Layer Modeling and Its Code Generation based on Context-oriented Programming

Chinatsu Yamamoto[1], Ikuta Tanigawa[1], Kenji Hisazumi[2], Mikiko Sato[1], Takeshi Ohkawa[1], Nobuhiko Ogura[3] and Harumi Watanabe[1]

[1]*School of Information and Telecommunication Engineering, Tokai University, 2-3-23, Takanawa, Minato-ku, Tokyo 108-8619, Japan*

[2]*Department of Advanced Information Technology, Faculty of Information Science and Electrical Engineering, Kyushu University,744, Motooka, Nishi-ku, Fukuoka 819-0395, Japan*

[3]*Graduate School of Environmental and Information Studies, Tokyo City University, 3-3-1 Ushikubo-nishi, Tsuzuki-ku, Yokohama, Kanagawa 224-8551, Japan*

Keywords: Model-driven Development, Context-oriented Programming, Runtime Cross-cutting Concerns.

Abstract: This paper contributes to the runtime cross-cutting concerns problem by a layer structure model based on UML (Unified-Modeling Language) and code generation to COP (Context-Oriented Programming). For software development, the cross-cutting concerns problem is well-known to cause complicated models. The reason is that one cross-cutting concern affects multiple objects. Also, the problems occasionally occur at runtime. Recently, this problem has become more challenging. Modern software such as IoTs usually connect with many machines and devices and change context-dependent behavior at runtime. Thus, runtime cross-cutting problems will occur increasingly. To solve this problem, we focus on the COP. It can gather scattered cross-cutting concerns in one module called the layer and change the layer at runtime. However, UML lacks the notation involving COP and also the code generation. Therefore, the first step to solve the runtime cross-cutting concerns problem is to propose a layer structure model on UML and COP code generation from its model.

1 INTRODUCTION

Recently, modern software for IoT or Industry 4.0 is required to change their behavior in contexts dynamically. In this paper, the context means the surrounding environments of the system.

For example, we consider that the context of self-driving cars is the state of the road. On sunny days, self-driving cars run at usual speeds; on rainy days, self-driving cars run slower than usual; on snowy days, self-driving cars must also change wheel operation and speed from usual. In this example, one concern about the weather changes behavior. The behavior is processed by multiple modules: steering, engine, and brake. Thus, one concern is scattered to multiple modules. That is the cross-cutting concern. Also, in this example, the software is expected to change its behavior according to the situation at runtime. It means the cross-cutting concern should change at runtime. We call this problem runtime cross-cutting concerns. In modern software such as IoTs, this problem has become more challenging because they usually connect with many machines and devices and change context-dependent behavior at runtime.

COP (Context-Oriented Programming) is well-known to solve this problem in the surrounding environment at runtime (Hirschfeld, Costanza and Nierstrasz, 2008). COP is a programming language that includes the mechanism of software reconstruction. The mechanism is to activate or deactivate the program group according to the surrounding environment when the program is executed. To develop practical software, we consider MDD (Model-Driven Development) a candidate because MDD will impact the near future, as mentioned in Ebert (Ebert, 2018). Additionally, MDE (Model-Driven Engineering) can properly handle the design of many embedded systems, and MDD is known for its technology suitable for developing complex systems (Wehrmeister, Pereira, and Rammig, 2013). However, UML and MDD function lacks the runtime cross-cutting concerns of COP. This paper solves the following problems: lack of (1) a

Yamamoto, C., Tanigawa, I., Hisazumi, K., Sato, M., Ohkawa, T., Ogura, N. and Watanabe, H.
Layer Modeling and Its Code Generation based on Context-oriented Programming.
DOI: 10.5220/0010328303300336
In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2021)*, pages 330-336
ISBN: 978-989-758-487-9

notation of the runtime cross-cutting concern, (2) its MDD function. For example, the notation of COP-specific layers and the MDD function to generate layer programs have not been established.

To solve these problems, we propose a layer structure model for applying COP to MDD. The layer structure model is the expression of the layer for COP. We also propose a process for generating a COP program using that model. The process is generating a COP program that is modeled as a layer. Finally, a simple example shows the generation from the layer structure model to the COP program.

