Reusable Aspect Models versus Theme/UML: the Aspect-Oriented Code Generation Perspective

Abid Mehmood and Dayang N.A. Jawawi

Department of Software Engineering, Faculty of Computing
Universiti Teknologi Malaysia (UTM), 81310 Skudai, Johor, Malaysia.
mabid4@live.utm.my, dayang@utm.my

Abstract

The integration of aspect oriented modeling approaches with model-driven engineering process achieved through their direct transformation to aspect-oriented code is expected to enhance the software development from many perspectives. This study aims to assess the existing UML-based aspect-oriented modeling techniques from the perspective of their suitability with regards to integration into model-driven engineering process through aspect-oriented code generation. For this purpose, we selected two mature aspect-oriented modeling approaches, Reusable Aspect Models and Theme/UML, and proceeded to evaluate them in a detailed way from the specific perspectives of design and its mapping to the implementation code. The in-depth comparison reveals some points equally shared by both approaches, and identifies some areas where one has advantage over the other. The study concludes that the Reusable Aspect Models approach may be seen as a preferred approach to handling the task of integration using aspect-oriented code generation.

Keywords: model-driven engineering, model transformation, aspect-oriented software development, code generation, reviews

1. Introduction

Aspect-oriented software development (AOSD) approaches [1-3] are essentially intended to improve the handling of a specific type of concerns, the crosscutting concerns, which represent the functionality that cuts across the primary modularization of a system. Such concerns usually seem to originate from non-functional requirements such as logging, security, persistence etc. A good example of this is the behavior related to authentication/authorization of a particular role in a software system. A straightforward representation of this behavior at modeling and implementation levels would most likely cut across all modules wherein updates are performed. Such representation results in problems associated with the phenomenon of scattering (a concern spread out over several modules) and tangling (one module representing more than one distinct concern) of behavior. Aspect-oriented techniques provide mechanism to explicitly identify, separate and encapsulate such crosscutting behavior (usually referred to as aspect behavior). Once the aspect behavior is encapsulated separately, composition mechanisms are also provided to control where and when this behavior is to be integrated with non-crosscutting behavior (usually referred to as the base). Therefore, in practice, aspect-oriented techniques are applied during analysis using Early Aspects [1], during design using Aspect-Oriented Modeling (AOM) [2], and using Aspect-Oriented Programming [3] for implementation. A few of the benefits of applying aspect orientation to software development may be found linked with maintainability, extensibility and reusability of the system, see for example [4-6].

The current study is specifically related to aspect-oriented design approaches (i.e., AOM approaches) and their subsequent integration with Model-Driven Engineering.
(MDE) process. In essence, MDE process makes models the primary development artifact and uses them as basis for obtaining an executable system. It emphasizes on subjecting models to a refinement process, through automatic transformations, until a running system is obtained. An integration of AOM approaches with MDE process (i.e., obtaining a final executable system from aspect-oriented models) can be realized through two different ways. First approach works purely at the modeling level and uses a model weaver to integrate the aspect and base models in such a way that a non-aspect-oriented (i.e., object-oriented) model is obtained. Subsequently, object-oriented code generation approaches are applied to generate code into one of the object-oriented programming languages. On the other hand, second approach directly transforms the aspect-oriented model into code of an Aspect-Oriented Programming (AOP) language and relies on weaver provided by the target language to deal with aspects. We focus on the second approach to integrating aspect-orientation with MDE process in this paper, i.e., on aspect-oriented code generation. The motivation behind this selection is provided in the following section. In this context, while we move towards aspect-oriented code generation, the modeling notations used to support aspect-orientation play a vital role, since the completeness of generated code is directly dependent on comprehensiveness supported by the modeling notation used. Moreover, since no modeling language has been adopted as a standard for aspect-oriented modeling, we consider it relevant to conduct an evaluation of the prominent aspect-oriented modeling approaches for their suitability to the purpose of code generation. Therefore, this study is focused on evaluating two well-published AOM approaches, namely Reusable Aspect Models [7-10] and Theme/UML [11-13], in the context of aspect-oriented code generation. In general, this comparison focuses on investigating the suitability of these approaches to serve the goal of integration of MDE and aspect orientation. For the purpose of comparison, we have taken a common example from aspect-oriented modeling literature and modeled it using both approaches. Further, we have also used the same example to investigate the model-code relationship of these two approaches, by mapping the design models of our example to AspectJ code.

In the following, Section 2 provides the motivation for this study and briefly discusses the related work. In Section 3, we define the strategy used for evaluation of selected AOM approaches, and describe the example system that we are going to use to support our discussion of the approaches. Section 4 is dedicated to detailed comparison of Reusable Aspect Models and Theme/UML approaches by means of developing design models of our example system and mapping the models to code. Further, we discuss the results in Section 5 and finally, conclude the paper in Section 6.

