View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universiti Teknologi Malaysia Institutional Repository

An Approach for Mapping the Aspect State Models to Aspect-Oriented Code

Abid Mehmood
Department of MIS
King Faisal University
Al-Ahsa, Saudi Arabia
aafzal@kfu.edu.sa

Dayang N.A. Jawawi
Department of Software Engineering
Universiti Teknologi Malaysia
Johor Bahru, Malaysia
dayang@utm.my

Furkh Zeshan
Department of Computer Science
COMSATS Institute of Information Technology (CIIT)
Lahore, Pakistan
drzfurkh@ciitlahore.edu.pk

***Abstract*— Model-driven code generation allows rapid generation of precise code thus reducing the development effort and the delivery time. Consequently, model-driven code generation has been a topic of interest in varying perspectives. While model-driven code generation has been explored well in many domains, its full potential has not been exploited in the context of aspect-oriented software development. The existing approaches have mainly focused on code generation from class diagrams only. Code generation from class diagrams is straightforward as majority of the constructs involved in these diagrams are directly mapped to those in programming languages. However, code generated using class diagrams is limited to skeletons of classes and methods only and does not contain behavior. In this study, we use the state diagrams and propose a mapping of its constructs to AspectJ language. We use the Reusable Aspect Models notation for this purpose. The approach addresses the mapping of both structure and behavior, however, owing to their strength with respect to modeling the system behavior, it essentially focuses on the state diagrams encapsulated in these models. A detailed mapping of different features of the state diagrams is proposed. The approach is illustrated by means of aspect models and corresponding mapped code from an aspect-oriented implementation of a Remote Service Caller example. The implementation shows that the approach can effectively be applied to obtain code for complete structure and behavior modeled using Reusable Aspect Models.**

Keywords*— *aspect-oriented models, model-driven code generation, state diagrams

I. INTRODUCTION

Model-driven engineering (MDE) aims at increasing the productivity of software development teams with the help of two fundamental concepts: *abstraction* and *automatic transformations*. The concept of abstraction allows reusable models to represent the core concept of the system under development, thus simplifying the design process as well as the communication between stakeholders. The concept of automatic transformations allows the manipulation and refinement of models into other models and eventually the application code. In this setting, on the one hand, model-driven code generation allows rapid generation of code thus reducing the development effort and the delivery time. On the other hand, automatically generated code is also deemed helpful in reducing programming errors, and to possess high consistency with the design [1, 2]. As a result, model-driven code generation has been a topic of interest in varying perspectives. Some examples include fully executable object-oriented code generation for UML models [3, 4], for web applications [5], for dynamically-adaptive systems [6], and for the Internet of Things (IoT) [7]. While model-driven code generation has been explored well in the mentioned domains, its full potential has not been exploited in the context of aspect-oriented software development (AOSD). Studies such as [8, 9] have found AOSD techniques as more effective than their counterparts in dealing with so-called *crosscutting concerns* which cut across the primary modularization of a system. Aspect orientation is particularly relevant in the context of code generation because it has been shown that while transforming an AO design into code, the approaches which target AO programming languages result in more compact, smaller, less complex and more modular implementations, see for example [10].

In past, few efforts have been made to achieve automatic AO code generation and initial results have been reported in the literature. It has been previously reported that each of these approaches formulates certain specific features of aspect-oriented model-driven code generation while eliminating others [11]. The existing approaches have mainly focused on code generation from structural models (specifically class diagrams), leaving the integration of behavioral models for future work. Code generation from class diagrams is straightforward as majority of the constructs involved in these diagrams are directly mapped to those in programming languages. However, code generated using class diagrams is limited to skeletons of classes and methods only and does not contain behavior. On the other hand, diagrams such as state diagrams which effectively model the behavior of systems are comparatively more complex and their implementation to obtain code is harder. Because the existing programming languages do not provide direct support for implementation of constructs supported by such diagrams.

In this study, we use the aspect state models developed using a well-documented AO modeling approach, Reusable Aspect Models [12] and propose a mapping of its constructs to AspectJ language. Aspect state models possess better modularization capabilities and are considered more robust as compared with traditional state models, see for example [13, 14].

