AJFCode: An Approach for Full Aspect-Oriented Code Generation from Reusable Aspect Models

Abid Mehmood[1*] and Dayang N.A. Jawawi[2]
[1] Department of Management Information Systems, College of Business Administration, King Faisal University, Al-Ahsa 31982, Saudi Arabia.
[e-mail: aafzal@kfu.edu.sa]
[2] Department of Software Engineering, Facutly of Computing, Universiti Teknologi Malaysia, Johor Bahru, Malaysia.
[e-mail: dayang@utm.my]
[*]Corresponding author: Abid Mehmood

Received March 16, 2022; revised May 22, 2022; accepted June 1, 2022; published June 30, 2022

Abstract

Model-driven engineering (MDE) and aspect-oriented software development (AOSD) contribute to the common goal of development of high-quality code in reduced time. To complement each approach with the benefits of the other, various methods of integration of the two approaches were proposed in the past. Aspect-oriented code generation, which targets obtaining aspect-oriented code directly from aspect models, offers some unique advantages over the other integration approaches. However, the existing aspect-oriented code generation approaches do not comprehensively address all aspects of a model-driven code generation system, such as a textual representation of graphical models, conceptual mapping, and incorporation of behavioral diagrams. These problems limit the worth of generated code, especially in practical use. Here, we propose AJFCode, an approach for aspect-oriented model-driven code generation, which comprehensively addresses the various aspects including the graphical models and their text-based representation, mapping between visual model elements and code, and the behavioral code generation. Experiments are conducted to compare the maintainability and reusability characteristics of the aspect-oriented code generated using the AJFCode with the most comprehensive object-oriented code generation approach. AJFCode performs well in terms of all metrics related to maintainability and reusability of code. However, the most significant improvement is noticed in the separation of concerns, coupling, and cohesion. For instance, AJFCode yields significant improvement in concern diffusion over operations (19 vs 51), coupling between components (0 vs 6), and lack of cohesion in operations (5 vs 9) for one of the experimented concerns.

Keywords: aspect-oriented software development (AOSD), AspectJ, automated code generation, model-driven engineering (MDE), software design.

This work was supported through the Annual Funding track by the Deanship of Scientific Research, Vice Presidency for Graduate Studies and Scientific Research, King Faisal University, Saudi Arabia [Project No. AN000160].

1. Introduction

Model-Driven Engineering (MDE) is a software development approach that makes use of models as the primary development artifact. The main idea is to transform the models automatically from one level of abstraction into another more detailed level and to continue such automatic transformations until the final application code is obtained. The resultant higher level of abstraction in systems development essentially leads to an improved understanding of complex systems. In the MDE context, automatically generated code offers some obvious benefits such as reduced time to develop, less unintentional syntax mistakes, greater consistency between code and design [1, 2]. Some recent examples of the use of MDE in various contexts include the approaches presented for fault injection in Java code [3], for simplifying the design and development of IoT-based monitoring systems [4], for engineering cyber-physical systems [5], for performance testing in mobile applications [6], and for distributed ledger deployment [7]. Aspect-Oriented Software Development (AOSD) [8, 9] provide a software engineering approach that allows an explicit way of identifying, separating, and encapsulating the so-called crosscutting concerns. Some typical examples of such concerns are the concerns related to non-functional requirements such as logging, security, and persistence. As these concerns cannot be fully decomposed from the main functionality using object-oriented techniques, they cannot be effectively modularized. With AOSD, on the other hand, the crosscutting concerns are implemented as individual modules and then composed into primary modules when their behavior is to be applied. The effective modeling of concerns improves the reusability and maintainability of software, which in turn lead to greater flexibility and extensibility [10-12]. Recently, studies have reported the benefits of AOSD in the development of middleware for IoT [13], in user activity detection [14], in supporting organizational patterns [15], and in smart contract development in the blockchains context [16], to name a few.

MDE and Aspect-Oriented Software Development AOSD have some complementary properties. MDE elevates the abstraction level, but it has limitations with regards to refining and integrating the system perspectives. AOSD is particularly effective in modularizing and composing concerns, but it lacks proper abstraction techniques. Therefore, an integration of the two is deemed to provide a two-fold benefit: adding the excellent abstraction mechanisms of MDE to AOSD and augmenting the MDE with strength of AOSD with regards to modularizing and composing the concerns. The aforementioned integration has been explored in the literature in two different ways: by employing model weavers and by directly generating aspect-oriented code. Model weaver approaches (sometimes referred to as weave-then-generate (WTG) approaches such as [17-19] take the base model and the aspect model and weave them together to obtain an object-oriented model. The resultant model is then transformed into the code of an object-oriented programming language using standard code generation techniques (e.g., [20, 21]). An integration carried out in this way may work effectively when analyzing or executing models. However, the resultant object-oriented code lacks the aspect features of the model and hence loses the separation of concerns. This defies the purpose for which aspect modeling was initially adopted and again exposes the system to maintenance and other issues [10, 22].

To address this issue, the aspect-oriented (AO) code generation approaches (also referred to as generate-then-weave (GTW) approaches) such as [23, 24] focus on the transformation of source AO model directly into the code of an AO language. The weaving of concerns is done by the weaver provided by the target programming language. Thus GTW approaches inherently benefit from strengths of AO-based methods as investigated empirically by studies

(e.g., [25, 26]).

Aspect technologies at the implementation level are already well-known in industrial circles. Some leading implementation framework designers (e.g., JBoss, Spring) have adopted them. However the present AO code generation methodologies are not without some significant flaws. First, they fail to fully elaborate and exploit the model-code relationship to address issues specific to code generation. Second, the existing AO code generation approaches do not provide a formal way to transform visual models into text-based models, which are program-savvy, and thus can be used for systematic generation of code. Third, the existing approaches lack support for the "base" part of the model, which characterizes the non-crosscutting (base) feature set of a system. Fourth, none of the present approaches generates code related to object's behavior. Because of these weaknesses, the existing research does not provide a practical and adequate integration of AOSD and MDE.

This paper proposes a solution that addresses all the weaknesses of existing GTW approaches stated above. Specifically, the research proposes AJFCode—an approach to integrate MDE and AOSD by means of AO code generation. It takes models developed using Reusable Aspect Models [27] notation and generates AspectJ code for structure and behavior represented by the model. The key contributions of this work are given in the following.

- We elaborate a comprehensive approach considering all facets of a model-driven code generation approach including the graphical models and their text-based representation, mapping to code, and behavioral code generation.
- We extend a method of mapping complete AO models comprising structural details and object's behavior to AO code.
- AJFCode exploits a well-defined text-based model to represent the graphical model in text form to enable the subsequent transformation into code.
- We provide a method of obtaining aspect-oriented code from aspect-oriented state chart diagrams.
- We provide a method for AO code generation that generates structure and behavior code for the base as well as crosscutting (aspectual) segments of the model. To the best of our knowledge, AJFCode is the first approach to generate AO behavioral code.

Following this introduction, other works related to the current study are discussed in Section 2. The details of code generation approach are presented in Section 3. Section 4 presents the results of the evaluation of AJFCode. Section 5 concludes the paper.

2. Related work

The existing research related to AO code generation can be viewed in two broad categories. First, some approaches explicitly address the code generation from models. The second type of works do not address the code generation directly, but they contribute significantly to the AO code generation goal in another way, e.g., by formalizing mechanisms for transformation between AO models and AO code. In the following, we describe the existing AO code generation approaches.

