JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 649-670 (2018)
DOI: 10.6688/JISE.201805_34(3).0005

From Early Aspect to Aspect-Oriented Programming: A State-Based Join Point Model Approach[*]

KUO-HSUN HSU AND CHANG-YEN TSAI
Department of Computer Science
National Taichung University of Education
Taichung, 403 Taiwan
E-mail: glenn@mail.ntcu.edu.tw; ss8805733@hotmail.com.tw

Aspect-oriented approaches have recently had a tremendous impact on the characterization of crosscutting concerns during the development of software systems. However, issues in transforming early aspects in the design stage into final codes still remain to be addressed. In this paper, we focused on operator conversion rules, which convert the early aspects described in state diagrams into AspectJ codes systematically. By weaving these aspect codes into base codes, a final system can be obtained that is enhanced by early aspects in a systematic manner. A tool that supports the conversion rules of the operators was developed. The tool transforms the aspect behaviors described in the aspect-enhanced state diagram into aspect code. To validate our proposed method, a meeting scheduler system was designed and implemented.

Keywords: early aspect, goal, use case, state diagram, join point model, software engineering, requirement engineering

1. INTRODUCTION

Aspects as a means to address separation of concerns (SoC) problems [1], are attracting increasing interest from aspect-oriented software development (AOSD) researchers [2]. In the implementation phase, an aspect [3-5] is implemented by a piece of code representing the realization of a crosscutting concern that may span or be scattered across multiple functional units such as classes or modules in a software system. By managing these crosscutting concerns within a modular design, the coupling strength between codes or modules and the complexity of system functionalities can be decreased substantially.

In our previous studies [6, 7], the relationships between goals, use cases, and early aspects were used to discover early aspects in the analysis stage. Interactions between goals and use cases were analyzed in a numerical manner, and early aspects were described in an aspect-enhanced use case diagram. In addition, the use case specification was augmented for describing the properties of early aspects represented as aspectual use cases in the diagram. Furthermore, an aspect-enhanced sequence diagram and extended state-based join point model (eSJPM) were proposed to solve the problem of representing aspects at the design stage. This previous work addressed the use of aspects between requirement analysis and system design, however, the problems faced at the transition from the design to implementation stages remain, which motivated this study.

We propose a state-based join point model approach for the transformation of as-

Received November 23, 2016; revised January 27, 2017; accepted February 21, 2017.
Communicated by Chang-Shing Lee.
[*] This research was supported by Ministry of Science and Technology (Taiwan) under grants MOST 105-2221-E-142-006.

pectual behaviors described in aspectual use cases into aspectual codes based on ESJPM. Three crucial elements are depicted within this model: state transitions for guiding the weaving sequence, join points for representing weaving points, and advice of the determining actions to be taken.

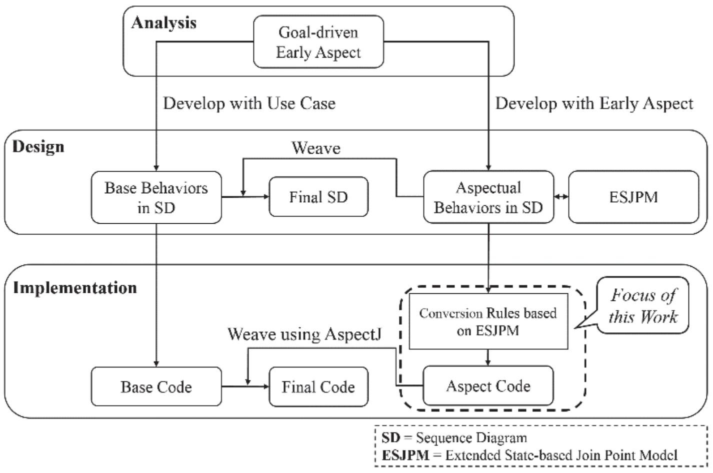


Fig. 1. Concept of the proposed approach.

Fig. 1 illustrates the concept of the proposed approach. Traditionally, aspects are written separately and then woven into the base code in the implementation stage. In our previous work [6, 7], we proposed employing use cases and early aspects derived from the goal-driven early aspect approach to model base behaviors and aspectual behaviors in sequence diagrams (SDs). That is, aspectual use case specification is augmented with aspectual properties to document the responsibilities of an aspectual use case. Aspectual behaviors related to those responsibilities are modeled in the aspectual sequence diagram, which makes explicit the inclusion of aspects with the help of the ESJPM for specifying weaving semantics. Moreover, an ESJPM that described weaving behaviors was established to assist in the weaving of base behaviors and aspectual behaviors and to obtain the final SD in the design stage. However, an approach more suited to comply with the software development process (SDP) is to address aspects during all stages. Therefore, in this work, six conversion rules based on ESJPM are proposed to bridge the gap between the design and implementation stages, as shown by the dashed line in Fig. 1. The final code is obtained by weaving the aspect code generated by applying the conversion rules into the base code. Thus, we not only bridge the gap between the design and the implementation stages but also complement the SDP by addressing aspects in all stages.

Background work particularly relevant to this paper is outlined in Section 2. Section 3 discusses the modeling of aspectual behaviors and base behaviors in the SD, and the establishment of the ESJPM. The conversion rules used to convert the ESJPM into aspect code are introduced in Section 4. In Section 5, a meeting scheduler system analyzed and designed within the goal-driven early aspect approach, and implemented by applying the conversion rules, is described to demonstrate the feasibility of the proposed approach. The conclusions and advantages of the proposed approach are detailed in Section 6.

2. RELATED WORK

Herein, we outline the literature that informed the present study, namely, that on the support of AOSD during the design stage, and the previous work on the conversion of aspectual behaviors into aspectual codes in the implementation stage.

2.1 Aspect-Oriented Development Method in Design Stage

The join point model [8] implemented in aspect-oriented programming (AOP) is the key concept in aspect orientation, and it dictates when and where crosscutting modularization occurs. Specifying a set of join points is a major task for aspect-oriented designers, and effectively representing join points is a critical task in the evaluation of aspect interaction throughout software development.

Boucke and Holvoet [9] indicated the need for high-level join points to capture abstract system states, a problem that was encountered in the development of an application to control automatic guided vehicles in a warehouse management system. They argued that the abstract states of SoC modules are necessary for the definition of join points in programs with crosscutting concerns. However, this work was only a preliminary attempt at the state-based join point modeling, where the authors outlined a Unified Modeling Language (UML) class diagram that corresponded to a concern and the software entities that constituted the concerns.

Ali *et al.* [10] proposed a state-based join point model (SJPM) that was motivated by the deficiencies of existing fixed code-based behavioral join point models when used to support the implementation of crosscutting concerns in systems that required constant state monitoring, such as safety-critical systems. To capture the crosscutting behaviors that are activated by some system state transitions, the SJPM defines a crosscutting system state as an abstract state machine, and uses its transitions to identify the join points of aspect superimposition. These state-based join points provide the foundation for a state-based aspect implementation.

Stein *et al.* [11, 12] proposed the aspect-oriented design model (AODM), which applies existing UML concepts to aspect-oriented concepts currently used in AspectJ. Aspects are represented as UML classes of a special stereotype called ⟪*aspect*⟫. Analogous to aspects in AspectJ, "pointcut" elements and "advice" operations can be defined as operations in the aspect class with ⟪*pointcut*⟫ and ⟪*advice*⟫ stereotypes, respectively. In the AODM, the crosscutting behavior of a program (implemented using advice in AspectJ) is visualized by highlighting messages in a UML SD. Araùjo *et al.* [13, 14] introduced aspects to scenario-based software requirement research. They focused on representing aspects during use case modeling and in particular on how to compose aspectual and nonaspectual scenarios in a way that enabled them to be simulated together. Non-aspectual scenarios were modeled using UML SDs, and aspectual scenarios were modeled as interaction pattern specifications [15]. Finite state machines were modeled as UML state machines and an aspectual finite state machine were modeled as state machine pattern specifications [16]. The aspectual and non-aspectual state machines were then composed using a state machine synthesis algorithm. The result was a new set of state machines representing the complete specification. The notion of join points was implicit in the synthesis algorithm, however, how the aspectual scenarios crosscut the non-aspectual scenarios was not explicitly modeled in the SDs.

From these studies, we can summarize that in some work, join points were specified directly by marking the woven actions in UML SDs, which cannot easily be achieved in state-sensitive systems. Support for specifying how, where, and when aspectual behaviors occur is rather limited. Furthermore, specifying one type of join point at which the aspectual behaviors do not depend on the specific actions performed, but on a specific state transition, is difficult.

2.2 Converting Aspectual Behaviors into Aspectual Code

Groher and Schulze [17] extended UML by including a new notation for separating aspect code from base code, which also supported the generation of code from UML models. The extended notation was proposed to provide guidelines for modeling, and the design methodology was based on AspectJ, which can be easily applied with numerous CASE tools.

Hecht *et al.* [18] researched automatic generation in the implementation phase based on XSLT. They first produced an XMI file output from class diagrams drawn by a UML editor and manually edited it. Using the rules of the aspect program code described in the XMI file, it was then transformed into program code using XSLT. This method is suitable for programmers who are implementing actual code, but not for abstract requirements in the development phase.

