3 Extending the Concrete Syntax

As discussed in Sect. 3.2, an EMFText text syntax specification can be extended along with the metamodel. For this a new syntax rule has to be introduced for each new concrete metaclass. As discussed above, the extended language contains different subclasses of the four metaclasses representing the template concepts `PH`, `IF`,

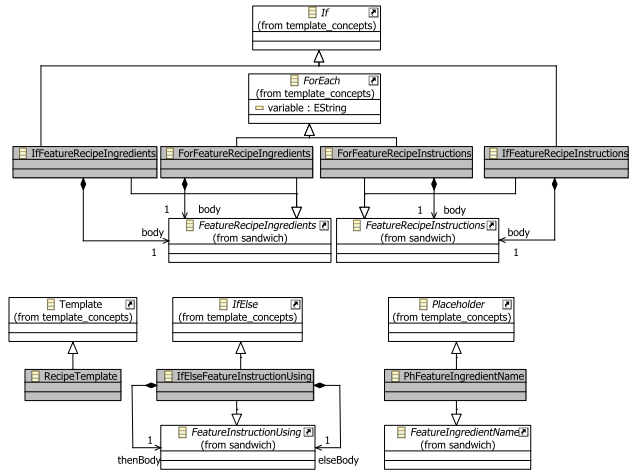

Figure 5. Metamodel for sandwich templates

IFELSE and FOR. The syntax, however, can look similar for all concrete subclasses of each concept.

Listing 3 shows a reusable template syntax specification. It contains one prototypical rule for each template concept. The left-hand side of each rule should identify the metaclass for which the syntax is defined. Thus, a rule can be repeatedly used for all metaclasses that inherit the same template concept (e.g., IF rule for IfFeatureRecipeIngredients and IfFeatureRecipeInstructions) by replacing the left-hand side of the rule with the corresponding concrete metaclass. Figure 6 shows a template defined in the extended syntax.

The rules specified in Listing 3 can be used as prototypes for arbitrary languages extended with template constructs, as long as the new syntax (escape symbols <% and %>) does not interfere with the syntax of the extended language. If it does, a different syntax can be used for that particular case. Another solution for such conflicts is to use a stateful lexer that switches between the object language and the expression language if a delimiter is detected or to use a scannerless parser. Both are, however, currently not provided by EMFText. Nonetheless, the configuration of the escape characters allows to perform the syntax extension for any language. It is important to note here that different escape symbols might be needed for different metaclasses, because the syntax extension can potentially introduce ambiguities.

An interesting thing to point out is that multiple usage of a reference (with multiplicity $*$) on the right-hand side of a rule is handled well by our approach. An example of such a reference is instructions in Listing 1 (Line 2) where it is expressed that all elements except the first one from the instructions list should be prepended with a comma. Such lists, where the first element is to be handled differently than the others, impose problems when templates are written to produce concrete syntax—or instances of a context-free grammar—directly; the first element always requires special treatment. In our approach, where the concrete syntax is discarded after the model is parsed, this is not a problem. A FOR loop can be inserted at any place in the ingredients list to produce additional elements, without caring about the commas in the concrete syntax. When the FOR loop is executed during template instantiation, the concrete syntax was already dropped. It is then correctly created, when the instantiated model is printed into text.[2]

```
1 PH      ::= "<%=" expression[] "%>"
2 IF      ::= "<%IF" expression[] "%>" body "<%ENDIF%>"
3 IFELSE  ::= "<%IFELSE" expression[] "%>" thenBody
4             "<%ELSE%>" elseBody "<%ENDIFELSE%>"
5 FOR     ::= "<%FOR" (variable[] ":")? expression[] "%>"
6             body "<%ENDFOR%>"
```

