Towards Modular Code Generators Using Symmetric Language-Aware Aspects

Steffen Zschaler
King's College London,
Department of Informatics, London, UK
szschaler@acm.org

Awais Rashid
School of Computing and Communications,
Lancaster University, Lancaster, UK
awais@comp.lancs.ac.uk

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 to improve modularisation support.

Categories and Subject Descriptors

D2.3 [**Software Engineering**]: Coding Tools and Techniques

Keywords

model-driven engineering, code generation, symmetric aspects

1. INTRODUCTION

In model-driven engineering (MDE), domain-specific languages (DSLs) 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 [14]. 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 [6], performance, security, different target platforms, etc., which are not captured in the source models. 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 [16]. 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., [4, 7, 8, 11]) 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 [10,15]. However, they are not fully adequate for all needs of code-generator modularisation: Because they are asymmetric approaches [5], 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, 15], 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.

FREECO'11, July 26, 2011, Lancaster, UK.

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 and present a prototype implementing these concepts based on the Epsilon Generation Language (EGL) [12]. Section 4 surveys some related work and Sect. 5 concludes the paper. Due to space limitations, we have omitted an example as well as a more detailed discussion of the benefits and drawbacks of our approach. These can be found in an accompanying technical report [17].

2. MOTIVATION: MODULARISING CODE GENERATION TEMPLATES

Many code-generation frameworks, such as [4, 7, 8, 11], 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 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 [16], 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 [3] 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, there 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 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 [13] 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, 15]. These are asymmetric aspect-oriented approaches [5] allowing the definition of `around`, `before`, and `after` advice for individual code-generation rules or operations in a base template. 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 `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.[1] 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, [15, p. 9] shows an example of such an empty generator rule (the fact that [15] in principle allows aspects on aspects does not strongly affect the need for scaffolding). In [2] 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.

[1] Oldevik and Haugen [10] describe this: "[...] there must be constraints on the base transformation [...], e.g. that domain-specific rules should be clearly separated from general rules".

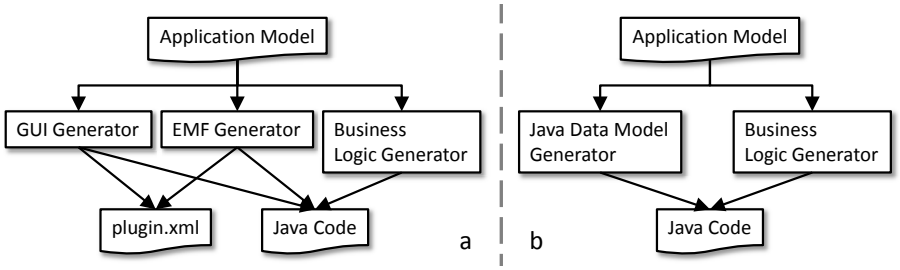

Figure 1: An example decomposition of code generators for generating Eclipse plugin code. The figure shows two possible configurations: a) using EMF for data management as well as including a graphical user interface, and b) using plain Java objects for data management and providing a programming interface only

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. Only `before`, `after`, and `around` are supported, making correct generation of implemented interfaces 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 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 [5], addressing the short-comings 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, addressing the issues identified above. Then, we present a prototype we have implemented based on the Epsilon Generation Language [12].

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 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. 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 [9] 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 com-

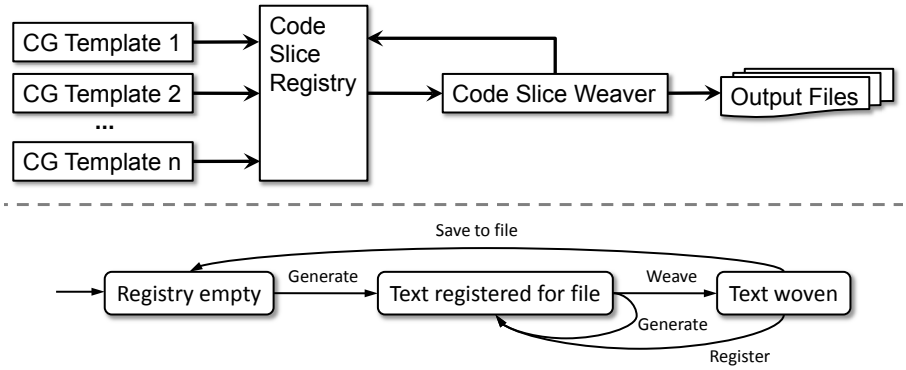

Figure 2: Abstract architecture for a symmetric aspect-oriented code-generation infrastructure. The upper part shows the components involved. The lower part shows the possible states of the code-slice registry for a particular target file

mon ancestor is also used during the merge process." [9]

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." [9]

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 context of our architecture, only two-way merging can be used, as there is no common ancestor available. This also implies using a state-based merge algorithm. In order to make composition language aware (which also helps address issues of weaving context as discussed above), we need to use syntactic or even semantic merging. Superimposition—for example as implemented in FEATUREHOUSE [1]—is an interesting candidate. It supports syntactic merging based on feature-structure trees (syntax trees whose terminal nodes correspond to blocks of code—for example, entire methods) and semantic merging for the terminal nodes of these trees.

