Aspect-Enhanced Goal-Driven Sequence Diagram

Jonathan Lee,1,* Chia-Ling Wu,1,† Wen-Tin Lee,1,‡ Kuo-Hsun Hsu2,§
1Software Engineering Laboratory, Department of Computer Science and Information Engineering, National Central University, Jhongli, Taiwan 320
2Department of Computer and Information Science, National Taichung University, Taichung, Taiwan 403

Recently, aspect-oriented approaches have resulted in a tremendous impact on the processing of broadly scoped properties during the development of software systems. However, the weaving mechanism of these crosscutting concerns cannot be easily represented with the extant unified modeling language (UML) notation at the early stage of software development life cycle. As an attempt toward the investigation of how the crosscutting behavior takes place, we proposed, in this work, an aspect-enhanced goal-driven approach to modeling the aspectual behavior in UML state transition diagrams and sequence diagrams with the proposed interaction operators based on the aspectual weaving semantics. By introducing the proposed interaction operations in the UML combined fragment, UML sequence diagrams can be further enhanced to support the modeling of the interactions between aspectual and base behavior in the analysis and design stage of software development. To further exemplify our points, the meeting scheduler system is chosen as a vehicle to illustrate the proposed approach. © 2010 Wiley Periodicals, Inc.

1. INTRODUCTION

Recently, aspect-oriented approaches have resulted in a tremendous impact on the processing of the system properties, such as recurring properties or important stakeholders’ concerns during the design and coding phases. By addressing these properties, such as performance, reusability, reliability, scalability, and so on, in an effective way, many aspect-oriented researches have been proposed and addressed in the coding phase1−6 as well as in the design and requirements phases.6,7
In this work, we propose a goal-driven approach enhanced with early aspects as an attempt toward the analysis of software systems. The proposed approach is an extension to our previous work on goal-driven use cases (GDUC)\cite{8-10}, in which use cases are derived on the basis of the analysis of goals interactions.

By introducing early aspects, the goal-driven approach can be further enhanced to address crosscutting properties in the early stage of software development. There are three main features involved in the proposed approach shown in Figure 1:

1. To specify the responsibilities of use cases in GDUC diagram (see Figure 1, “specify the responsibilities of each use cases”).
2. To specify the aspectual behavior based on the extended state-based join point model and clarify the weaving behavior in aspect-enhanced sequence diagrams (see Figure 1, “specify the aspectual behavior in aspectual sequence diagram,” “specify the aspectual states and construct the aspectual state machine,” and “specify detail weaving behavior in extended use case specification”).
3. To compose the early aspects behavior with the base behavior to provide an overall picture of the systems behavior (see Figure 1, “construct a system-level aspect-enhanced sequence diagram”).

We choose the meeting scheduler system\cite{11} as an illustrative example throughout this paper to demonstrate the idea of aspect-enhanced goal-driven sequence diagram modeling.

In the sequel, we first describe the related work on early aspects in Section 2. The weaving of early aspects is discussed in terms of the modeling of the aspectual
behavior in sequence diagrams with interaction operators in Section 3. In Section 4, the aspectual behavior of aspectual use cases is further elaborated using the notion of extended state-based join point model (ESJPM). To demonstrate the enhancement of the sequence diagrams with early aspects, a case study is elaborated in Section 5. Finally, we conclude by outlining benefits of the proposed approach and our future research plan in Section 6.

2. RELATED WORK

The following text is extracted from a paper written in 200512 to elucidate the aims of aspect-oriented software development (AOSD).

Aspect-oriented software development is emerging as an important new approach to software engineering. It sprang forth from a rethinking of the relationship between modularization and the long-honored principle of separation of concerns. Any separation-of-concerns criterion will lead to a different partitioning.

The AOSD provides an explicit and systematic means to identify, represent, and weave concerns that tend to crosscut other concerns which maybe artifacts elements in the requirements phase or system components in the design or implementation phase. At the beginning, work in early aspects focused on the identification of these concerns among system requirements. Recent work has tried to represent the interactions between aspectual and nonaspectual behaviors. In particular, most of these efforts have concentrated on the issues of how to model aspectual behaviors as well as how to specify the weaving courses between aspectual and nonaspectual behaviors explicitly. As a result, there is an increasing attention on the research of providing a suitable representation for join points that specify where or when to compose the separated concerns with the rest13–17 and giving a concise manner to model how aspectual and nonaspectual behaviors to be composed so that they can be simulated as a whole,14,15,18–20 which are outlined below.