Listing 3. Reusable template syntax

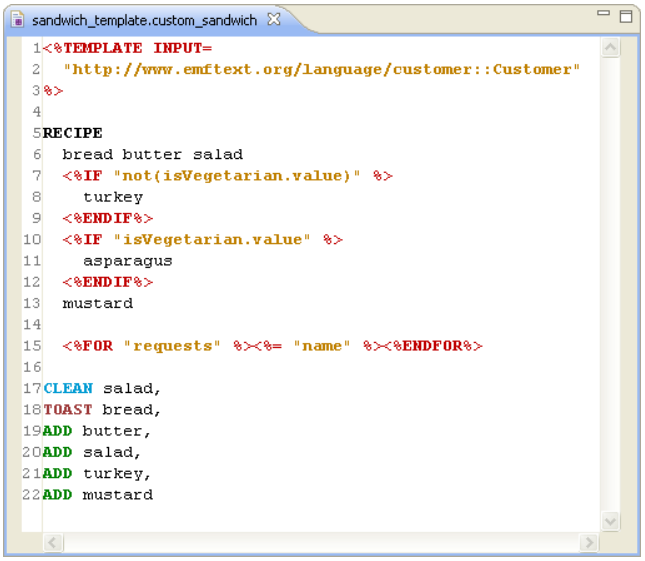

Figure 6. A template for sandwiches

4.4 Extending the Static Semantic Analysis

To reflect the impact of extending a language with template concepts on its static semantics the existing static semantics need to be enriched in correspondence to the classes introduced during extension. As described in Sect. 3.3 this semantics extension can be realised in a modular way. Therefore we generate semantic resolvers for every metaclass reference introduced during language extension. To avoid invalid results during semantics analysis, these resolvers implement a conservative strategy for propagating context-sensitive properties. For metaclasses that extend the template concepts IF, IFELSE and FOR new scope blocks are introduced. Since semantics analysis walks up the containment hierarchy of the parsed model it still behaves as originally defined within each body block of a template construct that inherits the visible names and types of the surrounding block. All name and type definitions made within a template body block are hidden to the outside. They can not be assumed to be addressable from outside the template block, since depending on the template parameters they may not be included in the generation result.

The described strategy can lead to false negatives, that is failing name or type resolutions that should succeed. An example is an IFELSE template which declares an identifier with the same name in both branches. In this case the identifier is definitely included in the generation result and can thus be assumed visible in the surrounding block. It is also challenging to check more complex semantic properties like duplicate declarations or unreachable statements, since such problems are highly language specific and many other constellations need to be considered. Currently, we address such issues by refining the generated semantics resolvers manually and specific for every language extension. Future work will have

[2] EMFText does not only generate a parser, but also a printer based on the (extended) text syntax specification which is utilised to print template instances after instantiation on the model level (cf. Sect. 4.7).

to investigate how techniques used, for example, in attribute grammars can be transferred to generalise the idea of modular static semantic analysis.

4.5 Restricting the Extension

While we discussed the whole set of possible extensions that can be performed to obtain a template language in the previous sections, there are reasons to restrict this extension process. First, the variability that is introduced by the template concepts might not match the expectations of the template designer. Too much variability, which might be caused by the amount of introduced concepts, can confuse developers. A restricted set of constructs for parts of the language where variability is actually needed might be better suited. Second, the option to insert loops, conditions, and placeholders wherever theoretically possible, allows template designers to write deeply nested and complex templates, which are hard to maintain.

To avoid these problems, a selection process can be employed that allows to tailor the language extension. By manually selecting the references for which variability must be provided, a custom template language can be obtained. The general extension process mentioned before is thus executed identically, but only for a subset of the metamodel, the syntax specification, and the static semantics analysis.

4.6 Extending the Tool Support

When a language is extended with new concepts, the impact on tools that operate on the language must be carefully analysed. Ideally, existing tools should continue to work. In the case of textual languages important objectives of such tools are parsing, editing and analysing. When it comes to extending these tools, metamodelling and generative approaches come in handy.

In our particular case of introducing template constructs to languages, we can generate an extended parser, an enhanced editor and a default name resolving mechanism for the new language concepts with EMFText. Thus, typical aspects of editing (e.g., syntax highlighting and code completion) are generated from the extended language specification (see Fig. 6 for an example of syntax highlighting derived from the base language). Furthermore, code generated from the metamodel extensions is based on the existing metamodel code through which semantic analysis continues to work to the degree discussed in Sect. 4.4.

In addition to the above, EMFText also generates a printer for the extended language. This printer converts model representations of templates into their textual representations. Although not in focus of this paper, this enables the creation of templates by model transformation or step wise instantiation of templates—thus a deeper integration of templates into model-driven development processes.

