Abstract

Model-driven engineering, especially using domain-specific languages, allows constructing software from abstractions that are more closely fitted to the problem domain and that better hide technical details of the solution space. Code generation is used to produce executable code from these abstractions, which may result in individual concerns being scattered and tangled throughout the generated code. The challenge, then, becomes how to modularise the code-generator templates to avoid scattering and tangling of concerns within the templates themselves. This paper shows how symmetric, language-aware approaches to aspect orientation can be applied to code generation and how this improves modularisation support, in particular reducing dependencies between code-generation templates, over previously proposed asymmetric aspect-oriented approaches.

1 Introduction

In model-driven engineering (MDE) [1], domain-specific languages (DSLs) [2] are typically accompanied by code generators, which expand the abstract DSL concepts into source code in a target programming language, typically taking into consideration a particular target platform (e.g., middleware, hardware, distribution and concurrency model, etc.). Such expansions can be limited to the generation of standard, ‘boiler-plate’ code, but, more interestingly, may include so-called local-to-global or global-to-local transformations [3]. A local-to-global transformation means that information from one element in a DSL program may be scattered across multiple elements in
the generated source code—for example, a data description may affect user-interface code as well as data-definition scripts for setting up a database back end. A global-to-local transformation, on the other hand, means that information from a number of DSL elements may need to be tangled within one element in the generated source code—for example, both information from a layout policy and the data description must be tangled in user-interface code generated.

The challenge in the presence of local-to-global and global-to-local transformations is how to cleanly modularise the code-generation templates so that, while tangling and scattering occur in the code generated, the templates themselves do not suffer from it. Moreover, the code-generators themselves may need to address a range of technical concerns, such as consistency management [4], performance, security, different target platforms, etc. Of course, we would also like to avoid scattering and tangling of technical concerns in generation templates. This becomes particularly important where DSLs should be reused in different projects, which may require to address some technical concerns in a different manner or may require adaptations to the DSL concepts provided. In this case, generator modularisation becomes essential, as it enables a "mix and match" approach to DSL adaptation and reuse [5]. Ideally, we would like to modularise code generators such that each technical concern as well as each concern modelled in a DSL could be realised in a separate code-generation module as independently as possible of the other concerns.

Existing code-generation frameworks (e.g., [6–9]) assume that one target file will be produced by one code-generation template (even though this may import and invoke additional generation rules). Thus, any scattering and tangling that occurs in the generated code automatically also occurs in the generation templates. To address this issue, aspect-oriented approaches to code generation have been proposed [10,11]. However, they are not fully adequate for all needs of code-generator modularisation: Because they are asymmetric approaches [12], they require an explicit base template, often imply the use of scaffolding (e.g., empty rules in the base template whose only purpose is to serve as hooks for the aspect templates), and lack sufficient support for weaving contexts. Furthermore, because they are language-agnostic, any language-specific weaving semantics must be implemented in the generation templates, which is not always possible.

This paper proposes a novel approach for modularising code generators. This approach maintains the benefits of reduced tangling introduced by [10,11], but also addresses the short-comings of these approaches:

- It is a symmetric aspect-oriented approach, addressing the issues connected to asymmetry in current approaches.
- It is also a language-aware approach to address the issue of language agnosticism.

Thus, the contributions of this paper are the following:

1. We identify four shortcomings of existing asymmetric AO approaches to code generation that stand in the way of effective separation of concerns for code generators.
2. We present a novel approach and architecture for symmetric language-aware AO for code generation, which addresses the aforementioned shortcomings.

3. We present a prototype implementing this architecture.

The remainder of this paper is structured as follows: In the next section, we give a more detailed motivation for the need of symmetric aspect-oriented code generation based on the identification of four issues with current symmetric approaches. In Sect. 3, we present a generic architecture for symmetric aspect-oriented code generation, discuss specific issues involved in choosing an appropriate merge algorithm (this being the key challenge in applying symmetric aspect orientation to code generators), and present a prototype implementing these concepts based on the Epsilon Generation Language (EGL) [13]. In Sect. 4, we demonstrate our approach with an example generating code for an application based on Enterprise Java Beans (EJBs) [14]. Section 5 discusses the benefits and drawbacks of this novel approach. Section 6 surveys some related work and Sect. 7 concludes the paper.

2 Motivation: Modularising Code Generation Templates

Many code-generation frameworks, such as [6–9], already provide notions such as generator rules and operations. These are very useful for modularising a generator based on the structure of the files to be generated. However, information that is localised in one place in a model often needs to affect multiple places in the generated code. Thus, the knowledge of how to implement a model element is scattered throughout the generated code. As a consequence, unless specific measures are taken in modularising the code generators, this knowledge will also be scattered throughout the code generators. Furthermore, information from multiple model elements may need to be combined to generate a particular file. Thus, the generated code tangles a number of concerns from the DSL models. Unless specific measures are taken in modularising the code generators, these concerns will also be tangled within the code generators.

Scattering and tangling in code generator templates is particularly bad where we want to flexibly configure a generation workflow so that it can be adapted to different circumstances. This can happen, for example, when a DSL is to be reused in a different context [5], or when the system context changes (e.g., a new version of an underlying platform is released and must be integrated with the system under development).

