Abstract

In aspect-oriented model driven engineering (AOMDE), a software design model consists of a primary model that describes the business logic of the application and a set of aspect models each of which describes a crosscutting software feature. The complete design is realized by composing the primary model with the aspect models. A variety of AOMDE approaches are possible depending on how the principal tasks of the AOMDE process are interleaved. This paper presents and compares two distinct AOMDE approaches.

Keywords: model-driven engineering, aspect-oriented software development, software engineering.

1. Introduction

Model driven engineering (MDE) [4, 11, 12, 13, 14, 16, 17] advocates the use of models as first class entities in software development. Models facilitate the understanding of complex systems by supporting the representation of systems at a high level of abstraction.

In order to realize the potential benefits of models, MDE approaches must facilitate the specification and reuse of models. However, reusing software models is difficult in the presence of crosscutting software features. Aspect-oriented technologies can be used to support MDE by facilitating the separation of platform-specific designs from business functionality designs. In aspect-oriented software development (AOSD) [1, 2, 7, 9, 18] crosscutting functionality are those that are spread across and tangled with other design elements. In AOSD crosscutting functionality are modularized in aspects and a complete software design consists of a platform independent primary model that describe the business logic of the application and a set of aspect models that describes crosscutting software features.

To support an aspect-oriented approach to model driven engineering (AOMDE), lifecycle process models delineating the principal tasks undertaken in AOMDE are needed, along with the identification of temporal and logistic dependencies between tasks. AOMDE approaches may be classified into two broad groups [6]. In one approach called the Generate Then Weave approach, code is generated separately from primary model and aspect design models and a complete application is created by weaving aspect code and primary model code. In the second approach, a complete design model of the target application is created by composing aspect models with a primary model. Code is then generated from this composed design model. The second approach is called the Weave Then Generate approach. While these approaches were identified in our previous work [6], the approaches were not explored. This paper identifies several high-level AOMDE tasks and utilizes the tasks in describing and comparing the two approaches.

The rest of the paper is organized as follows. Section 2 gives an overview of the high-level AOMDE tasks. Section 3 describes AOMDE approaches while a comparison of approaches is presented in Section 4. Section 5 presents a discussion of the approaches and identifies outstanding issues. Conclusions and future work are presented in Section 6.

2. Principal AOMDE Tasks

In the presented AOMDE approaches, classical software development lifecycle activities such as analysis, design and implementation are reinterpreted in the context of AOSD, and several AOSD-specific activities are added.

1. Analysis. This includes need identification and scoping, analysis of the scoped problem, the specification functionality requirements, and the identification of
potential crosscutting functionality (so called early aspects).

2. Primary Model Design. The Requirements Specification resulting from analysis is used to create a platform-independent primary model as well as to identify additional crosscutting software features that may be modeled as aspects.

3. Primary Model Enhancement. In an AOMDE environment, a complete application design is obtained by composing a primary model with aspect models. For a successful composition, the primary model must contain structural and behavioral properties required by the target platform. These required properties differ from one platform to another. The Jini middleware, for example, requires the creation of a proxy and a remote interface for each service. CORBA on the other hand, requires the creation of an IDL file for each service. The process of customizing a primary model to include required platform-specific features is called enhancement and the generated artifact is called an Enhanced Primary Model. In MDE, a primary model may be subject to many transformations, depending on the development goals of software engineers. For example, one may wish to refactor a primary model to make it amenable to a particular architectural style. In this paper the term enhancement is used to refer to that class of transformations that make an artifact platform-specific.

4. Aspect Model Design. An aspect model is a modularization of a crosscutting software feature. During aspect model design, models are created that represent the crosscutting software features identified during analysis and design of the primary model. While it may be possible to identify some crosscutting features during analysis, a complete list of these features is only available after design of the primary model.

5. Aspect Model Enhancement. Aspect model enhancement involves transforming an aspect to make it platform specific. For example, transforming a platform independent 2-phase transaction model to make it CORBA-specific. Aspect model enhancement is required when an aspect is platform independent which should be typical in AOMDE.

6. Aspect Model Instantiation. Aspect enhancement involves adding application-specific information to the aspect to enable the composition of the aspects with the primary model of the application. The process of adding the application-specific information to an aspect is called binding or instantiation, and the resulting aspect is called a mapped of context-specific aspect. Examples of application-specific information include primary model attribute names and operations.

7. Weaving (i.e. Composition). A complete application model is generated by composing application-specific aspects with an enhanced primary model. Composition may occur at either the design level or the code level.