This paper is organized in five sections. Section II explains the mapping approach for structure whereas Section III describes the mapping of behavioral aspects. Section IV discusses the results of evaluation using a model setting. Related work is discussed in Section V. Section VI concludes the paper.

II. MAPPING STRUCTURE TO CODE

An overview of the mapping of the core properties and structural units is given in Table I. The structure of an aspect is defined by the classes in the structural part of RAM aspect.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

Authorized licensed use limited to: UNIVERSITY TEKNOLOGI MALAYSIA. Downloaded on April 07,2021 at 02:11:22 UTC from IEEE Xplore. Restrictions apply.

Classes (complete and incomplete) specify attributes, operations and associations with other classes in the model.

A. General Structure, Classes, Attributes and Operations

We create a java package at the highest level to include all code artifacts related to the project. Inside this package we create a separate subpackage (named as the aspect) to represent each aspect.

We provide full implementation of all classes (complete and incomplete) by mapping the class diagram and associated state diagram given in the state view to code. We do not map a class to a standard class in Java to cater for the possibility of merging the mapped class to other classes in model as an effect of binding directives. A class implemented as a standard class in Java cannot be merged with other classes as Java does not support multiple inheritance. Therefore, we implement complete classes in RAM models by creating a public Java interface and employing inter-type declaration mechanism of AspectJ to introduce fields and methods into that interface. An interface with by the name of the complete class is created and an AspectJ aspect is introduced in the same file. Name of the aspect is determined by appending "Aspect" to the interface name.

Attributes are mapped to plain Java fields.

Methods are defined with the signature given in the model. Our approach distinguishes between two types of operations: (i) operations which are mentioned in the state view as part of the set of events or operations picked up by the state diagram and (ii) operations for which state view provides no details. Implementation of the former type is thoroughly discussed in Section III below. Operations that are not defined in the state view are implemented in one of two different ways: (1) if it is possible to determine the behavior of an operation on basis of its signature (for example setters and getters), then it is fully implemented, or otherwise (2) only a stub is generated and the coder is required to provide the implementation of the method.

As shown in Table 1, properties on attributes or operations such as access modifiers, i.e., +, ~ and – are mapped to public, protected and private, respectively.

B. Mapping Associations

As shown in Table 1, associations between classes specified are implemented using Java fields. Such fields are named exactly as the role name that they correspond to and their type is determined based on multiplicity of the relationship. Thus, an association having only one role name is interpreted as a unidirectional association and is implemented by introducing a field with the same name into the interface that represents the other end of the association. An association having role names for both directions of the relationship is considered bidirectional and is implemented by following the same technique for both interfaces that correspond to two sides of association.

A "`0..1`" multiplicity of a role is implemented by initializing the resulting field to `null`, whereas a field in case of multiplicity of "1" is initialized to an object of associated class. An "*" multiplicity is implemented using `java.util.Set`.

TABLE I. MAPPING THE RAM STRUCTURAL VIEW TO ASPECTJ

RAM construct	Mapped construct	Description
Aspect	Package	A RAM aspect encapsulates several classes that achieve the functionality related to the concern modeled by the aspect. A package in Java/AspectJ is a similar type of module. A top-level package refers to the application, whereas a subpackage is defined to encapsulate classes and interfaces related to each aspect.
Complete class	Interface, Aspect and Implementati	A *complete* class is mapped to an interface. An aspect is defined which introduces fields and methods into the interface using inter-type declarations. A class implements the interface, and thus allows instantiation of the *complete* class.
Incomplete class	Interface and Aspect	An incomplete class is mapped to an interface. An aspect is defined which introduces fields and methods into the interface using inter-type declarations.
Operation	Method	Operations define functionality for the classes of RAM models. They are mapped to methods which are defined into the interfaces using inter-type declarations.
Attribute	Field	Attributes are mapped to plain Java fields.
Property on attribute/ operation	Field/method access modifier	Properties on attributes or methods are mapped to corresponding access modifiers.
Association	Reference field(s)	A unidirectional association is implemented by defining a field of corresponding type on one side of the relationship, whereas a bidirectional association is mapped by defining fields on both sides, i.e., to both interfaces corresponding to two sides of the relationship.
Instantiation/ binding directive	Type inheritance	Inheritance relationship is defined between the interfaces that correspond to two sides of an instantiation or binding directive.