3.2 Prototype

We have implemented the architecture from Fig. 2 as an extension of the Epsilon Generation Language (EGL) [12].[2] 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 [8] 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.

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.

An example applying our prototype to the generation of EJB code from a class model can also be found in [17].

4. RELATED WORK

As mentioned previously, aspect-oriented approaches to code generation already exist [10, 15]. 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. [6] 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

[2]The full prototype as well as the example is available from `http://www.steffen-zschaler.de/publications/symmetric_ao_cg`.

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). 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.

5. CONCLUSIONS

In this paper, we have proposed an approach applying notions of symmetric aspects to the domain of code generation. This approach separates the specification of code generators from the specification of their composition, providing more flexibility in choosing code generators that should be used for a particular generation task. Because it is a symmetric approach, it does not require any template to be identified as the base template, enabling us to flexibly leave out or re-arrange code generators as required. Because the weaving happens at the level of generated code, there is no need for scaffolding in the form of empty generator rules. Because we use syntactic merging, the approach can take into account syntactic constraints of the language of the generated code. We have successfully applied our approach to the generation of EJB code from a class model [17], and are looking to further evaluate it with more case studies as well as in a comparative study with other generation approaches.

Acknowledgements

This work has been funded by the European Commission under FP6 STREP AMPLE and FP7 Marie-Curie IEF RIVAR. The authors 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.

6. REFERENCES

[1] Sven Apel, Christian Kästner, and Christian Lengauer. FEATUREHOUSE: Language-independent, automated software composition. In Stephen Fickas, Joanne Atlee, and Paola Inverardi, editors, *Proc. 31st Int'l Conf. on Software Engineering (ICSE'09)*, pages 221–231. IEEE Computer Society, 2009.

[2] Ruzanna Chitchyan, Phil Greenwood, Americo Sampaio, Awais Rashid, Alessandro Garcia, and Lyrene Fernandes da Silva. Semantic vs. syntactic compositions in aspect-oriented requirements engineering: An empirical study. In *Proc. 8th ACM Int'l Conf. on Aspect-Oriented Software Development (AOSD'09)*, pages 149–160. ACM, 2009.

[3] Eclipse Foundation. Eclipse web site. Published on-line: `http://www.eclipse.org/`.

[4] Sven Efftinge, Peter Friese, Arno Haase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, Patrick Schönbach, Moritz Eysholdt, and Steven Reinisch. openarchitectureware xpand documentation. Published on-line: `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html#xpand_reference_introduction`, 2009.

[5] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research, 2002.

[6] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: A case study in transformation modularity. *Software and Systems Modelling*, 9(3):375–402, June 2010. Published on-line first at www.springerlink.com.

[7] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: A DSL for the specification of textual concrete syntaxes in model engineering. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *Proc. 5th Int'l Conf. on Generative Programming and Component Engineering (GPCE'06)*, pages 249–254. ACM, 2006.

[8] Dimitrios Kolovos, Richard Paige, Louis Rose, and Fiona Polack. *The Epsilon Book*. Published on-line: `http://www.eclipse.org/gmt/epsilon/doc/book/`, 2009.

[9] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[10] Jon Oldevik and Øystein Haugen. Higher-order transformations for product lines. In Tomoji Kishi and Dirk Muthig, editors, *Proc. 11th Int'l Software Product Line Conf. (SPLC'07)*, pages 243–254. IEEE Computer Society, 2007.

[11] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal, and Arne-J. Berre. Toward standardised model to text transformations. In A. Hartman and D. Kreische, editors, *European Conf. on Model Driven Architecture – Foundations and Applications (ECMDA-FA'05)*, pages 239–253, 2005.

[12] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. Polack. The Epsilon generation language. In Ina Schieferdecker and Alan Hartman, editors, *Proc. 4th European Conf. on Model Driven Architecture (ECMDA-FA'08)*, pages 1–16. Springer, 2008.

[13] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2009.

[14] Jonne van Wijngaarden and Eelco Visser. Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, May 2003.

[15] Markus Völter and Iris Groher. Handling variability in model transformations and generators. In *Proc. of the 7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[16] Jules White, Jeff Gray, and Douglas C. Schmidt. Constraint-based model weaving. *Transactions on Aspect-Oriented Software Development, Special Issue on Aspects and Model-Driven Engineering*, 5560(6):153–190, 2009.

[17] Steffen Zschaler and Awais Rashid. Symmetric language-aware aspects for modular code generators. Technical Report TR-11-01, King's College London, Department of Informatics, 2011.