2. Motivation and Related Work

In previous section, we have described two different approaches that can be taken to obtain an executable from aspect models, i.e., weaving the aspect model to obtain a non-aspect model followed by generation of non-aspect code, and direct transformation of aspect model into aspect-oriented code. The core idea behind the first approach is to provide composition mechanism so that models can be simulated, tested and debugged prior to execution. Several model weavers have been proposed to achieve this. However, the main drawback in this approach is that it does not support separation of concerns that have once been composed during the weaving process. This means that benefits of clear separation of concerns become unavailable after the composition has been performed. This dilemma of losing clear boundaries of concerns while translating models into implementation further leads to problems related to evolvability, traceability, reusability and understandability of the developed software systems (cf. [14]). In contrast, second approach that proposes transformation of an aspect-oriented model directly into aspect-oriented code is mainly inspired by the benefits resulting from existence of a direct mapping between constructs of design model and the programming language. Moreover,
several empirical studies for example [4, 5, 15, 16] have reported the potential benefits of using aspect-oriented techniques in software development. Moreover, Hovsepyan et al., [6] have discovered that approaches that target aspect-oriented programming languages result in compact, smaller, less complex and more modular implementations. Hence, keeping in view the benefits of integration of aspect-oriented techniques with MDE through aspect-oriented code generation, we consider it worthwhile to evaluate prominent AOM approaches with respect to their suitability to serve as input to an aspect-oriented code generation process.

Previously, in [17], we have conducted a systematic mapping study of the related area with the aim to identify and classify existing research in context of aspect-oriented model-driven code generation. The results of study indicated the underdevelopment of the area of aspect orientation in general, and aspect-oriented code generation in particular. Some other work has presented comparison of aspect-oriented modeling approaches pursuing some distinguished goals. In this regard, Wimmer et al., [18], Chitchyan et al., [19], Reina et al., [20] and Op de beeck et al., [21] have all defined an evaluation framework to evaluate existing AOM approaches with focus on comparability in general.

As far as the integration of aspect-oriented modeling techniques into an MDE environment is concerned, the specific question of the suitability of various approaches for this integration by means of aspect-oriented code generation has not been explicitly investigated so far.

3. Evaluation Framework

In this section, we describe the evaluation framework used for this study. For this purpose, first the rationale behind the selection of approaches is given. This is followed by a description of the comparison approach used.

3.1. Selection of Approaches

In order to select two effective approaches for aspect-oriented modeling, we have adopted the criteria used for evaluation in one of the studies given in previous section, that is, Wimmer et al., [18]. We applied the Maturity criteria defined in their study to all existing AOM approaches in order to select two most matured approaches. It is to be emphasized here that the objective of this activity was only to select two effective approaches among existing ones, and that an extensive comparison was to be conducted in the current study. For this reason, we considered it appropriate to apply only the maturity-related criteria in this step. At this stage, two modeling approaches, which scored high against their criteria and appeared to be the most matured approaches, are Reusable Aspect Models and Theme/UML. These approaches are briefly described in the following.

3.1.1. Reusable Aspect Models

Reusable Aspect Models (RAM) [7-10] is a multi-view modeling approach that combines existing aspect-oriented approaches to model class, sequence and state diagrams into a single approach. Multi-view modeling provides means to describing a system from multiple points of view, using different modeling notations, thus allowing the use of the most appropriate modeling notation to describe facets different views of a system. RAM is different from all other AOM approaches in a sense that it views aspects as concerns that are reused many times in an application or across several applications. Therefore, this approach models any functionality that is reusable by means of an aspect. Hence, different views (i.e., structure, message, and state views) of a reusable concern are encapsulated in the form of an aspect model, which is essentially a special UML package. This aspect model comprises of three different compartments, representing the structural
view, state view and message view. These views are expressed using a UML class
diagram, state diagram and sequence diagrams, respectively.

3.1.2. Theme/UML

The Theme/UML [11-13] approach has basically evolved from work on composition
patterns [22, 23], and is considered one of the early approaches to aspect-oriented
modeling. In this approach, a new declaratively complete unit named “Theme” is
proposed at the design level to represent system concerns, which are seen as collections
of structures and behaviors inferred from requirements. A distinction has been made between
the “base” themes and the “aspect” themes, where aspect themes refer to crosscutting
behavior. An aspect theme may define some behavior that is triggered by behavior in
some other theme. Aspect themes are modeled by representing them as a new complete
unit of modularization, which is similar to a package in standard UML, with stereotype
<<theme>>. This theme may comprise of any of the standard UML diagrams to model
different views of the structure and behavior required for a concern to execute. Thus, an
aspect theme design is similar to a standard UML package, except for one difference, that
is, it contains specification of templates listed inside the theme package notation and a
sequence diagram for each of the templates grouping in the theme package. Currently,
package and class diagrams are used for structure modeling, whereas sequence diagrams
are used for behavior modeling.