The transformation-based approaches define a model to code transformations based on existing transformation techniques. Hecht, et al. [28] develop an XML representation of Theme/UML [29] models and apply the Theme's transformation approach to map models to code. The XSLT-based code generator uses XMI to implement transformation from UML to XML. In a similar work [30], visual models of multi-modal scenario-based system specifications have been transformed using a pattern-based technique to transform Live Sequence Charts into AspectJ. The template-based approach of Evermann et al. [31, 32]

obtains the code from the UML-based specification of AspectJ meta-model. The actual code generation is implemented using UML XMI model interchange abilities. Bennett et al. [24, 33] exploit some of the obvious advantages of graph-based transformations to generate AO code. This approach makes use of automated transformations first to transform the models developed in FDAF [34] into XML-based models and then transform these text models into AspectJ code. The aforementioned approaches provide basic support for AO code generation, but they do not address the advanced issues such as the model-to-code relationship for behavioral elements at model and code level.

Another category of approaches defines a direct mapping of the source model onto the constructs of a programming language. Therefore, they do not address the details of the transformation process. An approach to generate AspectJ code stubs from extended UML models has been proposed by Groher and Schulze [35]. They have integrated the Borland's Together CASE tool. The approach first obtains the object-oriented code for the base elements and then uses the extension mechanisms of the tool to implement AO code generation. Haitao et al. [36] interpret the AO domain-specific models to obtain AO code in AspectC++. This approach models the crosscutting concerns as separate aspects and then defines a model interpreter to generate code by traversing the aspects. Kramer and Kienzle [23] have provided a mapping of Reusable Aspect Models (RAM) into Java and AspectJ. Similarly, mapping of Theme/UML models to AspectJ has been provided in [29] and [37] essentially analogously. Some work has specifically focused on identifying and solving the common problems in mapping the design concepts to the constructs of code. Modeling Aspects using a Transformation Approach (MATA) [19, 38] is a graph transformation-based approach to composing UML aspect models, which also elaborates the mapping of MATA models to AO code in AspectWerkz. Loukil, et al. [39] have presented support for AO code generation from models of their AO extension (AO4AADL) of the Architectural Description Language (ADL). Even though their aim in this work is not to propose an AO code generation approach, they have defined a set of transformation rules for mapping of AO4AADL aspects into AspectJ.

As evident from discussion above, the key focus of GTW techniques has been structural models (more specifically class diagrams) only. Though obtaining such code is straightforward, it is extremely limited in extent (skeletons only). On the other hand, the diagrams that enable a more effective modeling of system's behavior such as state diagrams or sequence diagrams are quite complex and difficult to transform into code. While sequence diagrams suffice to model the behavior of a controller object, state chart diagrams are considered the most appropriate notation for representing the detailed behavior of objects. A more detailed discussion of the situations where state machines are preferable to other behavioral diagrams can be found in [40]. Recent studies have achieved significantly better results in various contexts with the use of state diagrams, see for example [41],[42]. A wide number of approaches such as [21, 43-45] provide implementation of state diagrams in object-oriented way, i.e., in a WTG setting. The existing approaches to generate code from state charts have achieved a lot, yet the same needs to be extended for AOSD and done in the GTW setting. Therefore, AJFCode makes use of state charts to generate behavioral aspect-oriented code.

3. The proposed approach

Fig. 1 shows the key components of this research. It depicts how the proposed solutions contribute to solving the problem addressed by the current research. It also shows the models and techniques and other relevant approaches utilized to conduct the research. Determining an effective modeling approach was the foremost task since no AOM notation has been adopted

as a standard. As shown in partition 1, we investigated all existing approaches based on the criteria specifically tailored to the needs of this research. In the first step, we compared a wider set of 14 AOM approaches based on a well-defined evaluation framework inspired by some existing surveys. We assigned an assessment weight and selected the studies to be evaluated in the next step. In the second step, we used a case study to further evaluate the set of four studies that scored the highest assessment weight in the previous step. Following this step, the Reusable Aspect Models (RAM) [27, 46] notation was selected for its advantages over other notations. After the selection of RAM as modeling approach, the method of mapping its models to AspectJ code was developed (see partition 2). Having developed the conceptual mapping, the next step was to develop a text-based implementation model for RAM as an XML schema, as shown in partition 3. The code generation algorithm was developed in a way that it systematically iterates over the textual representation of the models and generates code. The code generation technique elaborates the details of code snippets to support the generation of a workable code (see partition 4). Finally, as shown in partition 5, the code obtained for two case studies, i.e., OBSS and RSC, was compared to the same obtained by applying the other (WTG) approach based on reusability and maintainability metrics.

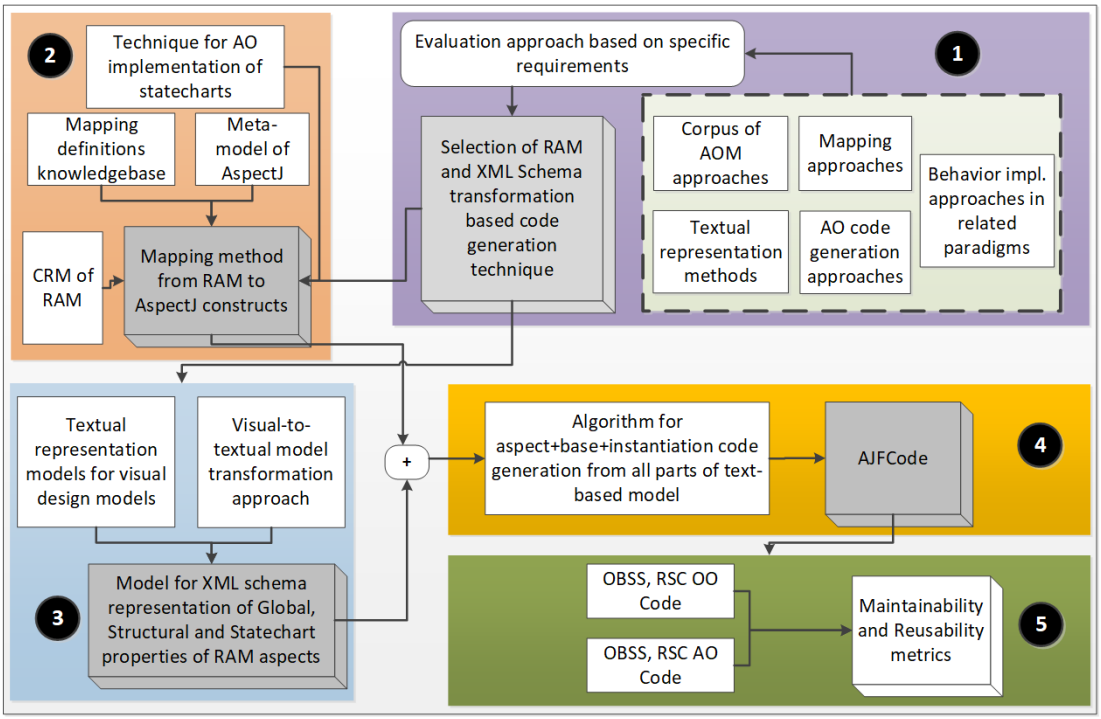

Fig. 1. Development process of AJFCode

3.1 Mapping RAM models to code

The development of a mapping method is a prerequisite for developing the code generation approach because mapping serves to close the gap between models and their representation at the code level in a programming language. An example of this gap can be seen in no direct support for representation of state diagram in languages like Java or AspectJ. The motivation for the selection of state diagrams for behavior representation was provided in the related work section. The translation of models involving state diagrams into code is not straightforward, and it requires an explicit definition of how the constructs of a model will be translated

(mapped) into constructs of a programming language. We adopted the mapping approach presented in [47] as it effectively models the structure and behavior found in RAM models. The approach has two main parts. The first part maps the core and structural units, whereas the second part deals with the mapping of behavior.