4.7 Instantiating the Templates

Since each template metaclass in an extended language inherits from one template-concept metaclass (cf. Fig. 4), an object-language independent template interpreter can be provided. This interpreter takes a template model—that is, the model presentation of the template obtained using the extended parser—and an input model. The interpreter then walks the containment hierarchy of the template model. It ignores all elements whose types do not inherit one of the template concepts. When it encounters a template element—that is, an element whose metaclass extends a template concept—it interprets it following the semantics of the corresponding template concept as sketched in Sect. 4.1.

The instantiation for each template element is performed by replacing the element itself with elements from the input model (in the case of `PH`) or elements from a `body` reference of the template element (in the case of `IF` or `IFELSE`). These elements are determined by the value of the expression attribute (cf. `TemplateConcept.expression` in Fig. 4). This value is interpreted as an Object Constraint Language (OCL) expression. We decided to use OCL as query language over the input models, because these models are (as the templates themselves) EMOF models that can naturally be queried by OCL. The queries are passed to the MDT OCL Interpreter[3] together with the input model. The interpreter returns the element or value addressed by the query. Our template engine then replaces the template element it currently interprets according to the query result. Note that an OCL expression always operates in the context of one model element. In the global context of a template instantiation it is the root element of the input model. In the context of a `FOR` it is the current element of the iteration.

In the end, the template model itself was turned into an instantiation of the template. All template elements were removed or replaced with their body or with input model elements. This model can be printed into text by the printer that was also generated from the object language's syntax specification by EMFText.

5. Safe Templates for Java

To evaluate our language extension mechanism on a real world example, we chose Java as it is widely used and well known. Based on EMFText, JaMoPP[4] provides both a metamodel and a text syntax specification for Java. In addition, JaMoPP contains a set of reference resolvers that implement name and type resolution rules similar to attribute grammars. These artefacts served as a basis for our template extension. In this section we will first discuss the problems that emerged from the extension and then explain the key benefits of generating a safe template language for Java. In particular, the types of properties that can be guaranteed (i.e., the level of safety) are discussed along with application domains that particularly benefit from these guarantees.

5.1 Extending Java with Template Concepts

In Sect. 4 the general approach to extend languages to obtain a template language was presented. Using the sandwich language the basic steps were explained. In principle, the same procedure is also applicable for larger languages. However, the Java language (and its concrete implementation JaMoPP) raised some issues, which we will discuss in this section.

5.1.1 Dealing with Existing Metamodels

Since the JaMoPP metamodel was not explicitly designed for extensibility, the refactorings described in Sect. 3 needed to be applied to obtain a more extensible version of the metamodel. The set of refactorings can be split into two groups—reference abstraction and primitive type wrapping.

The first type of refactoring is harmless from the perspective of model-based tools. For example, loading model instances that were created before the refactoring is unproblematic. Even EMF-based tools, such as editors or viewers continue to work. Other tools, that directly rely on the code that was generated from the metamodel may break after the refactoring. Consider, for example, the introduction of an abstract type for a reference that used a concrete type beforehand. Code that accesses the reference must cast to the concrete type after the refactoring even though there might only be one valid subtype. As JaMoPP is accompanied by a set of classes that use the generated metamodel code, we had to adapt these according to the type changes. In theory however, the