2.1. Join Point Model

The join point model (JPM)21 that is implemented in an aspect-oriented programming language dictates when and where the crosscutting modularization takes place. The join point represents the key concept in aspect orientation. Specifying a set of join points is a major task for aspect-oriented designers, and effectively representing join points is a critical task in any aspect-oriented approach to evaluating the interactions of aspects throughout the software development life cycle. For years, many works have sought to elucidate and apply the concept of JPM in the early stage of software development in the hope that important stakeholder’s concerns could be better addressed in the requirements analysis and design phases.

Deubler et al.13 employed the AspectJ definition and nomenclature for join points, which are well-defined, single points in the execution flow, where two concerned models are connected with each other. Join points specify when a weaving activity of a crosscutting concern should be executed. Crosscutting concerns arise under four conditions (before, after, around, and instead of a certain action).
Therefore, graphical elements are introduced into the unified modeling language (UML) sequence diagrams to specify when an aspect is invoked—before, after, wrapping, and overriding join points. A join point is represented as a lifeline with a circle head symbol to specify the interaction of aspectual and base behaviors with messages passing between the join point and system instances. The drawback of this approach is that it cannot model the join point as a concrete instance with a lifeline. Since join points are associated with different semantics, and the time ordering imposed in the lifeline is disturbed and chaotic.

Stein et al.\textsuperscript{14,15} specified fixed code-based behavioral join points in the UML sequence diagrams. Their work adhered strongly to the semantics of the AspectJ program language, and they attempted to model the execution of program at a high level of abstraction. They argued that behavioral crosscutting takes place at the link that is used to communicate the message (which is the stimulus that dispatches the action associated with the message.) By utilizing links as fixed points for behavioral crosscutting, this work will specify the join points where crosscutting occurs in the UML sequence diagrams with some stereotypes, such as execute, initialize, create, set, and set. However, the deficiency of fixed code based behavior join points is that those join points triggered not depending on specific actions but specific system state transitions are difficult to represent.

Boucke and Holvoet\textsuperscript{16} pointed out the need for high-level join points to solve the problems in capturing abstract system states that are encountered in the development of an application to control automatic guided vehicles in a warehouse management system. They argued that abstract states of software entities (a concern consists of software entities) are a promising instrument for defining join points for crosscutting concerns rather than fixed points in program execution. However, this work represents only a preliminary attempt at state-based join points modeling. It outlines a UML class diagram corresponding to a concern and the software entities, which makeup the concerns. In the initial sketch of state-based join points, they considered the use of an event-based approach to model the specified state-based join points. However, the semantics of crosscutting concerns and join points are informally presented in UML notes. The dynamic behavior of aspectual and nonaspectual behavior cannot be represented.

Noorazean Mohd Ali\textsuperscript{17} proposed a state-based join point model (SJPM) that was motivated by the deficiencies of existing fixed code-based behavioral JPMs when used to support the implementation of crosscutting concerns in systems that need constant state monitoring such as the safety-critical systems. To find ways to capture the crosscutting behaviors that are activated by some system state transitions, SJPM defines a crosscutting system state as an abstract state machine and uses the transitions of this abstract state machine to identify the join points for aspect superimposition. This work specifies three types of state-based join point in the SJPM: They are dependency based, scope based, and transition flow join points, and they provide the foundation for state-based aspect implementation. The major disadvantage of this approach is the lack of a visual means of representing the interactions between aspectual and nonaspectual behaviors. A state machine diagram cannot easily be used to represent how aspectual behaviors are imposed on these system behaviors that they crosscut.
Some findings considering the aforementioned approaches are summarized as follows: (1) In some work, join points are specified intuitively by marking the woven actions in UML sequence diagrams, which cannot be easily applied to state-sensitive systems. (2) Support for specifying the semantics of how, where, and when aspectual behaviors occur is rather limited.