C. Instantiation and Binding Directives

We implement the instantiation directives, which assign classes to mandatory instantiation parameters, by type hierarchy modifications. To explain the idea, let us consider two aspects `aspectA` and `aspectB` such that `aspectA` depends on `aspectB`. Suppose that the structural view of `aspectA` contains a class `ClassOne` and an instantiation directive `ClassTwo` → `ClassOne` where `ClassTwo` is a mandatory instantiation parameter of `aspectB`. The effect of this instantiation is acquired by mapping it to an inheritance relationship between `ClassOne` and `ClassTwo` such that `ClassOne` *extends* `ClassTwo`.

Binding directives in RAM are used to bind complete classes to other classes in the model. They are also

Authorized licensed use limited to: UNIVERSITY TEKNOLOGI MALAYSIA. Downloaded on April 07,2021 at 02:11:22 UTC from IEEE Xplore. Restrictions apply.

implemented using type inheritance. Let us consider `aspectA` and `aspectB` again such that `aspectA` depends on `aspectB`. Suppose that the structural view of `aspectA` now contains a class `ClassOne` and a binding directive `ClassOne` → `ClassTwo` where `ClassTwo` is a class in `aspectB`. This binding directive means that the two classes `ClassOne` and `ClassTwo` should be merged. As the corresponding interfaces already contain all the structure (and behavior), this relationship can simply be translated into an inheritance between `ClassOne` and `ClassTwo`.

III. MAPPING BEHAVIOR

A. Mapping the Basic State Diagram

As previously mentioned, state diagrams cannot be implemented by linearly converting constructs at model level to those at the code level. Therefore, first we describe the mapping of basics and set the context in this subsection, and then move on to implementing some advanced concepts of state diagrams in the following subsections. An overview of the mapping of a state diagram is given in Table 1. For an illustration of the mapping of basic constructs, we use a very simple form of a state diagram, `StateDemo1`, shown in Fig. 1, which contains only two states, `StateA` and `StateB` and two transitions namely `t1` and `t2`. Transition `t1` changes the state from `StateA` to `StateB`, whereas the transition `t2` has the opposite effect.

A complete implementation hierarchy of the interfaces and associated aspects for the state view `StateDemo1` is shown in Fig. 2. We use two different types of objects to conceptually implement the state diagram: one to provide an entry point to the state diagram and maintain the context between states, and the other to provide a generalization of all states in the state diagram. We call the former as *context* and the latter as (state) *controller* object. Each state in the state diagram is implemented as a distinct object.

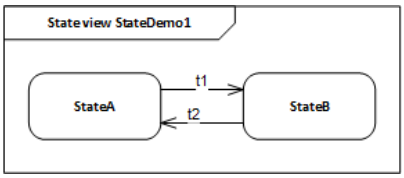

Fig. 1. State view StateDemo1

As the *context* serves as a single-entry point, we define the classes which implement different interfaces corresponding to states inside the *context* aspect. Events in the state diagram are implemented as methods; they are received by the *context* and delegated to the *controller* for processing. The *controller*, which points to the current state at any given time, processes the event based on the current state of system. Polymorphism is used to ensure the delegation of an event to an appropriate state for processing. The *controller* serves as an interface to all states in the state diagram. The name of the *controller* is generated by appending `State` to the name of the *context* interface.

Concrete states in the state diagram are mapped in a similar way. Each interface that represents a state in the state diagram *extends* the general state interface described previously. Such state objects are named exactly as the state they correspond to. These objects include all state-specific fields and implement the entire state-specific behavior. Internal transitions and entry/exit operations are also implemented in the corresponding state class. Guard conditions are mapped to appropriate `if` statements.