3.2. Modeling Example: The Online Book Store System

As an appropriate modeling example to illustrate the RAM notation, an Online Book
Store (OBS) System has been adopted from MDE literature [24]. This system has
previously been used in AOSD literature to explain some AOM notations (e.g., [25]). It
has to be emphasized here that by selection of this example system, our intention is not to
prove the strength of the selected modeling notations, since it has already been shown by
means of the case studies and modeling examples evaluated in other reviews such as [18].
Instead, we have taken this example to apply the guidelines of both approaches with
regards to modeling and mapping, and consequently, to get a basis for conducting a
detailed analysis and comparison of both approaches.

In [24], the authors identify several use cases for the OBS System, for example,
ordering of books, cancelling of order, approving of charge, delivering of order etc.
However, following the presentation of [25], we will focus on the ordering of books,
which is described below:

1) Ordering of books use case starts by the customer selecting a book and its desired
   quantity.
2) The customer adds more books if desired.
3) When the customer is finished with adding selected books to the order and wants to
   checkout, a message is sent to the credit card company to process the payment. This
   adds a few related requirements as well:
   a) Since all purchases in OBS System are made in euros whereas the credit card
      company carries out all operation in US dollars, currency conversion is required
      while exchanging messages with the credit card company.
   b) For privacy reasons, all communication with the credit card company must be
      encrypted.
4) If the payment is processed successfully, a shipping order is created, followed by
   delivering a message to the delivery company notifying that a new order is ready.
5) In case an order is changed, the change is persisted. This adds another global requirement:

a) All update operations for order must be persisted.

An excerpt of high level architecture of the OBS System is shown in Figure 1. Apart from providing the core functionality of the OBS System, the OBS Core component is also responsible for handling the persistence of orders. The external services of payment and delivery are connected through a public network. The details of shipment delivery are uninteresting; therefore, we will concentrate only on payment, encryption and persistence of an order while performing functionality related to a new order.

![Figure 1. Excerpt of the High Level Architecture of OBS System](image)

Figure 1. Excerpt of the High Level Architecture of OBS System

Figure 2 shows the high level communication of objects to handle a new order. The crosscutting functionality has been highlighted in grey regions. It should be observed that these aspectual parts will be woven by the weaver, and will not be explicitly called. For example, a chargeCreditCard request dispatched to a credit card payment controller will result in inherently invoking a currency converter, which will convert euros to dollars and pass the order further, and an encrypter, which will encrypt the charge request to make it suitable for dispatching on public network. Therefore, if the aspectual part is eliminated, an order will directly be communicating with a credit card payment controller. This means that the chargeCreditCard request is not actually sent to CurrencyConverter. Similarly, paymentApproved is not returned to an encrypter.

![Figure 2. High Level Communication of Objects to Handle a New Order](image)
In the following, we model the crosscutting concerns of OBS System using the selected modeling notations and proceed to mapping the models to AspectJ code.

4. In-depth comparison of Theme/UML and Reusable Aspect Models

In this section, we take our example application and model some of its relevant parts using both the RAM and Theme/UML approaches. Then, we map these models to AspectJ code in order to carry out a thorough comparison of the obtained code.

4.1. The Reusable Aspect Models (RAM) Approach

As we briefly discussed in Section 3.1.1, the RAM approach views aspects as concerns that are reused many times in an application, or across several applications. Therefore, this approach models any functionality that is reusable by means of an aspect. In OBS System, the crosscutting concerns of Persistence, Currency Conversion, and Encryption are modeled as separate reusable aspects. A brief description of these aspect models follows.

4.1.1. Design of Online Book Store System using RAM

The Persistence aspect model for the OBS System is shown in Figure 3. The aspect, which is represented in the form of a special UML package, comprises of different compartments. First compartment represents the structural view, and is expressed using an extended UML class diagram. It is to be noted here that the classes in this compartment do not necessarily have to be complete. They may include methods and attributes that are relevant to the particular concern only. Such incomplete classes which are later to be composed with other classes by the weaver at time of instantiation of aspect, or its binding to a base model, and methods whose names and signatures are to be determined later, are declared as mandatory instantiation parameters. The structural view of the Persistence aspect contains two classes, DBManager and |Persistable. The latter is an incomplete class and hence, in addition to having its name prepended with a "|" character, it is made prominent by specifying it as UML template parameters on top right corner of the structural view section. Persistence provides methods to handle functions related to persistence.

![Figure 3. The Persistence Aspect Modeled using RAM](image-url)
The public method `isUpdateRequired` intends to identify situations where orders have really been updated, and thus need persistence. In case an update operation has been performed, the persistence is carried out by `persist` method of `DBManager`, which is responsible for handling all the database-specific functionality related to persistence of objects.