For example, when generating Eclipse [15] plugins from an application model, a number of different concerns must be addressed by these generators: i) generating the Java implementation of the application’s business logic, ii) generating code implementing the data model, and iii) generating user interface code, among others. It is immediately clear that there is a number of local-to-global transformations in this scenario—for example, information about a data structure affects code in the user interface, the data model, as well as, possibly, the business logic code. Furthermore, the are a number of tangled concerns—for example, both user-interface code and data-model code may have to provide configuration information, which means these concerns will be tangled in Eclipse’s plugin.xml configuration file. Separating these concerns into distinct generation modules, would enable us to choose from different variations
Figure 1: An example decomposition of code generators for generating Eclipse plugin code. The figure shows two possible configurations: a) using EMF for data management as well as including a graphical user interface, and b) using plain Java objects for data management and providing a programming interface only.

for implementing each concern. Figure 1 shows an example of how we may want to decompose the code generators. In particular, we have defined one generator for the user-interface concern, one for the concern of business logic, and two different variations for the concern of data-model implementation—one using EMF [16] and one using plain Java objects. The figure shows two possible configurations making specific choices about these code generators. It can be seen how each of the generators uses information from the application model (which could be provided using a number of DSLs) and generates code in one or more files; that is, concerns are scattered throughout the generated code. Some code generators also need to provide code to the same file; that is, different concerns are tangled within these files.

In summary, we require a modularisation of our code generators, such that each concern can be realised in a separate code-generation module as independently as possible of the other concerns, even if these concerns need to be scattered and tangled in the generated code. Ideally, this should enable us to modify, remove, or add a code-generation module without impact on any of the other modules in our code generator.

To support the modularisation of tangling in code generators, some authors have proposed using aspect-oriented techniques in the definition of code-generator templates [10, 11]. These are asymmetric aspect-oriented approaches [12] allowing the definition of around, before, and after advice for individual code-generation rules or operations in a base template.\(^1\) While this can successfully solve some of the modularisation issues, it also has important drawbacks:

1. **Base template needed.** Every aspect template requires a base template against which it is defined and which it modifies. This makes it difficult to define code generators for files that do not need to exist for all variants of a system. In the example of Fig. 1, depending on which concerns we include in the generation, not all of the plugins generated will actually require a plugin.xml file. At the same time, a number of concerns will have to make contributions to

---

\(^1\)In aspect-oriented software development, we can distinguish two essentially different types of approaches: Symmetric approaches make no formal distinctions between aspects and non-aspect parts of the system, allowing them to be composed freely. Asymmetric approaches on the other hand distinguish aspects and components (i.e., elements of the base system); aspects can be woven into the base (and sometimes into other aspects as well), but there always has to be a base and it cannot directly influence any of the aspects.
plugin.xml if they are selected. Using asymmetric aspect-oriented code-generation techniques, we need an empty code-generation base template for plugin.xml, even though this file would often not be needed. Note that, because we want to allow free choice of code generators and because this selection may be made based on information in the application model rather than based on a static configuration choice of the generation workflow, we cannot guarantee a priori that any specific code generators will be used in a concrete workflow, and, thus, cannot make any generator the base generator.

2. **Scaffolding needed in base template.** Because every advice needs to be attached to some rule or operation in the base template, we can only influence those parts of a file for which explicit code-generation rules have been defined. This typically leads to the definition of a number of empty generation rules, which only serve as hooks to be referenced in aspect templates adding additional code to the file generated.\(^2\) For example, in Fig. 1, the empty base template for plugin.xml needs to provide empty rules for generating extension and extension-point descriptions, even though we do not declare any of them in the base template. The only purpose of these rules would be to enable aspect templates to add extensions and extension points of their own. The need for such additional constructs in base code has been previously identified—for example, [11, p. 9] shows an example of such an empty generator rule (the fact that [11] in principle allows aspects on aspects does not strongly affect the need for scaffolding). In [17] such rules have been referred to as scaffolding. They have been shown to contribute strongly to the instability of pointcuts and to break encapsulation of base and aspects.

3. **Insufficient support for weaving contexts.** In the example, different generators affect the final Java code. Some of these generators may need to add implemented interfaces to certain classes (e.g., to be consistent with requirements of frameworks used). This demonstrates another problem of asymmetric aspect orientation for code generation. As the only weaving modes supported are before, after, and around, correct generation of implemented interfaces becomes impossible as soon as more than one aspect needs to add to this list unless the base template already provides a non-empty list. The first aspect to be applied would need to introduce the implements keyword (we will call this the weaving context). All following aspects must not produce this weaving context. Of course, one could extend the aspect languages to support advice ordering and use ordered post-advice only. Still, if one decided not to generate code for the first feature (where the aspect introduces the implements keyword) the setup would break and one would have to change the code of another aspect to make it work again.

In addition to these drawbacks mainly caused by the asymmetric nature of current approaches, there is also an issue because these approaches are language agnostic. Because the code generators and aspect weavers have no knowledge of the language to

---

\(^2\) Oldevik and Haugen [10] describe this for their approach: "[...] there must be constraints on the base transformation [...] e.g. that domain-specific rules should be clearly separated from general rules").
be generated, the weaving result can easily be wrong. In the interface-implementation example from above, it could, for example, easily happen that the same interface is named more than once in the list of implemented interfaces that is generated.

To address these issues, we propose a novel aspect-oriented approach to code generation. This approach is symmetric [12], addressing the shortcomings related to the asymmetric nature of current approaches. Our approach, further, is language aware, so that it can provide better weaving based on language syntax and semantics. The following section discusses our approach from an architectural perspective and presents a working prototype.

3 An Architecture for Symmetric Language-Aware Aspects for Code Generation

We begin this section by proposing a general architecture for symmetric aspect-oriented code generation, partially addressing the issues related to asymmetry above. Next, we discuss the choice of weaving algorithm, which is instrumental in addressing these issues completely as well as making the overall approach language aware. Finally, we present a prototype we have implemented based on the Epsilon Generation Language [13].