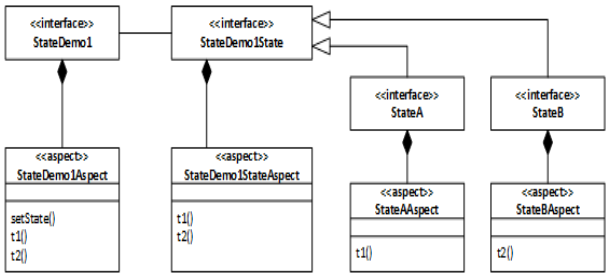

Fig. 2. Implementation hierarchy for StateDemo1

B. Mapping Composite States

A composite state contains other substates in the state diagram. For example, Fig. 3 shows two composite states, i.e., `StateB` and `StateC`. Here `StateB` contains four substates, whereas `StateC` contains two substates. There are two types of substates: sequential (nonorthogonal) and concurrent (orthogonal).

TABLE I MAPPING RAM'S STATE VIEW TO ASPECTJ

RAM construct	Mapped construct	Description
Context class	*Context* and *Controller* objects	Context class of a RAM model is the class with which state diagram is associated. The behavior of a class is captured by two objects: a *context* and a *controller*, each represented as an interface and an associated aspect.
State	State object	A concrete state is mapped to an object which is a combination of an interface and an associated aspect.
Event	Method	An event is defined as a method into the interface that corresponds to the *context*.
Action	Method	An action is defined as a method into the interface that corresponds to the *context*.
State hierarchy	Type hierarchy	Hierarchy between states is mapped by defining inheritance relationship between the corresponding interfaces.

Sequential substates partition the state space of a composite state into disjoint states. In Fig. 3, `StateC` contains two sequential substates namely `StateC1` and `StateC2`. Transition `t4` activates the substate `StateC1` as that is the default state within composite state.

Concurrent substates let modelers specify any number of state machines enclosed in concurrent regions which execute in parallel. This means that on entering a composite state of this type, an object will enter two or more states at a time. `StateB` in Fig.3 is a composite state with two concurrent regions. On transition `t1`, `StateB1` and `StateB3` will be entered concurrently.

1) Mapping the Composite States with Sequential Substates

The implementation hierarchy of interfaces and associated aspects for the state view `StateDemo2` is shown in Fig. 4. We create a *context* here that represents the composite state itself. Thus, referring to Fig. 3, a *context* object `StateC` is created to correspond to the composite

Authorized licensed use limited to: UNIVERSITY TEKNOLOGI MALAYSIA. Downloaded on April 07,2021 at 02:11:22 UTC from IEEE Xplore. Restrictions apply.

state `SateC` as shown in Fig. 4. This *context* object maintains a reference of the substate that is active at any given time. A state *controller* object `StateCState` encapsulates all state-related information and is inherited by the concrete state objects, i.e., interfaces corresponding to `StateC1` and `StateC2` in Fig. 4. The composite state contains two references: one for the object representing the initial (super) *context* and the other for composite state class. Therefore, `StateCState` maintains reference to `StateC` as well as to the global *context.*