Fig. 2 shows the overall mapping of the core and structural parts of a RAM model. The aspect structure comprises complete and incomplete classes that in turn define class members and associations. In this way, the structural view of the model essentially contains UML class diagrams supporting some additional features, such as mandatory instantiation parameters, declaration, and binding of structural elements across different models. AJFCode adopts a Java interface to implement complete classes. The fields are methods are introduced in the interface using inter-type declaration. Attributes defined by classes in the structural view are mapped to plain Java fields. Similarly, operations declared in the structural view are defined in the interface that corresponds to the class with the same signature. Further, simple operations with no details in the state view (e.g., getters and setters) are fully generated, whereas other operations that are not presented in the state view are implemented by only a stub. On the other hand, operations consisted in the state view are fully implemented with complete behavior using the techniques described in the following.

Fig. 2 depicts a high-level view of the state diagram's mapping (see behavioral part). In general, the state diagram is conceptually implemented using two objects: the context and (state) controller. The context serves as the entry point into the state diagram. The controller enumerates the various objects corresponding to different states represented in the state model. To enable instantiation of RAM aspect and merging of state classes, these conceptual objects are implemented as interfaces with inter-type declarations to insert methods. The methods defined within the state controller objects correspond to the state diagram's events. So, the context receives events and passes them on to the controller for processing. The controller being aware of the current state of the system, handles the event accordingly.

As far as composite states (states containing other substates) are concerned, they may contain two different substates: sequential (non-orthogonal) substates and concurrent (orthogonal) substates. The technique for mapping sequential substates is closely related to the one described above for the handling of states in general. The only difference is that unlike the basic approach in which the super state class maintains a reference only to the context object, the composite state contains references to the object representing the initial context as well as to the composite state class. The concurrent composite state is implemented as context for all concurrent regions, and it contains references to all active substates within each region. Behavior in each concurrent region is encapsulated by a separate (super) state object. This state object declares methods to handle all events of states in this concurrent region.

3.2 Aspect-oriented code generation technique

The development of AJFCode involves two transformations as shown in **Fig. 3**. To develop the textual model (the first transformation), we used the text-based implementation model for RAM previously presented by the authors in [48]. The text-based implementation model can be employed here as the XML notation. The related standards used in this work have traditionally been used for code generation, see for example [24]. The model is validated using meta-models of both the RAM and XML notations. Moreover, we have designed an XML schema which standardizes the aspect models and can be used by the transformation model.

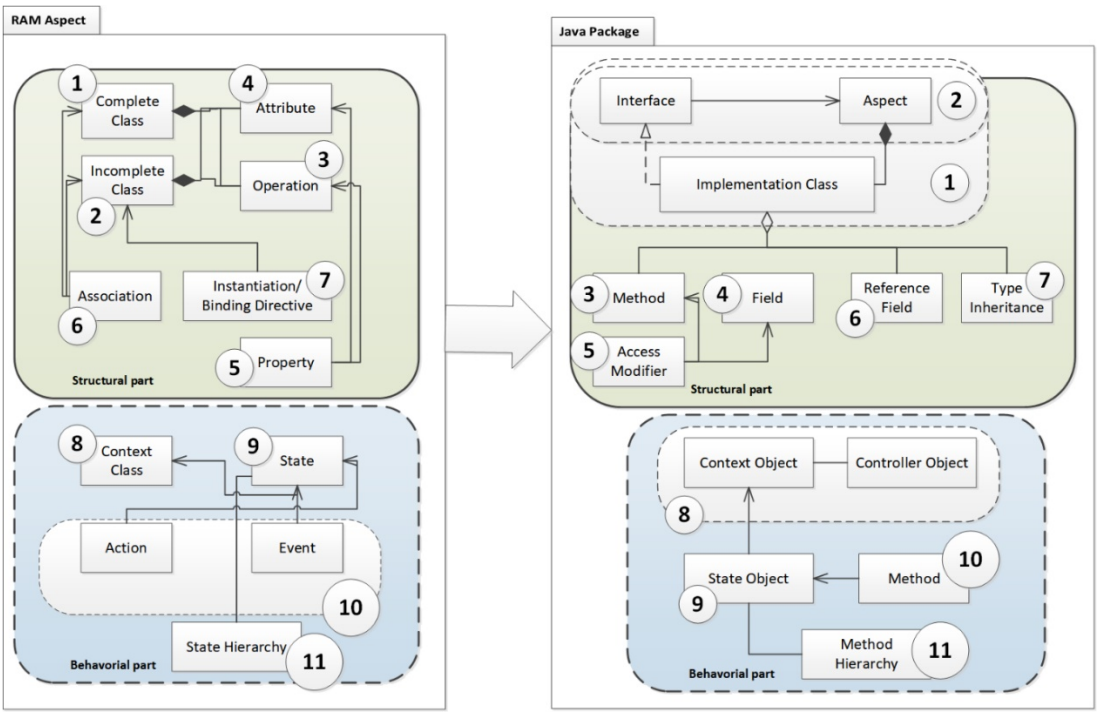

Fig. 2. RAM to AspectJ mapping definition

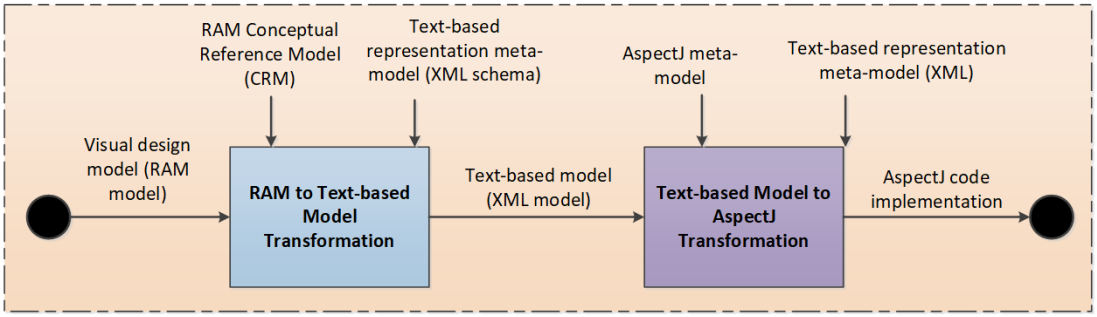

Fig. 3. Transformation from RAM models to AspectJ code

The second transformation shown in **Fig. 3** involves taking the text-based representation of RAM models as input and employs the meta-model of AspectJ to generate code that is syntactically correct. For this purpose, a code generation algorithm has been defined to take care of all facets of the RAM model. We can divide the actual code generation process into three key activities: (1) implementation of the core of aspect, (2) implementation of the structural part, and (3) incorporation of the behavior into structural units by implementing the state chart part. The code generation algorithm, which controls all these activities, has been designed in a way that it is more aligned with the target implementation, and fetches the required information from XML representation by traversing it in a non-sequential manner.