3.1 Registration + Weaving = Symmetric Aspects for Code Generation

We propose a code-generation infrastructure that separates the step of generating code from the step of writing this generated code to a target file. This separation enables us to inject additional behaviour. In particular, we can decide to have a number of code generators generate code for the same target file and inject a code-weaving step that merges the generated code before it is written to the target file.

Figure 2 gives an overview of such an architecture. For code generators to become as independent as possible, each one needs to generate an operationally complete slice. These slices are then registered against their intended target file name in a code-slice registry from where they can be extracted for weaving by the code-slice weaver (removing the slices from the registry in the process). The result of the weaving can then be written to the target file. As the code-slice weaver is a code generator itself, we may also decide not to write the weaving result to a target file, but rather to register it again in the registry. This allows us to build hierarchical compositions of code generators, encapsulating the number and responsibility of the individual code generators.

This architecture provides symmetric aspect orientation for code generation. In particular, because all generation results are registered against their target file names in the same way, there is no distinction between base templates and aspect templates. Also, because the weaving happens based on the generation result no details of the template structure need to be exposed. This means there is no need for scaffolding through empty generator rules. Weaving context is also not a problem, because it can be generated by every template and its resolution can be left to the weaving algorithm.
Because generators are loosely coupled through the common registry only, they can be developed independently and it is easy to add or remove a generator for the same target file. As already discussed above, this requires, however, that each generator creates operationally complete slices of code, which may lead to some redundancy between templates for the same target file. The issues of scaffolding and weaving context are only partially addressed by the proposed architecture. How well they can be addressed—and how language-aware the code generation can become—depends strongly on the weaving algorithm used in the code-slice weaver. This will be discussed next.

### 3.2 A Choice of Weaving Techniques

The code-slice weaver is given a number of (code) text streams and faces the task of merging them into one stream that can then be either written to a target file or re-registered against the original target file. In this subsection, we discuss different alternatives for implementing such a code-slice weaver. The aim of this discussion is to decide on a suitable weaving algorithm.

Combining two or more text streams into one has been discussed in the literature quite extensively in particular in connection with software configuration management. Tom Mens [18] provides a good overview of the state of the art in software merging. He also proposes a number of dimensions providing a framework for classifying and discussing merging algorithms:

1. **Two-Way vs Three-Way Merging** “Two-way merging attempts to merge two versions of a software artifact without relying on the common ancestor from which both versions originated. With three-way merging, the information in the common ancestor is also used during the merge process.” [18]

2. **State-Based vs Change-Based Merging** “With state-based merging, only the information in the original version and/or its revisions is considered during the
merge. In contrast, change-based merging additionally uses information about the previous changes that were performed during evolution of the software.” [18]

3. Textual vs Syntactic vs Semantic Merging

Different merge algorithms use different amounts of knowledge about the language in which the text streams to be merged are expressed. Textual merge merges two texts independently of their language. Syntactic merge takes into account the syntax of the language of the texts to be merged, while semantic merge even considers the language’s semantics.

In the following, we discuss each of these dimensions in turn.

3.2.1 Two-Way vs Three-Way Merging

Because in the code-generation context we do not have a common source for the text streams to be merged, we cannot use three-way merge; only two-way merge algorithms are applicable. Note that using the empty string as a common source does not help, because three-way merge from an empty common source is effectively the same as a two-way merge.

Two-way merge might not seem sufficient, as we really need an algorithm that can merge \( n \) streams of text into one. However, this can typically be reduced to \( n - 1 \) applications of an appropriate two-way merge algorithm. The main caveat of this reduction is that typically we will have to be explicit about the order in which we perform merges because the two-way merge algorithm may not be commutative or associative [19]. This depends on the specific algorithm as well as on the language of the texts to be merged.

3.2.2 State-based vs Change-Based Merging

“Obviously, two-way merge techniques are always state-based since they can only compare two revisions without taking into account how these revisions have been obtained” [18]. Consequently, in the context of code generation we can only consider state-based merging algorithms.

3.2.3 Textual vs Syntactic vs Semantic Merging

Because textual merging does not take into account the language in which text streams have been expressed, it cannot make use of the syntax or semantics of this language to improve the merge results. It can also typically not provide any support for the kind of weaving context discussed in Sect. 2. There are cases where language specifics or weaving contexts may not be an issue for a specific code generation task. For example, when generating plain text documentation it may be sufficient to simply concatenate texts contributed from different sources. In these cases, textual merging may be useful and even preferred because of its efficiency and low configuration overhead.

For the typical case, however, we require at least syntactical merging algorithms for symmetric language-aware aspect-oriented code generation. An interesting example of syntactic (and partially semantic) merging is superimposition [19, 20]. Here, merging
of two text streams is based on a partial parse of each stream into a so-called feature structure tree (FST)—effectively an abstract syntax tree whose terminal nodes correspond to blocks of code rather than individual tokens. Then, merging is performed in two steps: FSTs are merged using a generic algorithm that considers only the structure of the tree and the names of the nodes, merging nodes (and their corresponding subtrees) of the same name. Terminal nodes are merged using language-specific merge strategies. Syntax (and partially semantics) of the language affect this algorithm in two ways: a) by determining the structure of the FSTs and the names of the nodes (both terminal and non-terminal), and b) by selecting merge strategies that are appropriate for the syntax and semantics of the language. Superimposition has been used in the context of feature-oriented development [21], which has similar requirements on merging as aspect-oriented code generation. It, therefore, seems to be a good candidate algorithm for the code-slice weaver.

Note that superimposition is not always commutative [19], especially for languages where a) the order of subtrees of an FST non-terminal node is relevant, or b) FST-terminal-node merge strategies are not commutative. An example for a non-commutative merge strategy is the merging of method bodies: Here, there is a need to indicate which method body should wrap the other one and where in the code the other method body should be integrated. For our architecture, this implies that the code slices may have to be ordered before weaving. On the other hand, there are cases (e.g., the generation of plugin.xml in Fig. 1), where the order of weaving is irrelevant.