[3] http://www.eclipse.org/modeling/mdt/ocl

[4] http://jamopp.inf.tu-dresden.de

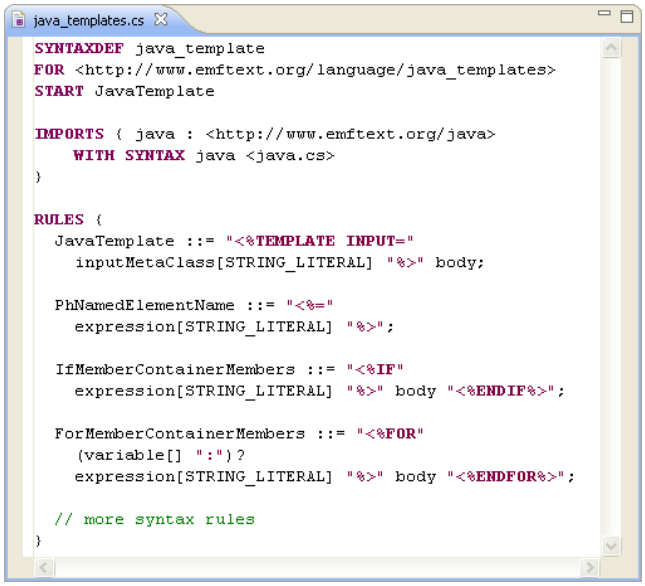

Figure 7. Extract from the syntax specification for Java templates

refactoring of the code—that is the introduction of type checks and casts—can be automated similar to the metamodel refactoring.

The second step towards a more extensible Java metamodel is wrapping primitive types. This change is more invasive since it introduces an additional forward which may invalidate existing model instances that were serialized using a standard syntax based on the metamodel (such as XMI). However, this is not a problem when models are serialized in their text syntax, as it is the case for Java where model instances are saved as Java source code. It was sufficient to refactor JaMoPPs concrete syntax definition according to Sect. 4.3 to parse Java code into instances of the refactored metamodel. Nonetheless, the existing code that works on instances of the metamodel had to be adapted to work with the refactored metamodel. To give an example, the class `NamedElement` contained an attribute `name` of type `String`. This attribute was replaced by a reference to a wrapper class called `NamedElementName`. The code that accessed the attribute `name` directly had to be modified to use the wrapper object instead. This code refactoring—changing each access to a primitive type to use the corresponding wrapper—can be automated as well.

5.1.2 Extending Syntax and Semantics

To introduce the text syntax for the new classes representing template concepts, we imported the original Java syntax provided by JaMoPP and generated one concrete syntax rule for each new class. Each template concept uses the same syntax (cf. Listing 3). However, since each concept is instantiated once for each compatible reference, the same syntax must be added multiple times. A fragment of the syntax for Java templates is shown in Fig. 7.

After refactoring the metamodel to be extensible, the syntax extension was straightforward and executed fully automatically. This was partially due to the choice of the new syntax, that did not interfere with Java's original syntax. To extend other languages without difficulty as well, we pass the choice of the syntax to the template language designer. Thus, one always obtains a parseable template language. If the initially chosen syntax interferes with the original language, the generator informs the language developer about that such that he can modify the template syntax until all ambiguities are removed. After generating the syntax extension for the reformed metamodel, we used EMFText to regenerate tool support for the Java template language. Thus, a parser and an editor (including syntax highlighting) were instantly available.

To preserve the static semantic analysis for name and type resolving, no additional analysis module for template concepts was required. This is because the name and type resolution walks up the parse tree to search for declarations of the element to be resolved and skips nodes of unknown type during this process. Instances of the new template constructs are therefore ignored and the search is continued further up the tree. Consequently, the name and type resolution does not search for declarations inside of `IF`, `IFELSE` and `FOR` elements. This is correct, because the values of the conditions and collections that control the existence of the elements in the body in the template instance is not known at template design time. Elements that are declared inside of these elements can therefore not be assumed to be present in every possible template instance.

Other static semantic properties (e.g., detection of duplicate declarations) must be extended with custom logic to take the template concepts into account. For example, an element that is declared inside a `FOR` loop—without using placeholders in the declaration—will most likely be declared several times in many template instances; even though the original analysis does not detect this in the template. Here, language-specific extensions of the static semantics are inevitable. In the future we will explore this further on the example of Java.

5.2 Using Java Templates

Now that an extended version of Java with template concepts was obtained, the pros and cons of using such a language can be discussed. The checks that can be performed both on the semantical and syntactical level are highlighted in this section.

5.2.1 Static Syntax Checks

In contrast to template languages that are agnostic of the object language, building a custom template language has several advantages. One of which is the possibility to check the syntax of templates. By doing so, one can make sure that all instances of a template conform to the syntax of the object language. This approach was shown earlier [Arnoldus et al. 2007] and extended in this paper to treat textual modelling languages. The specific construction of an extended syntax (shown in Sect. 4.3) carefully extended the Java language such that template instantiation must yield a valid object sentence.