The remainder of this paper is as follows. Section 2 compares related work to this study. Section 3 defines the runtime cross-cutting concerns. Section 4 explains the goal of our study. Section 5 explains the COP briefly. Section 6 proposes a method. Section 7 argues each step of the proposed method in detail. Section 8 shows the simple example that the source code is generated using the proposed method automatically. Section 9 summarizes this paper and describes future works.

2 RELATED WORKS

This section compares related work to this study. First, COP is an extension of AOP. Many MDD for AOP (Aspect-Oriented Programming) has been proposed (Wimmer, Schauerhuber, Kappel, et al., 2011). However, there are few studies on MDD for COP compared to AOP. For example, UML4COP: UML-based DSML for Context-Aware Systems proposes to express COP based on UML (Unified-Modeling Language) proposes to express the COP model. In this study, the COP code is generated manually. Thus, there is no work from the COP model to the automatic generation of the COP code. In this section, the following mentions MDDs of AOP and just COP languages.

The most extensive survey of the AOM (Aspect-Oriented Modeling) approaches is provided by Chitchyan et al. (Wimmer, Schauerhuber, Kappel, et al., 2011). Aspect-Oriented Model-Driven Engineering for Embedded Systems Applied to Automation Systems (Wehrmeister, Pereira, and Rammig, 2013) aims to design real-time and embedded automation systems by combining UML and AOSD (Aspect-Oriented Software Development). The proposed method uses a tool that can perform everything from specifications to automatic source code generation. Thus, the transition from implementation can be done smoothly. Improved encapsulation of non-functional requirements has increased reusability. Moreover, cross-cutting concerns were concentrated on a small number of elements. However, cross-cutting concerns were improved, and the problem of scattering was reduced.

A Component Model for Model Transformations proposes a method for reusing model transformations between different modeling languages. In this proposed method, a component model for model transformation is designed (Cuadrado， Guerra and Lara., 2014). That is, the related work proposes component-based development. Transformation reuse, binding development, and component development were improved by using this proposed method.

An Approach for Mapping the Aspect State Models to Aspect-Oriented Code proposes a mapping of its constructs to AspectJ language using the state machine diagrams (Mehmood, Jawawi, and Zeshan., 2019). This related study uses the Reusable Aspect Models notation for this study. In this proposed method, the source code of the modeled structure and operation is generated using a reusable aspect-oriented model. The conceptual separation of state machine diagrams is directly mapped to the code level. Therefore, the source code obtained from this approach is the same as the model. Traceability is high, and maintenance is easy by using this proposed method.

COP is a language that has evolved around programming (Salvaneschi, Ghezzib, and Pradellab, 2012): such as ContextJ (Appeltauer, Hirschfeld, Haupt, et al., 2011), JCOP (Appeltauer, Hirschfeld, Masuhara, et al., 2010), EventCJ (Kamina, Aotani and Masuhara, 2011). There are also several studies on modeling languages (Kamina, Aotani, and Masuhara, 2011). However, as previously mentioned, there are no studies on COP that directly connect the model and the source code than aspect-oriented technology. Therefore, in this paper, we propose a layer structure model as the COP model. The layer structure model is for expressing COP in existing UML. Moreover, we also propose a method for generating COP code from the layer structure model. Thus, our novelties are the notation involving COP on UML and COP code generation.

3 RUNTIME CROSS-CUTTING CONCERNS

This section explains the problem of the runtime cross-cutting concerns. The cross-cutting concerns

are to scatter of processing related to one concern across multiple modules. In this paper, the runtime cross-cutting concern is defined as cross-cutting concerns, which changes at runtime.

Figure 1 shows an example of the runtime cross-cutting concerns. This concern is Sunny, Rainy, and Snowy. A self-driving car changes its driving according to the weather. The modules related to driving include Steering, Engine, and Brake. Each module must change its behavior for each concern of Sunny, Rainy, and Snowy. In this paper, this module is a class, and the behavior is a method of each class. For example, on a Sunny day, the self-driving car uses the methods of Steering and Engine class. The method of each class is changing at runtime because the weather changes during driving. This change is the runtime cross-cutting concern.

In a real system, there are many classes, and many cross-cutting concerns occur in those classes. Therefore, the runtime cross-cutting concerns cause system complexity.