3.2.4 Conclusion

In conclusion, our architecture for symmetric, language-aware aspect-oriented code generation should use a code-slice weaver that implements a two-way, state-based, syntactic merge of two (code) text streams—for example, superimposition as realised in FeatureHouse [20]. It is still useful to allow developers to use different weaving algorithms for specific situations, for two reasons:

1. For some scenarios, it may be sufficient to use a textual merge algorithm. Alternatively, in some cases a semantic algorithm may be required.

2. We have chosen superimposition as a specific algorithm because it is relatively well known in the area of feature-oriented software development. However, other syntactic two-way merges may be more useful for specific generation scenarios.

In the next subsection, we discuss a prototype which we have implemented as a proof of concept of our approach. This prototype is then used in Sect. 4 to implement an example of symmetric, language-aware aspect-oriented code generation.

3.3 Prototype

We have implemented a prototype of the architecture from Fig. 2 as an extension of the Epsilon Generation Language (EGL) [13]. Note that we have chosen EGL because
of our previous expertise, but in principle could have used any other code-generation language.

Our prototype provides additions to the workflow component of EGL. EGL (and Epsilon in general) uses Ant-based workflow descriptions [22] to co-ordinate different tasks required to solve a particular model-management problem. EGL provides a single workflow component epsilon.egl, which takes a model and an EGL template and writes the result of evaluating the template to a specified file. We provide a specialisation of this component—epsilon.eglRegister—which has the same interface and functionality, but instead of storing the generation to a file, uses a workflow-wide registry to register the generated code against the specified file name. EGL allows templates to instantiate and execute other templates, allowing the result of these executions to be stored in separate files. If an EGL template is evaluated in the context of epsilon.eglRegister any code generated in this way will also be registered rather than written to the file system.

Registered code slices can then be woven and finally written to a file (or registered again) using the epsilon.eglMerge component. To select the files for which to merge code slices, epsilon.eglMerge uses standard ANT file sets that allow the user to use wildcards to express the set of files for which to weave. This is important, because when templates are invoked from other templates, the names of the target files may depend on the contents of the models from which code is generated. Therefore, in the generation workflow, these file names cannot be statically encoded. Using wildcards in file names addresses this issue.

As discussed in Sect. 3.2.4, other weaving algorithms than superimposition may occasionally be required. To support such configuration, epsilon.eglMerge allows users to choose a weaving algorithm from a set of pre-defined algorithms, but also allows the definition of additional weaving algorithms. Our prototype currently supports a simple concatenation algorithm (a textual merge) and superimposition based on the FEATUREHOUSE implementation [20].

Finally, epsilon.eglMerge supports the definition of an order in which different code slices registered for the same file should be woven. To this end, code slices can be associated with a feature id in epsilon.eglRegister. epsilon.eglMerge then provides a means to define partial orderings between slices using their associated feature ids.

The next section shows our approach applied to an example case.

4 Example

As an example application of our approach, we construct a basic code generator for an EJB-based [14] application. The code generator consumes an annotated UML class diagram\(^4\) and produces skeleton code for both entity beans and corresponding session beans.\(^5\) We have implemented the following features:

\(^4\)We use EMF annotations to simplify implementation of the code generator, but could just as well have used standard UML stereotypes.
\(^5\)The templates presented in this paper have been simplified slightly to fit the available space and omit unnecessary detail.
• For any class that is not annotated as `session`, appropriate entity-bean code is generated, marking all attributes of the class as persistent. For read-only attributes, only a getter method is generated and the attribute is also marked as non-changeable and non-nullable in the database.

• For any class annotated as `session`, session-bean code is generated. The name of the session bean is the name of the class plus ‘Bean’. The session bean provides skeletons for all operations defined in the class.

• For any class that is not annotated as `session`, but is annotated as `group`, an additional session bean is generated. The name of the session bean is the value of the `group` annotation plus ‘Bean’. Additional debug and query annotations are used to determine what is generated into such a session bean: A `debug` annotation results in support operations for debugging (e.g., printing out all existing instances of the entity bean including their attributes) to be generated. A `query` annotation creates a support operation for a particular fixed query (specified in the value of the annotation).

In the following, we discuss each of these features in more detail.

### 4.1 Generating Entity Beans

Since the publication of EJB version 3 [14], entity beans are plain Java objects that have been annotated with persistence information. Consequently, we have split the entity-beans generator into two parts: One generator produces standard Java implementations of getters and setters for the attributes of a class. A second generator then generates the required annotations only. This way, the first generator can be reused in other situations, where no EJB annotations are required.

Listing 1 shows the base template for generating attribute getters and setters for plain Java classes. This template is invoked for every suitably annotated class in the class model (the class is mapped to the `currentClass` parameter for use in the template). Note that this is a standard EGL [13] template that does not need to make any reference to aspect-oriented extensions or even requires knowledge of the fact that another aspect will be woven into the final generation result.

Listing 2 shows an aspect template that adds entity-bean annotations to the plain Java code generated from the template in Listing 1. It can be seen that this template is only concerned with those elements of the generated code that need to be manipulated to add the necessary annotations: An `@Entity` annotation is added to the class as a whole and a `@Column` annotation is added to the getters of read-only attributes. Furthermore, the generator adds `import` statements for these annotations and adds `Serializable` to the list of implemented interfaces of the generated class.