3.2.1 The code generation algorithm

The key parts of the algorithm for code generation from an aspect model are presented in **Fig. 4** to **Fig. 9**. The activities carried out by the algorithm are repeated for each of the aspects of the design model. First, a source code file is created for each interface and class in the aspect (as shown in **Fig. 4**). The algorithm distinguishes between the classes having an associated state chart and the others. We implement the classes with no state chart specification straightforwardly (see **Fig. 5**). Here, we generate a marker interface with a corresponding aspect. Within the aspect, we declare relationship(s) of this class with other classes and interfaces (if any) and implement constructors, fields, and methods containing no functionality.

```
01   # Generate source file for interfaces and classes
02   repeat
03      for each xStructUnit in xGlobal do
04         set sFile.name=xStructUnit
05         generate sFile
06      endfor
07   until xGlobal is empty
```

Fig. 4. Steps for generation of source code files

```
01   # Process class types
02   # Generate package details
03   repeat
04      for each xClassType in structView do
05         set sFile.name=xClassType.structName
06         convert xPackageDetails to sPackageDetails
07      # Generate marker interface and associated aspect
08      set sContext.name=xClassType.structName
09      set sContextAspect.name=xClassType.structName
10      generate sContext
11      generate sContextAspect
12      # Generate code for relationships
13      if xParent is not null
14         insert declareExtParents in sContext, xParent
15      if xRealizes is not null
16         insert declareIntParents in sContext, xParent
17      # Generate fields, constructors, and methods
18      for each xField in xData do
19         convert xField to sField
20      for each xConstructor in xOperations do
21         convert xConstructor to sConstructor
22      for each xMethod in xOperations do
23         convert xMethod to sMethod
24   until structView is empty
```

Fig. 5. Steps for processing the class types

As far as classes associated with a state chart are concerned, we divide the task of state handling to a set of classes, including a class each for the context, the controller, and the state (see **Fig. 6**). The implementation of state view involves the generation of local classes for context and controller objects pertaining to each state. Next, the state controller class is generated that contains signatures of the methods to be introduced in the state classes (see **Fig. 7**). Finally, as shown in **Fig. 8**, states in the state diagram are implemented by creating a dedicated class for each of them. Here, the composite states are implemented as a context

object along with a dedicated state controller. Also, the hierarchical relationship of all state classes with the state controller is implemented. The interfaces in the aspect model are implemented as the standard Java interfaces (see **Fig. 9**). We obtain the details of methods from the structural definitions and directly generate the signatures. As the parts of algorithm corresponding to core and structural parts of RAM model can be implemented by utilizing the language constructs directly, our discussion in this section will be mainly focused on code generation for the behavioral part.

```
# Process the state view
# Generate local classes for context and controller
set contextClassName=xClassType.structName+'Class'
generate locInstClass of contextClassName
set contClassName=xClassType.structName+'State'
set name=contClassName+'Class'
generate locControllerClass of name
insert var:locControllerClass
repeat
   for each xStateName in xStateView do
      set xStateName=xStateName+'Class'
      generate locStateClass of xStateName
      insert var:locStateClass
      # Declare parents for state classes
      insert declareIntParents in locInstClass,sContext
      insert declareIntParents in locStateClass,contClassName
      insert declareIntParents in locStateClass,xStateName
      insert declareIntParents in locStateClass,locControllerClass
      # Generate methods for initialization and setting states
      generate sInitializeMethod
      generate sSetStateMethod
until xStateView is empty
```

Fig. 6. Steps for processing the state view

```
# Create the controller source code file and generate code in it
set contClassName=xClassType.structName+'State'
generate contSFile of contClassName
# Generate state controller
do in contSFile
   generate sController
   generate sContAspect
   insert var:locControllerClass
   # Declare methods in controller class
   for each xMethod in xOperations
   convert xMethod to sMethod
```

Fig. 7. Steps for generating the controller's code

3.2.1.1 State chart implementation

The implementation involves more than one implementation-level entities to represent a single conceptual entity, mainly to handle states and coordination between them during transitions. In the following, we explain the construction process of each placeholder related to implementing state view in our algorithm.

`xStated`: Refers to the case when the value of `isStated` attribute in `xClassType` definition is `true`. It will be the case when the model will have a state chart for a class.

`contextClassName`: Refers to the name of the local context class to be generated within

the aspect corresponding to the context object. This name is generated by appending `Class` to the string value of `structName`.

`locInstClass`: The local class defined for the context object to support instantiation of context by other classes including the state controller and state classes. The code excerpt in **Fig. 10** defines a local instantiation class for a context object named `MyContext`.

`locControllerClass`: The local class definition for controller class to allow instantiation of the controller by other classes. `Class` is appended to `contClassName` to generate the name of this class.

`xStateName`: Individual states found as the string value of `<stateName>` element within `<state>` element in `xStateView`.

`locStateClass`: It refers to code that is generated to provide a local implementation of a class which is used for instantiation. This code is repeatedly generated for each state in the state chart. The code in **Fig. 11** shows the implementation of the local class for a state named MyState in the scope of the context object `MyContext`.

```
01  # Create a source file for each state and generate code in it
02  generate stateSFile of xStateName
03  for each stateSFile do
04     # Generate a controller class for composite substates
05     if xCompState=true
06        set xStateName=xStateName+'State'
07        generate stateContSFile of xStateName
08        insert sCompStateHandling
09        # Generate state controller
10        do in contSFile
11           generate sController
12           generate sContAspect
13           insert var:locControllerClass
14        do in stateSFile
15           generate sState
16           generate sStateAspect
17        # Extend the state class from state controller
18        insert declareExtParents in sState, sController
19        if xSMappedFrom is not null
20           insert declareExtParents in sState, xMappedFrom
21           do in sStateAspect
22              for each xIntEvent in xState do
23                 # Process transitions and generate method
24                 generate sStateMethod
25                 for each xEvent in xState do
26                    generate sStateMethod
27           # Generate the .aj file
28           generate stateInstSFile
29           in stateInstSFile do
30              generate stateInstAspect
31              in stateInstAspect
32                 generate checkStateUsedPointcut
33                 generate excludeInternalCallsPointcut
34           pointcut=checkStateUsedPointcut+excludeInternalCallsPointcut
35                 insert pointcut
36                 generate sInstState
```

Fig. 8. Steps for generating the states' code

`sInitializeMethod`: It is declared within the *context* class and is used to initialize the *state controller* class as well as all the classes representing states in the state chart. It invokes

the `getInstance` of states and assigns the result an instance of the respective class. If the state chart specifies a default state, the `initialize` method also makes a call to the `entry` method of state controller.

`sSetStateMethod:` Refers to the source code responsible for changing states encapsulated in a method named `setState` within the *context* class. A typical `setState` method generated for `MyContext` is shown in **Fig. 12**.

`contSFile:` Refers to the file containing the source code of controller class. The file is named as the value of `contClassName.`

`sController:` Refers to the source code representing the state controller class (defined as a marker interface).

```
01   # Implement interfaces
02   repeat
03      for each xInterfaceType in structView do
04         set sFile= xInterfaceType.structName
05         in sFile do
06            convert xPackageDetails to sPackageDetails
07            generate sInterface of structName
08            if xParent is not null
09               for each xParent do
10                  insert interfaceParent in sInterface, xParent
11            in sInterface do
12               for each xFunction in xOperations do
13                  convert xFunction to sFunction
14   until structView is empty
```

Fig. 9. Steps for implementation of interfaces

```
 static class MyContextClass {
       public static MyContextClass getInstance() {
              return new MyContextClass();
       }
}
```

Fig. 10. Code generated for a local context class

```
static class MyStateClass {
       MyStateClass(MyContext mc) {
              myContext = (MyContextClass) mc;
       }

       public static MyStateClass getInstance() {
              return new MyStateClass(mc);
       }
}
```

Fig. 11. Code generated for a local state class

```
public void MyContext.setState(MyContextState st) {
             state = (MyContextStateClass) st;
                      state.entry();
}
```