The advantages of this procedure are especially important in application domains where either the syntactical correctness of the instantiation result can not be easily checked or wherever the template cannot be modified if an error is found. Both situations occur when Java templates shall be reused or supplied for reuse. Both the supplier and the consumer of third-party templates depend on guarantees about the syntactical correctness of template instances. With the Java template language based on JaMoPP, templates can be safely distributed with a guarantee about syntactical correctness. As this basic syntactic correctness was shown in [Arnoldus et al. 2007], we will explore which additional checks can be performed on Java templates, that exceed the syntactical level.

5.2.2 Static Semantic Checks

In Sect. 4.4 we discussed how the modular specification of static semantics can be integrated with the language extension on the syntactical level. As static semantics is concerned with non context-free aspects of a language, it is most important for languages that intensively make use of such aspects. Java, being an object-oriented language, has lots of constraints that involve context. One typical example for an analysis that must consider context is name resolution. Method calls, variable accesses and type references must be resolved to their respective declarations with the correct name

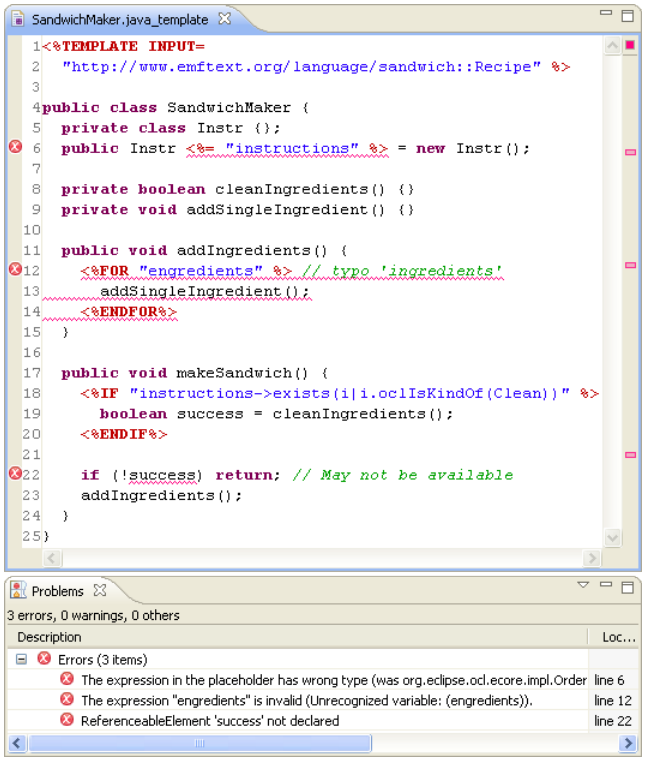

Figure 8. Errors detected in a Java template

or identifier. As programming languages heavily rely on such relations between declarations and usage of elements, checking that these references are valid for all instances of a template substantially decreases the number of errors found upon template instantiation time. In Fig. 8 an example for an error that is detectable by static analysis of Java templates is shown. In Line 22 the variable `success` that was declared inside of an `IF` is referenced. As this variable may not be available in all template instances—namely not the ones where the condition is false—an error is reported.

As argued for the syntactic checks, the reuse of templates can particularly benefit from early semantic checks. However, the extension of the semantics can not be automated as much as its syntactical counterpart. This is basically due to the nature of semantics, which are highly language specific. Only common semantical concepts, such as name resolution, can be automatically extended.

5.2.3 Static Input Model Checks

As stated in Sect. 4.7, we use OCL as query language over the input models. To ensure the correctness of all queries contained in a template (i.e., placeholder expressions, collection selectors and conditions), a reference to the metamodel of the input must be given. As all queries refer to this metamodel—one cannot evaluate queries over unknown models—an explicit declaration seems feasible. This specification is done at the very beginning of a template by declaring a special element. Using this declaration, we can check the OCL queries statically against the metamodel.

For example, accessing types or references that are not present in the metamodel is not possible, as shown in Fig. 8. In Line 12 an expression contains a typo which is correctly reported as an error, because the input model does not contain a feature called `engredients`. A second error is reported for Line 6 where the type of the expression (`OrderedSet`) is not compatible with the position of the placeholder. Instead of a set, a primitive string or a `StringObject` is expected. These checks allow to safely port templates from one input metamodel to another. After changing the declared input metamodel the queries are checked and developers can correct erroneous expressions.