Wehrmeister *et al.* [19] presented a model-driven engineering approach, AMoDE-RT, to design real-time and embedded automation systems, which combined UML, AOSD, platform-based design [20], and code generation techniques in a consistent set of activities and tools. AMoDE-RT enables straightforward progress from requirements engineering to the system's implementation.

These studies demonstrate that the procedure used in code creation concerns high-level products that are developed artificially. Therefore, UML is suitable to describe the abstract requirements of human thought. The conversion has been mainly addressed on a code level and it would be difficult for these approaches to adopt early aspects because early aspects represent high-level and abstract requirement descriptions. Therefore, methods must be devised that bridge the gaps between the analysis, design, and implementation phases.

3. ASPECT-ENHANCED SEQUENCE DIAGRAM

To address the gaps and incorporate aspects between the design and implementation stages, it is crucial to clarify how to model aspectual behavior in an aspectual SD with an ESJPM. In our previous study [7], an augmentation of the use case specification to include aspectual properties was proposed for addressing the issue of representing aspects in the requirement stage. Once the aspectual use case specification is designed, we can begin to model the aspectual behavior in the aspectual SD with interaction operators. Aspects can then be woven into the base use cases and the ESJPM can be appended with the weaving semantics. The types of weaving operators used serve as the background of our conversion rules.

3.1 Augmented Use Case Specification with Aspects

Each use case specifies what must be done to perform its functionalities (also called responsibilities). A basic use case specification includes a use case ID, name, pre-conditions, post-conditions, actors, a basic flow and alternative flows. Fig. 2 shows a use case specification augmented to include aspectual properties, which can address the responsibilities of aspectual use cases.

Use Case ID	<A unique identification that is used to organize all the use cases>		
Use Case Name	<The name of this use case>	**Type**	<The type that is used to specify if this use case is an aspectual or not>
Woven Base Use Cases	<A collection of base use cases that are affected by this aspectual use case>		
Actors	<A list of roles that are involved in this use case>		
Pre-Condition	<A list of conditions that must be true before the use case starts>		
Pre-Condition	<A list of conditions that must be true when the use case ends>		
Join-points	<A description of joinpoints belonging to the woven use cases where the respective behavior of this aspectual use case will interleave>		
Basic Flow	<The basic path that is written as if everything goes right>		
Alternative Flow	<The alternative path that allows an alternative sequence of events>		
Types of Weaving Operators	<The type of weaving operators that is used to express what kinds of weaving operation this aspectual use case performs>		

Fig. 2. Detailed use case specification template.

3.1.1 Type

Type is used to specify the type of category a use case belongs to, which is either an aspectual use case that interleaves with base use cases or a base use case that may be affected by aspectual use cases.

3.1.2 Woven base use cases

Woven base use cases are a set of use cases that an aspectual use case crosscuts. Unlike extend or include relationships, an aspectual use case crosscuts more than one base use case.

3.1.1 Join points

Join points describe when and where the corresponding aspectual behavior weaves into the base use cases that the aspect crosscuts. They are specified in the aspectual use case specification, not the base use case specification.

3.1.1 Types of weaving operators

The three categories of weaving operators are “insert behavior,” “replace behavior,” and “impose constraint behavior” operators, further elaboration of which can be found in Section 4.

Fig. 2 is a template of an aspectual use case specification for documenting aspectual properties; however, it is up to system analysts to determine their own format, including information that is necessary for the development of a software system that is suitable for their organization or application.

3.2 Aspectual Sequence Diagram

UML interaction diagrams [21, 22] model the dynamic behaviors of a system. SDs are a specific type of interaction diagram that portray the timing of messages dispatched among a set of object instances. They are also the most direct and intuitive means for describing how a group of object interact with each other. The proposed extension to UML SDs is to introduce weaving operators that are implemented by an instance of a weaver object. This incorporates aspectual behavior into object instances in the base SD and the three types of weaving operators. The focus of the proposed extension to the SDs is twofold: (1) to express the weaving semantics by modeling what and how the notion of aspectual behavior can be interleaved into the sequence of event occurrences that are defined in the woven use case specification, and (2) to specify the join points by modeling where the aspectual behavior will be interleaved into those behaviors defined in the woven SDs.

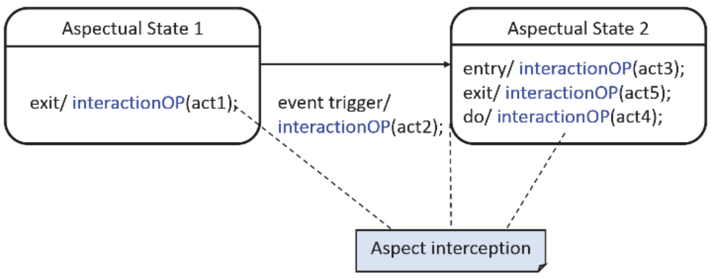


Fig. 3. Extended state-based join point model.

3.3 Extended State-Based Join Point Model

In AOP [8], a join point is a well-defined location within the primary code where a concern crosscuts an application. Join points can be method calls, constructor invocations, exception handlers, or other points in the execution of a program. In aspect-oriented modeling, how to denote join points and their corresponding behaviors with suitable notation is the main focus of aspect modeling. Ali *et al.* [10] proposed an SJPM that supports, from a system dynamics behavior perspective, the implementation of crosscutting concerns in systems that must be monitored constantly, such as safety-critical systems. This addressed the deficiency of existing fixed code-based behavioral join point models. The SJPM offers a high-level conceptual method to define and capture the concerns of a software system, and assists analysts in representing nonfixed code-based aspectual behavior in a systematic manner. An extension to the SJPM, called ESJPM (shown in Fig. 3), was proposed in [7] to facilitate the capture and representation of the dynamic behavior of aspectual use cases with aspect weaving in the early stage of software development, which has the following three features:

1. Attachment of state machines to aspectual use cases to represent aspectual behavior and join points;
2. Enrichment of the weaving semantics by introducing weaving operators into action expressions in each early aspect state; and
3. Augmentation of the possible weaving location (called aspect interception in the ESJPM) to include the entry point, inside state, and state exit point, to more effectively include aspectual behavior in the software.

The proposed conversion rules used to bridge the gap between the design and implementation stages are discussed in detail in the following section.

4. CONVERSION RULES

In the conversion mechanism, the ESJPM, which describes the weaving behaviors, is used to provide the input information. In addition to the ESJPM, a defined code is required to assist the conversion of aspectual behaviors to aspect code. In this section, we introduce the steps of the conversion mechanism and conversion rules of applying weaving operators, and depict the code fragment that supplements the weaving operator.

Six weaving operators are classified into three operator type categories, as shown in Table 1. Through the use of weaving operators, the system analyst can explicitly specify the kind of weaving operation to be performed by an aspectual use case and the effects to base use cases.

Table 1. Weaving operator categories.

Operator Type	Operator Name
Insert Behavior	insert insertPar (insert parallel)
Replace Behavior	replace
Impose Constraint Behavior	IDC (impose duration constraint) ITC (impose timing constraint) ISI (impose state invariant)

1. Insert behavior includes two weaving operators, $WeaveOP_{insert}$ and $WeaveOP_{insertPar}$. $WeaveOP_{insert}$ operator is applied when the woven aspect owns the control of executing the base use case till the end of performing the aspectual behavior. $WeaveOP_{insertPar}$, on the contrary, does not interfere with the control flow of the base behavior. The aspectual behavior will be executed in parallel with the woven behavior.
2. Replace behavior has only one weaving operator, named $WeaveOP_{replace}$. It is applicable only when aspectual behavior is used to replace the base behavior at a given joinpoint.
3. Impose constraints include three weaving operators, namely, $WeaveOP_{IDC}$, $WeaveOP_{ITC}$ and $WeaveOP_{ISI}$. $WeaveOP_{IDC}$ is applied when an aspect imposed a duration constraint on the execution of base behavior. $WeaveOP_{ITC}$ is used when an aspect imposes a timing constraint on the base behavior. And $WeaveOP_{ISI}$ is used to introduce and maintain an invariant state described by an aspect to the base use case's behavior.

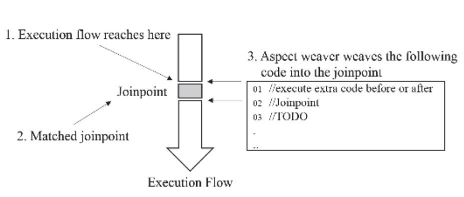
The definition of the aspectual behaviors and how they are transformed into conversion rules are as follows:

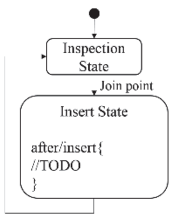
1. Depicting the weaving operator: It is paramount to know which weaving operator is appropriate. An appropriate scenario for each weaving operator is depicted and where the aspect is to be woven is defined.
2. Transforming the scenario into the ESJPM: To provide weaving operator information to the conversion mechanism, the weaving operator scenario is modeled in the ESJPM.
3. Converting the ESJPM into aspectual code: A generic sample code for each weaving operator is provided as a guideline to transform the EJSPM into aspectual code.

In the following, rules of applying the weaving operators are all expressed in a generalized format. However, to express the weaving operators more convincingly, a realistic example is also provided using AspectJ [3-5].