As described previously, in RAM, following the structural compartment, several state view compartments are added, where each one corresponds to some class defined in the structural view of the aspect model. State view contains UML state diagrams to describe the internal states of the class which are relevant within the concern. For complete classes (i.e., standard classes) in the structural view, the state diagram is a standard state diagram. However, for incomplete classes, in which concerns are to be injected later, an *aspect state diagram* is defined, which contains two parts: a pointcut and an advice. The pointcut part is used to define the states and transitions that are required in target state diagram, whereas the state diagram that replaces the occurrence of pointcut in the target state diagram is defined by the advice part. Similar to structural view, states are designated to be mandatory instantiation parameters in state view if their binding to states in standard state diagram does not exist. Such states are also annotated in a way similar to structural view. The pointcut view in the state view of Figure 3 shows three relevant states within the *Persistence* aspect, i.e., `UpdateRequired`, `Updated` and `Any`. The advice part shows the capability of relevant states to call `update`, `isUpdateRequired` and `isUpdated` methods. The state view compartments in RAM are followed by message view compartments. The message view uses a UML sequence diagram to describe the sequence of messaging between different entities during the execution of a public method given in structural view of the aspect model. The message view compartment in the *Persistence* aspect model defines only one message view, i.e., `update` message view. The pointcut in this message view states that all such situations are relevant in the context of this aspect where an update method is called on an instantiation of the `Persistable` while some form of it is passed as an argument to this method. The advice states that all such method executions must also “persist” the updated order, by initializing a connection to the `DBManager` and calling its implementation of the `persist` method. It should be noted that prior to making a call to `persist` method, the method `initialize` is explicitly invoked in order to avoid repeated execution of functionality to establish and maintain database connectivity.

In order to instantiate and use an aspect, its mandatory instantiation parameters need to be mapped to concrete model elements in a target model. Therefore in this case, the class `Persistable` shown as the UML template parameter on the top right corner of the *Persistence* aspect package must be mapped to some class in the target model.

Figure 4 shows the *CurrencyConversion* aspect modeled as a reusable aspect using the RAM notation. The structural view defines one incomplete class `Convertible` to handle currency conversion. The `convert` method of this class is intended to be woven into all classes that need its functionality. In this regard, the specific details of exchange rates etc. as well as the details of conversion are to be handled by a complete class `Currency`. The *CurrencyConversion* aspect defines two state views, one for the incomplete class `Convertible` and the other for complete class `Currency`. The `Convertible` state view defines two relevant states `ConversionRequired` and `Converted`. The advice suggests that in order to accept calls to `convert`, an implementing object must be in `ConversionRequired` state. Since the `Currency` state view defines the protocol for a standard class (and not an incomplete one), it takes the form of a standard state diagram. Specifically, it defines three relevant states namely `CurrencyToConvert`, `ConvertedCurrency` and `Any`. First state is the one in which a call to `convert` method is possible, whereas the state diagrams shows that an implementing object may be in `Any` state to make a call to `getConversionRate`.  


In order to use the \textit{CurrencyConversion} aspect, a modeler must map the class \textit{Convertible} shown as a mandatory instantiation parameter to one of the elements in target model. The message view compartment in the \textit{CurrencyConversion} aspect model defines only one message view, \textit{i.e.}, \texttt{convert} message view. In line with the description of the payment functionality given above, the pointcut in this message view takes all such calls to accept method as relevant wherein \texttt{convert} method of a \textit{Convertible} object is called. The advice specifies that in all such cases, a conversion will need to be made by calling \texttt{convert} method of the \textit{Currency} class that will handle the finer details of the required conversion.

Figure 5 shows the \textit{Encryption} aspect which contains only one class \textit{Encryptable}. The model is self-explanatory in view of the previous discussion on RAM notation. Therefore, we skip discussing it here.

![Figure 4. The Currency Conversion Aspect Modeled using RAM](image)

In a similar way to previously described aspects, the \textit{Encryption} aspect may be instantiated and used by mapping the mandatory instantiation parameter, \textit{i.e.}, the class \textit{Encryptable} to one of the elements in target model.
Figure 5. The Encryption Aspect Modeled using RAM

4.1.2. Mapping of RAM design to AspectJ Code

In [26], the authors propose a mapping scheme for mapping of RAM models to AspectJ code. In this section, we adopt their guidelines to the specific needs of this study, and present a mapping of aspects developed using RAM to AspectJ language.