Note that the base template in Listing 1 did not have to make any provisions for the addition of annotations, implemented interfaces or import statements in its template structure. To allow for successful weaving, both templates have to follow the same naming conventions for classes, attributes, and getter and setter methods. To ensure these are used consistently across a project, they have been encapsulated in helper
Listing 1: EGL [13] template for plain Java getters and setters

```java
import 'Names.eol';

public class [%=getJavaClassName(currentClass)%] {

   for (prop : Property in currentClass.getAllAttributes()) {

      private [%=prop.type.name%] [%=getJavaFieldName(prop)%];

      [%=prop.visibility%] [%=prop.type.name%] [%=getJavaGetterName(prop)%]() {
         return [%=getJavaFieldName(prop)%];
      }

      if (not prop.isReadOnly) {
         void [%=getJavaSetterName(prop)%] (%=prop.type.name% [%=prop.name%]) {
            this. [%=getJavaFieldName(prop)%] = [%=prop.name%];
         }
      }
   }
}
```

Listing 2: EGL [13] template for entity-bean annotations

```java
import 'Names.eol';
import javax.persistence.Entity;
import javax.persistence.Column;
import java.io.Serializable;

@Entity
public class [%=getJavaClassName(currentClass)%] implements Serializable {

   for (prop : Property in currentClass.getAllAttributes()) {

      if (prop.isReadOnly) {
         @Column(updatable=false,nullable=false)
         [%=prop.visibility%] [%=prop.type.name%] [%=getJavaGetterName(prop)%]() {
            return [%=getJavaFieldName(prop)%];
         }
      }
   }
```


4.2 Generating Session Beans

We generate session beans in three situations: 1) For classes explicitly annotated using the `session` annotation, 2) for classes annotated using `group` and `query`, and 3) for classes annotated using `group` and `debug`. Each of these situations corresponds to a separate feature of our code generator. Therefore, we modularise each into its own generation template. A specific session bean may combine contributions from all three features, but may also be made up of only one feature. Because for any given session bean none of the templates is mandatory, we cannot view anyone of them as the base template and construct the others as aspectual extensions of this base. Instead, the three templates must be completely independent of each other.

Listing 3 shows the template used to generate session-bean code for classes explicitly annotated as `session` in the model. It can be seen that session beans are annotated as stateless or stateful depending on the value of the `session` annotation and that they are normal Java classes otherwise. The name of the session bean is derived from the name of the class in the model using another operation encapsulating a naming convention. Furthermore, skeletons for all operations in the modeled class are generated.

Listing 4 shows the template used to generate session-bean code for classes not annotated as `session`, but annotated using `group` and `query`. The `group` annotation is used to derive the name of the session bean to generate. For each `query` annotation, an operation evaluating the query is generated. Note that the class name generated may well be identical to the name of a class generated through the template.
Listing 4: EGL [13] template for session beans with query code

```java
[import 'Names.eol';]
import javax.ejb.*;

@Stateless
public class [%=makeBeanName(currentClass.getAnnotationValue('group'))%] {
  for (%sQuery in currentClass.getAnnotationValues('query')) {
    /** Generated by query generator */
    public Object query[%=getQueryName(sQuery)%]() {
      // Execute query...
    }
  }
}
```

Listing 5: EGL [13] template for session beans with debug code

```java
[import 'Names.eol';]
import javax.ejb.*;

@Stateless
public class [%=makeBeanName(currentClass.getAnnotationValue('group'))%] {
  /** Generated by debug generator */
  public String getDebugInfo() {
    return original() + ...
  }
}
```

in Listing 3, in which case the two code slices would be merged later on.

Finally, Listing 5 shows the template used to generate session-bean code for classes not annotated as session, but annotated using group and debug. The group annotation is again used to derive the name of the session bean to generate. Then, the template simply generates a standard method for obtaining debug information about the entity bean in question. Note the use of original() to refer to other versions of getDebugInfo() that may have been generated for the same session bean. In merging, this will be replaced by an invocation of these previous versions. This does not imply any dependencies between different templates generating different versions of getDebugInfo(). All that each of these templates need to be aware of is that another version of the same method may be generated. To ensure that no base template is needed to ground this potential chain of method definitions, we extended the merging algorithm from [20] with a post-processing step removing any left-over invocations of original().

14
Listing 6: Excerpt from the workflow specification for the EJB generation workflow

```xml
<project name="GenerateEJBSkeletons" default="generate_all" basedir="."/>
<target name="generate_all"
depends="register_sessions, register_entities, register_associations, register_attributes, init">
<epsilon.eglMerge>
<file>
<include name="${dir}/*.java"/>
</file>
</epsilon.eglMerge>
<target name="register_sessions" depends="init">
<epsilon.eglRegister src="templates/EntityBeans.egl" feature="entity">
$model ref="classes" as="Model"/>
</epsilon.eglRegister>
</target>
<target name="register_entities" depends="init">
<epsilon.eglRegister src="templates/Associations.egl" feature="assoc">
$model ref="classes" as="Model"/>
</epsilon.eglRegister>
</target>
<target name="register_associations" depends="init">
<epsilon.eglRegister src="templates/EntityBeans.egl">
$model ref="classes" as="Model"/>
</epsilon.eglRegister>
</target>
<target name="generate_entities" depends="init">
<epsilon.egl src="templates/EntityBeans.egl">
$model ref="classes" as="Model"/>
</epsilon.egl>
</target>
... 
</project>
```

4.3 Putting It All Together

Listing 6 shows an excerpt from the ANT file controlling the generation workflow for our case study. The targets named register_xx trigger generation from a particular template and store the results of this generation in the code-slice registry. Target generate_all merges the contents of the code-slice registry for every Java file using Java-specific superimposition. Finally, target generate_entities has been introduced for debugging purposes to enable generation using the basic EntityBeans template only. Notice that this target (lines 33–37) differs only marginally from the corresponding target using the code-slice registry (lines 21–25): It uses epsilon.egl instead of epsilon.eglRegister and it does not define a separate feature attribute. It should particularly be noted that both targets use the exact same template file.

epsilon.eglRegister is invoked without providing a file name for the file.
to be generated (see lines 22–24). It is possible to provide an optional target attribute defining a file name for the direct output from the template. In our example, the templates derive the target file names from model information, creating one .java file per class to be generated. Therefore, we do not use the target attribute. In the case of epsilon.eglMerge (lines 8–17), it is necessary to say for which files the code-slice weaver should be invoked. Two things should be noticed, however:

1. Where the exact file names are not known (e.g., because they have been determined based on model information), wildcards can be used to specify the files to be merged. These wildcards will be matched against all files for which some generated code has been registered.