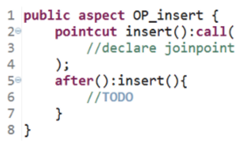
4.1 OP Insert Rule

An OP insert is used to assist developers to insert extra code at a specified point in an execution flow. The scenario of where an OP insert operator should be applied is shown in Fig. 4 (a). When the execution flow reaches a join point, the aspect weaver obtains the Join Point information from the base code and aspect code and creates a new class with the aspect code as depicted in the code block at the right-hand side of the figure.


1. Execution flow reaches here
Joinpoint
2. Matched joinpoint
3. Aspect weaver weaves the following code into the joinpoint
01 //execute extra code before or after
02 //Joinpoint
03 //TODO
Execution Flow
(a) Scenario.

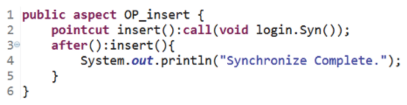


(b) ESJPM.



```java
public aspect OP_insert {
    pointcut insert():call(
        //declare joinpoint
    );
    after():insert(){
        //TODO
    }
}
```

(c) General example code.



```java
public aspect OP_insert {
    pointcut insert():call(void login.Syn());
    after():insert(){
        System.out.println("Synchronize Complete.");
    }
}
```

(d) Example code with AspectJ.

Fig. 4. Insert operator.

The ESJPM of the OP insert shown in Fig. 4 (b) provides weaving information while generating aspectual codes. The ESJPM is composed of a transition condition and two states: the "Inspection State" and the "Insert State." The Inspection State examines whether the aspect weaver encounters a join point. If it is at a join point, the aspect weaver transits to the "Insert State," which describes the advice, the type of operators, and the extra code in the entry action. In the figure we use "after" advice as an example. After the extra code is inserted, the aspect weaver returns to the "Inspection State."

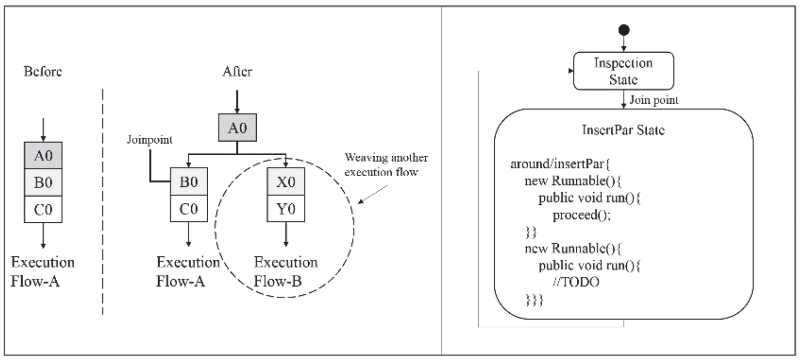
In the previous step, the ESJPM is converted into aspectual codes. As depicted in Fig. 4 (c), aspectual code of the OP insert is generated from the “Join Point” and the entry action information within the “Insert State”. The first line of code is equivalent to the declaration of a class in Java. Lines 2-7 are all directly converted from the pointcut *insert()*, the “after” advice, and “TODO” in the entry action in the “Insert State” in the ESJPM (Fig. 4 (b)). The join point declared in the pointcut *insert()* is also converted from the transition condition between the “Inspection State” and “Insert State” in Fig. 4 (c).

Fig. 4 (d) illustrates an example of OP insert use for inserting a message that reads “Synchronize Complete” when *login.Syn()* is completed.

4.2 OP InsertPar Rule

OP inserPar is used to assist developers to fork a new thread from an original thread to execute a new insert code. The two threads run in parallel without affecting each other. The OP insertPar scenario is shown in Fig. 5 (a). In the origin thread, the order of execution in flow-A is A0, B0, C0. Using OP insertPar, a new thread can be forked to execute the sequence X0, Y0 at the join point without affecting the original execution sequence B0, C0.

The OP insertPar ESJPM is shown in Fig. 5 (b) and is composed of a transition condition and two states. The “Inspection State” has the same meaning as previously, and the other is called the “InsertPar State.” In this state, “around” advice should be used because it calls *proceed()* to execute the original execution flow in AspectJ. The two *Runnable()* commands contain the codes to be executed in parallel by the two threads, one of which calls *proceed()* and the other of which is used to execute the inserted code.

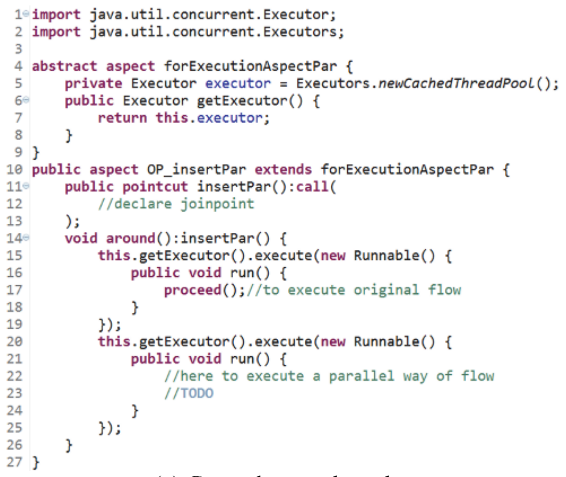


```
around/insertPar{
    new Runnable(){
        public void run(){
            proceed();
    }}
    new Runnable(){
        public void run(){
            //TODO
    }}}
```

(a) Scenario. (b) ESJPM.
Fig. 5. OP insertPar Scenario and ESJPM.

By converting the ESJPM of OP insertPar, aspectual code can be generated as shown in Fig. 6 (a) between lines 10 and 12. The code fragment depicted within the dashed line block is provided to supplement OP insertPar.

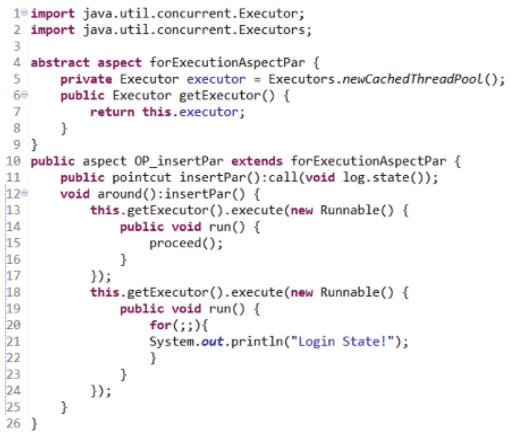
Fig. 6 (b) shows an example code using AspectJ. There are two threads in the code, starting at lines 13 and 18. The first thread executes the original method *log.state()* and the other thread displays the message “Login State!”. In this case, the message “Login State!” is shown when *log.state()* is executed.



```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

abstract aspect forExecutionAspectPar {
    private Executor executor = Executors.newCachedThreadPool();
    public Executor getExecutor() {
        return this.executor;
    }
}
public aspect OP_insertPar extends forExecutionAspectPar {
    public pointcut insertPar():call(
        //declare joinpoint
    );
    void around():insertPar() {
        this.getExecutor().execute(new Runnable() {
            public void run() {
                proceed();//to execute original flow
            }
        });
        this.getExecutor().execute(new Runnable() {
            public void run() {
                //here to execute a parallel way of flow
                //TODO
            }
        });
    }
}
```

(a) General example code.



```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

abstract aspect forExecutionAspectPar {
    private Executor executor = Executors.newCachedThreadPool();
    public Executor getExecutor() {
        return this.executor;
    }
}
public aspect OP_insertPar extends forExecutionAspectPar {
    public pointcut insertPar():call(void log.state());
    void around():insertPar() {
        this.getExecutor().execute(new Runnable() {
            public void run() {
                proceed();
            }
        });
        this.getExecutor().execute(new Runnable() {
            public void run() {
                for(;;){
                System.out.println("Login State!");
                }
            }
        });
    }
}
```

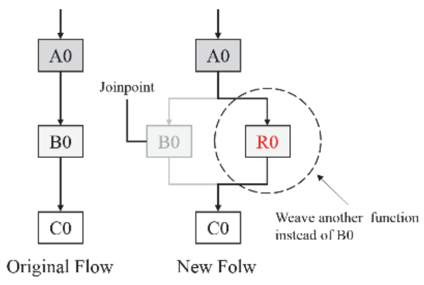
(b) Example code with AspectJ.

Fig. 6. OP insertPar example codes.

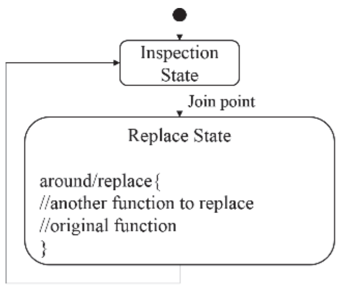
4.3 OP Replace Rule

Developers can replace an original process with a new process by using OP replace. As shown in Fig. 7 (a), the original execution flow is A0, B0, C0. Using OP replace, the aspect weaver weaves in an aspectual behavior R0 to replace behavior B0, which diverts down a new execution flow with the sequence A0, R0, C0.

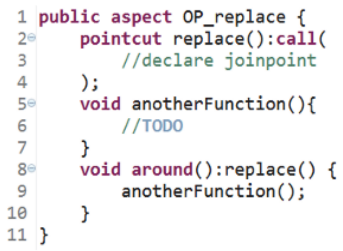
The OP replace ESJPM is shown in Fig. 7 (b). In the replace state, "around" advice is recommended because replacement is built into its semantics.



(a) Scenario.