2.2. Modeling Aspectual Behavior

Stein et al.\textsuperscript{14,15} proposed a design model, called the aspect-oriented design model (AODM), that extends existing UML concepts to aspect-oriented concepts, which are found in AspectJ. Aspects are represented as the UML classes of a special stereotype called aspect. Analogous to aspects in AspectJ, the “pointcut” elements and “advice” operation can be defined as operations in the aspect class with pointcut and advice stereotypes, respectively. In the AODM, crosscutting of the behavior of a program (implemented by means of advice in AspectJ) is visualized by highlighting messages in a UML sequence diagram. Behavioral crosscutting takes place at the link that is used to communicate the message. The crosscutting behavior is specified by standard UML collaborations. Weaving of behavioral crosscutting is considered to be a matter of splitting and composing collaborations. Splitting always takes place in the collaboration describing the crosscut behavior, at the link at which the crosscutting behavior is to be executed. A new collaboration, which implements the crosscutting behavior, is composed to the split collaboration before or after (or “around”) the link at which the collaboration was split. However, the work inherits the deficiency of fixed code-based behavior join points, which is mentioned in Section 2.1.

Aráujo et al.\textsuperscript{18,19} introduced the notion of aspect in scenario-based software requirement research. They focused on representing aspects during use case modeling. In particular, they focused on how to compose aspectual and nonaspectual scenarios in a way that enabled them to be simulated as a whole. They modeled nonaspectual scenarios using UML sequence diagrams. Aspectual scenarios are modeled as interaction pattern specifications (IPSs).\textsuperscript{22} Finite state machines will be modeled as UML state machines and an aspectual finite state machine will be modeled as state machine pattern specifications (SMPSs).\textsuperscript{23} The aspectual and nonaspectual state machines are then composed using a state machine synthesis algorithm. The result is a new set of state machines representing the complete specification. A state-based approach is used to merge aspectual and nonaspectual scenarios, each of which is translated into a set of state machines, which are combined to generate a complete validated state machine description of the system requirements. The notion of join points is implicit in the synthesis algorithm. How the aspectual scenarios crosscut nonaspectual scenarios is not explicitly modeled in sequence diagrams.

Georg et al.\textsuperscript{20} proposed an aspect-oriented design approach that defines an aspect using role models based on two aspect views—static and interaction views. An aspect’s static view leverages UML class diagrams to define the structural properties of the aspect. The interaction view utilizes UML collaboration diagrams to specify the interaction patterns that are associated with the aspect. Aspects are treated as design patterns and modeled by the role models. However, only instantiation for
interaction role models is considered herein. The composition of role models with nonaspectual interactions is not considered.

Deubler et al. proposed aspect-oriented modeling techniques to model interacting components that are implemented as Web services within service-oriented architectures. A service is defined by the interaction among entities that are involved in establishing a particular functionality. Hence, a service is a behavior or functionality that is provided by collaborative interworking system entities. This work introduces an explicit notation for modeling before, after, around, and instead join points within UML sequence diagrams. The drawback of this approach is that it cannot model a join point as a concrete instance with a lifeline in sequence diagrams. Because join points are associated with different semantics.

Some of the findings associated with the related work are summarized as follows: (1) hard to specify a join point at which the aspectual behavior hinges on a specific state transition and (2) limited support for specifying explicitly the interactions between aspectual and nonaspectual behaviors.

### 3. ASPECTUAL BEHAVIORS MODELING

In this work, we proposed to augment the use case specification with aspectual properties to document the responsibilities of an aspectual use case and model the aspectual behavior in an aspectual sequence diagram to make explicit the inclusion of aspects with an ESJPM for specifying weaving semantics. Early aspects are documented in a use case diagram, which is further enhanced to represent early aspects. An early aspect is denoted as a rounded rectangle, and an associated use case is created to realize the early aspect.

A weave stereotype is proposed to model how the notion of early aspects can be integrated into the use case diagram. The tenet of weave stereotype is twofold:

1. To specify that an aspectual use case, which realizes an early aspect, can crosscut many base use cases and
2. To stress that there may exist many join points inside a base use case for corresponding aspectual use cases to be woven in.