2. The call to epsilon.eglMerge remains the same regardless of how many templates have contributed text towards any one of the selected files. Implicitly, everything that has been registered for a particular file will be merged. This means, we can easily add new code generators contributing additional features towards the same java files selected on line 10.

We have previously mentioned that the order in which code slices are woven may be important. Therefore, epsilon.eglMerge allows this order to be specified using an arbitrary number of order tags. Each usage of order defines the ordering for one pair of code slices, so that all tags together induce a partial ordering of code slices. So that the code slices can be referred to precisely in these tags, they need identification that is unique at least within the set of slices registered for a particular file. This is achieved by associating each generator that requires explicit ordering information with a feature identifier (see line 22 for an example). These feature identifiers can then be used in order tags to refer to code slices. Note that it is legal to specify ordering constraints for features for which no code slice is registered in the registry. In this case, such a constraint will be silently ignored. This is useful, because callers of epsilon.eglMerge do not need to know what code slices are registered for the files they ask to be merged, so generators can be easily removed from and added to the overall work flow.

In our discussion of the selection of a particular weaving technology, we have stated that different projects may require different weaving technologies. To provide this flexibility, callers of epsilon.eglMerge must explicitly state the weaving technology to be used for a particular set of files (see line 12). The prototype currently supports simple concatenation as well as superimposition, but new weaving technologies can be easily added. The superimposition implementation is based on FeatureHouse [20] and reuses FeatureHouse’s notion of artifact handlers to encapsulate language-specific knowledge. This leads to very simple configuration of the weaving technology (a single parameter is required to identify the language to use) while maintaining flexibility (new artifact handlers can be easily implemented and registered in the system).

Listing 7 shows how the work-flow specification can be changed slightly so that the woven code will not be written to a file, but instead will be entered back into the code-slice registry. In particular, we have modified line 9 by adding the attribute register="ejb". This means that all merged code will be re-registered with the code-slice registry using the provided value as the feature id for purposes of ordering
later weavings. This has the effect of hiding what generators (and ‘features’) and code-
slice weaver were used to produce a particular piece of code; in fact, after this point
such a woven and re-registered piece of code cannot be distinguished from a piece
of code that has been generated directly. This can be particularly useful where a set
of code slices need to be woven together into one output file using different weaving
strategies.

5 Discussion

In this section, we provide an analytical evaluation of our proposed approach to sym-
metric language-aware aspect-orientation for code generation. We begin our discussion
by revisiting the issues we identified in Sect. 2 and discussing how our approach ad-
dresses them. After this, we discuss other benefits and drawbacks of our approach.

5.1 Addressing the Issues of Asymmetric AO for Code Generation

In Sect. 2, we identified four issues with asymmetric aspects for code generation. Here,
we discuss how our approach addresses these issues:

1. **Base template needed.** Every generator is treated equally and produces an op-
erationally complete code slice. Therefore, none of the templates needs to be
(artificially) declared the base template, but all templates can be treated the same
way. Ordering of code slices may be needed before weaving (cf. Sect. 3.2.3), but
this is not the same as making one template the base template. In particular, any
code slice can still be left out without invalidating the weaving definition. This is
not true for asymmetric techniques, where at least the base template must always
be included. In our example, we saw this in the generation of session beans (cf.
Sect. 4.2). We used three independent generation templates for session beans,
none of which could be identified as a base template and none of which was mandatory for the generation of a particular session bean.

Note that this does not mean we cannot have a base template if this is more convenient. We can always agree to treat one template as the base and assume it will always be used. We have done so, for example, in Sect. 4.1, where we assumed the template from Listing 1 to be the base template. However, such assumptions are made by the users of the modularisation technique and are not formally encoded in the templates or the specification of the code-generation work flow.

2. **Scaffolding needed in base template.** The weaving is performed on generated code slices rather than on generator templates. Thus, the structure of the templates is hidden from the weaver. This means that the possible joinpoints are only limited by the structure of the language of the code slices and the specific merging algorithm used, but not by the structure of the code-generator templates. Therefore, every generator can use exactly the template structure that is best for its specific concern independently of other generators that might affect the same file. In the example, we saw this in the generation of entity beans (cf. Sect. 4.1). The EJB-specific template introduces import statements, annotations and implemented interfaces to the generated class definition. In an asymmetric approach such as [10, 11] we would have needed to provide appropriate hooks in the base template already. Instead, our base template (Listing 1) has been written completely independently of EJB concepts and did not need to include any additional hooks. Rather, any extensions have been identified by matching the generation results and merging them where they are different.

3. **Insufficient support for weaving contexts.** Weaving in our approach is based on the generated text rather than the template structure and is aware of the language in which this text is expressed. This means that every template can generate the weaving context it requires. The weaver can then ensure this context is included at most once in the weaving result. In the entity-bean example (cf. Sect. 4.1), we have used this to ensure that there will only be one implements clause regardless of how many different templates contribute implemented interfaces to the same entity bean.