Fig. 12. Code generated for a setState method

sContAspect: It is the source code representing the aspect that accompanies sController and is used to add code to it. The state controller contains a reference to the *context* class and declares methods corresponding to the entry, exit and all other events in the statechart.
stateSFile: Refers to the source code file containing the interface and object used to represent the state.
xCompState: Refers to the condition when type attribute for a state (in <state> element within <stateView>) bears the value "composite". Semantically, when set like this, it asserts that the current state is a composite state.
stateContSFile: The source code file for the *state controller*.
sCompStateHandling: Refers to the code segments that deal with the existence of a composite state. Generation of code for standard states is described below for placeholders named sState and sStateAspect.
sState: It is the source code produced declaring the marker interface that corresponds to the state. The interface is named as its corresponding state name.
sStateAspect: Refers to the source code generated to define an aspect within stateSFile to introduce behavior into sState. Within code of sStateAspect, we use the set of elements under <state> element to generate code for internal events, transitions, as well as guard conditions.
xSMappedFrom: It specifies which state a state has been mapped from.
xIntEvent: This element uses a combination of event and action(s) to specify an internal event.
xState: This element hosts all details regarding the specification of a state including its type, name, internal event(s), and transition(s).
sStateMethod: It refers to the source code generated to define a method corresponding to an event or an action within sStateAspect.
xEvent: Refers to <event> element enclosed by xState.
stateInstSFile: Source code file for the aspect that handles the instantiation of states by other states by merging the events and operations. Unlike all other source code files generated previously, this file is generated as an AspectJ source code file (.aj extension). The AspectName is combined with the name of StateView to obtain the name.
stateInstAspect: The aspect in StateInstSFile that contains the code for pointcut and manages the state instantiations. The aspect in **Fig. 13** is intended to instantiate objects of target states and delegate all method calls to the same in case the state named StateOne has been used.
checkStateUsedPointcut: Pointcut that checks whether a state has been instantiated and thus needs to be merged for transitions with the instantiating state. MyAspect in **Fig. 13** contains the code for checkStateUsedPointcut in a pointcut named stateOneUsed, which detects any call to a method that has target as an object of StateOne.
excludeInternalCallsPointcut: Pointcut that ensures that calls internal to a state are ignored by the checkStateUsedPointcut code, see for example excludeInternalCalls in **Fig. 13**.

```
public aspect MyAspect {
       pointcut stateOneUsed(StateOne s):
              execution (* *(..)) && target (s);
       pointcut excludeInternalCalls():
              !within (MyContextAspect) ||
              !within (MyContextStateAspect) ||
              !within (StateOneAspect);
       pointcut finalPointcut(StateOne s) =    stateOneUsed(s)&&
       excludeInternalCalls();

       before(StateOne s): finalPointcut(s) {
              // instantiations
       }
}
```

Fig. 13. Code generated for a `stateInstAspect`

3.3 Code generation tool

The AJFCode tool has been implemented as an Eclipse plugin using Java. It is invoked using a dedicated menu, as shown in **Fig. 14**. The *Editor* (zone①) allows designers to create RAM models in XML notation as specified by the schema definitions. It provides the features of syntax coloring and syntax error reporting. The zone ②shows the menu bar, which can create a RAM model from scratch or load an existing model. Once a syntactically correct and valid model has been created, the menu bar can be used to generate the code for an aspect. The generated code is shown in zone③. The generated code is presented in an editable mode to allow the developer to modify it if needed.

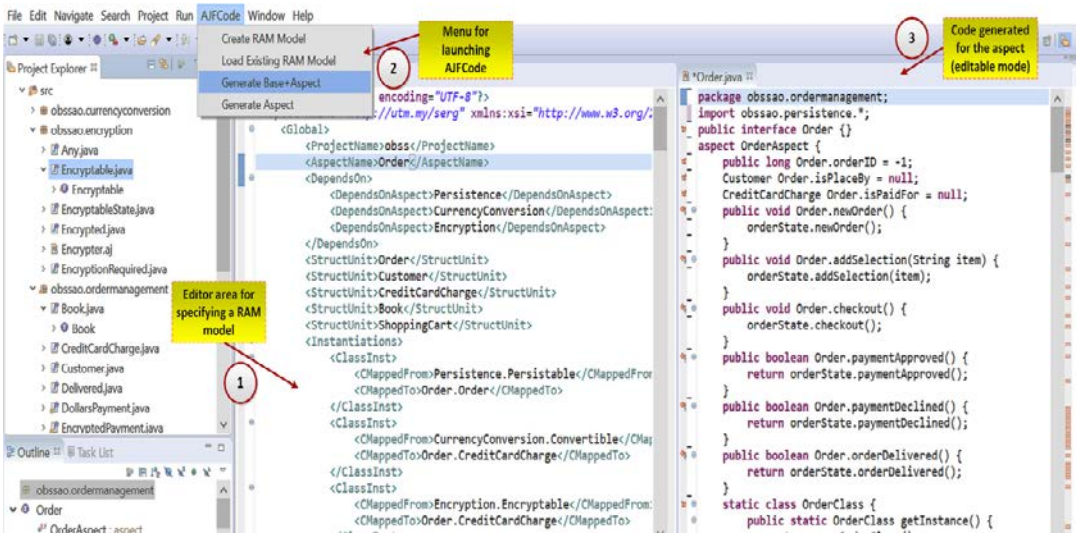

Fig. 14. AJFCode tool

4. Evaluation of approach

4.1 Evaluation methodology

We have validated the AJFCode approach in two different ways.

Evaluation of AJFCode relative to GTW approaches: A previous study [49] has evaluated

AO code generation approaches according to five criteria, i.e., transformation, models, validation, the extent of code, and tool-support. To highlight the advantages or disadvantages of the AJFCode approach, we have evaluated it based on the same criteria set.

Evaluation of AJFCode relative to WTG approaches: In the context of the current work, the effectiveness of a WTG approach means the projected efficiency of the approach when existing object-oriented code generation approaches are applied to generate code. The comparison with a WTG approach was imperative for two reasons: (i) none of the existing GTW approaches supports the implementation of behavioral diagrams, and (ii) as AJFCode is an approach for integrating AOSD and MDE using AO code generation, it is meaningful to compare it with an approach that serves the same purpose but uses OO code generation. To this end, a thorough analysis of the existing OO code generation was conducted approaches to identify an approach that is most comprehensive and suitable for the comparison. Therefore, seven existing OO code generation approaches were compared based on six criteria sets, i.e., models, design, implementation, validation, extent of code, and tool support. Consequently, JCode [43, 50-52] approach was selected to compare its performance with JFCode. None of the existing weavers support code generation, therefore, the woven model was manually developed, and then subjected to the JCode's code generation. AJFCode was applied to generate executable code for two different systems adopted from MDE literature: an online book store system (OBSS) [53], and a remote caller system (RSC)[19]. Finally, the quality metrics related to the quality of the application code [54], i.e., metrics for reusability and maintainability, were applied to measure the quality of code generated using JCode and AJFCode. The results are presented in the following.

4.2 AJFCode vs GTW approaches

In this section, mimicking the presentation of the assessment results for WTG approaches in [49], we present the performance of AJFCode based on the same criteria. We also show the evaluation results for other two significant works, i.e., RAM [23] and FDAF [24]. The results of comparison are shown in **Table 1**. AJFCode provides a detailed mapping definition for both structure and behavior (T-MD) and augments the conceptual mapping with a comprehensive implementation model (T-IM). AJFCode supports both static and dynamic views of models (T-SV), for both aspectual and non-aspectual (base) parts (T-SC). Unlike RAM and FDAF, the advanced interactions between aspects and encapsulated components are supported using the instantiation and binding directives (T-AN). Implementation of ADTs and collections is also provided at all levels of the approach (T-AN). Even though the implementation model and the code generation algorithm can generate executable code as verified by test executions, the correctness and performance of algorithm have not been focused formally (T-AM).