Once the enhanced GDUC diagram is obtained, specifying the dynamic behavior of early aspects becomes the main theme in the analysis of software systems. As use cases are often used to identify the dynamic behavior of a software system, it is vital to clarify the difference between how an aspect and a goal are represented in the enhanced GDUC diagram. There are two types of use cases in the diagram: One is base use case that specifies a slice of system functionalities that is visible to actors, and the other is aspectual use case that specifies a part of system functionalities that are interleaved, or crosscut in aspect-oriented research term, with other system functionalities. Consequently, it is important to specify not only how system objects interact to achieve the goal of each base use case, but also how these objects interact to implement the crosscutting concerns of each aspectual use case as well as where the effects take places with respect to the aspectual use case’s corresponding base use cases.
As a continuation of our previous research, we augment the use case specification with aspectual property to document the responsibility of an aspectual use case. Through the specification of aspectual property, aspectual behavior can be modeled within the proposed aspectual sequence diagrams together with three types of aspectual interaction operators to perform on the base use cases. An aspect weaver is also introduced to the aspectual sequence diagram to specify the weaving semantics by depicting which weaving operator to be applied.

### 3.1. Augment Use Case Specification with Aspects

Each use case includes details about what needs to be done to achieve its functionalities (also called responsibilities), which are documented and maintained in each use case specification. The basic properties of a use case specification include use case id, use case name, preconditions, postconditions, actors, a basic flow, and alternative flows.

Figure 2 shows the use case specifications augmented with aspectual property to address the responsibility of aspectual use cases, including

- **Type**: Type is used to specify what kinds of categories a use case belongs to. A use case belongs to either an aspectual use case that interleaves with base use cases or a base use case that may be affected by aspectual use cases.
• **Woven base use cases**: A set of use cases that an aspectual use case crosscuts. Unlike *extend* or *include*, an aspectual use case can crosscut more than one base use case.

• **Join points**: Join points describe when and where the corresponding aspectual behavior would weave into base use cases that an aspect crosscuts. In addition, the description of join points is specified in the aspectual use case specification instead of in a base use case specification.

• **Types of weaving operators**: There are three categories of weaving operators in the proposed approach. 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. The three categories of weaving operators are “insert behavior,” “replace behavior,” and “impose constraints” which is further elaborated as follows:

1. **Insert behavior** includes two weaving operators, \( \text{WeaveOP}_{\text{insert}} \) and \( \text{WeaveOP}_{\text{insertPar}} \). \( \text{WeaveOP}_{\text{insert}} \) operator is applied when the woven aspect owns the control of executing the base use case till the end of performing the aspectual behavior. In contrast to \( \text{WeaveOP}_{\text{insert}} \), \( \text{WeaveOP}_{\text{insertPar}} \) 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 \( \text{WeaveOP}_{\text{replace}} \). It is applicable only when aspectual behavior is used to replace the base behavior at a given join point.

3. **Impose constraints** include three weaving operators, namely, \( \text{WeaveOP}_{\text{IDC}} \), \( \text{WeaveOP}_{\text{ITC}} \), and \( \text{WeaveOP}_{\text{ISI}} \). \( \text{WeaveOP}_{\text{IDC}} \) is applied when an aspect imposes a duration constraint on the execution of base behavior. \( \text{WeaveOP}_{\text{ITC}} \) is used when an aspect imposes a timing constraint on the base behavior. And \( \text{WeaveOP}_{\text{ISI}} \) is used to introduce and maintain an invariant state described by an aspect to the base use case’s behavior.

### 3.2. Aspectual Sequence Diagram

The UML interaction diagrams are intended to model the dynamic behaviors of a system. A sequence diagram is one of the interaction diagrams, which emphasizes the time line of messages dispatched among a set of object instances. It is also the most direct and intuitive way to describe how a group of objects interact with each other. The proposed extension to UML sequence diagrams is to specify the weaving operations by introducing weaving operators performed by a weaver object instance to incorporate aspectual behavior into object instances in the base sequence diagram in three sorts of weaving ways, such as behavior insertion, modification, or constraints imposition effects. The focus of the proposed extension to the sequence diagrams 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 behavior defined in the woven sequence diagrams.