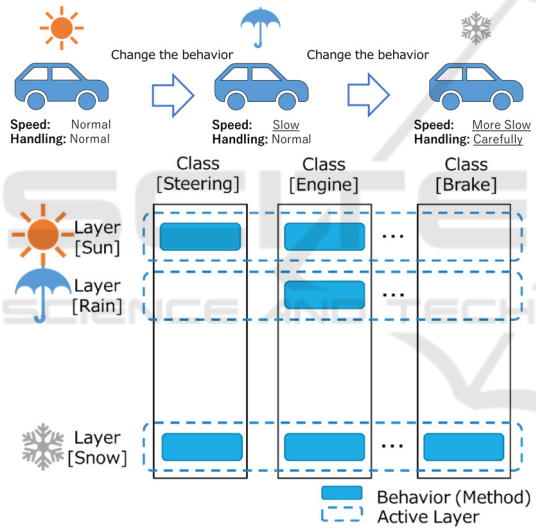


Figure 1: The runtime cross-cutting concerns problem.

4 GOAL

This section explains the goal of this study. The purpose is to reduce system complications due to the runtime cross-cutting concerns. We focused on COP as a way to address the runtime cross-cutting concerns. COP gathers each environment's operation as a layer and changes the operation by activating and deactivating the layer. In this paper, we propose an approach using COP to treat the runtime cross-cutting concerns. There are two goals in this paper.

(1) Layer structure model: We propose a layer structure model for the run time cross-cutting concerns of COP.
(2) COP code generation: We propose COP Extractor to generate COP code on an MDD tool automatically. The source code is automatically generated from the model of using the layer structure model as a simple example.

By achieving the goals (1) (2), we show that the layer structure model can modularize cross-cutting concerns in one layer and generate from this model to COP code.

5 CONTEXT-ORIENTED PROGRAMMING

This section explains the outline of a COP by using RTCOP (Tanigawa, Hisazumi, Ogura, et al., 2019). COP can use modules called layers to modularize context-dependent cross-cutting concerns. The layers consist of one base layer and others.

The base layer aims to give a structure of classes and the behavior of multiple methods. The base layer is overwritten by a different new layer at runtime when its new layer is activated. Figure 2 shows an example of RTCOP. In this example, firstly, the *Sunny Layer* is activated. At this time, *Sunny Layer* overwrites *Base Layer* and runs the method *Run( )* on *Sunny Layer*. Then, *Sunny Layer* is deactivated and activated *Rain Layer*. The method *Run()* changes to *Run()* on the *Rainy Layer*.

Usually, one layer includes multiple classes. Thus, COP can change multiple classes at runtime. In other words, COP can deal with the runtime cross-cutting concerns.

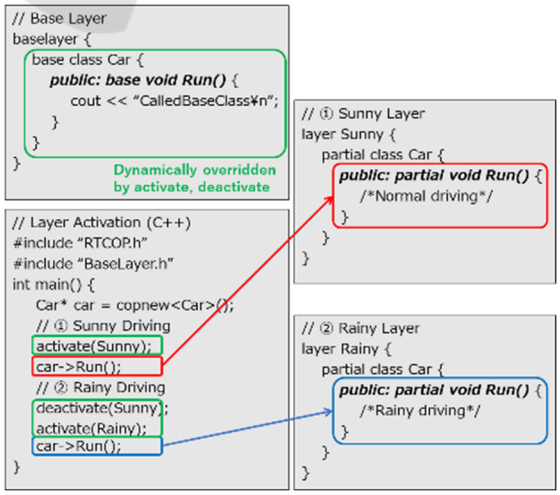


Figure 2: Outline of COP.

6 LAYER STRUCTURE MODEL

In this section, we propose to extend UML as a layer structure model. In the layer structure model, we can gather scattered methods in one module called layer. To express the layer, we define the stereotype <<layer>>. This stereotype is attached to the package of UML. By this stereotype, we can distinguish the layers from the original packages. Figure 3 shows an example of the layers. In this example, there is a one-layer *AA_Layer* and one package *AA*. Additionally, *AA_Layer* contains one class diagram that holds two classes.