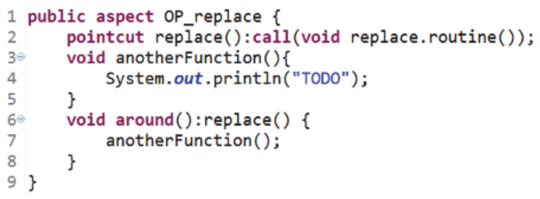


(b) ESJPM.



```
public aspect OP_replace {
    pointcut replace():call(
        //declare joinpoint
    );
    void anotherFunction(){
        //TODO
    }
    void around():replace() {
        anotherFunction();
    }
}
```

(c) General example code.



```
public aspect OP_replace {
    pointcut replace():call(void replace.routine());
    void anotherFunction(){
        System.out.println("TODO");
    }
    void around():replace() {
        anotherFunction();
    }
}
```

(d) Example code with AspectJ.

Fig. 7. Replace operator.

Fig. 7 (c) displays the general code derived from the OP replace ESJPM. The content of *anotherFunction()* replaces the original behavior coded at line 3. Developers can also code the replacing code directly in the block of *void around():replace()*.

An example code using the OP replace operator is provided in Fig. 7 (d). *AnotherFunction()* displays the message "TODO," replacing the execution of *replace.routine()*. Using "replace and modify" in "around" advice, both OP insertPar and OP replace can be implemented easily.

4.4 OP IDC (Impose Duration Constraint) Rule

The OP IDC operator is used to limit the time a specific execution process is allowed to take; otherwise, the process fails. As shown in Fig. 8, OP IDC checks the execution time of *doSomething()* against the constraint time at the checkpoint *receiveResult()* to determine whether *doSomething()* is finished in time.

There are two types of OP IDC operators: in type A, the checkpoint is set at the end of a process, which means that the process is not checked until it is finished; in type B, the process is interrupted immediately when the constraint time has run out.

Fig. 9 (a) displays OP IDC ESJPM type A, where the join point of the transition is the process that must be limited. The "IDC Before State" records the start time of the

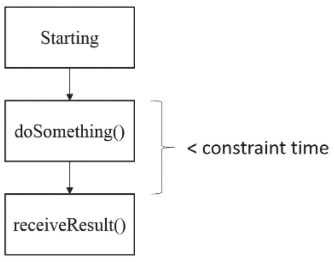
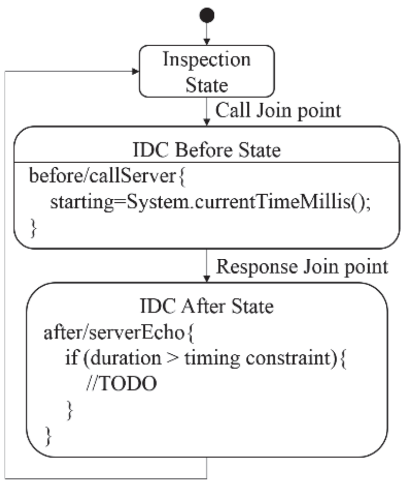
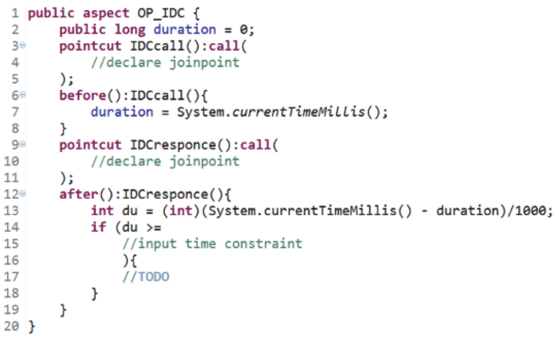


Fig. 8. OP IDC scenario.





```
public aspect OP_IDC {
    public long duration = 0;
    pointcut IDCcall():call(
        //declare joinpoint
    );
    before():IDCcall(){
        duration = System.currentTimeMillis();
    }
    pointcut IDCresponce():call(
        //declare joinpoint
    );
    after():IDCresponce(){
        int du = (int)(System.currentTimeMillis() - duration)/1000;
        if (du >=
            //input time constraint
            ){
            //TODO
        }
    }
}
```

(a) Type A ESJPM. (b) Type A general example code.

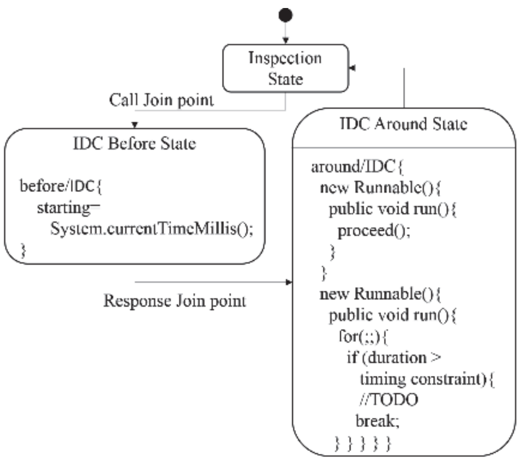
Fig. 9. Type A of OP IDC ESJPM and general example code.

process and the “IDC After State” records the time it is finished. These are then used to determine whether the process is completed within the constraint time. There are also two states for setting the constraint time in IDC ESJPM type B (Fig. 10 (a)). The “IDC Before State” is the same as that for type A, and the other state is named “IDC Around State,” which introduces multiple threads as used in OP insertPar. The first thread in the “IDC Around State” is *proceed()*, which executes the process, and the second thread uses a “for” loop to check the constraint time. When the time is up, the code represented by “TODO” is executed, and the “for” loop is terminated immediately.

A general example code of type A is shown in Fig. 9 (b). The block from lines 6 to 8 is converted from the “IDC Before State” to mark the starting time, and conversion of the “IDC After State” is listed from lines 12 to 19. The variable *duration* declared at line 2 is used to record the execution time of the process. Join points are declared at lines 4 and 10. Before the execution of a join point, *duration* records the current time, and after the execution, it is checked to determine whether the constraint time set at line 15 was exceeded.

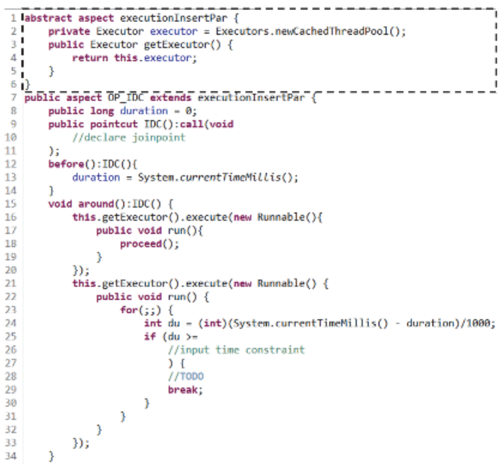
For the example code in Fig. 11 (a), a developer declares *Client.clientCallServer()* and *Server.serverResponse()* at the two pointcuts. The starting time is recorded before the execution of *Client.clientCallServer()*, and the end time is recorded after the execution of *Server.serverResponse()*. If the process time exceeds 3 seconds, the system displays the message “timeout!”.

Fig. 10 (b) shows the general example code for type B. The effect of the block between lines 12 and 14 is the same as that for type A. The portion from lines 15 to 34 is converted from “IDC Around State” in Fig. 10 (a). The constraint time is set at line 26, and the next instruction is coded in the block after line 28. The join point is declared at line 10. Before the execution of the join point, the variable duration records the current time, and then the *proceed()* method in the first thread executes the original function declared at line 10. Simultaneously, the second thread executes a “for” loop to check whether the constraint time has been exceeded.



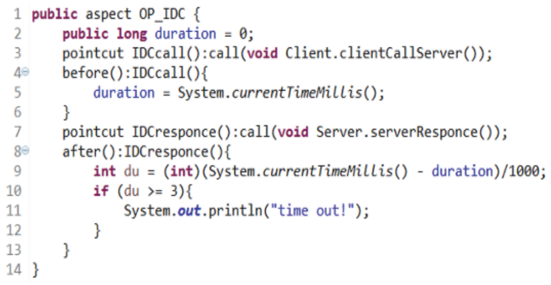
```
before/IDC{
  starting=
    System.currentTimeMillis();
}
```

```
around/IDC{
  new Runnable(){
    public void run(){
      proceed();
    }
  }
  new Runnable(){
    public void run(){
      for(;;){
        if (duration >
          timing constraint){
          //TODO
          break;
  } } } } }
```



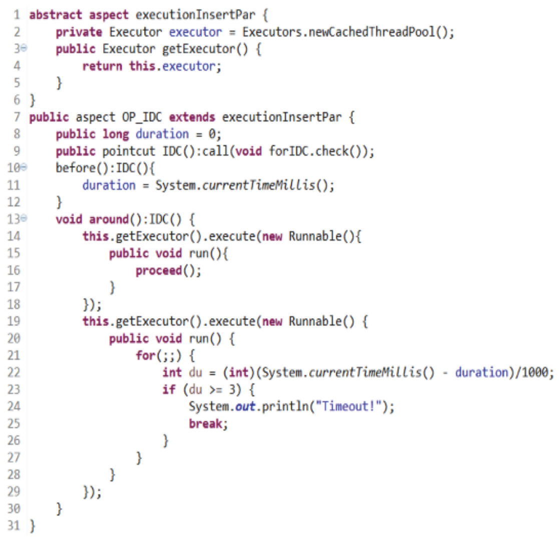
```
1  abstract aspect executionInsertPar {
2      private Executor executor = Executors.newCachedThreadPool();
3      public Executor getExecutor() {
4          return this.executor;
5      }
6  }
7  public aspect OP_IDC extends executionInsertPar {
8      public long duration = 0;
9      public pointcut IDC():call(void
10         //declare joinpoint
11     );
12     before():IDC(){
13         duration = System.currentTimeMillis();
14     }
15     void around():IDC() {
16         this.getExecutor().execute(new Runnable(){
17             public void run(){
18                 proceed();
19             }
20         });
21         this.getExecutor().execute(new Runnable() {
22             public void run() {
23                 for(;;) {
24                     int du = (int)(System.currentTimeMillis() - duration)/1000;
25                     if (du >=
26                         //input time constraint
27                         ) {
28                         //TODO
29                         break;
30                     }
31                 }
32             }
33         });
34     }
```