5.2.4 Template Language Restrictions

The previous sections showed the application of language extension to introduce template concepts for arbitrary languages, explained the realisation of a generic interpreter to generate template instances and discussed different advantages of that approach. However, the drawbacks and limitations also need be recapitulated. A first restriction is that the interpreter interprets each construct locally and no environment to define global variables during instantiation is available. This could be helpful for more convenient access to the input model in particular in nested loops, where currently only the innermost element is directly accessible. Adding this is straightforward since the OCL environment allows the declaration of variables.

A second limitation is that our template languages do not offer possibilities to define functions for computations on the input models inside a template. This limitation is however desired as discussed in [Parr 2004] to separate logic and template code. It implies that the input model must explicitly contain all information that needs to be embedded in the template instance. This does also apply to simple generation tasks such as creating names for getters and setters. Adding the prefixes `get` and `set` can not be done in the template, but needs to be performed externally. The Eclipse Modeling Framework, for example, uses dedicated generator models to provide derived names. Another solution for this problem is to use external OCL `def` expressions for input model metaclasses to perform such small localised computations.

A third limitation can be observed when metamodels use cardinalities other than the ones mentioned in Sect. 4.2. For example, a reference having a minimum cardinality of 2 can not be extended with a template concept yet. None of the four concepts is strictly compatible with such a reference. However, checking that every template contains at least two static elements (or placeholders and `IFELSE` elements) can solve this problem and allows to use template concepts for such a reference. The same applies to fixed maximum cardinalities. Extending the `FOR` concept with a maximum loop count can ensure that the maximum number of elements created by the loop is not exceeded.

A similar restriction is caused by the need for type safety. Currently, placeholders can be replaced with elements of the same type only. While this type might even be a complex one, type conversions of complex types are not investigated yet—only primitive types and primitive type wrappers can be converted by the engine. Extending the template engine with information about compatible types and conversions between them could certainly ease using the templates. String-based engines do not face this problem since all elements basically have the same type.

6. Related Work

There exists a number of language-specific extensions that add template (or template-like) mechanisms for specific programming languages. For example, MorphJ [Huang and Smaragdakis 2008] or ConceptC++ [Gregor et al. 2006]. While these approaches go much further regarding language specific semantic analysis, they always have a fixed object and input model language and miss a common base. Our approach could serve as a common base for similar approaches in the future. This can ease development, prototyping and implementation. It can also help to find commonalities or port language-specific solutions to other languages.

Text-based template engines like *StringTemplate* [Parr 2004] or *XVCL* [Jarzabek et al. 2003] provide template languages that

are object-language-agnostic in the sense that they solely process strings. They are not aware of the type system and syntax of the object language. *StringTemplate* inspired our work with regard to restricting the available constructs in the template language so that logic cannot be mixed with template code (e.g., changing the input model from the template or compute values within the template). In contrast to text-based template engines, we do not operate on strings but on typed models conforming to metamodels.

Repleo [Arnoldus et al. 2007] is a syntax-safe template engine that is based on the Syntax Definition Formalism (SDF) [Heering et al. 1989] and the term rewriting system ASF+SDF [van den Brand et al. 2001]. *Repleo* uses SDF's module mechanism to explicitly define extensions of the object language with template-language specific constructs and the SGLR parser [Visser 1997a] to parse the combined language. Based on the strongly typed equations of ASF, the syntactic correctness of the expanded templates can be ensured. In contrast to our work, *Repleo* does not cover static analysis of templates and cannot guarantee type correctness of the constructs used within templates. Since we use OCL for querying in templates, ill-formed references to concepts of the input metamodel are detected at template definition time, whereas *Repleo*'s XPath-like queries are not checked against any metamodel. Furthermore, our metamodel-based approach and the algorithm presented in Sect. 4 enables us to automatically extend existing languages with template concepts and generate tooling for the extended language (e.g., editors with syntax highlighting).