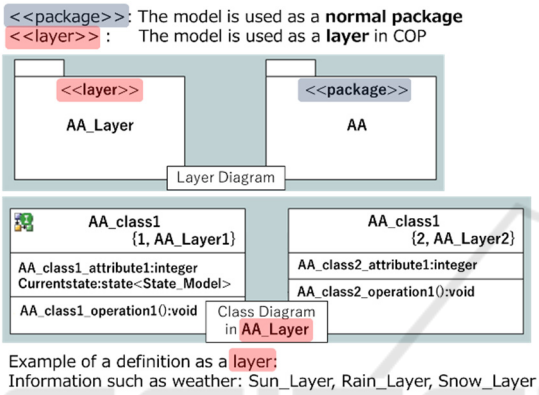


Figure 3: Layer structure model.

7 COP GENERATION

This section proposes a process of generating the COP program from the layer structure model. Figure 4(a) shows an overview of the process for generating the source code. This process consists of two parts: the original MDD tool xtUML / Bridgepoint (xtUML.org, 2020)., and the proposing COP Extractor. xtUML/Bridgepoint is a tool based on Executable UML. Executable UML is Object-Oriented technology. Executable, translatable UML (xtUML) is an extension to UML based upon the Shlaer-Mellor Method of MDA, which supports a powerful approach to MDD (xtUML.org, 2020).

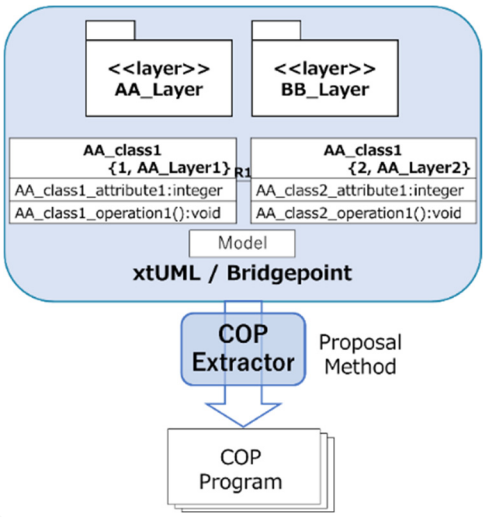
The Extractor changes the intermediate code of xtUML for generating COP code.

Figure 4(b) shows the steps of the code generation process. The purpose of each is as follows:

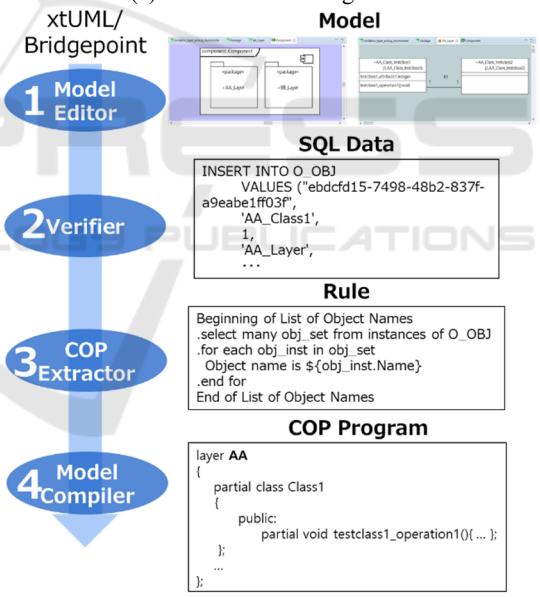
(1) Model Editor: Separate a COP model and the normal model
(2) Verifier: Generate model data from models,
(3) COP Extractor: Preparation for generating COP program with COP Extractor
(4) Model Compiler: Generate the COP program.

Our original process is (3) COP Extractor. Others are normal processes of xtUML / Bridgepoint.

The detailed process for generating COP code is explained in the following subsection. The mechanism of generation is mentioned in Section 7.



(a) Overview of COP generation.



(b) Steps to generate COP program.

Figure 4: Overview and Steps of COP generation.

7.1 Layer Structure Model on XtUML/ Bridgepoint

This subsection explains (1) Model Editor: separate a COP model and the normal model. The developer draws a layer structure model based on UML. The developer needs to include at least the component diagram, package diagram, class diagram, and state

machine diagram. There are restrictions on package diagrams and class diagrams.

First, the restrictions on the package diagram will be explained. In the package diagram, it is necessary to separate the standard model and the COP model. The layer structure model creates a layer in the package diagram. Moreover, the layer structure model creates a layer using the stereotype <<layer>>. The layer name as a layer structure model needs to include *_Layer* to the name of the package diagram. Next, the restrictions of the class diagram will be explained. *Class Key Letters* is one of the properties of the class diagram notation in BridgePoint. For the class included in the layer, it is necessary to write the included layer name in the *Class Key Letters*. For example, *AA_class1* needs to write *AA_Layer1* in *Class Key Letters* because *AA_class1* is included in AA_Layer. Figure 5 shows to summarize the restrictions of each diagram.