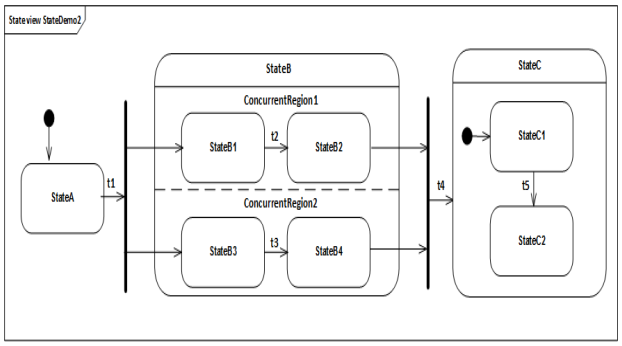

Fig. 3. State view StateDemo2 with composite states

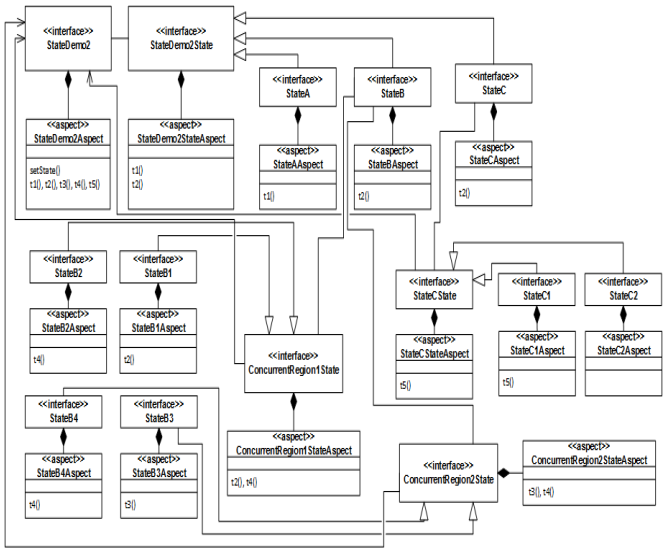

Fig. 4. Implementation hierarchy for StateDemo2

2) Mapping the Composite States with Concurrent Substates

In the first step, we implement the `StateB` conceptual object by creating an interface named `StateB` and an aspect named `StateBAspect` in the file of this interface as shown in Fig. 4. `StateBAspect` introduces the entry and exit actions as well as internal transitions, if required, into the `StateB` interface. Moreover, since `StateB` serves as the context for both concurrent regions, `StateBAspect` also introduces references to the substates within each concurrent region. In the next step, we implement the super state objects by creating two interfaces `ConcurrentRegion1State` and `ConcurrentRegion2State`. Both interfaces introduce abstract methods against the transitions `t2`, `t3`, and `t4`. Each interface maintains a reference to the implementation classes that capture context for the state machine and the composite state `StateB` as shown by the association of both with `StateDemo2` and `StateB`, respectively, in Fig. 4. In the last step, `StateB1` and `StateB2` interfaces are created along with the corresponding aspects to capture behavior associated with `ConcurrentRegion1`; `StateB1` and `StateB2` extend the `ConcurrentRegion1State` interface. Similarly, `StateB3` and `StateB4` extend the `ConcurrentRegion2State` interface and implement the behavior of `ConcurrentRegion2`.

Our approach implements a *fork* by setting the active substates of each concurrent region inside the entry method of the composite state. The approach implements a *join* in the entry methods of source states.

IV. EVALUATION

To determine the applicability of the mapping approach, we have applied it to obtain behavioral code for two nontrivial cases that involved crosscutting behavior from the literature. In the following, for space reasons, we briefly describe the system (Remote Service Handler [15]) that has relatively simpler implementation[1]. It contains functionality related to two use cases (described below) where behavior of one crosscuts the functionality of other.

Call remote service use case: This use case calls a remote service with no regard for scenarios where a call to remote service may not be successful.

(i) The use case starts when a service request is made.
(ii) Prior to calling the remote service, a GUI is disabled.
(iii) In case the call to remote service is acknowledged, a return value is logged and the GUI is updated.
(iv) Next, the GUI is enabled. Note that having a service request acknowledged is the only case when the GUI will be enabled.

Handle network failure use case: This use case is intended to capture the functionality for handling a network failure as described below:

(i) The use case starts whenever a call to remote service is made.
(ii) The use case sets a limit on the number of retries to call a service.
(iii) GUI is enabled in two mutually exclusive scenarios: when the service is not acknowledged or the limit to retry the remote service has reached.

The second use case crosscuts the first in the sense that whenever a call to remote service is made, the network failure handling behavior should be used to handle a failure in call. Composing the failure-handling would mean that the GUI should be re-enabled whether the remote service is successfully called or not. GUI will be enabled even if the call was not successful but the limit to retry the service was reached. However, logging of value and updating of GUI will only be carried out if the call succeeds.

[1] All aspect models as well as the entire source code for both systems may be accessed at https://git.io/fx69e

Authorized licensed use limited to: UNIVERSITY TEKNOLOGI MALAYSIA. Downloaded on April 07,2021 at 02:11:22 UTC from IEEE Xplore. Restrictions apply.

Fig. 1 shows the RAM model of network failure handling use case. The `NetworkFailureHandler` aspect is to be instantiated by the aspect that models the calling of remote service. This aspect basically expresses the reusable functionality to handle a network failure.