In his article on Language Oriented Programming from 2004 [Dmitriev 2004], Dmitriev describes that an automatic template language generator that copies the object language and adds template concepts to it. This generator is included in JetBrains MPS [JetBrains 2009]. MPS was however just released during the writing of this paper. So far, only the *MPS base language*, which is a base for building languages in MPS that is very close to Java, comes with build-in template constructs. It is not clear how easy it is to integrate a completely new base language into MPS, nor what the conditions for this language are to allow the template language generator to work with them.

SafeGen [Huang et al. 2005] is another approach to create safe generators for Java programs based on a theorem prover that proves the well-formedness of the generated code for all possible inputs. Similar to our approach both syntax and type correctness are ensured. But in contrast to our work—which intrinsically works for all existing languages with the properties described in Sect. 3.1—SafeGen is designed to require Java programs as input and produces Java programs as output.

The XML Template Language (XTL) [Hartmann 2006, 2007] is a template engine that enables safe template authoring with XML languages that are described by XML Schema. The available constructs in the template language are similarly restricted based on [Parr 2004] to ensure safety properties of templates. Unlike our approach, XTL is limited to languages with XML syntax and does not generate any tooling for the extended object language.

MetaBorg [Bravenboer and Visser 2004, Bravenboer et al. 2006] is a system for embedding domain specific languages in arbitrary host languages. It is not a template engine per se but the concepts embodied by MetaBorg can be used to embed a domain-specific language (DSL) featuring template constructs into a host language. In MetaBorg, the DSL constructs and their embedding in the host language are described with SDF2 [Visser 1997b, Heering et al. 1989]. Syntactic errors in the program using the combined language are detected during parsing. Semantic errors in programs written in the combined language are detected by a manually extended type checker of the host language which supports the semantics of the embedded language and the bindings between the host language and the embedded language. In an assimilation phase—which is implemented with the program transformation language and toolset Stratego/XT [Bravenboer et al. 2008]—the embedded fragments are translated to the host language. It is not guaranteed that the assimilation phase always produces well-formed code.

All previous mentioned approaches are not based on a common implementation platform but used specific and diverse supporting technologies for implementation. As we base our work on the Eclipse/EMF platform, we also want to discuss existing template approaches that work based on Eclipse technologies. Prominent examples of tools for model-to-text transformations are the Epsilon Generation Language (EGL) [Rose et al. 2008], MOFScript [Oldevik 2006] and Xpand [oAW Project Team 2009]. These tools take EMF models as input models and produce text based on the parameterised templates. The templates are again basic mixtures of escaped template constructs and text where neither syntactic nor semantic errors can be detected during template definition. In contrast to these approaches, we use model-based syntax and the underlying metamodels to create templates which produce well-formed output in the object language.

7. Conclusion

The contribution of this paper is threefold. First, we investigated the fundamental requirements to prepare languages for extensibility. We introduced general design principles to prepare metamodels for syntactic and semantic extensibility and a technique to automatically incorporate these design principles into existing metamodels. Second, we applied the extension technique to introduce template concepts for an exemplary language while preserving compatibility with existing tools (e.g., parsers and editors). The result is a template language that is safe w.r.t. syntax and properties of static semantics (e.g., name and type analysis). Finally, we evaluated the extension mechanism and its limits by applying it to generate a template language for the general-purpose programming language Java.

The generic interpreter that performs template instantiation operates on typed models. Replacing placeholders, inserting bodies of loops or conditions is performed on the respective models (i.e., input model, template model, and template instance model). This supplies features that can not be accomplished with engines that use basic string concatenation. Future work will explore how development processes, where traceability is of utter importance, can employ our template interpreter to track the connections between elements of the input model, the template, and the template instance. Another application, which strongly depends on correct and up-to-date tracing information, is certification where verified system properties are propagated along the tracing chains. Finally, to overcome the limitations of traditional forward oriented code generation approaches Round-trip Engineering can also benefit from tracing changes made to generated code back to templates and the corresponding elements in the input models.

The design principles for language extensibility are not only relevant for building safe template languages. However, it is hard to force language designers to conform to the design principles for language extensibility. To give language designers more control where and how to apply language extensions we plan to adapt our automatic extension technique to be more configurable. We argue, that the extension technique strongly depends on the language extension to realise. Future work in this direction has therefore two objectives. First, language extension scenarios need to be classified to construct a language extension framework that provides means to address the specific requirements of each extension scenario. Second, extensions for existing metamodelling languages to provide built-in support to design extensible languages need to be explored.