In the following, we define the overall structure of the mapped code and some global items, and then we proceed to mapping different modeling elements along different views in aspects defined using RAM. Specifically, the global structure of the code is defined as follows: a main package named `obss.ram` is created for the RAM project as a whole. This package will contain code for all aspects defined so far. Within this main package, two subpackages named `obss.ram.aspects` and `obss.ram.conflictresolution` are created. In the proposed mapping technique, a sub package is created in the aspects package for each ordinary RAM aspect which has the same name as that of the aspect. This package contains all the artifacts of the RAM aspect. Figure 6 shows three packages representing the three aspects modeled for OBS System.

```
1 obss.ram.aspects.persistence
2 obss.ram.aspects.encryption
3 obss.ram.aspects.currencyconversion
```

Figure 6. Ordinary Aspect Packages

Following the definition of subpackages that contain all artifacts of aspects, three required aspects are created directly into the main package. The three required aspects are shown in Figure 7.

```
1 obss.ram.AnnotationInheritance
2 obss.ram.ConfigurationEnforcement
3 obss.ram.AspectPrecedence
```

Figure 7. Required Aspects in the Main Package

After defining the overall code architecture, we proceed to mapping different views belonging to individual aspects to AspectJ code, and start with the Persistence aspect. While mapping classes in the structure view, first, the complete classes are mapped. The mapping approach of Kramer and Kienzle [26] propose a reuse mechanism to make use of the Java library, for classes and interfaces that resemble to existing ones in Java. Since this is not the case for our modeled classes, we move on to generating their Java implementation from scratch. Therefore, for every complete class, we create a new public
Java interface and an AspectJ aspect within the source file of this interface. In the next step, fields and methods are introduced into this interface using AspectJ’s inter-type declaration mechanism. The code for DBManager complete class in the Persistence aspect is shown in Figure 8.

```java
public interface DBManager {
    //signatures of public methods in structural view
    void initialize();
    void persist (Persistable persistableObj);
}

aspect DBManagerAspect {
    //methods without message views
    public void DBManager.initialize() {}
    public void DBManager.persist(Persistable persistableObj) {}
}
```

**Figure 8. Mapping of the Complete class DBManager**

Apart from generating an interface, for complete classes in RAM aspects, a public class is created that implements that interface (generated in previous step). The name of this class is generated by appending the string Impl to the name of the modeled class (see Figure 9). It should be noted that these classes are used for instantiation purpose only, and are empty except for possible constructors.

```java
public class DBManagerImpl {
    public DBManager() {}
}
```

**Figure 9. Implementation Class for DBManager Interface**

```java
public interface Persistable {
    //public methods in structural view
    boolean isUpdated();
    void update(Persistable data);
    boolean isUpdateRequired();
}

aspect PersistableAspect {
    //attributes and associations in the structural view
    private boolean Persistable.updated = false;
    private DBManager Persistable.persister = new DBManagerImpl();
    //methods without message views
    public boolean isUpdated() {}
    public boolean isUpdateRequired() {}
    //methods containing message views
    public void Persistable.update(Persistable data) {
        DBManager persister = new DBManager();
        persister.persist(data);
    }
}
```

**Figure 10. Mapping of Incomplete class Persistable**

Following the complete classes, we map the incomplete classes in our aspects. For these classes, the process of matching to standard Java library is skipped, and an implementation class in not generated, since the incomplete classes cannot be instantiated. Figure 10 shows the code obtained by mapping the incomplete class Persistable. Specifically, it shows that the attributes of Persistable are mapped as variables of the respective types within the aspect using AspectJ’s inter-type declarations. Associations are mapped as a special case of introducing Java fields. The persister field in Persistable interface represents an association of this class with DBManager (see Figure 10). Code stubs are generated for methods that are not
associated with a message view. Just like attributes, methods that have a message view are added using the inter-type declaration mechanism of AspectJ. The implementation of such methods is generated by adding method calls specified in the message view. See for example, the initialization of a persisted object and its invocation of the persist method given in Figure 10, which is based on the message view of Persistence aspect.

Currency interface and CurrencyAspect are implemented in a similar way. The code output from this step will be just like that obtained for DBManager in Figure 8, therefore, we have not presented it here for space reasons. Convertible interface in Figure 11 defines a method for the only public method in its structural view and provides its implementation within the aspect ConvertibleAspect. The details of conversion were considered irrelevant to this study and hence omitted from the message view.

```java
public interface Convertible {
    //signatures of public methods in structural view
    void convert(Currency source, Currency target);
}

aspect ConvertibleAspect {
    //methods with message views
    public void Convertible.convert(Currency source,
                                     Currency target) {
        int amount = new Currency().convert(source, target);
    }
}
```

Figure 11. Mapping of the Convertible Incomplete Class

As far as the Encryption aspect in our example is concerned, it is a rather simple aspect and its mapping is straightforward. Specifically, Encryptable interface will contain three methods: encrypt, decrypt and isEncrypted. The EncryptableAspect will provide implementation of these methods besides declaring a field named encrypted.

4.2. The Theme/UML Approach

In this section, the Theme/UML models of the selected aspects of the OBS System are presented. This is followed by a brief description of the mapping process, and the mapping of design to code.