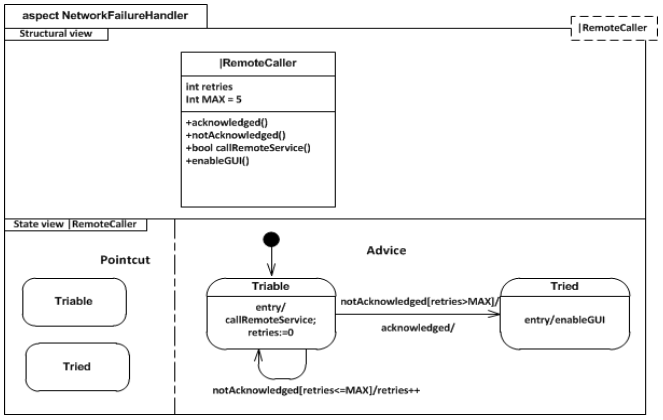

Fig. 5. Fig. 1: NetworkFailureHandler aspect

The `ServiceController` aspect presented in Fig. 6 depends on the functionality of `NetworkFailureHandler`.

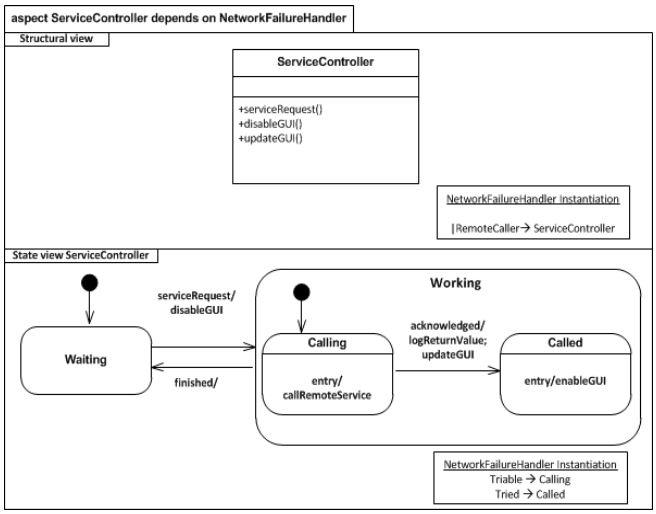

Fig. 6. ServiceController aspect instantiating the NetworkFailureHandler

Fig. 7 presents the code for `RemoteCallerAspect` defined in the file of `RemoteCaller` interface, which serves as the context object for the entire state diagram. It defines a local class to correspond to the controller object, `RemoteCallerStateClass`, and maintains a reference to it to point to current state.

V. Related Work

To the best of our knowledge, this is the first work in the literature that focuses on defining a detailed mapping of various constructs of the state diagrams in context of AO implementation. Nevertheless, some closely-related work, which relates to the mapping of AO behavioral models to AO code, is discussed in this section.

Kramer and Kienzle [16] present a mapping of RAM to AspectJ code. They have used annotation-based style of aspect declaration instead of using AspectJ code-based style, which we have used with the aim to obtain more concise and readable code. They implement both complete and incomplete classes as interfaces. The only difference between the techniques of implementation of these classes is that they provide a class that implements the interface corresponding to a complete class. No such class is provided for incomplete classes. In our approach, we required a coherent way in which all conceptual objects, whether specified in the structural view or those required to implement the state machine behavior, can be implemented. We map both complete and incomplete classes in a similar way and use type inheritance and method delegation to merge structure and behavior. For mapping of behavior, instead of mapping state diagrams, they have concentrated only on the sequence diagrams.

```
//interface RemoteCaller
aspect RemoteCallerAspect {
// declaration of fields and methods into interface
        ...
  static class RemoteCallerClass {
        public static RemoteCallerClass getInstance() {
                return new RemoteCallerClass();
        }
  }
//aspect-specific interface implementations to provide
//access to instances of classes that correspond to
//interfaces
  static class RemoteCallerStateClass {
        static RemoteCallerStateClass getInstance() {
                return new RemoteCallerStateClass();
        }
  }
  static class TriableClass {
        TriableClass(RemoteCaller rc) {
                remoteCaller = (RemoteCallerClass) rc;
        }
  static TriableClass getInstance(RemoteCaller      rc)
{
        return new TriableClass(rc);
  }
 }
        static class TriedClass {
                TriedClass(RemoteCaller rc) {
                        remoteCaller =
(RemoteCallerClass) rc;
                }
                static TriedClass
getInstance(RemoteCaller rc) {
                        return new TriedClass(rc);
                }
        }
//declare associations of classes to interfaces and
other classes
declare parents: RemoteCallerClass implements
RemoteCaller;
declare parents: RemoteCallerStateClass implements
RemoteCallerState;
declare parents: TriableClass implements Triable;
declare parents: TriedClass implements Tried;
declare parents: TriableClass extends
RemoteCallerStateClass;
declare parents: TriedClass extends
RemoteCallerStateClass;
}
```

