Chapter 6
Security Aspect Specification

As mentioned in the introduction of this book, security should be addressed during the early phases of the software development life cycle. From the state-of-the-art survey presented in Chap. 5, we have concluded that AOM is the most appropriate approach to achieve this objective. In this context, we propose, in this chapter, an AOM approach for specifying and systematically integrating security solutions into UML design models, and therefore enabling secure code generation. The targeted security concerns are those high-level requirements that are usually specified and verified on software, and for which a security solution can be provided as an aspect. Examples of such requirements are: confidentiality, integrity, authentication, authorization, access control, etc. In the proposed approach, the security expert specifies the needed security solutions as application-independent aspects. In addition, he/she specifies how these aspects should be integrated into the design models. The developer then specializes the application-independent aspects to his/her design. Finally, our framework automatically injects the application-dependent aspects at the appropriate locations in the design models.

In this chapter, we focus on the specification of security aspects. To this end, we devise a UML profile that assists security experts in specifying security solutions as aspects. The proposed profile covers the main UML diagrams that are used in software design, i.e., class diagrams, state machine diagrams, sequence diagrams, and activity diagrams. In addition, it covers most common AOP adaptations, i.e., adding new elements *before*, *after*, or *around* specific points, and removing existing elements. Moreover, we define a high-level and user-friendly pointcut language to designate the locations where aspect adaptations should be injected into base models.

The remainder of this chapter is organized as follows. Section 6.1 summarizes our approach for specifying and weaving aspects into UML design models. Afterwards, we present our AOM profile in Sect. 6.2. The related work on AOM is given in Sect. 6.3. Finally, Sect. 6.4 concludes this chapter.

D. Mouheb et al., *Aspect-Oriented Security Hardening of UML Design Models*,
DOI 10.1007/978-3-319-16106-8_6

6.1 Proposed AOM Approach for Security Hardening

In this section, we present an overview of our proposed AOM approach for security hardening of software. The proposed approach assists security experts in designing security solutions in a precise way without altering the software functionalities. In addition, the proposed approach allows developers with limited security knowledge to reuse those solutions with minimal intervention. The approach architecture is depicted in Fig. 6.1. The main steps of the proposed approach are the following:

- *Security Aspect Specification*: A security expert designs security solutions as application-independent aspects. By analogy, these aspects are generic templates representing the security features independently of the application specificities and presented in a security aspects library. This design decision is useful in order to support reusability of aspects in different application domains. Since there is no standard language to specify aspects in UML, a UML profile is developed as part of our framework in order to assist security experts in designing security aspects. This profile is designed to allow as many modification capabilities as possible. These capabilities include the common modification capabilities characterizing the most prominent AOP languages (AspectJ [113] and AspectC++ [189]). As part of this UML profile, we have developed a high-level language to specify the pointcuts that designate the locations in the base model where the aspect adaptations should be performed. The details about the design of this profile are provided in Sect. 6.2.