8. Code Generation involves generating application code from design models.

9. Testing, Deployment and Evolution. During these phases executables are generated and the application is tested, deployed and evolved. For the scope of this paper, these phases have been omitted since the salient features of the two approaches may be represented without these tasks.

3. MDE Approaches

Templates have been used for describing design patterns [5] and testing patterns [3]. Similarly, in this paper, a template is used for describing MDE approaches. The template contains the items listed below.

1. Name. The name of the approach is a word or phrase that is used for identifying the approach and for suggesting its general intent.

2. Intent. The intent of an approach is a statement that highlights the objectives and distinctive features of the approach.

3. Structure. Each description of a MDE approach includes a graphical representation of the approach, (its structure), along with a textual description of the interrelationship of its subparts. The high-level AOMDE tasks defined in section 2 are represented in each graphical representation.


Each AOMDE approach is described using an activity diagram where action nodes are classified into six partitions (See Figure 1 and Figure 2). The partitions provide a high-level description of the AOMDE approaches with the high-level tasks being distributed among the partitions as shown in the figures.

3.1. The Weave-Then-Generate (WTG) Approach

Name: Weave-Then-Generate.

Intent: Create an integrated design model and use the model as the basis for generating application code.
Figure 1. The Weave-Then-Generate Approach.

Structure: The overall structure of the Weave-Then-Generate approach is depicted in Figure 1. The figure illustrates that code is generated from a composed design model created during weaving. In this figure, weaving precedes code generation.

Consequences: This approach may be considered ideal in AOMDE because a composed design model that facilitates design-time model analyses and model execution is created. For example, the composed model can be used for trade-off and error analyses as well as for scenario-based 'what-if' analyses. It is through these types of analyses that many of the expected benefits of MDE will be realized.

When a design aspect is transformed into a code aspect, it may not be possible to convert the design aspect into a single corresponding code aspect. The ability to effect this transformation is constrained by the sophistication of the target aspect-oriented programming language. However, this approach is not constrained by this limitation since code is not generated for individual aspect models.

One potential weakness of this approach is a potentially large semantic gap between individual design models (enhanced PM, mapped aspects) and code. This may be important if the code is to be manually maintained. In the ideal case, software maintenance in MDE will entail model debuggers and associated tools that eliminate the need to manually edit and evolve code. However, even when these tools are available, some developers may still wish to generate code from models and then manually evolve the code and the associated design models.

3.2. Generate-Then-Weave Approach

Name: Generate-Then-Weave (GTW).

Intent: Create a complete application by integrating aspect code and primary model code generated separately from aspect models and a primary model respectively.

Structure: The Generate-Then-Weave Approach is depicted in Figure 2. The structure differs from the Weave-Then-Generate (WTG) approach described in section 3.1 in two respects. First, there are two code generation phases, one for the primary model and one for the aspect models. Secondly, in this approach, code generation precedes weaving.

Consequences: The advantages of this approach include:

1. A potential reduction in the semantic gap between individual design models and code because a design aspect is translated directly into a code aspect and a primary model is also transformed directly into code. Until MDE become a mature discipline, one would expect the need for manual code evolution in many cases, and as such, this semantic gap may be important in such cases.
2. Improved traceability between code errors and design artifacts (enhanced PM, mapped aspects). When an error is detected in code, it may be easier to relate the error to the aspect models or the primary model since the code resulted directly from these models. This differs from the Weave Then Generate approach where code is generated from a complex integrated design model and therefore before an error can be traced to the mapped aspect of
the enhanced primary model it may be necessary to first trace the error to a component in the integrated design model.

3. Scattering and tangling is completely eliminated at the design level.

This approach has several weaknesses. First, the approach is constrained by the sophistication of available aspect-oriented programming language. Two important questions in this regard are: (1) whether there are cases where a design aspect cannot be directly translated into a code aspect in the target language, and (2) how these cases can be identified. For example, inner classes have been a problem for AspectJ [15] aspect developers. Secondly, the absence of an integrated design model constrains the capacity for model analysis and model execution.

4. Comparison of Approaches

In this section a comparison of the AOMDE approaches is presented. The goal of the comparison is to provide guidance to developers as to when and how the approaches may be used.

4.1. Tool Requirements

The Weave-Then-Generate approach requires the following tools:

1. Model development and editing tools for creating and modifying design models.
2. Model transformation tools to support both vertical and horizontal transformation of models, including the compilation of models into code.
3. Model composition tools to be used to compose aspect models with a primary model. These tools will need to support a variety of composition strategies, for example, signature based composition [10].
4. Aspect model instantiation tools.
5. Model debugging tools to facilitate traceability of code-level errors to design artifacts.
6. Model execution tools to facilitate identification of faults before code is generated from the models.
7. Model analysis tools to support the identification and visualization of apparent and emergent software properties, both static properties and behavioral properties.