4.2.1. Design of OBS System using Theme/UML

As briefly described in Section 3.1.2, the Theme/UML approach makes a distinction between the “base” themes, and the “aspect” themes which refer to the crosscutting behavior. As far as modeling process is concerned, first the triggered behavior needs to be identified and captured in the form of templates, and then the crosscutting behavior related to those templates is modeled [11].

The design of Persistence aspect model in Theme/UML notation is shown in Figure 12. As shown, the aspect theme design is similar to a standard UML package along with structural and behavioral diagrams, but is different in the way that it contains templates listed inside the theme package notation, and contains a sequence diagram for each of the templates grouping in the theme package. It should be noted that each theme in this approach is intended to show only the classes and behavior that are necessary to represent its concepts. This means that each theme includes classes that it requires from its own
perspective, and specifies them regardless of whether other themes also have classes to represent the same concepts. Any conflicts and overlaps that may arise are considered at a later stage. Therefore, initially, a design may contain different themes having different versions of the same class, each representing what is of interest for that specific theme.

However, to crosscut a theme, the crosscutting theme requires abstract knowledge of the themes it crosscuts; crosscutting themes cannot operate independently. This external behavior in the crosscut theme is referred to by means of an extension of UML’s notion of templates in a parameterized way; see for example the update operation of Persistable class exported as a template in Figure 12. The semantics of this declaration specify that the functionality of update method be augmented with some additional behavior defined by the sequence diagram given in the theme. Consequently, as given in the sequence diagram, a call to update method will essentially trigger execution of the persistChange method, which will in turn call initialize and persist functions on the Persister object.

Next, the functionality related to currency conversion is modeled as a crosscutting theme named CurrencyConverter (See Figure 13). Specifically, this theme declares all such situations where a conversion is required as the crosscut points, and specifies the same by means of a template method aConversionRequired of the Converter class. Any invocation of this scenario results in supplementing the real behavior with behavior contained in convert method of the Converter class, which indeed invokes the convert function of Currency class and so on. The details of currency conversion have been intentionally omitted since we consider them uninteresting with regard to the discussion in this paper.

The theme/UML model of the Encryption aspect is not different from the RAM model of the same, mainly because it contains no behavioral logic. Therefore, we have not reproduced it in this section.

4.2.2. Mapping of Theme/UML Design to Code

Unlike the mapping approach for Reusable Aspect Models which starts with defining a global architecture of code, the Theme/UML approach directly maps the individual aspect themes. The Theme/UML mapping for the Persistence theme is shown in Figure 14. We start with defining an abstract aspect to represent the Persistence aspect. Within this
aspect named Persistence, an interface is declared for the template class Persistable. Further, the mapping approach requires that an abstract pointcut be declared for each template method that initiates a sequence diagram.

```java
abstract aspect Persistence {
    // interfaces
    public interface Persistable { }
    // abstract attributes
    boolean Persistable.updated = false;
    Persistor actualPersistor = null; // association
    // aspect methods
    public boolean Persistable.isUpdatedRequired() { }
    public void Persistable.persistChange(Persistable p) { }
    // abstract pointcut
    public abstract pointcut update(Persistable persistableObject);
    // advice
    after (Persistable persistableObject) oncall:
    update(persistableObject){
        actualPersistor = new Persistor();
        actualPersistor.initialize();
        actualPersistor.persist(persistableObject);
    }
}

// the Persistor class
class Persistor {
    private boolean initialized = false;
    public void initialize() { }
    public void persist(Persistable p) { }
}
```

**Figure 13. The Currency Conversion Aspect Modeled using Theme/UML**

Therefore, we have declared the update pointcut that is associated with after advice relative to the sequence of behavior defined in the sequence diagram in Figure 12.

Associations are defined using Java’s native mechanism by defining variable of the target type (see e.g. Persistable.updated). Attributes and operations that are not declared as templates are implemented by defining an inter-type declaration on the class interface (see e.g., Persistable.updated and Persistable.isUpdateRequired).
Finally, classes without template operations are directly mapped as ordinary classes. This is in contrast to RAM’s approach in which a class is mapped to an interface definition. The same set of guidelines is applied to map the CurrencyConversion theme. The resultant code is shown in Figure 15. Notice how the one-to-many association named currencyConverter in Figure 13 has been mapped to a single reference of Currency class in the CurrencyConverter aspect. Theme/UML mapping approach does not give much detail on mapping different types of associations.

5. Discussion

In the following subsections, we summarize the key findings of this study and discuss some effects of these findings on the future research in this area. We divide our discussion into two parts: first we provide discussion from the modeling perspective and then we consider the code perspective.

5.1. The Model Perspective

As far as the models of Online Book Store System are concerned, there is a huge resemblance between the concepts supported by RAM and Theme/UML approaches. Specifically, both approaches use complete units of modularization, i.e., aspects and themes respectively. Both approaches explicitly work to make their aspects as generic as possible through the use of UML templates with template parameters. In both approaches, the generic aspect model is instantiated by binding the aspect model’s template parameters to elements of the target model. Nevertheless, in the process of applying guidelines provided by each approach to model our system, we have observed that the two approaches differ from some perspectives, and that in different cases one approach may have advantage over the other. These observations are briefly described in the following.