(a) Type B ESJPM. (b) Type B general example code.
Fig. 10. Type B of OP IDC ESJPM and general example code.



```
public aspect OP_IDC {
    public long duration = 0;
    pointcut IDCcall():call(void Client.clientCallServer());
    before():IDCcall(){
        duration = System.currentTimeMillis();
    }
    pointcut IDCresponce():call(void Server.serverResponce());
    after():IDCresponce(){
        int du = (int)(System.currentTimeMillis() - duration)/1000;
        if (du >= 3){
            System.out.println("time out!");
        }
    }
}
```

(a) Type A example code with AspectJ.



```
1  abstract aspect executionInsertPar {
2      private Executor executor = Executors.newCachedThreadPool();
3      public Executor getExecutor() {
4          return this.executor;
5      }
6  }
7  public aspect OP_IDC extends executionInsertPar {
8      public long duration = 0;
9      public pointcut IDC():call(void forIDC.check());
10     before():IDC(){
11         duration = System.currentTimeMillis();
12     }
13     void around():IDC() {
14         this.getExecutor().execute(new Runnable(){
15             public void run(){
16                 proceed();
17             }
18         });
19         this.getExecutor().execute(new Runnable() {
20             public void run() {
21                 for(;;) {
22                     int du = (int)(System.currentTimeMillis() - duration)/1000;
23                     if (du >= 3) {
24                         System.out.println("Timeout!");
25                         break;
26                     }
27                 }
28             }
29         });
30     }
31 }
```

(b) Type B example code with AspectJ.

Fig. 11. OP IDC Type A and B example code with AspectJ.

Fig. 11 (b) is an example code for type B. The developer declares *forIDC.check()* in the pointcut, and sets integer 3 at line 23 as the constraint time. When the *forIDC.check()* function has taken longer than the constraint time (3 seconds), the message "Timeout!" is printed and *forIDC.check()* is terminated.

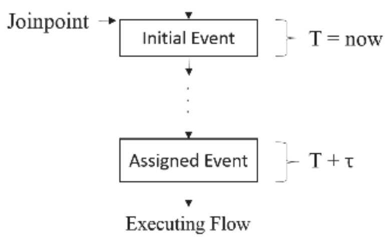
4.5 OP ITC (Impose Timing Constraint) Rule

The weaving operator OP ITC is used to constrain the time of a process, similar to OP IDC; however, in this case, it is used to execute a specified process at a preset time (Fig. 12 (a)). When encountering the initial event, a join point and a time interval are declared. The process continues to be executed until the preset time interval is reached, at which point the assigned process is executed.

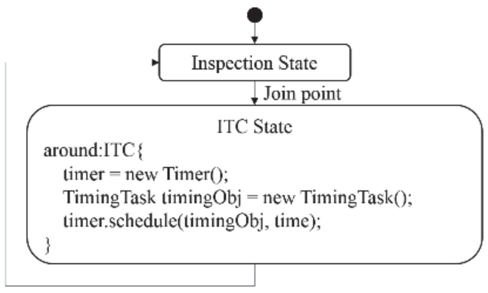
The ESJPM of OP ITC is shown in Fig. 12 (b). The preset time is defined in the "ITC State." Asides from the "ITC State," the general code requires a supplementary code fragment to be converted from the ESJPM of OP ITC.

Fig. 12 (c) depicts a general example code for OP ITC. The code fragment between lines 9 and 18 is converted from the ESJPM of OP ITC. The join point is declared at line 10, and the preset time at line 16. The code between lines 2 and the 8 is the supplementary code fragment that is executed when the preset time is up, and should be coded in the block at line 5.

Particular example code for the use of the OP ITC operator is provided in Fig. 12 (d). When *forITCtest.run()* has begun execution, the clock has started. After 5 seconds, "5 seconds after timingObj" is displayed. Because the "around" advice leads to a replacement, the original process *forITCtest()* is replaced with the code coded in the "around" advice. If the developer wants to execute the original process, we recommended that *proceed()* be used here.

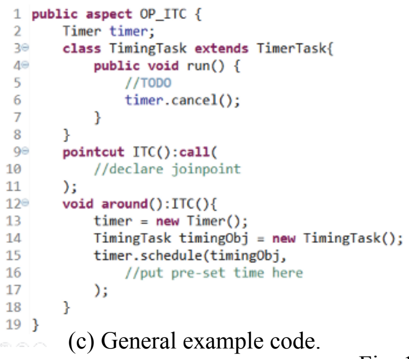


(a) Scenario.



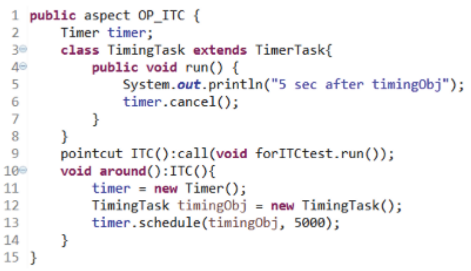
```
around:ITC{
    timer = new Timer();
    TimingTask timingObj = new TimingTask();
    timer.schedule(timingObj, time);
}
```

(b) ESJPM.



```
public aspect OP_ITC {
    Timer timer;
    class TimingTask extends TimerTask{
        public void run() {
            //TODO
            timer.cancel();
        }
    }
    pointcut ITC():call(
        //declare joinpoint
    );
    void around():ITC(){
        timer = new Timer();
        TimingTask timingObj = new TimingTask();
        timer.schedule(timingObj,
            //put pre-set time here
        );
    }
}
```

(c) General example code.



```
public aspect OP_ITC {
    Timer timer;
    class TimingTask extends TimerTask{
        public void run() {
            System.out.println("5 sec after timingObj");
            timer.cancel();
        }
    }
    pointcut ITC():call(void forITCtest.run());
    void around():ITC(){
        timer = new Timer();
        TimingTask timingObj = new TimingTask();
        timer.schedule(timingObj, 5000);
    }
}
```

(d) Example code with AspectJ.

Fig. 12. ITC operator.

4.6 OP ISI (Impose State Invariant) Rule

The OP ISI operator is used to ensure that an invariant is not violated before or after the execution of a process. For example, banking systems may have an invariant declaring that deposits must be greater than or equal to withdrawals. Fig. 13 (a) illustrates an OP ISI scenario, in which the invariant is not violated before or after the execution of a process.

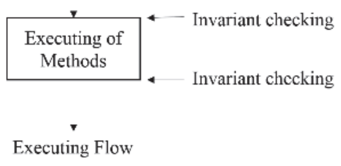
The ESJPM of OP ISI is shown in Fig. 13 (b). The "ISI State" has two advices, "before throws Exception" and "after throws Exception," which throw *Exception()* messages if the constraint conditions are not satisfied.

Fig. 13 (c) provides the general code for the OP ISI operator. The join point is declared at line 3. Lines 7 and 13 examine the conditions to determine whether the invariants are violated. If the invariants are violated, *Exception()* messages at line 9 or line 15 are thrown and can be used to assist developers in maintaining or debugging the system.

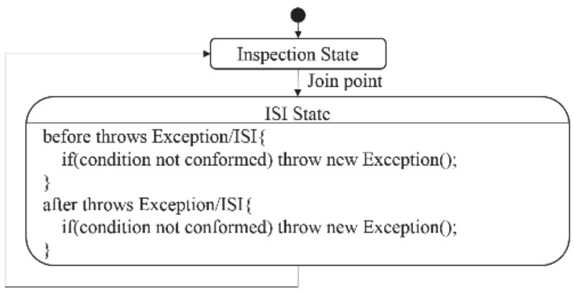
Before and after the withdrawal of money from a bank, the balance must be checked to ensure that it is greater than or equal to 0. The example code is shown in Fig. 13 (d). Before and after the execution of *Bank.withDrawMoney*, *user.getBlance()* is called to check whether the balance is less than 0, and if it is, the exception message "Error!" is displayed.

4.7 Guidelines of Applying Conversion Rules

To assist developers to apply the proposed conversion mechanism, the following steps are provided as a guideline for selecting appropriate conversion rules:

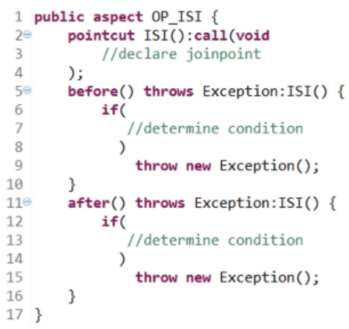


(a) Scenario.