#### 3.2.1. Extend Weaving Semantics with Interaction Operators

In the UML, a sequence diagram describes a set of possible traces of message passing among objects that are bound to the interaction specification. Interactions include a number of constructs called combined fragments for representing
possible behavior, such as conditionals, loops, parallels, and so on. A combined fragment is the construct for representing a set of possible traces within an interaction that includes an interaction operator keyword and one or more interaction operands, each of which is a fragment of an interaction. The combined fragment is shown as a nested region within a sequence diagram in Figure 3, which shows two possible traces of message passing among three object instances, which are ob1, ob2, and ob3 respectively. It uses a combined fragment with an alternative interaction operator to express that one of the two interaction segments described by the two interaction operands is allowed in run-time circumstance.

An interaction operator is an enumeration designating the various kinds of operators of combined fragments. The interaction operator defines the type of operator of a combined fragment. The original literal values are alt, opt, par, loop, critical, neg, assert, strict, seq, ignore, and consider. The value of the interaction operator is given in text in a small compartment in the upper left corner of a combined fragment.

On the basis of the concept of combined fragment, we introduce interaction operators in the combined fragment to express the aspectual behavior to impose constraints on the behavior of the woven use cases. With the proposed interaction operators, we can further specify the weaving semantics by using these operators in the aspectual sequence diagrams.

The proposed interaction operators are developed to provide a one-to-one mapping to the weaving operators mentioned in Section 3.1, which are summarized as follows:

Figure 3. Combined fragment example.
• **WeaveOP\_insert** is denoted as **insert**. An **insert** fragment has a subfragment that defines an aspectual scenario to be inserted into the trace of message passing. When this **insert** fragment is reached and the guard condition of this fragment is satisfied, namely the corresponding join point condition is satisfied, the weaver will insert the aspectual behavior into the sequence of messages of the base behavior. The insertion will interrupt and take over control of execution until the insert fragment ends its execution.

• **WeaveOP\_insertPar** is denoted as **insertPar**. An **insertPar** fragment has only one subfragment that defines an aspectual scenario to be inserted into the trace of message passing. When this **insertPar** fragment is reached and the guard condition of this fragment is satisfied, the weaver will insert the aspectual behavior into the sequence of messages concurrently. The insertion will not interrupt the execution of base behavior.

• **WeaveOP\_replace** is denoted as **replace**. A **replace** fragment has two subfragments: One defines a piece of base scenario to be replaced, and the other defines an aspectual scenario. When **replace** fragment is reached and the guard condition of this fragment is satisfied, the weaver will replace the base behavior specified in the first subfragment with the aspectual behavior described in the second subfragment.

• **WeaveOP\_idc** is denoted as **idc**. An **idc** fragment has two subfragments: One defines a piece of base scenario to be imposed on a duration constraint, and the other defines the scenario that the duration constraint has been affected. When **idc** fragment is reached and the guard condition of this fragment is satisfied, the weaver will replace the base behavior specified in the first subfragment with the second subfragment.

• **WeaveOP\_itc** is denoted as **itc**. An **itc** fragment has two subfragments: One defines a piece of base scenario to be imposed on a timing constraint, and the other defines the scenario that the timing constraint has been influenced. When **itc** fragment is reached and the guard condition of this fragment is satisfied, the weaver will replace the base behavior specified in the first subfragment with the second subfragment.

• **WeaveOP\_isi** is denoted as **isi**. An **isi** fragment has two subfragments: One defines a piece of base scenario to be imposed on a state invariance constraint, and the other defines the scenario upon which the state invariance constraint has been impacted. When **isi** fragment is reached and the guard condition of this fragment is satisfied, the weaver will replace the base behavior specified in the first subfragment with the second subfragment.

### 3.2.2. Model Weaving Behavior

Figure 4 provides a conceptual view of the proposed aspect weaving mechanism to facilitate the merge of aspectual behavior into base use cases’ behavior. There are three roles involved in this sequence diagram: **GoalImp**, **Weaver**, and **AspectImp**. The **GoalImp** role represents a set of objects in a base use case. These objects cooperate with each other to achieve the goal associated with **GoalImp**. The **AspectImp** role represents a set of objects that are meant to be woven into the base use case to address the crosscutting concerns modeled in **AspectImp**. The weaving activities are conducted by executing the trace of the message flows of the **Weaver**. The activities carried out by **Weaver** are as follows:

1. The **Weaver** sends an inquiry to **AspectImp** to obtain information needed for performing weaving operation, including join points, aspectual behavior, and the types of weaving operations.
2. The **Weaver** performs a series of inspections on the states of objects associated with base use cases to locate appropriate weaving points. The notion of appropriateness is meant to describe a situation that the states of objects match the conditions described in the join point acquired from **AspectImp**.