Unlike RAM, Theme/UML approach does not support the current version of UML. Apart from the observation that current UML tools often provide support for the current standard, there are few other differences in two standards, which may have some repercussions for the models developed using this approach. For instance, sequence diagrams are used as the main behavior modeling tool in Theme/UML, but several new features for UML 2.x sequence diagrams such as fragments, interaction occurrences, explicit representation of loops, notation for showing creation and destruction of objects etc., are not supported [27]. Due to these features, in contrast with UML 1.x sequence diagrams, UML 2.x sequence diagrams can work in two different forms: instance form

```
abstract aspect CurrencyConverter {
   //interfaces
   public interface Converter { }
   Currency curr = null; //association: details of one-to-many
    // association are absent
   //aspect methods
   public boolean Converter.convert() { }
   //abstract point cut
   public abstract pointcut aConversionRequired(Converter objectToConvert); //advice
   after (Converter objectToConvert) oncall: convert(objectToConvert) {
      curr = new Currency();
      curr.convert(objectToConvert);
   }
}
// the Currency class
class Currency {
   public void getConversionRate() { }
   public void convert(Currency target) { }
}
```

Figure 15. Mapping of the Currency Conversion Theme
and generic form. The instance form describes one possible interaction in a specific scenario, whereas the generic form documents all possible alternatives in a scenario.

In terms of relationship of aspect-oriented design models with overall software development life cycle, Theme/UML provides a detailed set of rules and guidance from analysis to implementation phases. In fact, a distinct part of theme approach, i.e., Theme/Doc is used to handle analysis phase. In this regard, although RAM does not propose a complete methodology, but the existing large case studies provide sufficient insight into the design process, and guidelines to model systems effectively.

As far as the modeling diagrams are concerned, both approaches support structural as well as behavioral diagrams. Specifically, both approaches make use of class diagrams to model structure. For behavior modeling, RAM uses state diagrams and sequence diagrams. On the other hand, although theme approach is intended to support any type of diagrams, the support is not precisely defined except for sequence diagrams. Sequence diagrams are often suitable to showing collaborations among various objects involved in a single use case. However, they are not so good at precisely defining the behavior of an object [28]. The state diagrams and state machine specifications are considered the most effective and widely used method to specify behavior of a system [29, 30]. This way, the use of state diagrams in RAM approach is significant.

Despite the fact that the focus of this study is not on model weaving, some findings are interesting. The aspects in RAM approach are woven using explicitly-defined directives for the instantiation and binding, which improves internal traceability of models. No such directives are provided in Theme/UML models which use a merge operator to specify weaving of models. However, until now, the weaving with Theme/UML approach must be done manually.

As its name suggests, the reusability of aspect models is a major strength of the Reusable Aspect Models approach, whereas Theme/UML approach does not elaborate the reuse mechanism for its unit of modeling, i.e., theme. In this regard, RAM supports the reuse of its aspect packages through creation of aspect dependency chains. An aspect providing complex functionality is modeled by decomposing it into many aspects that provide simpler functionality, and vice versa. This phenomenon also helps to hide the indirect dependencies of aspects from the user of an aspect.

Finally, tool support is another important factor while considering an aspect-oriented modeling approach. As we mentioned previously, unless supported by appropriate tools, an AOM approach may raise the complexities of a modeling process. Theme/UML does not provide any tool support, neither for aspect modeling nor for aspect weaving. On the other hand, RAM comes with a tool developed in Kermeta [31] environment, which runs within the Eclipse Modeling Framework the use of Eclipse tools to edit, store and visualize models. The current RAM tool also supports the reusability of aspects and provides an inherent mechanism for consistency checks.

5.2. The Code Perspective

Reusable Aspect Models and Theme/UML share some common points, e.g., both are asymmetric approaches, both use a dedicated unit for encapsulation of aspects; both extend the UML Meta model and define new constructs etc. As a result the code obtained from mapping to AspectJ is similar in many ways. However, few notes on the resultant code, which may lead to selection of one approach over the other, are presented in the following.