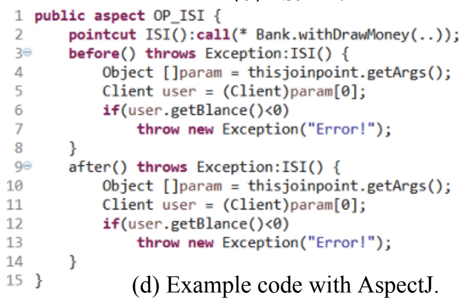
```
before throws Exception/ISI{
   if(condition not conformed) throw new Exception();
}
after throws Exception/ISI{
   if(condition not conformed) throw new Exception();
}
```

(b) ESJPM.



```
public aspect OP_ISI {
    pointcut ISI():call(void
        //declare joinpoint
    );
    before() throws Exception:ISI() {
        if(
           //determine condition
          )
            throw new Exception();
    }
    after() throws Exception:ISI() {
        if(
           //determine condition
          )
            throw new Exception();
    }
}
```

(c) General example code.



```
public aspect OP_ISI {
    pointcut ISI():call(* Bank.withDrawMoney(..));
    before() throws Exception:ISI() {
        Object []param = thisjoinpoint.getArgs();
        Client user = (Client)param[0];
        if(user.getBlance()<0)
            throw new Exception("Error!");
    }
    after() throws Exception:ISI() {
        Object []param = thisjoinpoint.getArgs();
        Client user = (Client)param[0];
        if(user.getBlance()<0)
            throw new Exception("Error!");
    }
}
```

(d) Example code with AspectJ.

Fig. 13. ISI operator.

1. Matching the scenario of the proposed operators and the system to be built. In the aspectual use case document, there is a field of "types of weaving operators", which suggests the operator to be applied. By matching the scenario of the operator proposed in the previous sections with the one in the use case document, developers can determine whether the operator in the use case document is appropriate or not.
2. Looking for the ESJPM of the aspectual sequence diagram corresponding to the use case in the previous step to find out weaving information. By examining the ESJPM, the following information can be retrieved, including join points, advices and "TO-DO" operations.
3. Transforming information in the ESJPM into aspect code. According to the general sample code given in each conversion rules, the information obtained from EJSPM can be converted into aspect code as depicted in each conversion rules example.

5. CASE STUDY: A MEETING SCHEDULER SYSTEM

To demonstrate the feasibility of the proposed approach, a meeting scheduler system [23] that has been adopted as a benchmark by Potts *et al.* [24], which illustrates typical requirements and real system problems is introduced in this work. The purpose of this system is to support the scheduling of meetings in organizations, that is, to determine, for each meeting request, a meeting date and location so that most of the intended participants can join the meeting. The meeting data the location should be as convenient as possible to all participants. The system should assist meeting initiators in re-planning a meeting dynamically to support flexibility. All participants should be allowed to modify their preference set, exclusion set or preferred location before a meeting date or location is proposed. The system should also support conflict resolution according to the resolution policies specified by the initiator. Physical constraints, such as a person should not be in two different meetings at the same time, should not be broken. Fig. 14 shows the aspect-enhanced goal-driven use case diagram of the system. The meeting scheduler system was then further analyzed and specified with aspect-enhanced sequence diagram and ESJPM.

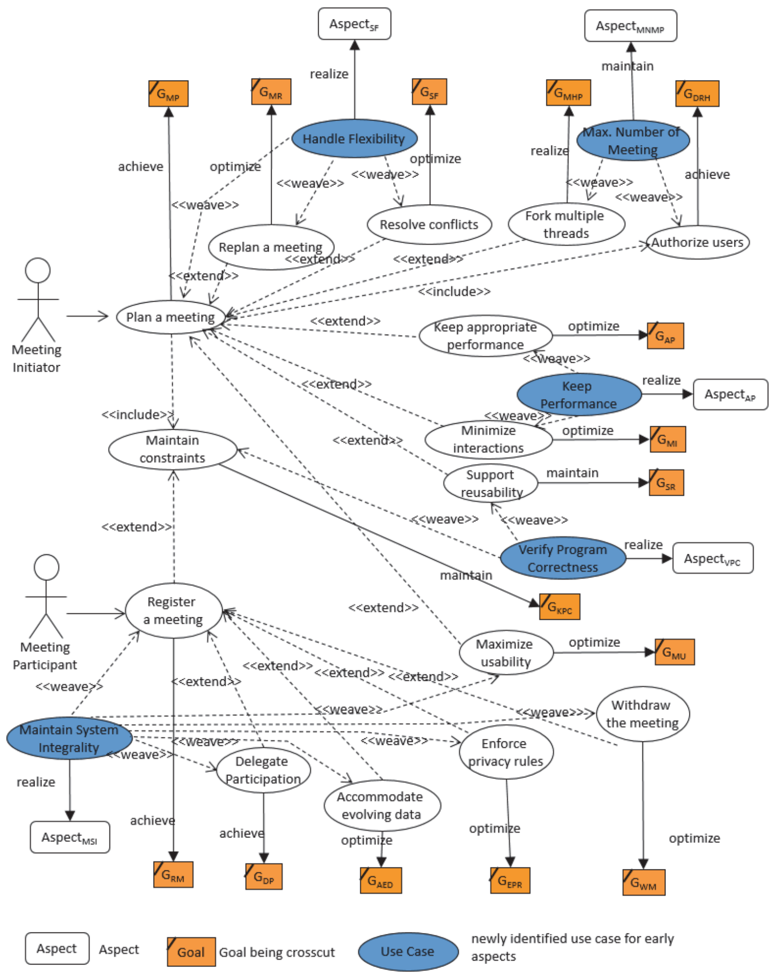
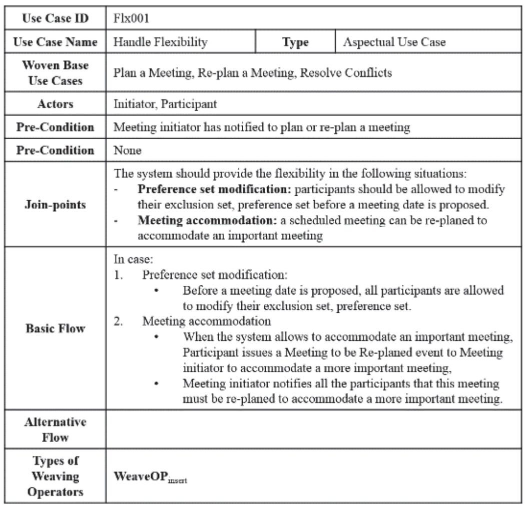


Fig. 14. Handle flexibility early aspect.

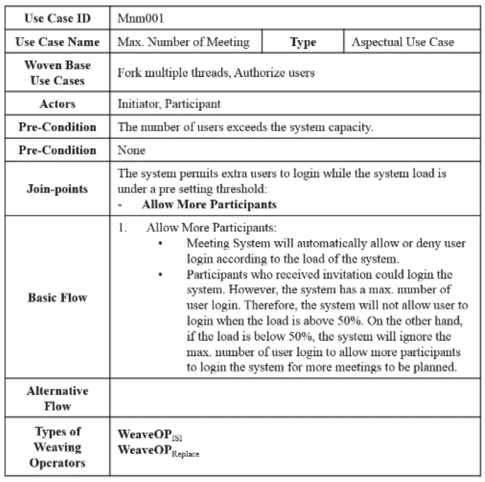
We use the handle flexibility early aspect as an example. The aspect $Aspect_{SF}$ is to be realized by the use case $UC_{HandleFlexibility}$ and crosscuts three goals, namely G_{MP}, G_{MR}, G_{SF}. G_{MP} denotes the goal to plan a meeting, G_{MR} denotes the goal to re-plan a meeting, and G_{SF} denotes the goal to support flexibility. We focus on the interaction between the aspectual use case, $UC_{HandleFlexibility}$, and base use case, $UC_{ReplanaMeeting}$, for explaining the proposed approach. Fig. 15 (a) shows the aspectual use case specification of the $UC_{Handle\text{-}Flexibility}$ aspectual use case. There are two join points specified: one is "Preference set modification" that allows all participants to modify their exclusion/preferences sets before the meeting is held, and the other is "Meeting accommodation" that allows all participants to issue a 're-plan the meeting request' for accommodating a more important meeting. The type of weaving operator to be used is *WeaveOP*$_{insert}$, which means that the aspectual behavior addressed in this specification will be inserted into the behavior of base use case.

After coding a general meeting scheduler, we proceeded as follows: (1) A handle flexibility early aspect was identified from a goal-driven use case (shown in the dashed rectangle in Fig. 14); (2) The use case specification shown in Fig. 15 was written; (3) According to the operator definition, the aspect-enhanced sequence diagram and ESJPM that fulfill the specification in the UML editor were constructed as shown in Figs. 16 and 17; (4) We exported the *.xmi file and (5) converted it into AspectJ code using the conversion tool developed for this study; (6) Finally, by adjusting the code to fit the situation, the requirements defined by the goal-driven use case specification were satisfied.