Like RAM and FDAF, AJFCode supports class diagrams and aspects modeled as enhanced package diagrams (M-SM). Nevertheless, in contrast with RAM and FDAF, where the former supports behavioral models (sequence diagrams) for conceptual mappings only, and the latter does not support them at all, AJFCode provides a full implementation of state chart diagrams (M-BM). AJFCode employs standard quality metrics to verify its effectiveness against the other approaches (V-AT). Unlike RAM and FDAF, which are limited to the generation of skeletons of code only, AJFCode generates both full as well as skeleton code (for all other classes and methods) in a model (E-SC). Code for behavior diagrams is generated (E-BC), for both aspect and the base parts (E-FC). The tool support provided is sufficient for validation and was developed using industry-standard plugin mechanism enabling integration with other tools.

4.3 AJFCode vs WTG approaches

In this section, we compare the code generated by AJFCode and the code produced by applying JCode to the woven (object-oriented) models. For this purpose, we selected a set of metrics (see **Table 2**) to measure the reusability and maintainability of the generated code. These metrics were adopted because of their suitability for assessing the aforementioned properties, and their extensive use in the literature for the same purpose. Note that these properties can be linked with other important quality factors such as understandability, flexibility, and extensibility [11, 55]. The results of applying the metrics are given in **Table 3**.

Table 1. Results of comparison with generate-then-weave (GTW) approaches

Main criteria	Sub-criteria		RAM	FDAF	AJFCode
Transformation	Mapping definition (T-MD)		✓	×	✓
	Implementation model (T-IM)	General-purpose	×	✓	✓
		Extendable	×	✓	✓
		Behavior support	×	×	✓
	Supported views (T-SV)	Static view	✓	✓	✓
		Dynamic view	✓	×	✓
	Supported concerns (T-SC)	Aspect	✓	✓	✓
		Base	×	×	✓
	Approach advanced-ness (T-AN)	Other interactions	×	~	✓
		ADTs, Collections	×	~	✓
	Algorithmic maturity (T-AM)	Correctness	×	~	×
		Performance	×	×	×
Models	Structure models (M-SM)	Class diagrams	✓	✓	✓
		Other diagrams	✓	×	✓
	Behavior models (M-BM)	Statechart	×	×	✓
		Sequence	✓	×	×
		Other	×	×	×
Validation	Approach transparency (V-AT)	Standard inputs	×	✓	×
		Standard outputs	×	✓	×
		Open comparison mechanism	×	✓	✓
Extent of code	Structure code (E-SC)	Full code for class diagram	×	×	✓
		Skeleton code	✓	✓	✓
	Behavior code (E-BC)	Code for behavior diagram	✓	×	✓
	Full code (E-FC)	Aspect+base code	×	×	✓
		Trade-off analysis	×	✓	✓
Tool support	Tools sufficiency	Validation	×	✓	✓
		Real-word apps	×	~	~
	Tools integration	Standard dev frameworks	×	✓	✓

AJFCode positively affects the quality of the final code by improving its reusability and maintainability. **Table 3** shows that more components are required for implementation of a single concern with WTG approach. AJFCode requires a smaller number of components to implement crosscutting concerns (5 classes vs 7 classes) because it models these concerns as independent aspects, and the components maintain this independence down to the code level. Therefore, a component is instantiated and used only if it is required to be woven. Similarly, the difference between the number of operations contributing to implementing a concern using

different implementation approaches shown in CDO column is noteworthy. For instance, CDO values for concern 1 of OBSS were recorded to be 92 and 38 for WTG and JFCode, respectively. This difference is mainly because of the special attention given by AJFCode to the weaving mechanism that instantiates only the required components and excludes internal calls to the same. This prevents intermingling of concerns at all levels of the generated code.

The components resulted by the AJFCode possess more reusability and maintainability qualities, as they are less coupled (see the columns named CBC and DIT). The difference is due to context handlers in WTG approach keeping references to all components that they correspond to. AJFCode does not require any explicit references to such objects. Instead, it makes use of local classes, which exist only within the aspect thus providing a mechanism to instantiate the interfaces. Objects of these classes are generated using inter-type declarations. So far as the cohesion of the components generated by AJFCode is concerned, **Table 3** shows that the approach generates components that are more cohesive than their object-oriented counterparts (see the column named LCOO). In this regard, the approach is particularly effective for implementing composite states since, in that case, the extent of lack of cohesion in operations is zero, meaning that the components are fully cohesive.

Table 2. Summary of metrics applied to AJFCode and WTG approaches

Metric Type	Metric	Description of Metric
Separation of Concerns (SOC) [56]	Concern Diffusion over Components (CDC)	The total number of primary components used to implement the concern added to the number of other aspects or classes accessing them.
	Concern Diffusion over Operations (CDO)	Number of methods and advices which exist mainly to contribute to implementing a concern, and the number of other methods/advices which access them.
Coupling [57]	Coupling Between Components (CBC)	The number of other classes and aspects that a class or aspect is associated with.
	Depth of Inheritance Tree (DIT)	Measure of how far down in the inheritance hierarchy a class or an aspect is declared.
Cohesion [58]	Lack of Cohesion in Operations (LCOO)	Determined by the number of method or advice pairs that do not access the same instance variable.
Size [58]	Vocabulary Size (VS)	The number of classes and aspects of the system.
	Lines Of Code (LOC)	The number of lines of code.
	Number of Attributes (NOA)	The total number of attributes of each class or aspect excluding the inherited attributes.
	Weighted Operations per Component (WOC)	The total complexity of a component determined by the number of methods, their parameters and advices of each class or aspect.

Similarly, with regards to size metrics, as the smaller vocabulary size indicates less complexity, and in turn, high reusability and maintainability, the results (see the column named VS) highlight the positive impact of AJFCode. That AJFCode requires more lines of implementation code (see LOC) than WTG approach is essentially a side effect of the use of local classes, which have contributed positively regarding all other metrics. However, since the stated mechanism is well-defined and is to be applied exactly in the same way to all different applicable scenarios, it will be justified to expect that its effect on complexity and consequently on reusability and maintainability will be minimum.

5. Conclusion

This paper presents AJFCode, an approach to apply aspect orientation in combination with MDE through AO model-driven code generation. To provide a comprehensive mapping of aspect models, both the structure and the object behavior have been supported. The code generation technique has been developed in a way that the process followed by the employed algorithm is aligned with the target implementation and allows fetching of the required information by a non-sequential traversal of the textual representation. Structural as well as the behavioral code has been generated for both aspectual and non-aspectual parts. AJFCode has been validated using two systems from the literature. First, the comprehensiveness of the approach is determined regarding: (i) its ability to address all features of the design model, and (ii) its strength to address areas which are not addressed by other AO code generation approaches. Next, the object-oriented code for the same systems is obtained and compared with the code generated by AJFCode using several metrics. The results show AJFCode gives better results against about 78% of the applied metrics.

Table 3. Results of metrics applied to AJFCode and WTG approaches

System	Concern	Approach	CDC	CDO	CBC	DIT	LCOO	VS	LOC	NOA	WOC
RSC	1	WTG	7	51	6	0	9	7	45	7	0
		AJFCode	7	19	0	0	5	2	63	5	0
	2	WTG	7	51	1	1	7	13	33	1	0
		AJFCode	5	21	0	1	0	5	19	1	0
OBSS	1	WTG	17	92	13	0	14	17	45	13	2
		AJFCode	15	38	3	0	7	11	63	3	1
	2	WTG	11	82	1	1	9	12	33	1	1
		AJFCode	7	21	0	1	0	7	19	1	0
	3	WTG	12	90	9	0	8	9	55	25	1
		AJFCode	6	14	2	0	1	2	28	4	1
	4	WTG	11	87	5	1	10	7	71	11	0
		AJFCode	6	19	1	1	2	1	45	2	0

There are some limitations of the current work that may be addressed by the future research. First, AJFCode adopts the state diagrams only to implement the behavior of generated classes. Even though the state diagrams are considered a more efficient way of representing behavior, incorporation of other diagrams such as sequence diagrams may increase the amount of the generated code, by generating code for controller objects. Second, the transformation of the visual design models into XML-based form can be automated by extending the modeling support of the existing environments.