4. **Weaving performed independent of language syntax or semantics.** Our approach uses a syntactic text merging algorithm to implement the weaving. Because we are effectively free to define the merging strategies for individual parts of the language, we can even define a semantic merging strategy, which, for example, knows for EJB-specific annotations that @Stateful and @Stateless are alternative annotations and that @Stateful should be preferred, if both are present in different versions of the same class.

### 5.2 Reducing Inter-template Dependencies

Using a symmetric language-aware approach to aspect-oriented code generation, we have reduced dependencies between the different templates. Because weaving occurs

18
based on the generation result, templates can define their internal structure independently.

Moreover, we have decoupled the specification of the templates and the specification of their composition. In the asymmetric aspect approaches proposed so far [10,11], aspect templates explicitly mention the base template (and rules in that template) to which they apply. In our approach, we have made the decision about which templates to compose external to the templates themselves. Each template is, thus, defined without explicitly mentioning other templates that its generation result is to be merged with. Instead, the templates that are evaluated and the composition of generation results are specified as part of the generation work flow. Listing 6 shows each template being evaluated in its own target (Lines 21–31). The composition is then specified effectively in the `depends` clause of target `generate_all` (Lines 5–6). This makes it easy to modify the generation work flow to generate the results of a single template or to merge only a subset of the templates to focus on some generation problem.

Reduced dependencies between code-generation templates means that these templates can be reused in new contexts more easily. For example, we have seen in Sect. 4.1 how we reused a template generating basic getters and setters for a class’s attributes in the context of entity-bean generation.

5.3 Complexity of Composition

It can be argued that in moving from asymmetric to symmetric aspect orientation, we simply shift the complexity of the composition from the templates to the choice and configuration of the weaving algorithm. At this point, there is no empirical evidence in either direction, but we can provide some analytical arguments towards this discussion. In general, complexity will be moved, but the less configuration of the weaving algorithm is required, the better this complexity can be encapsulated, thus relieving the developer from having to deal with it. We can distinguish at least three different classes of code-slice weavers:

1. **Textual-Merge Weavers.** In rare situations, simple textual merges (such as concatenation, textual two-way merge, or even substitution) may be sufficient. As these do not require any further configuration, all merging complexity is effectively hidden. We expect this to be useful mainly for textual artefacts—for example, documentation materials.

2. **Generic Syntactic or Semantic Weavers.** The merge algorithms we used in the example in Sect. 4 belong into this category: While they are syntactic (or, in the case of merging EJB-specific annotations, even semantic), they can be configured using simple parameters, such as choice of programming language or target platform. For example, Line 12 in Listing 6 shows how we configure the superimposition weaver, which is a generic syntactic weaver. The only configuration necessary is to provide a single parameter "java15" to indicate the language of the files to be merged as Java version 1.5. "java15" actually refers to a set of Java 1.5 specific merging algorithms, as described in more detail in [20]. These may need to be developed for each new language or platform feature, but once developed they can be packaged separately, registered with the
system, and reused between projects, reducing complexity again to that of simple parametric configuration. For FEATUREHOUSE, Apel et al. [20] note that the effort for integrating a new language was “on the order of hours” (p. 227). They note further that “[i]n practice, only a few composition rules are needed, which can be reused by different languages and which follow even fewer rule patterns” (p. 229). Where weavers are based on our prototypical integration of FEATUREHOUSE, we expect these statements to carry over directly.

We expect this to be by far the largest category of cases. All of the weavers shown in the example fall in this category. They should be easily reusable for other projects, as they only depend on specifics of the target language and platform. Notice that it is still possible to encode project-specific translations from models into code by providing project-specific generation templates. This, however, does not require project-specific code-slice weavers to be used as well.

3. Project-Specific Weavers. These are the most complex cases, as they require merging algorithms to be developed specifically for a particular project to include (semantic) knowledge about this project. Where project-specific merging is required, using our approach to build code generators may become uneconomic.

We expect project-specific weavers to be required most rarely. Most customisation of the generation process would be expected to happen in the code generators, or possibly in setting up the overall generation work flow. Project-specific weavers would only be required where projects define specific target-language constructs the semantics of which need to be taken into account by the code-slice weaver.

From the discussion above it can be seen that in what we expect to be the most typical scenario, configuration can be kept to a minimum, ideally only requiring the specific target language to be specified.

5.4 Scaffolding vs Slice Completeness

We have argued above that because weaving happens on the generation result it is independent of the internal structure of generation templates used and, therefore, no scaffolding rules are required within these templates. It can be argued that this benefit has been traded against the need for each code slice to be operationally complete, which leads to a certain level of redundancy between code slices. This redundancy must be managed, and consistency between redundant parts in different code slices must be ensured so that the weaving can be performed successfully.

However, most of the redundant code generated is skeletal code (e.g., class and method declarations, or the topmost element of a plugin.xml file). To synchronise different code slices, it is mainly necessary to ensure they use the same names to refer to the same classes or operations etc. These names are typically derived from model data. Therefore, much of the required synchronisation can be encoded in naming conventions to be used consistently throughout a project. As we have seen in the
example in Sect. 4, these naming conventions can be conveniently encoded using traditional modularisation techniques such as operations or generator rules. We have used standard Java naming conventions for getters, setters, and bean classes to generate the names of methods and classes in different code slices. These naming conventions were sufficient to ensure that the code slices could be woven successfully and correctly.