Use Case ID	Flx001		
Use Case Name	Handle Flexibility	**Type**	Aspectual Use Case
Woven Base Use Cases	Plan a Meeting, Re-plan a Meeting, Resolve Conflicts		
Actors	Initiator, Participant		
Pre-Condition	Meeting initiator has notified to plan or re-plan a meeting		
Pre-Condition	None		
Join-points	The system should provide the flexibility in the following situations: - **Preference set modification:** participants should be allowed to modify their exclusion set, preference set before a meeting date is proposed. - **Meeting accommodation:** a scheduled meeting can be re-planed to accommodate an important meeting		
Basic Flow	In case: 1. Preference set modification: • Before a meeting date is proposed, all participants are allowed to modify their exclusion set, preference set. 2. Meeting accommodation • When the system allows to accommodate an important meeting, Participant issues a Meeting to be Re-planed event to Meeting initiator to accommodate a more important meeting, • Meeting initiator notifies all the participants that this meeting must be re-planed to accommodate a more important meeting.		
Alternative Flow			
Types of Weaving Operators	**WeaveOP**$_{insert}$		

(a) Handle flexibility aspectual use case.



Use Case ID	Mnm001		
Use Case Name	Max. Number of Meeting	**Type**	Aspectual Use Case
Woven Base Use Cases	Fork multiple threads, Authorize users		
Actors	Initiator, Participant		
Pre-Condition	The number of users exceeds the system capacity.		
Pre-Condition	None		
Join-points	The system permits extra users to login while the system load is under a pre setting threshold: - **Allow More Participants**		
Basic Flow	1. Allow More Participants: • Meeting System will automatically allow or deny user login according to the load of the system. • Participants who received invitation could login the system. However, the system has a max. number of user login. Therefore, the system will not allow user to login when the load is above 50%. On the other hand, if the load is below 50%, the system will ignore the max. number of user login to allow more participants to login the system for more meetings to be planned.		
Alternative Flow			
Types of Weaving Operators	**WeaveOP**$_{ISI}$ **WeaveOP**$_{Replace}$		

(b) Max, number of meeting aspectual use case.

Fig. 15. Aspectual use case specification.

The meeting conflict scenario is defined as a situation in which the number of people present at a meeting is insufficient and some meeting attendees have other meetings scheduled that overlap with the meeting. Fig. 18 shows the current meeting schedule and the activity diagram describing the process of scheduling a meeting. If the number of possible attendees is 11, and at least half of the possible attendees can attend, then six or more people are required to successfully establish a meeting. Attendee B is invited to two simultaneous meetings; therefore, if B attends Meeting-1, there will be a conflict.

According to the basic flow of a goal-driven use case specification, two processes should be performed: (1) preference set modification and (2) meeting accommodation. Fig. 16 shows the aspect-enhanced SD for planning a meeting. In the SD, after the execution of *Meeting()*, an aspect is inserted to identify which meeting has higher priority. Focusing on the aspect in the aspect-enhanced SD, the ESJPM is depicted in Fig. 17.

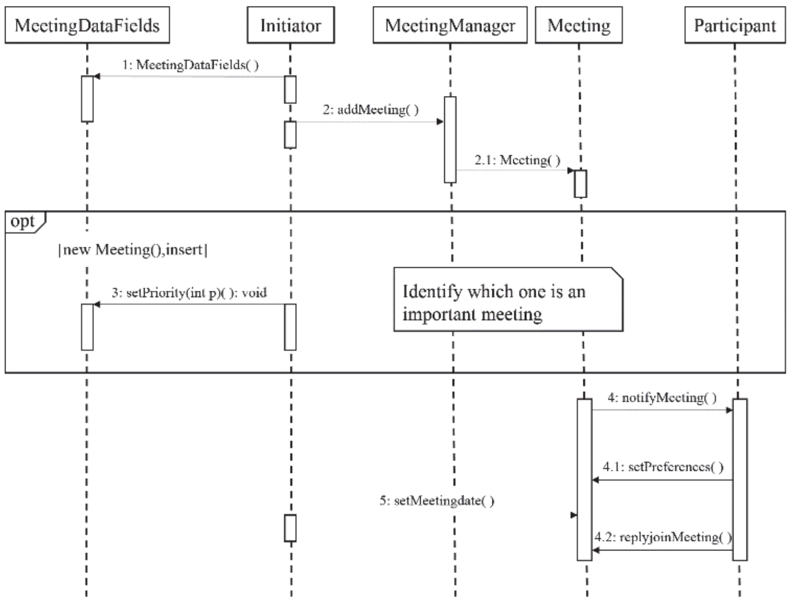
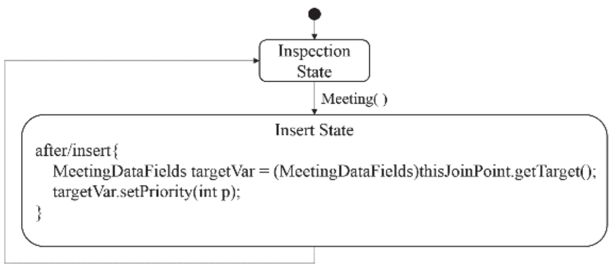


Fig. 16. Handle flexibility SD crosscut for the scenario of planning a meeting.



```
after/insert{
    MeetingDataFields targetVar = (MeetingDataFields)thisJoinPoint.getTarget();
    targetVar.setPriority(int p);
}
```

Fig. 17. ESJPM of handle flexibility aspect.

Meeting Name / Date	Meeting – 1	Meeting - 2
7/1 - 08:00-10:00	A J K	---------------
7/2 - 10:00-12:00	A **B** C D E F	**B** G H I J K
7/3 - 13:00-15:00	G H I J	C D E F J
7/3 - 15:00-17:00	A	A B

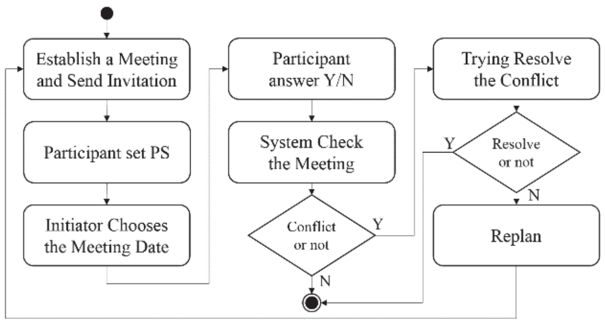
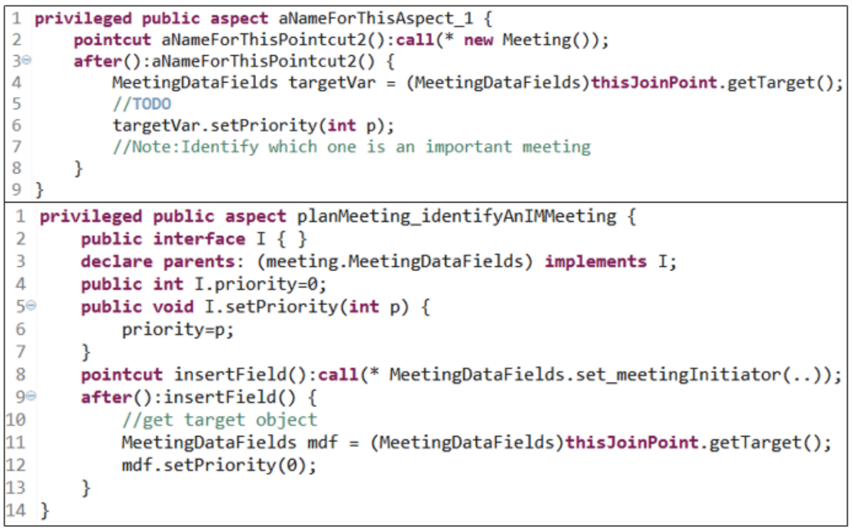


Fig. 18. Meeting schedule and activity diagram for scheduling a meeting.

Using our conversion mechanism, a code template was generated and then modified based on the AspectJ requirements (Fig. 19, top and bottom respectively).

Before the addition of the AspectJ content, the system could only inform separately the initiator of both meetings that there would be a conflict. The initiators would have to negotiate and replan the meeting.

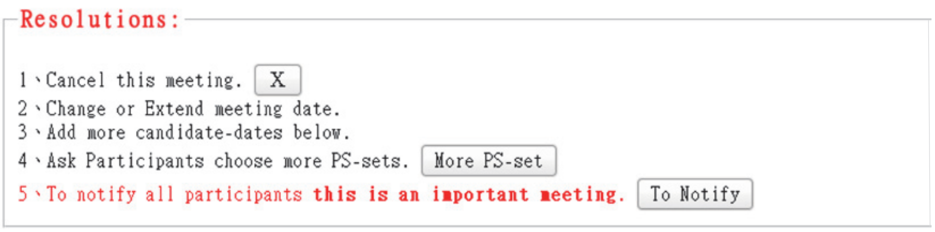
After the handle flexibility of the early aspect is weaved in, the system can assign each meeting a priority level. If there are any conflicts, participants can execute dynamic attendance preferences according to the priority settings, without being limited by constraints (shown in Fig. 20).



```
privileged public aspect aNameForThisAspect_1 {
    pointcut aNameForThisPointcut2():call(* new Meeting());
    after():aNameForThisPointcut2() {
        MeetingDataFields targetVar = (MeetingDataFields)thisJoinPoint.getTarget();
        //TODO
        targetVar.setPriority(int p);
        //Note:Identify which one is an important meeting
    }
}
```

```
privileged public aspect planMeeting_identifyAnIMMeeting {
    public interface I { }
    declare parents: (meeting.MeetingDataFields) implements I;
    public int I.priority=0;
    public void I.setPriority(int p) {
        priority=p;
    }
    pointcut insertField():call(* MeetingDataFields.set_meetingInitiator(..));
    after():insertField() {
        //get target object
        MeetingDataFields mdf = (MeetingDataFields)thisJoinPoint.getTarget();
        mdf.setPriority(0);
    }
}
```

Fig. 19. Code template and modified version of code.