References

[1] B. Karakostas and Y. Zorgios, *Engineering Service Oriented Systems: A Model Driven Approach*, IGI Global, 2008. Article (CrossRef Link)

[2] M. Afonso, R. Vogel, and J. Teixeira, "From code centric to model centric software engineering: practical case study of MDD infusion in a systems integration company," in *Proc. of Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, Fourth and Third International Workshop on*, p.10(pp.-134), 2006. Article (CrossRef Link)

[3] E. Rodrigues, L. Montecchi, and A. Ceccarelli, "Model-Driven Fault Injection in Java Source Code," in *Proc. of 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 414-425, 2020. Article (CrossRef Link)

[4] M. Ziaei, B. Zamani, and A. Bohlooli, "A Model-Driven Approach for IoT-Based Monitoring Systems in Industry 4.0," in *Proc. of 2020 4th International Conference on Smart City, Internet of Things and Applications (SCIOT)*, pp. 99-105, 2020. Article (CrossRef Link)

[5] T. B. l. Fosse, Z. Cheng, J. Rocheteau, and J. M. Mottu, "Model-Driven Engineering of Monitoring Application for Sensors and Actuators Networks," in *Proc. of 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 553-560, 2020. Article (CrossRef Link)

[6] L. Silva and D. Lopes, "Model Driven Engineering for Performance Testing in Mobile Applications," in *Proc. of 2020 5th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, pp. 1-7, 2020. Article (CrossRef Link)

[7] T. Górski and J. Bednarski, "Applying Model-Driven Engineering to Distributed Ledger Deployment," *IEEE Access*, vol. 8, pp. 118245-118261, 2020. Article (CrossRef Link)

[8] T. Elrad, O. Aldawud, and A. Bader, "Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design " in *Proc. of Generative Programming and Component Engineering*, pp. 189-201, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002. Article (CrossRef Link)

[9] K. Hoffman and P. Eugster, "Trading obliviousness for modularity with cooperative aspect-oriented programming," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 1-46, 2013. Article (CrossRef Link)

[10] A. Hovsepyan, R. Scandariato, S. V. Baelen, Y. Berbers, and W. Joosen, "From aspect-oriented models to aspect-oriented code?: the maintenance perspective," in *Proc. of the 9th International Conference on Aspect-Oriented Software Development, Rennes and Saint-Malo*, France, pp. 85-96, 2010. Article (CrossRef Link)

[11] E. K. Piveta, A. Moreira, M. S. Pimenta, J. Araújo, P. Guerreiro, and R. T. Price, "An empirical study of aspect-oriented metrics," *Science of Computer Programming*, vol. 78, no. 1, pp. 117-144, 11/1/ 2012. Article (CrossRef Link)

[12] S. A. Vidal and C. A. Marcos, "Toward automated refactoring of crosscutting concerns into aspects," *Journal of Systems and Software*, vol. 86, no. 6, pp. 1482-1497, 2013. Article (CrossRef Link)

[13] S. V. S, "Introducing Aspect-Oriented Programming in Improving the Modularity of Middleware for Internet of Things," in *Proc. of 2020 Advances in Science and Engineering Technology International Conferences (ASET)*, pp. 1-5, 2020. Article (CrossRef Link)

[14] F. Moreno, S. Uribe, F. Álvarez, and J. M. Menéndez, "Extending Aspect-Oriented Programming for Dynamic User's Activity Detection in Mobile App Analytics," *IEEE Consumer Electronics Magazine*, vol. 9, no. 2, pp. 57-63, 2020. Article (CrossRef Link)

[15] P. Berta and V. Vranić, "Synergy of Organizational Patterns and Aspect-Oriented Programming," in *Proc. of 2019 IEEE 15th International Scientific Conference on Informatics*, pp. 000439-000444, 2019. Article (CrossRef Link)

[16] C. Hung, K. Chen, and C. Liao, "Modularizing Cross-Cutting Concerns with Aspect-Oriented Extensions for Solidity," in *Proc. of 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pp. 176-181, 2019. Article (CrossRef Link)
[17] L. Fuentes and P. Sánchez, "Dynamic Weaving of Aspect-Oriented Executable UML Models," in *Transactions on Aspect-Oriented Software Development VI*, pp. 1-38, 2009. Article (CrossRef Link)
[18] T. Cottenier, A. v. d. Berg, and T. Elrad, "Motorola WEAVR: Aspect Orientation and Model-Driven Engineering," *Journal of Object Technology*, vol. 6, no. 7, pp. 51–88, 2007. Article (Web Link)
[19] J. Whittle, P. Jayaraman, A. Elkhodary, A. Moreira, and J. Araújo, "MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation," in *Transactions on Aspect-Oriented Software Development VI*, pp. 191-237, 2009. Article (CrossRef Link)
[20] A. Stavrou and G. A. Papadopoulos, "Automatic Generation of Executable Code from Software Architecture Models," in *Information Systems Development*, pp. 447-458, 2009. Article (CrossRef Link)
[21] R. Pilitowski and A. Dereziñska, "Code Generation and Execution Framework for UML 2.0 Classes and State Machines," *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, pp. 421-427, 20007. Article (CrossRef Link)
[22] P. Papotti, A. Prado, W. Souza, C. Cirilo, and L. Pires, "A Quantitative Analysis of Model-Driven Code Generation through Software Experimentation," in *Proc. of CAiSE 2013: Advanced Information Systems Engineering*, pp. 321-337, 2013. Article (CrossRef Link)
[23] M. Kramer and J. Kienzle, "Mapping Aspect-Oriented Models to Aspect-Oriented Code," in *Proc. of MODELS 2010: Models in Software Engineering*, pp. 125-139, 2011. Article (CrossRef Link)
[24] J. Bennett, K. Cooper, and L. Dai, "Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach," *Science of Computer Programming*, vol. 75, no. 8, pp. 689-725, 2010. Article (CrossRef Link)
[25] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena, "Composing design patterns: a scalability study of aspect-oriented programming," in *Proc. of the 5th international conference on Aspect-oriented software development*, Bonn, Germany, pp. 109-121, 2006. Article (CrossRef Link)
[26] P. Greenwood et al., "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study," in *Proc. of ECOOP 2007 – Object-Oriented Programming*, pp. 176-200, 2007. Article (CrossRef Link)
[27] J. Kienzle, W. Al Abed, F. Fleurey, J.-M. Jézéquel, and J. Klein, "Aspect-Oriented Design with Reusable Aspect Models," in *Transactions on Aspect-Oriented Software Development VII*, pp. 272-320, 2010. Article (CrossRef Link)
[28] M. V. Hecht, E. K. Piveta, M. S. Pimenta, and R. T. Price, "Aspect-oriented Code Generation," in *Proc. of the XX Brazilian Conference on Software Engineering*, 2005. Article (Web Link)
[29] S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison Wesley Object Technology, 2005.
[30] S. Maoz and D. Harel, "From multi-modal scenarios to code: compiling LSCs into aspectJ," in *Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, Portland, Oregon, USA, pp. 219-230, 2006. Article (CrossRef Link)
[31] J. Evermann, "A meta-level specification and profile for AspectJ in UML," in *Proc. of the 10th international workshop on Aspect-oriented modeling*, Vancouver, Canada, pp. 21-27, 2007. Article (CrossRef Link)
[32] J. Evermann, A. Fiech, and F. E. Alam, "A platform-independent UML profile for aspect-oriented development," in *Proc. of The Fourth International C Conference on Computer Science and Software Engineering*, Montreal, Quebec, Canada, pp. 25-34, 2011. Article (CrossRef Link)
[33] J. D. Bennett, "An approach to aspect-oriented model-driven code generation using graph transformation. MS Thesis," MS, The University of Texas at Dallas, 2007.