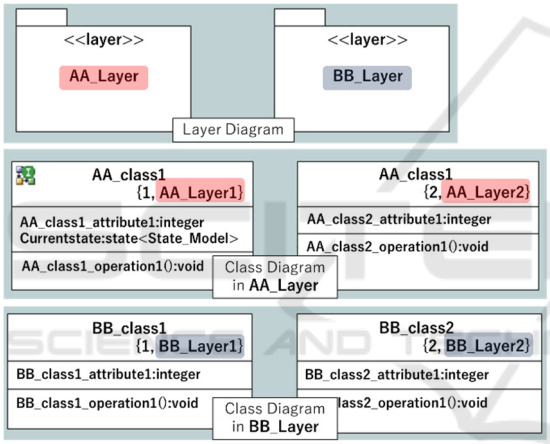


Figure 5: Creating Layer Structure Model.

7.2 Model Data Generation

This subsection explains (2) Verifier: generate model data from models. The model data is generated by SQL from the function of xtUML / BridgePoint. The model data is stored as a database. The model data is a construct of the created model. This model data is necessary to generate a COP program. Using the model data stored as the database, COP Extractor searches for the classes contained in the layer. COP Extractor is explained in the next subsection. Table 1 shows an example of the model data construct. This model data is the construct of a layer. All models are generated as model data by following the metamodel of xtUML / BridgePoint. COP Extractor uses Name and Descrip in Table 1. The ID of the class that has the layer ID is searched and extracted.

Table 1: Model data by SQL.

Create Table	Insert Data
Package_ID	0ab5d488-bd73-4805-9403-0cfd63d85c24
Sys_ID	00000000-0000-0000-0000-000000000000
Direct_Sys	e8d942bb-de5e-48da-b726-4635f3a3e2c8
Name	AA_Layer
Descrip	<<layer>>
Num_Rng	0

7.3 COP Extractor

This subsection explains (3) COP Extractor: Preparation for generating COP program with COP Extractor. This part is our original. A COP program is generated using the model data explained in the previous subsection. The COP program is generated in C++. Programs such as components and classes generate C++ source code using the Model Compiler specific to xtUML / Bridgepoint. COP Extractor creates the necessary parts for the layer program. COP Extractor modifies and adds the arc file, which is the source code generation rules for xtUML / Bridgepoint. COP Extractor is described by RSL (Rule Specification Language). COP Extractor accesses the model data stored in the database and extracts the elements for generating a COP program. Figure 6 below shows the step for generating the source code of COP Extractor.

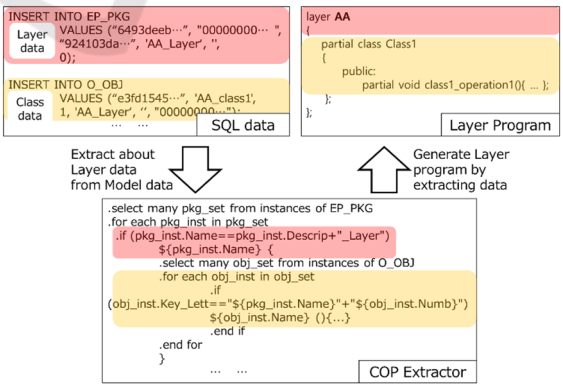


Figure 6: COP Extractor.

7.4 COP Program Generation

This subsection explains (4) Model Compiler: generate the COP program. COP Extractor first

extracts the layer name from model data. The stereotype <<layer>> is used for extracting as an extraction method. Next, the classes related to the layer are extracted using a layer name. The layer name of the class is described in Class Key Letters. The proposed process uses Class Key Letters to extract the necessary classes for layer structure. After extracting the information about all layers, the class is described in each layer. Besides, the layer program is output as a COP program. Moreover, the necessary information is described before generating source code that information is for activating and deactivating the layer in COP. Therefore, it adds detailed information about the layer. Figure 7 shows an example of the created model and the generated COP program.