5.5 Benefits of Code-Slice Registry

It could be argued that the same modularisation benefits could be achieved without a specialised architecture including a code-slice registry and code-slice weaver by simply generating code for each feature into a separate file and using a standard symmetric weaver to compose these files later on. Here we argue that the indirection offered by the code-slice registry and code-slice weaver play an important role in enabling flexible inclusion and exclusion of features in a generation run. Without them, adding a feature to the generation work flow requires a number of changes throughout: The feature’s generation template must be added to the generation work flow and all files generated by it must be made known to the separate symmetric weaver (e.g., by adding an additional command-line option). Similarly, removing a feature requires changes in all of these places. In contrast, the code-slice registry dynamically maintains a list of text streams to be merged into the same file. Therefore, adding or removing a feature in our architecture requires a change in exactly one place: The code-generation template for this feature must be included into or excluded from the code-generation work flow.

The problem is compounded where a template generates more than one file (e.g., generating one class file per class in a model); that is, where part of the generation work flow is encoded in the templates and controlled by the model for which code is generated. In this case, it can be impossible to statically encode the list of files to be merged in the invocation of the standalone symmetric weaver. Again, because the code-slice registry builds the list of files dynamically, it does not suffer from this problem.

In Sect. 4.2, we have used an additional post-processing step after superimposition to clear up any dangling uses of original(). Such post-processing is not normally required in standalone uses of superimposition as there we have a stable set of features including a core (or base) feature, so that we can ensure no calls to original() are left dangling. In the code-generation environment, and especially where we require the flexibility of adding and removing generators as required for a particular model and product, such post-processing is paramount to ensuring the production of well-formed code. Therefore, we cannot directly use a standalone superimposition weaver for our problem.

6 Related Work

As mentioned previously, aspect-oriented approaches to code generation already exist [10, 11]. These approaches are asymmetric and language agnostic and their issues and how our approach addresses them have been the focus of this paper.
Hemel et al. [4] discuss modularisation of code generation and model transformation in the context of Stratego/XT and their WebDSL case study. One of their modularisation scenarios is closely related to the work presented in this paper: Because the modularisation of code generators is driven by the structure of the target metamodel (a single element or artefact in the target code must be the result of a single code-generation rule), Hemel et al. were forced to use very large ‘God rules’ incorporating generation logic for all features that affect a particular artefact. This is exactly the same problem that motivated our work. They introduce an intermediary language providing more advanced constructs for modularisation (e.g., partial classes and operations). They then use a staged generation strategy: A first code generator produces code in the intermediary language from WebDSL code. A second code generator then merges (superimposes) partial classes and operations and produces the final code in the target language. Again, this is essentially the same as our approach. However, because Stratego/XT does not provide any direct support for aspect-oriented transformations or code generations, Hemel et al. needed to introduce an explicit new intermediary language, containing a large set of constructs only relevant for the subsequent superimposition (e.g., @Class for representing a partial class). With the exception of the original() operator, we did not require any constructs specific to the subsequent merging step, so that code generators can be used regardless of whether merging will happen. Their approach is eased by the use of Stratego/XT, where generated code is always handled as an abstract syntax tree rather than plain text. Instead, in our approach, code is generated as text, which is then merged in a separate step. This requires this text to be parsed again in preparation for merging, making our approach perhaps a little less time efficient. At the same time, however, it also allows the use of simpler textual merges based on the same infrastructure.

Framed aspects [23] use code generation to parameterise AspectJ aspects and, thereby, make them reusable in new contexts. The key difference between their approach and ours is that they focus on generating aspects, whereas we focus on using aspects at the code-generator level to support modularisation of code generators. While our approach could also generate aspects, it is just as useful when generating normal OO code, or HTML files. In general, work on aspect-oriented programming [24] is complementary to the work presented in this paper. As discussed in Sect. 7, it may be very useful for some generator modules to produce aspect code, which is then woven at compile time. This is the case particularly where the quantification properties of asymmetric aspect-oriented programming techniques can be used to simplify and improve modularisation.

7 Conclusions and Future Work

In this paper, we have proposed an approach applying notions of symmetric aspects to the domain of code generation. This has enabled us, chiefly, to reduce dependencies between code-generation templates, making this approach an interesting technique for the modularisation of large code generators in the MDE context. Reduction of dependencies between templates also makes these templates more easily reusable and simplifies debugging of code-generation workflows. We have shown how the symmet-
ric nature of our approach resolves some problems identified for existing asymmetric aspect-oriented approaches to code generation, namely the need for an explicit base template, the need for scaffolding in this base template, and insufficient support for weaving contexts. We have also shown how the use of language-aware weaving algorithms helps to completely address these issues as well as allowing more intelligent weavings that depend on language syntax or semantics.

Asymmetric aspects remain a complementary approach as they can contribute to code-generator modularisation in areas not easily covered with a symmetric approach as the one proposed here. In such scenarios, it may be more economic to use a combination of symmetric and asymmetric approaches to code-generation. For example, a code generator adding tracing or transaction support to every method generated in a class definition—regardless by which code-generator template—may be more efficiently expressed by generating an asymmetric aspect to be woven with the class definition at compile time. In some scenarios, it may also be useful to combine asymmetric aspect templates with the approach presented here. When exactly such combinations can be beneficial and how they should best be realised, remains an area open for further research.

Acknowledgements

This work has been funded by the European Commission under FP6 STREP AMPLE and FP7 Marie-Curie IEF RIVAR. The authors would wish to thank Dimitrios Kolovos, Louis Rose, and Richard Paige for help with and discussion about the Epsilon framework, and Jon Whittle for useful feedback on an earlier draft of this paper.

References

[6] Sven Efftinge, Peter Friese, Arno Haase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnelein, Dieter Moroff, Karsten Thoms, Markus Völter, Patrick Schönbach, Moritz Eysholdt, and Steven Reinisch. openarchitectureware xpand documentation. Published