The tool set required by all the Generate-Then-Weave approach differs from that listed above in two general ways. First, model composition tools are not required since a composed design model is not created. Instead, code composition tools are used. Secondly, tools that can generate aspect-oriented code from design models are needed.

4.2. Design and Code Reuse Potential

Reuse potential of both approaches may be examined from both code-centric and design-centric viewpoints. From the design viewpoint, the Weave-Then-Generate approach has more reuse potential because the composed application design generated during weaving may be used multiple times for model execution and model analysis purposes. Conversely, the Generate-Then-Weave approach is superior with respect to code reuse potential because (1) the code generated for the primary model may be reused with many relevant aspects for the target platform, and (2) the aspect code may be reused with many different applications.
4.3. Error Correction and Modifiability

The traceability of application errors to code and design artifacts will depend in part on the semantic gap between the composed application code and the more abstract code and design artifacts. Since the Generate-then-Weave approach generates a code artifact for each individual aspect model as well as for the primary model, it may be easier to trace code errors to one of the individual design artifacts. This contrasts with the Weave-Then-Generate approach where a composed design model is intermediate between the application code and the individual design models. On the other hand a composed design model provides an opportunity for the design-time observation and analysis of the semantic coupling [8] inherent in an aspect-oriented design. The approach selected for a specific project will depend on availability of tools and the purpose and intent of the design models. For example, the availability of tools that can execute designs and a need to perform scenario analyses may mitigate against using the second approach.

5. Discussion

The two approaches presented represent two distinct AOMDE lifecycle models. The models presented support the incremental development of applications with feedback loops. Incremental development of the primary model is facilitated through the morePM guard. The morePM condition is true whenever either primary model features for one or more requirements or one or more sub-systems remain to be designed. Aspects may also be developed iteratively. This is facilitated through the moreAspects guard. This condition is true whenever one or more aspect models remain to be designed. Therefore weaving can proceed as long as there are at least one available primary model feature and one associated aspect model.

In an aspect-oriented environment, errors in design and code may originate from the primary model, from the aspect model, or as an emergent property of the application. In any of these cases modification will need to be made to the primary model, one or more aspect models or to both aspects and primary model. This is supported in the approaches through the use of special error conditions, for example createError and enhanceError guards.

MDE is sometimes conceived as a unified process where code is generated from models and evolution of applications is model-centric based on a forward-engineering-only process. However, the two approaches presented in this article suggest that a variety of options are available. For example, in the Generate-Then-Weave approach code is generated separately for each aspect model and for the primary model. As a result, an error in an aspect model would require modification of the aspect and regeneration of only the code for that aspect. In contrast, an error in an aspect in the Weave-Then-Generate approach may actually require modification to the aspect, recomposition with the primary model, and code regeneration using the composed model. The two approaches presented in this article suggest that the code artifacts present in MDE environments will differ depending on the MDE approach selected. This difference in the code is likely to impact the decision to develop application by a forward-engineering-only approach versus a combination of forward and reverse engineering.

In summary, an optimal program evolution strategy in a MDE environment may actually involve a discretionary mix of forward and reverse engineering. Forward engineering is the obvious choice for generating the initial application code. However, a number of potential factors may affect the choice of forward versus reverse engineering when the software is to be modified. These factors include: (1) the AOMDE approach used (Weave-Then-Generate, Generate-Then-Weave or an hybrid approach), (2) the scope of the modifications, (3) whether the modifications are being made to aspect models or primary model, and (4) the nature of the modifications needed - adaptive, corrective or perfective. Further research is needed to determine how these factors are likely to impact an AOMDE environment.

6. Conclusions and Future Work

A model-centric approach to software development is needed to leverage the benefits of models in software engineering. Aspect-oriented approaches to model driven engineering are needed to help manage the complexity of cross-cutting concerns in MDE. In this paper two aspect-oriented approaches to model driven engineering have been presented. The differences in the approaches suggest that MDE in practice may not need to be the cohesive unary process that is sometimes suggested. Instead, a variety of potential options are available. Further research is needed to explore the details of the two presented approaches. For example, the Generate-Then-Weave approach seems to be a family of approaches rather than a single approach. This can be observed by noting that it is possible to generate primary model code and aspect code before customization.

In the generate-Then-Weave approach, when a design aspect is converted to code, part of the aspect may have to be added to the primary model of the application depending on the sophistication of the target aspect-oriented programming language. Further research is needed to determine the constraints and limitations of mapping design aspects into code.
References