RAM approach explicitly considers the overall structure of code and provides few guiding principles to lead the mapping process [26]. Although no such information is available in current mapping approach of Theme/UML, a similar set of instructions may be applied, since the aspect package of RAM resembles the theme package of

1 http://www.cs.mcgill.ca/~joerg/SEL/TouchRAM.html
Theme/UML. With regard to overall structure, however, the role of required aspect packages in RAM (see Figure 7) is important. The AnnotationInheritance package is actually a repository of all declarations in an annotation hierarchy. As described previously, annotations are a way to map instantiation directives present in RAM models; these directives assign classes or methods to mandatory instantiation parameters (e.g., See Figure 11). As we discuss in the following, Theme/UML approach does not make use of annotations, hence no such structure mapping is available. The special AspectJ aspect named ConfigurationEnforcement is related to the reuse of aspects. Specifically, since RAM allows different configurations of an aspect at code level, each representing a variation of features, it needs some mechanism to keep a record of all possible variations. Thus for example, if two alternative variations of an aspect are not allowed to be used simultaneously, a definition of this configuration in the ConfigurationEnforcement aspect would specify that involved classes are not allowed to be marked with annotations corresponding to alternative variations. The AspectPrecedence aspect is also related to reuse hierarchies, and aspect dependency chains described in Section 5.1. Specifically, it keeps a list of all aspects that are reused in a context in the order of their reuse, to keep track of the precedence rules.

In Theme/UML ordinary classes, i.e., complete classes are mapped to corresponding classes in Java containing plain variable declarations and function stubs. The approach of RAM to map complete classes to Java interfaces and a corresponding interface-implementing class is interesting. In the first step, to allow reuse of Java library, complete classes are checked to find their equivalent in a custom-built library of supported classes and interfaces from Java standard library. In case an equivalent is not located, it is mapped to a public Java interface that contains the aspect declaration. By doing so, instead of mapping a class to a standard class in Java, the approach allows merging of this class with other classes based on binding directives. Since multiple inheritance is not supported by Java, merging of a class with other classes can only be made possible by the means of its support for implementing multiple interfaces. Theme/UML approach does not consider reuse of existing Java classes or interfaces, and relies only on its own mechanism to generate classes from scratch.

As far as incomplete classes are concerned, both approaches implement them with interfaces. In Theme/UML, an interface implementing an incomplete class (or more specifically a pattern class in Theme approach) contains all non-template operations as methods. A template operation is mapped to an abstract method (if it is not associated with a sequence diagram) or an abstract pointcut (if it has a sequence diagram). On the other hand, RAM uses the Java annotations for mandatory instantiation parameters. In our opinion, although the code resultant from Theme/UML looks concise, mapping template operations in Theme/UML’s way has some drawbacks. First, mapping a template operation to abstract method means that it needs to be bound to the target class by providing a corresponding implementation of the method with a delegating call. This is obviously more verbose than RAM’s approach of marking existing methods with annotations. Second, mapping a template operation (with supplementary behavior) to an abstract pointcut means that it requires the definition of concrete pointcuts for methods to be bound to target classes. Definition of concrete pointcuts will prevent this code from being used in other Java projects since pointcuts are not supported in pure Java. On the other hand, since annotations are supported by pure Java, the RAM’s implementation can be reused in other standard Java projects.

With regards to the mapping of sequence diagrams, since Theme/UML is based on UML 1.x, the mapping for an important feature of UML 2 sequence diagrams, i.e., notion of Sequence Fragments [27] (sometimes referred to as Interaction Frames cf. [28]) could not be provided yet. Sequence fragments are essentially a control flow construct that is represented by a rectangular box surrounding a portion of sequence diagram and overlapping the section in which a fragment occurs. There are several different types of
sequence fragments; currently only a few of them are supported by RAM approach. The supported fragments are: (1) option combination fragment, which is mapped to an *if* statement in Java, (2) alternation combination fragments, which are implemented using *if* followed by *else-if* statements in Java, (3) loop combination fragments, which are implemented using Java’s *for* and *while* loops. There are several other types of fragments which are not supported by RAM approach, *i.e.*, *ref*, *assert*, *break*, *neg*, *par*, *region*, *sd* fragments.

Having considered current support for feature set of sequence diagrams, it has to be emphasized again that the sequence diagrams are not considered the best tool to precisely defining the behavior of an object. In this regard, as we mentioned previously, state diagrams are considered the most effective and widely used method of specifying system behavior [28, 32]. Therefore, they can be considered a more effective tool for full code generation than any other UML diagram. That is why, a considerable volume of literature (cf. [29]) is devoted to studying the implementation of state diagram in different programming languages. Currently, both RAM and Theme/UML have not considered implementation of state diagrams.

6. Conclusion

In this paper we provide an evaluation of two AOM approaches with respect to their potential to integrate into an MDE environment by means of aspect-oriented code generation.

The results of our study show that Reusable Aspect Models and Theme/UML resemble in many ways with respect to handling modeling and mapping details of aspects. However, there are some points where RAM approach has advantage over Theme/UML such as: (1) its support for the latest UML version, (2) use of state diagrams for modeling behavior, (3) explicit definition of directives for the instantiation and binding, (4) its strength in terms of reusability of aspects both at design and code levels, (5) tool support, (6) improved handling of code structure, and (7) mapping of advanced features of sequence diagrams.

References