Fig. 7. RemoteCaller containing implementations of the state classes

Clarke and Baniassad [17] have proposed an approach to mapping AO models developed using Theme/UML to AspectJ. They implement pattern classes of Theme by means of interfaces and introduce the non-template methods of model into these interfaces. Template operations with no

Authorized licensed use limited to: UNIVERSITY TEKNOLOGI MALAYSIA. Downloaded on April 07,2021 at 02:11:22 UTC from IEEE Xplore. Restrictions apply.

supplementary behavior (defined as a sequence diagram) are mapped to abstract methods, whereas those with supplementary behavior are mapped to abstract pointcuts. In contrast to our approach, this makes it mandatory to implement methods with no supplementary behavior with a delegating call.

A wide number of approaches to implement state machine specifications exist in the literature which target object-oriented (mostly Java) code. A majority of these approaches extends the State design pattern to implement state machines, see for example JCode [18] and OCode [19]. However, OCode and JCode mainly focus on extending the State pattern to solve the problems in an object-oriented context. Reusability of the models in a larger context has not been addressed.

VI. CONCLUSIONS

Visual modeling techniques enhance the software systems development by allowing modelers to view a system from different perspectives and at different levels of abstraction. Model-driven code generation can further enhance the benefits of modeling techniques by reducing the coding effort and consequently the delivery time. Several existing proposals have addressed the challenges involved in obtaining executable object-oriented code from design models. However, studies in AOSD suggest transforming aspect models into code of one of the AO programming languages. In this context, we have conducted this study to outline a technique for mapping AO models to AO code.

The main contribution of this paper is to investigate the challenges involved in the mapping of state diagrams that contain aspectual features to AO code. To support model-driven code generation in a broader context, we have selected a mature AO modeling technique, Reusable Aspect Models (RAM), to elaborate the mapping process. We have illustrated the mapping of the structural part of RAM models, which involves UML class diagrams and an associated reuse mechanism, and behavioral part, which uses the aspect state diagrams.

To make reuse at any level possible, our approach makes use of Java interfaces in combination with inter-type declaration mechanisms of AspectJ. Therefore, each conceptual class in our approach works as a combination of an interface and an associated aspect. We represent states in the state diagram as individual classes, and transitions on states as methods in these classes. In this way, all the behavior related to a state is localized into one object, which makes transitions more explicit. We have exploited this conceptual separation of states and used it to implement the substates and corresponding transitions, by defining new subclasses. Because of having this conceptual separation directly mapped to the code level, the code resultant from our approach is fully consistent with the model, and thus easier to understand and maintain.

We believe that our mapping approach can be used in combination with a textual representation of aspect state models to achieve automatic transformation to AO code.

REFERENCES

[1] Afonso, M., Vogel, R., and Teixeira, J.: 'From code centric to model centric software engineering: practical case study of MDD infusion in a systems integration company', in Editor (Ed.)^(Eds.): 'Book From code centric to model centric software engineering: practical case study of MDD infusion in a systems integration company' (2006, edn.), pp. 10 pp.-134

[2] Papotti, P., Prado, A., Souza, W., Cirilo, C., and Pires, L.: 'A Quantitative Analysis of Model-Driven Code Generation through Software Experimentation', in Salinesi, C., Norrie, M., and Pastor, Ó. (Eds.): 'Advanced Information Systems Engineering' (Springer Berlin Heidelberg, 2013), pp. 321-337