Resolutions:

1、Cancel this meeting. X

2、Change or Extend meeting date.

3、Add more candidate-dates below.

4、Ask Participants choose more PS-sets. More PS-set

5、To notify all participants **this is an important meeting.** To Notify

Fig. 20. Handle flexibility crosscut: Part of the resolve conflict section.

When a possible attendee chooses whether they will attend a meeting, they can ignore the meeting constraints, attend the crucial meeting, and cancel others in order of priority (shown in Fig. 21). This achieves the preference set modification and meeting accommodation defined by the aspect requirement.

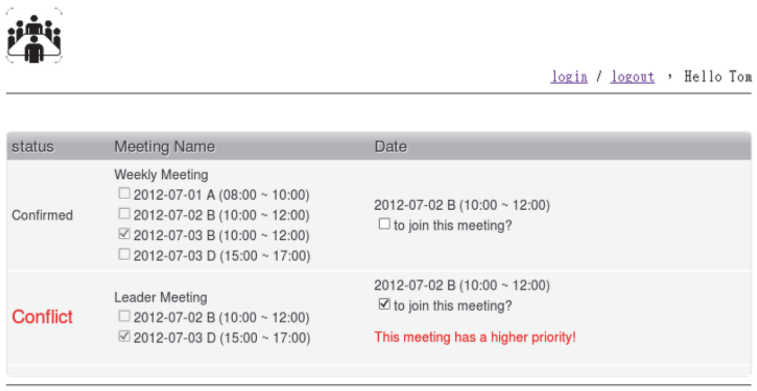
Fig. 15 (b) gives another example of the aspectual use case specification of the $UC_{Max.NumberofMeeting}$ aspectual use case with different weaving operators. In this example, one join point is specified: "Allow More Participants" that allow more users to login the system for more meetings to be planned. Two different weaving operators are used, one is $WeaveOP_{ISI}$ and the other is $WeaveOP_{Replace}$. $WeaveOP_{ISI}$ is used to guarantee that the system won't exceed the preset loading threshold while $WeaveOP_{Replace}$ allows the system to have more participants to login when the number of login users is exceeding the origin design but under the system loading threshold.

6. CONCLUSIONS

We present a mechanism for converting the ESJPM into AOP code, which enables system development personnel to focus on developing state diagram models, and through the support of tools, incorporate aspects into their code. This bridges the gap between the design and code implementation phases of the development process.

We validated our conversion mechanism by incorporating an aspect into a meeting scheduler system, showed the benefits of implementing SoC [1] modularization in the SDP.

AOSD is emerging as a crucial approach to software engineering, and provides explicit means to model critical concerns that tend to crosscut multiple system components. Early identification of concerns prevents the development of a tangled, scattered code design, which reduces the cost of implementation and enhances maintainability.


login / logout , Hello Tom

status	Meeting Name	Date
Confirmed	Weekly Meeting ☐ 2012-07-01 A (08:00 ~ 10:00) ☐ 2012-07-02 B (10:00 ~ 12:00) ☑ 2012-07-03 B (10:00 ~ 12:00) ☐ 2012-07-03 D (15:00 ~ 17:00)	2012-07-02 B (10:00 ~ 12:00) ☐ to join this meeting?
Conflict	Leader Meeting ☐ 2012-07-02 B (10:00 ~ 12:00) ☑ 2012-07-03 D (15:00 ~ 17:00)	2012-07-02 B (10:00 ~ 12:00) ☑ to join this meeting? This meeting has a higher priority!

Fig. 21. Handle flexibility crosscut: Part of the meeting replanning section.

We proposed six operator transformation rules and converted the early aspect described in the ESJPM into aspect-oriented code in AspectJ, which can combine with the base code and generates final system programs. There are three main features of the proposed mechanism:

1. Using operator transformation rules, the early aspect of the state diagrams can be converted.
2. Using the provided AspectJ transform template, both automatic and semi-automatic conversions are supported.
3. Using the supporting tools, aspects can be imported into code, bridging the gap between the design and implementation phases of the development process.

REFERENCES

1. D. Orleans and K. Lieberherr, "Dj: Dynamic adaptive programming in java," in *Proceedings of International Conference on Metalevel Architectures and Reflection*, 2001, pp. 73-80.
2. J. Araùjo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdogan, "Early aspects: The current landscape," *Technical Notes*, *CMU/SEI and Lancaster University*, 2005.
3. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of European Conference on Object-Oriented Programming*, 2001, pp. 327-354.
4. D. Zook, S. S. Huang, and Y. Smaragdakis, "Generating aspectj programs with meta-aspectj," in *Generative Programming and Component Engineering*, Springer, 2004, pp. 1-18.
5. S. Akai, S. Chiba, and M. Nishizawa, "Region pointcut for aspectj," in *Proceedings of the 8th ACM Workshop on Aspects*, *Components*, *and Patterns for Infrastructure Software*, 2009, pp. 43-48.
6. J. Lee and K. H. Hsu, "Gea: A goal-driven approach to discovering early aspects," *IEEE Transactions on Software Engineering*, Vol. 40, 2014, pp. 584-602.
7. J. Lee, C.-L. Wu, W.-T. Lee, and K.-H. Hsu, "Aspect-enhanced goal-driven sequence diagram," *International Journal of Intelligent Systems*, Vol. 25, 2010, pp. 712-732.
8. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of European Conference on Object-Oriented Programming*, 1997, pp. 220-242.
9. N. Boucké and T. Holvoet, "State-based join-points: Motivation and requirements," in *Proceedings of the 2nd Dynamic Aspects Workshop*, 2005, pp. 1-4.
10. N. M. Ali and A. Rashid, "A state-based join point model for aop," in *Proceedings of the 1st ECOOP Workshop on Views*, *Aspects and Role*, *in 19th European Conference on Object-Oriented Programming*, Citeseer, 2005.
11. D. Stein, S. Hanenberg, and R. Unland, "Designing aspect-oriented crosscutting in uml," in *Proceedings of Workshop on Aspect-Oriented Modeling with the UML*, 2002.
12. D. Stein, S. Hanenberg, and R. Unland, "On representing join points in the uml," in *Proceedings of Aspect Oriented Modeling Workshop*, 2002.
13. J. Whittle and J. Araújo, "Scenario modelling with aspects," *IEE Proceedings – Software*, Vol. 151, 2004, pp. 157-171.
14. J. Araujo, J. Whittle, and D.-K. Kim, "Modeling and composing scenario-based requirements with aspects," in *Proceedings of the 12th IEEE International Requirements Engineering Conference*, 2004, pp. 58-67.
15. R. B. France, D.-K. Kim, S. Ghosh, and E. Song, "A uml-based pattern specification technique," *IEEE Transactions on Software Engineering*, Vol. 30, 2004, pp. 193-206.
16. D.-K. Kim, R. France, S. Ghosh, and E. Song, "A uml-based metamodeling language to specify design patterns," in *Proceedings of the 2nd Workshop in Software Model Engineering*, 2003.

17. I. Groher and S. Schulze, "Generating aspect code from uml models," in *Proceedings of the 4th AOSD Modeling With UML Workshop*, 2003.
18. M. V. Hecht, E. K. Piveta, M. S. Pimenta, and R. T. Price, "Aspect-oriented code generation," *Simpsio Brasileiro de Engenharia de Software*, 2006, pp. 209-223.
19. M. A. Wehrmeister, C. E. Pereira, and F. J. Rammig, "Aspect-oriented model-driven engineering for embedded systems applied to automation systems," *IEEE Transactions on Industrial Informatics*, Vol. 9, 2013, pp. 2373-2386.
20. A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design & Test of Computers*, Vol. 18, 2001, pp. 23-33.
21. G. Booch, J. Rumbaugh, and I. Jacobson, "The unified modeling language," *Unix Review*, Vol. 14, 1996, p. 5.
22. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley Professional, UK, 2010.
23. M. S. Feather, S. Fickas, A. Finkelstein, and A. Van Lamsweerde, "Requirements and specification exemplars," *Automated Software Engineering*, Vol. 4, 1997, pp. 419-438.
24. C. Potts, K. Takahashi, and A. I. Antón, "Inquiry-based requirements analysis," *IEEE Software*, Vol. 11, 1994, p. 21.

Kuo-Hsun Hsu (徐國勛) is an Assistant Professor in the Department of Computer Science at National Taichung University of Education in Taiwan. He received his Ph.D. degree in Computer Science and Information Engineering from National Central University, Taiwan, in 2003, his B.S. in Computer and Information Science from the National Chiao-Tung University, Taiwan, in 1996. His research interests mainly focus on software engineering, requirement engineering, software architecture, service-oriented architecture, and CMMI. He is a member of the IEEE.

Chang-Yen Tsai (蔡長諺) is a graduate student in the Department of Computer Science at National Taichung University of Education in Taiwan. He received his B.S. in Computer Science from the National Taichung University of Education, Taiwan, in 2015. His research interests mainly focus on software engineering and aspect-oriented development.