*International Journal of Intelligent Systems*  DOI 10.1002/int
3. Once the condition is matched, interaction operators, such as insert, replace, and impose constraint, can be applied to weave the aspectual behavior into the sequence diagrams of base use cases.

The *Weaver* plays a crucial role in the sequence diagram by representing the concept of aspect weaving with the three types of interaction operators with respect to their corresponding aspects to be woven. The weaving operators can be explicitly specified as a message delivered to *GoalImp*.

The sequence diagram in Figure 5 indicates a specific weaving scenario, the high-level aspectual sequence diagram of *HandleFlexibility* use case, which contains three lifelines, such as *replanAMeeting*, *handleFlexibility*, and *weaver*. The sequence diagram illustrates that there are two join points where aspectual behavior could be woven and the employed weaving operator is *WeaveOP*\textsubscript{insert}, which means the weaving operation will take over the control flow of the execution of replanning a meeting behavior until the end of performing the aspectual behavior.

### 4. EXTENDED STATE-BASED JOIN POINT MODEL

In AOP\textsuperscript{21} the key concept: 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 represent join points and their corresponding behaviors with a suitable notation is the main focus of aspect modeling. Noorazeeen Mohd Ali \cite{Ali17} proposed an SJPM that supports, from systems’ dynamic behavior point of view, the implementation of crosscutting concerns in systems that are needed to be monitored constantly, such as a safety-critical system, which complements the deficiency of existing fixed code-based behavioral JPM. The SJPM offers a high-level conceptual perspective to define and capture concerns of a software system and assists analysts to represent nonfix code-based aspectual behavior in a systematic way. An extension to the previous work, \cite{Ali17} called ESJPM, is proposed to assist the capturing and representing the dynamic behavior of aspectual use cases for aspect weaving in the early stage of software development, which is devised with the following three features:

- attaching state machines to aspectual use cases to represent the aspectual behavior and join points,
- enriching the weaving semantics by introducing weaving operators into action expressions in each state of early aspects, and
- augmenting the possible weaving location, called aspect interception in SJPM, to state entry point, inside state, and state exit point, to better address the inclusion of aspectual behavior into the software system.

\textit{International Journal of Intelligent Systems} DOI 10.1002/int
4.1. Attach State Machines to Aspectual Use Cases

To address the crosscutting behavior of early aspects in a software system using a state-based approach, state transition diagrams are adopted as a vehicle to represent the aspectual behavior of an aspectual use case identified in Section 3 by facilitating the state transitions as weaving conditions. A state machine is attached to an aspectual use case and specifies the behavior of the aspectual use case. An aspectual use case is a coherent unit that addresses crosscutting concerns provided by a classifier, which may be a subsystem, or a class. The use of an aspectual use case defines the concept of crosscutting of a classifier, which executes on behalf of a group of cooperative object instances without revealing the internal structure of the classifier itself. The behavior of an aspectual use case can be specified in three ways: (1) by an interaction describing communication sequences of objects, (2) by an activity specification, or (3) by an attached state machine. In our approach, the behavior of an aspectual use case is modeled by attaching a state machine to each aspectual use case and specifying the weaving behavior in a sequence diagram. By doing so, we can depict how an early aspect reacts to system state’s transition and elaborates the interactions between base and aspectual behavior with an aspectual sequence diagram from the system point of view.

The use of the state transition diagrams in Figure 6 is elaborated as follows: Weaving points are represented as events with/ or without guard conditions followed by different states and aspectual behaviors that are treated as actions/ or activities in each state. A weaving point stands for a place that an aspect participates in a system. However, a criterion is needed for determining whether it is suitable to include the aspect. Therefore, an attempt is made to leverage semantics of the event-transition in the state transition diagrams to represent the criterion for the aspect weaving mechanism. The event with/ or without a guard condition is used as a criterion for matching the weaving condition whether the aspect should be woven into the system. The actions/ or activities in the following state with respect to the event are

---

In UML, a classifier is a model element that describes behavioral and structural features, such as class, subsystem, collaboration and use case.