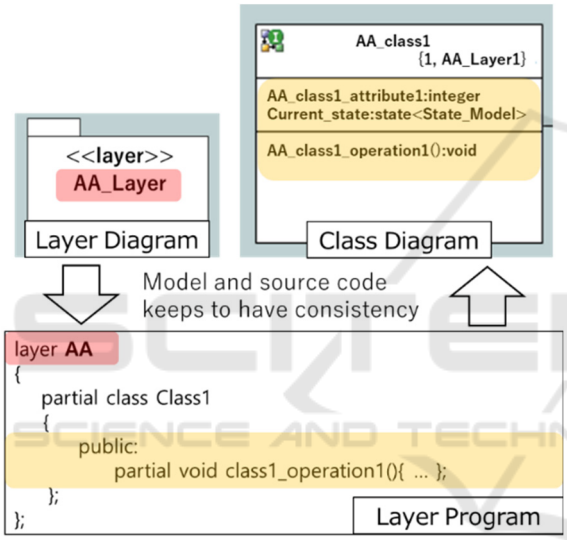
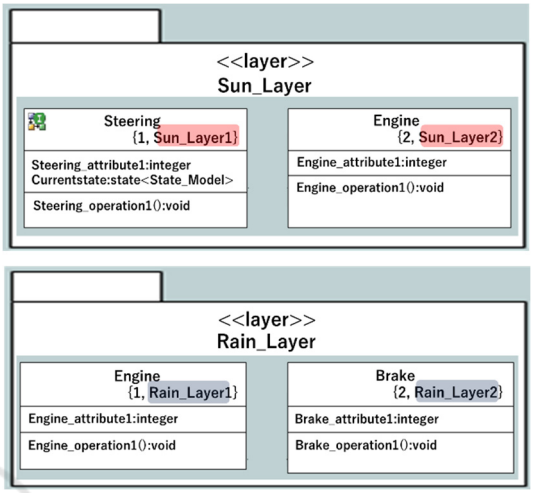


Figure 7: Model and source code.

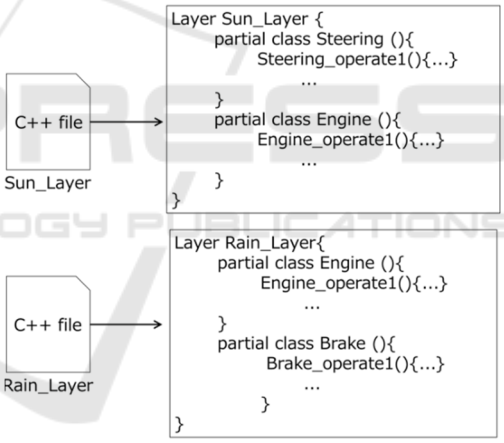
8 EXAMPLE OF COP GENERATION

This section shows the example that is to generate an RTCOP program from a layer structure model. As this example, we create two layers, Sun_Layer and Rain_Layer. Each layer includes two classes. For example, Sun_Layer includes two classes, Steering_class and Engine_class. COP Extractor generates the source code from each model using the proposed method. The purpose of this example is to confirm that each class is properly described in the source code.

Figure 8 shows an example of a layer structure model and a part of the RTCOP program. Figure 8(a) shows an example of the generated layer and class model. Figure 8(b) shows that a COP program is generated using COP Extractor and a part of programs of each class generated using xtUML / Bridgepoint. COP Extractor generates a program file with each layer name as a program for each one. From these figures, it can be confirmed that each layer is modularized.