[3] Stavrou, A., and Papadopoulos, G.A.: 'Automatic Generation of Executable Code from Software Architecture Models': 'Information Systems Development' (Springer US, 2009), pp. 447-458

[4] Niaz, I.A., and Tanaka, J.: 'An Object-Oriented Approach to Generate Java Code from UML Statecharts', International Journal of Computer & Information Science, 2005, 6, (2)

[5] Rahmouni, M., and Mbarki, S.: 'An end-to-end code generation from UML diagrams to MVC2 web applications', International Review on Computers and Software, 2013, 8, (9), pp. 2123-2135

[6] Loukil, S., Kallel, S., and Jmaiel, M.: 'An approach based on runtime models for developing dynamically adaptive systems', Future Generation Computer Systems, 2017, 68, pp. 365-375

[7] Patel, P., and Cassou, D.: 'Enabling high-level application development for the Internet of Things', Journal of Systems and Software, 2015, 103, pp. 62-84

[8] Giunta, R., Pappalardo, G., and Tramontana, E.: 'AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns'. Proc. Proceedings of the 27th Annual ACM Symposium on Applied Computing, Trento, Italy2012 pp. Pages

[9] Lindström, B., Offutt, J., Sundmark, D., Andler, S.F., and Pettersson, P.: 'Using mutation to design tests for aspect-oriented models', Information and Software Technology, 2017, 81, pp. 112-130

[10] Hovsepyan, A., Scandariato, R., Baelen, S.V., Berbers, Y., and Joosen, W.: 'From aspect-oriented models to aspect-oriented code?: the maintenance perspective'. Proc. Proceedings of the 9th International Conference on Aspect-Oriented Software Development, Rennes and Saint-Malo, France2010 pp. Pages

[11] Mehmood, A., and Jawawi, D.N.A.: 'Aspect-oriented model-driven code generation: A systematic mapping study', Information and Software Technology, 2013, 55, (2), pp. 395-411

[12] Kienzle, J., Al Abed, W., Fleurey, F., Jézéquel, J.-M., and Klein, J.: 'Aspect-Oriented Design with Reusable Aspect Models', in Katz, S., Mezini, M., and Kienzle, J. (Eds.): 'Transactions on Aspect-Oriented Software Development VII' (Springer Berlin / Heidelberg, 2010), pp. 272-320

[13] Ali, S., Yue, T., and Briand, L.C.: 'Does aspect-oriented modeling help improve the readability of UML state machines?', Softw Syst Model, 2014, 13, (3), pp. 1189–1221

[14] Ayache, M., Erradi, M., Freisleben, B., and Khoumsi, A.: 'Aspect-Oriented State Machines for Resolving Conflicts in XACML Policies', in Editor (Ed.)^(Eds.): 'Book Aspect-Oriented State Machines for Resolving Conflicts in XACML Policies' (Springer, 2017, edn.), pp. 166-171

[15] Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., and Araújo, J.: 'MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation', in Katz, S., Ossher, H., France, R., and Jézéquel, J.-M. (Eds.): 'Transactions on Aspect-Oriented Software Development VI' (Springer Berlin / Heidelberg, 2009), pp. 191-237

[16] Kramer, M., and Kienzle, J.: 'Mapping Aspect-Oriented Models to Aspect-Oriented Code', in Dingel, J., and Solberg, A. (Eds.): 'Models in Software Engineering' (Springer Berlin / Heidelberg, 2011), pp. 125-139

[17] Clarke, S., and Baniassad, E.: 'Aspect-Oriented Analysis and Design: The Theme Approach' (Addison Wesley Object Technology, 2005. 2005)

[18] Niaz, I.A.: 'Automatic Code Generation From UML Class and Statechart Diagrams'. PhD Thesis, University of Tsukuba, Ph.D. Thesis., 2005

[19] Ali, J., and Tanaka, J.: 'Implementing the dynamic behavior represented as multiple state diagrams and activity diagrams', ACIS Int. J Comp. Inf. Sci., 2001, 2, (1), pp. 24-36

Authorized licensed use limited to: UNIVERSITY TEKNOLOGI MALAYSIA. Downloaded on April 07,2021 at 02:11:22 UTC from IEEE Xplore. Restrictions apply.