[34] L. Dai, "Formal design analysis framework: an aspect-oriented architectural framework," University of Texas at Dallas, Ph.D. Dissertation, 2005.
[35] I. Groher and S. Schulze, "Generating aspect code from UML models," in *Proc. of The Third International Workshop on Aspect-Oriented Modeling*, 2003. Article (CrossRef Link)
[36] S. Haitao, S. Zhumei, and Z. Shixiong, "Mapping Aspect-Oriented Domain-Specific Model to Code for Real Time System," in *Proc. of The Sixth World Congress on Intelligent Control and Automation*, vol. 2, pp. 6426-6431, 2006. Article (CrossRef Link)
[37] A. Jackson, N. Casey, and S. Clarke, "Mapping design to implementation," *AOSD-Europe TDC-D111*. Article (CrossRef Link).
[38] J. Araújo and J. Whittle, "Aspect-Oriented Compositions for Dynamic Behavior Models," in *Aspect-Oriented Requirements Engineering*, 2013, pp. 45-60. Article (CrossRef Link)
[39] S. Loukil, S. Kallel, B. Zalila, and M. Jmaiel, "AO4AADL: Aspect oriented extension for AADL," *Open Computer Science*, vol. 3, no. 2, pp. 43-68, 2013/06/01 2013. Article (CrossRef Link)
[40] A. Mehmood, "Aspect-Oriented Model-Driven Code Generation Approach For Improving Code Reusability And Maintainability," Ph.D. Thesis, Department of Software Engineering, Faculty of Computing, Universiti Teknologi Malaysia (UTM), Ph.D. Thesis, 2014.
[41] W. M. Ma and W. S. Chao, "Structure-Behavior Coalescence Abstract State Machine for Metamodel-Based Language in Model-Driven Engineering," *IEEE Systems Journal*, vol. 15, no. 3, pp. 4105-4115, 2021. Article (CrossRef Link)
[42] M. Miroshnyk, A. Shkil, E. Kulak, D. Rakhlis, I. Filippenko, and A. Miroshnyk, "Verification of FPGA control systems by analyzing the correctness of state diagrams," in *Proc. of 2020 IEEE 11th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pp. 85-89, 2020. Article (CrossRef Link)
[43] I. A. Niaz, "Automatic Code Generation From UML Class and Statechart Diagrams," PhD Thesis, Graduate School of Systems and Information Engineering., University of Tsukuba, Ph.D. Thesis., 2005.
[44] A. Derezinska and R. Pilitowski, "Correctness issues of UML class and state machine models in the C# code generation and execution framework," in *Proc. of International Multiconference on Computer Science and Information Technology*, pp. 517-524, 2008. Article (CrossRef Link)
[45] V. S. E and P. Samuel, "Automatic Code Generation From UML State Chart Diagrams," *IEEE Access*, vol. 7, pp. 8591-8608, 2019. Article (CrossRef Link)
[46] J. Kienzle, W. A. Abed, and J. Klein, "Aspect-oriented multi-view modeling," in *Proc. of the 8th ACM international conference on Aspect-oriented software development*, Charlottesville, Virginia, USA, pp. 87-98, 2009. Article (CrossRef Link)
[47] A. Mehmood, D. N. A. Jawawi, and F. Zeshan, "An Approach for Mapping the Aspect State Models to Aspect-Oriented Code," in *Proc. of 2019 International Conference on Engineering and Emerging Technologies (ICEET)*, pp. 1-6, 2019. Article (CrossRef Link)
[48] A. Mehmood and D. N. A. Jawawi, "A Text-based Implementation Model for Reusable Aspect Models," *Journal of Theoretical and Applied Information Technology*, vol. 55, no. 2 pp. 209-224, 2013. Article (CrossRef Link)
[49] A. Mehmood and D. N. A. Jawawi, "Aspect-Oriented Code Generation for Integration of Aspect Orientation and Model-Driven Engineering," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 2, pp. 207-218, 2013. Article (CrossRef Link)
[50] I. A. Niaz and J. Tanaka, "Code Generation from UML Statecharts," in *Proc. of 7th IASTED International Conf. on Software Engineering and Application (SEA 2003)*, Marina Del Rey, USA, pp. 315-321, 2003. Article (CrossRef Link)
[51] I. A. Niaz and J. Tanaka, "Mapping UML Statecharts to Java Code," in *Proc. of IASTED International Conf. on Software Engineering (SE 2004)*, Innsbruck, Austria, pp. 111-116, 2004. Article (CrossRef Link)
[52] I. A. Niaz and J. Tanaka, "An Object-Oriented Approach to Generate Java Code from UML Statecharts," *International Journal of Computer & Information Science*, vol. 6, no. 2, 2005. Article (CrossRef Link)

[53] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*, Addison-Wesley Longman Publishing Co., Inc., pp. 368, 2002.
[54] C. Sant'anna, A. Garcia, C. Chavez, C. Lucena, and A. von Staa, "On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework," in *Proc. of XVII Brazilian Symposium on Software Engineering*, 2003. Article (CrossRef Link)
[55] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. v. Staa, "Modularizing design patterns with aspects: a quantitative study," in *Proc. of the 4th international conference on Aspect-oriented software development*, Chicago, Illinois, pp. 3-14, 2005. Article (CrossRef Link)
[56] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *Proc. of the 21st international conference on Software engineering*, Los Angeles, California, USA, pp. 107-119, 1999. Article (CrossRef Link)
[57] T. T. Bartolomei, A. Garcia, C. Sant'Anna, and E. Figueiredo, "Towards a unified coupling framework for measuring aspect-oriented programs," in *Proc. of the 3rd international workshop on Software quality assurance*, Portland, Oregon, pp. 46-53, 2006. Article (CrossRef Link)
[58] I. Sommerville, *Software Engineering*, Pearson, 2010.

Abid Mehmood received the M.Sc. degree in computer science from Quaid-i-Azam University, Islamabad, Pakistan, in 2001, and the Ph.D. degree in computer science from Universiti Teknologi Malaysia, Johor Bahru, Malaysia, in 2014. Prior to entering academia, from 2001 to 2009, he worked at the software development industry in different roles and contributed to the design and development of various high-performance enterprise applications. He is currently an Assistant Professor with the Department of Management Information Systems, King Faisal University, Saudi Arabia. His research interests include neural networks and deep learning, the Internet of Things, dynamic and self-adaptive systems, model-driven engineering, and aspect-orientation.

Dayang N. A. Jawawi received the bachelor's degree in software engineering from Sheffield Hallam University, U.K., and the master's degree in computer science and the Ph.D. degree in software engineering from Universiti Teknologi Malaysia (UTM), Malaysia. She is currently an Associate Professor at the Faculty of Engineering, School of Computing, UTM. Her main research interests include software engineering, software reuse, software quality, software testing, requirement engineering, and computing education. A major part of her research projects focuses on rehabilitation and mobile robotics, real-time embedded systems, and precision farming applications.