(a) Example of Layer Structure Model.




```
/*----------------------------------------------------------------------
 * File:  sample_comp1_Engine_class.cpp
 * Class:       Engine  (Sun_Layer1)
 * Component:   sample_comp1
 * your copyright statement can go here (from te_copyright.body)
 *----------------------------------------------------------------------*/
#include "sample_layerpackage_sys_types.h"
#include "sample_comp1.h"

/*instance operation:  Engine_operation1*/
void
sample_comp1_Sun_Layer1::sample_comp1_Sun_Layer1_op_Sun_class1_operation1( sam
ple_comp1_Sun_Layer1 * self, sample_comp1 * thismodule){}
/*----------------------------------------------------------------------
 * Operation action methods implementation for the following class:
 * Class:       Engine  (Sun_Layer1)
 * Component:   sample_comp1
 *----------------------------------------------------------------------*/
static sys_sets::Escher_SetElement_s
sample_comp1_Sun_Layer1_container[ sample_comp1_Sun_Layer1_MAX_EXTENT_SIZE ];
static sample_comp1_Sun_Layer1
sample_comp1_Sun_Layer1_instances[ sample_comp1_Sun_Layer1_MAX_EXTENT_SIZE ];
sys_sets::Escher_Extent_t pG_sample_comp1_Sun_Layer1_extent = {
  {0}, {0}, &sample_comp1_Sun_Layer1_container[ 0 ],
  (Escher_iHandle_t) &sample_comp1_Sun_Layer1_instances,
  sizeof( sample_comp1_Sun_Layer1 ), sample_comp1_Sun_Layer1_STATE_1,
sample_comp1_Sun_Layer1_MAX_EXTENT_SIZE
  };
```

(b) Sample Generation of COP.

Figure 8: Example of Layer Structure Model and source code.

9 CONCLUSIONS

This paper proposed (1) the layer structure model on UML and (2) COP code generation from its model. Those aim to solve the runtime cross-cutting concerns problem. COP is well-known to solve this problem; however, we cannot easily represent the runtime cross-cutting concern in UML. Also, MDD has never generated COP code. We showed that the layer structure model could modularize cross-cutting concerns in one layer and generate from this model to COP code.

In future work, we will challenge the following: (1) describe Activate and Deactivate at appropriate timings; (2) measure the degree of coupling and module cohesion of each class and layer; (3) apply metamodels to source code generation function on MDD (4) define how to verify the created layer structure model. Moreover, we consider the performance of the code generated by this method as future work.

REFERENCES

Hirschfeld, R., Costanza, P., Nierstrasz, O., 2008. *Context-Oriented Programming*, Journal of Object Technology, 7(3).

Ebert, C., 2018. *50 Years of Software Engineering: Progress and Perils*, IEEE Software, p. 94-101.

Wehrmeister, M., Pereira, C., Rammig, F., 2013. *Aspect-Oriented Model-Driven Engineering for Embedded Systems Applied to Automation Systems*., IEEE Transactions on Industrial Informatics, p. 2373-2386.

Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., Kapsammer, E., 2011. *A Survey on UML-Based Aspect-Oriented Design Modeling*, ACM Computing Surveys, p.1-59.

Ubayashi, N., Kamei, Y., 2012. *UML4COP: UML-based DSML for Context-Aware Systems*. Proceedings of the 2012 workshop on Domain-specific modeling, p.33-38.

Cuadrado, J., Guerra, E., Lara, J., 2014. *A Component Model for Model Transformations*. IEEE Transactions on Software Engineering, p. 1042-1060.

Mehmood, A., Jawawi, D., Zeshan, F., 2019. *An Approach for Mapping the Aspect State Models to Aspect-Oriented Code*. International Conference on Engineering and Emerging Technologies, p. 1-6.

Salvaneschi, G., Ghezzib, C., Pradellab, M., 2012. *Context-oriented programming: A software engineering perspective*, The Journal of Systems and Software, p. 1801-1817.

Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H., 2011. *ContextJ: Context-oriented Programming with Java*, Information and Media Technologies, p.399-419.

Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M., Kawauchi, K., 2010. *Event-Specific Software Composition in Context-Oriented Programming*, SC 2010: Software Composition, p. 50-65.

Kamina, T., Aotani, T., Masuhara, H., 2011. *EventCJ: a context-oriented programming language with declarative event-based context transition*, AOSD'11: Proceedings of the tenth international conference on Aspect-oriented software development, p. 253-264.

Lincke, J., Appeltauer, M., Steinert, B., Hirschfeld, R., 2011. *An open implementation for context-oriented layer composition in ContextJS*, Science of Computer Programming, p.1194-1209.

Tanigawa, I., Hisazumi, K., Ogura, N., Sugaya, M., Watanabe, H., Fukuda, A., 2019. *RTCOP: Context-Oriented Programming Framework based on C++ for Application in Embedded Software*, Proceedings of the 2019 2nd International Conference on Information Science and Systems (ICISS 2019), p. 65-72,

xtUML.org. *xtUML | eXecutable Translatable UML with Bridgepoint*,September 9, 2020, from " https://xtuml.org/".