Acknowledgments

This research has been co-funded by the European Commission within the FP6 project MODELPLEX #034081, the FP7 project MOST #216691 and by the German Ministry of Education and Research within the projects feasiPLe and SuReal.

References

Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. Repleo: a Syntax-Safe Template Engine. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE 2007)*, pages 25–32, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8.

Gilad Bracha and Gary Lindstrom. Modularity Meets Inheritance. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, 1992.

Martin Bravenboer and Eelco Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation without Restrictions. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, pages 365–383. ACM, October 2004. ISBN 1-58113-831-8.

Martin Bravenboer, René de Groot, and Eelco Visser. MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. In R. Lämmel and J. Saraiva, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, volume 4143 of *Lecture Notes in Computer Science*, pages 297–311, Berlin, Heidelberg, 2006. Springer.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. ISSN 0167-6423. Special issue on experimental software and toolkits.

Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm. White Paper, JetBrains, 2004. URL http://www.onboard.jetbrains.com/is1/articles/04/10/lop.

Torbjörn Ekman and Görel Hedin. Modular name analysis for Java using JastAdd. In R. Lämmel and J. Saraiva, editors, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, volume 4143 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2006. Springer.

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA 2006)*, pages 291–310, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4.

Falk Hartmann. An Architecture for an XML-Template Engine Enabling Safe Authoring. In *Proceedings of the 17th International Workshop on Database and Expert Systems Applications (DEXA 2006)*, pages 502–507. IEEE Computer Society, 2006.

Falk Hartmann. Ensuring the Instantiation Results of XML Templates. In *Proceedings of the IADIS International Conference WWW/Internet 2007*, 2007.

Jan Heering, P.R.H. Hendriks, Paul Klint, and Jan Rekers. The Syntax Definition Formalism SDF—Reference Manual—. *SIGPLAN Not.*, 24 (11):43–75, 1989. ISSN 0362-1340.

Florian Heidenreich, Jakob Henriksson, Jendrik Johannes, and Steffen Zschaler. On Language-Independent Model Modularisation. *Transactions on Aspect-Oriented Development, Special Issue on Aspects and MDE*, 2008. To Appear.

Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and Refinement of Textual Syntax for Models, Accepted for publication. In *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, 2009.

Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with morphj. *SIGPLAN Not.*, 43(6):79–89, 2008. ISSN 0362-1340.

Shan Shan Huang, David Zook, and Yannis Smaragdakis. Statically Safe Program Generation with SafeGen. In Robert Glück and Michael R. Lowry, editors, *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 2005. ISBN 3-540-29138-5.

Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. XVCL: XML-based Variant Configuration Language. *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 810–811, May 2003.

JetBrains. JetBrains Meta Programming System (MPS), 2009. URL `http://www.jetbrains.com/mps`.

Anneke Kleppe. *Software Language Engineering*. Pearson Education, 2009. ISBN 0321553454.

oAW Project Team. openArchitectureWare, 2009. URL `http://www.openArchitectureWare.org`.

Jon Oldevik. MOFScript Eclipse Plug-In: Metamodel-Based Code Generation. In *Eclipse Technology Workshop (EtX) at ECOOP 2006*, 2006.

OMG. MOF 2.0 core specification. OMG Document, January 2006. URL `http://www.omg.org/spec/MOF/2.0`. URL http://www.omg.org/spec/MOF/2.0.

Terence John Parr. Enforcing Strict Model-View Separation in Template Engines. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th International Conference on World Wide Web (WWW 2004)*, pages 224–233. ACM, May 2004. ISBN 1-58113-844-X.

Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. Polack. The Epsilon Generation Language. In *Proceedings of the 4th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2008)*, pages 1–16, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-69095-5.

Daniel A. Sadilek and Guido Wachsmuth. Using Grammarware Languages To Define Operational Semantics of Modelled Languages. In *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS 2009)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 348–356. Springer, 2009. ISBN 978-3-642-02570-9.

Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.

Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of the 10th International Conference on Compiler Construction (CC 2001)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer, 2001. ISBN 3-540-41861-X.

Eelco Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997a.

Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997b.