*International Journal of Intelligent Systems*  DOI 10.1002/int
tasks an aspect is intended to perform while weaving into the system. It is defined as aspectual behaviors to embody the crosscutting concerns represented by aspects. In state transition diagrams, we use the action/ or activity for the execution of these crosscutting concerns that may occur while entering, leaving, or during a state.

For example, Figure 7 shows a partial state transition diagram of the weaver for the meeting scheduler system. The state transits from “inspection” to “allow to set preferences” while the event of “replan” occurs. The “insert” weaving operator can be applied to the object that being inspected at the time that the state has been entered. Aspectual behavior described in the state is then being woven into the base object. If there is no more event, it will return to “inspection” state for weaving other aspects when appropriate events arrive.

4.2. State-Based Join Point Model

ESJPM uses the aspectual state transitions, state entries, states duration, and state exits of the aspectual state machine to identify the join points during the system execution. Using state guard conditions as a controlling mechanism, the state of the aspect changes when the guard condition is satisfied, which helps to locate the join point where aspectual behaviors should be superimposed on the woven behavior. The superimposed activities can be specified in four places, namely entry activities, exit activities, ongoing do activities, and activities attached to a state transition diagram, in the aspectual state machine. An aspectual activity that are partially realized the aspect may be attached to a state transition diagram. When the trigger of the transition is satisfied, the transition is fired and the aspectual activity attached on the transition is performed, then the source state is deactivated and the target state is activated. An ongoing do activity may be associated with an aspectual state. The do activity is executed as long as the aspectual state is active. Alternately, ongoing
activity may be modeled by a pair of actions, an entry activity that starts the do activity on entry to the state, and an exit activity that terminates the do activity on the exit point of the state.

5. MEETING SCHEDULER SYSTEM: AN EXAMPLE

To demonstrate the feasibility of the proposed approach, the meeting scheduler system\textsuperscript{11} that has been adopted as a benchmark by Potts et al.,\textsuperscript{27} which illustrates typical requirements and real system problems is being analyzed and specified with aspect-enhanced goal-driven sequence diagrams to show how the proposed approach helps to address the afore-mentioned issues on aspect weaving in AOSD.

In the meeting scheduler example, the purpose of this system is to support the scheduling of organization meetings, that is, to determine, for each meeting request, a meeting date and location so that most of the intended participants can participate the meeting. The meeting date and location should be as convenient as possible to all participants. The system should assist meeting initiators in replanning a meeting dynamically to support flexibility. For example, participants should be allowed to modify their exclusion set, preference set, or preferred location before a meeting date or location is proposed. The system should also support conflict resolution according to resolution policies stated by the meeting initiator. Physical constraints should not be broken, for example, a person should not be in two different meetings at the same time.

After analyzing the meeting scheduler system, 15 goals are identified among which six early aspects are discovered, which gives an aspect-enhanced GDUC diagram as shown in Figure 8. An early aspect is represented as a rounded rectangle and its associated use case is identified to realize the early aspect. A textual notation for representing an early aspect and its associated goals (i.e., goals being crosscut by the early aspect) is specified as the following:

\[
\text{Aspect}_a : (UC_u, [G_1, G_2, \ldots, G_n])
\]

where \text{Aspect}_a denotes the name of the early aspect; \text{UC}_u denotes the use case that realizes the early aspect with recurring properties or important stakeholders’ concerns; and \([G_1, G_2, \ldots, G_n]\) denotes the goals that are crosscut by the early aspect, which could be more than two goals in the square bracket.

For example,

\[
\text{Aspect}_{SF} : (UC_{HandleFlexibility}, [G_{MP}, G_{MR}, G_{SF}])
\]

means that the aspect \text{Aspect}_{SF} is to be realized by the use case \text{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 replan a meeting, and \(G_{SF}\) denotes the goal to support flexibility.
We focus on the modeling of the interaction between the aspectual use case, $UC_{HandleFlexibility}$, and base use case, $UC_{ReplanaMeeting}$, for explaining the proposed approach.

To document and specify the responsibilities of an aspectual use case, the extend use case specification is used to capture the characteristics of the aspectual
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
Precondition | Meeting initiator has notified to plan or re-plan a meeting
Postcondition | 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-planned 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 by.
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

Figure 9. Use case specification of handle flexibility aspectual use case.

use case. Accordingly, the flexibility handling aspectual use case is documented as in Figure 9. The type of the use case is aspectual. Woven base use cases are UC_PlanAMeeting, UC_ReplanAMeeting, and UC_ResolveConflicts. There are two join points specified: One is “preference set modification,” and the other is “meeting accommodation.” “Preference set modification” join point specifies a weaving point that allows all participants to modify their exclusion/ or preferences sets before the meeting is held. “Meeting accommodation” join point specifies a weaving point that allows all participants to issue a “replan 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 cases, where the weaving conditions are satisfied.

Once theaspectual use case specification is documented, we can proceed to model the aspectual behavior in the aspectual sequence diagram with interaction operators to be woven into the base use cases. In this stage, an aspect weaver is
introduced to the aspectual sequence diagram to specify the weaving semantics by depicting which weaving operator to be applied.

The aspect weaver can be realized by a weaving utility facilitating aspectual behavior weaving. Referring to Figure 5 again, the three roles involved in this sequence diagram are replanAMeeting, weaver, and handleflexibility. The replanAMeeting role represents a set of objects in the “replan a meeting” base use case. These objects cooperate with each other to achieve the goal $G_{MR}$ associated with replanning a meeting. The handleflexibility role represents a set of objects that are meant to be woven into the base use case to address the flexibility handling concerns. The weaving activities are conducted by executing the trace of the message flow of the weaver as follows:

1. The weaver sends an inquiry to handleflexibility to obtain information needed for performing weaving operation, including join points, aspectual behavior.
2. The weaver performs a series of inspections on the states of objects associated with the base use case to locate appropriate weaving points. The notion of appropriateness is meant to describe a situation that the states of objects match the conditions described in the join point acquired from handleflexibility.
3. Once the condition is satisfied, the insert weaving operator is applied to weave the aspectual behavior into the sequence diagrams of the base use cases.

A state transition diagram of the flexibility handling is shown in Figure 7. When the event of “replan” occurs, the state transits from “inspection” to “allow to set preferences.” The weaving operator “insert,” in this case, is applied to the objects being inspected when the state is entered. Aspectual behavior described in the state is then being woven into the base object. Furthermore, if another event, say “notifyTheScheduledMeeting,” occurs, the state will transit from “allow to set preferences” to “allow to accommodate an important meeting” for weaving additional concerns to the base objects. Otherwise, it will return to “inspection” state for weaving other aspects when appropriate events arrive.

The sequence diagram of the base use case $UC_{ReplanaMeeting}$ is shown in Figure 10. To weave the aspect $Aspect_{SF}$ represented in sequence diagrams (see Figure 5) and state machine (see Figure 7), the proposed interaction operators, insert$_{par}$ and insert, are applied and realized by the weaver. The weaver performs a concurrent insertion activity to recursively inspect the state of the meeting object instance. When the trigger condition, caused by a state transition, of a join point is satisfied, the aspectual behavior of the aspect is woven into the base behavior by the weaver in Figure 11.

6. CONCLUDING REMARKS

Aspect-oriented software development is emerging as an important approach to software engineering. It provides explicit means to model important stakeholders’ concerns that tend to crosscut multiple system components. The early identification of concerns helps develop untangled and nonscattered design and code, which reduces the cost of implementation and enhances maintainability.
In this work, we proposed an aspect-enhanced goal-driven approach to the modeling of the interactions between aspectual behavior and the base behavior in UML state transition diagrams and sequence diagrams as an attempt toward the investigation of how the crosscutting behavior takes place, in which three main features are devised:

1. specifying the responsibility with the augment use cases specification with early aspects,
2. describing the aspectual behavior based on the extended state-based join point model and clarifying the weaving behavior in aspect-enhanced sequence diagrams, and
3. constructing an aspect-enhanced sequence diagram by composing early aspect behavior with the base behavior to provide an overall picture of the system behavior.

Our future research plan will focus on the investigation of the issues of aspect-oriented programming language implementation to support the state-based and run-time weaving mechanism.
Figure 11. Aspect-enhanced sequence diagram of replan a meeting use case.

References


International Journal of Intelligent Systems DOI 10.1002/int