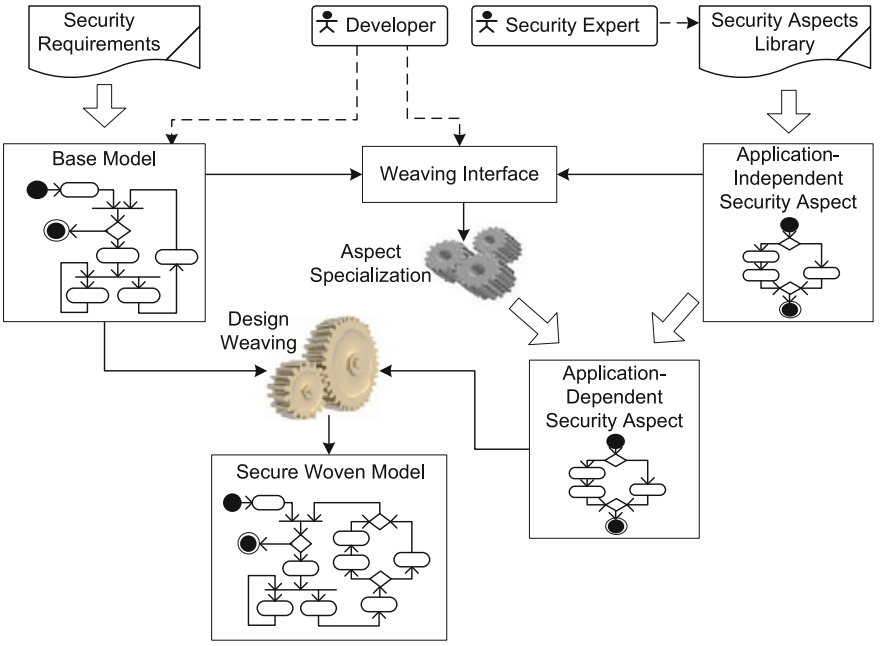

Fig. 6.1 Specification and weaving of UML security aspects

- *Security Aspect Specialization*: The developer has the possibility to specialize the application-independent aspects provided by the security expert according to the application-dependent security requirements and needs. To specialize the aspects, we provide a weaving interface, in which only the generic pointcuts are exposed to the developers. By doing so, the complexity of the security solutions is kept hidden from the developers. More details about security aspects specialization are presented in Sect. 7.2.
- *Join Point Matching*: A security aspect mainly consists of a set of adaptations that should be performed at some specific points (called join points in AOP) of UML design. Based on the pointcuts specified in the aspect by the security expert and specialized by the developer, our framework identifies, without any developer interaction, the join points from the base model where the aspect adaptations should be performed. More details about join point matching are presented in Sect. 7.3.
- *Security Aspect Weaving*: This represents the automatic injection of the security solutions into the design models at the identified join points. To provide a portable solution, we adopt a model-to-model transformation language; the QVT language [150]. QVT is an OMG standard compatible with UML and supports a large set of modifications on UML models. For each aspect adaptation and the corresponding base model elements, a set of QVT transformation rules are generated. The details about the aspect weaving step are provided in Sect. 7.4.

This chapter focuses on describing the security aspect specification step. The remaining steps of our security hardening approach, i.e., security aspect specialization, join point matching, and security aspect weaving are detailed in Chap. 7.

6.2 A UML Profile for Aspect-Oriented Modeling

This section presents our AOM profile that extends UML for security aspects specification. An aspect represents a non-functional requirement. It contains a set of adaptations and pointcuts. An adaptation specifies the modification that an aspect performs on the base model. A pointcut specifies the locations in the base model where an adaptation should be performed. The elements of this profile will be used by security experts to specify security solutions for well-known security problems. However, the profile is generic enough to be used for specifying non-security aspects. In our AOM profile, an aspect is represented as a stereotyped package (Fig. 6.2). For example, Fig. 6.3 shows a partial specification of an aspect designed to enforce RBAC mechanisms.[1] The RBAC aspect is modeled as a package stereotyped ≪*aspect*≫. In the following subsections, we show how adaptations and pointcuts can be specified using our AOM profile.

[1] The full specification of the RBAC aspect is presented in Sect. 7.6.1.2.

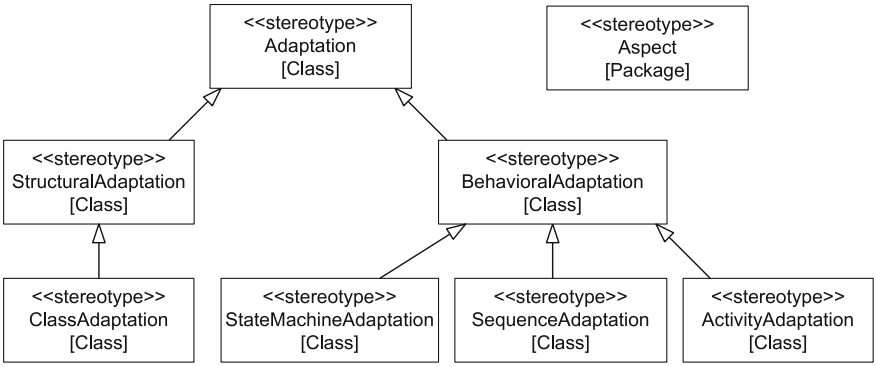

Fig. 6.2 Meta-model for specifying aspects and their adaptations

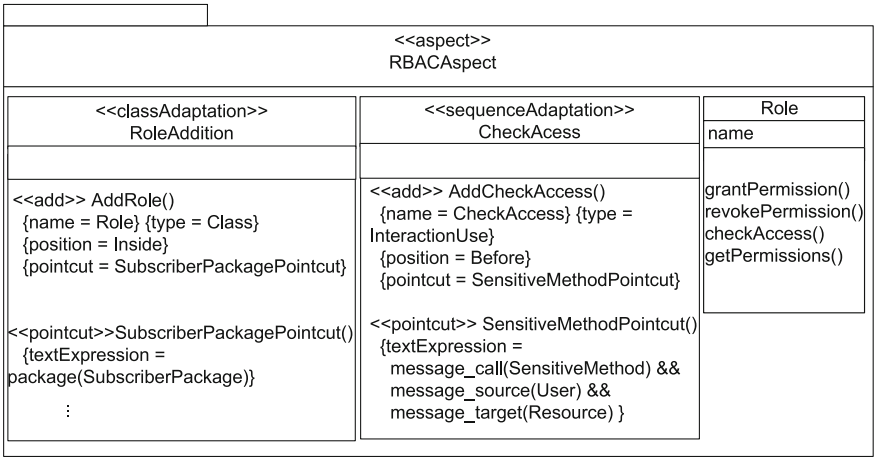

Fig. 6.3 Partial view of the RBAC aspect

6.2.1 Aspect Adaptations

As mentioned earlier, an adaptation specifies the modification that an aspect performs on the base model. We classify adaptations according to the covered diagrams and the modification rules that specify the effect of adaptations on the base model. UML allows the specification of a software from multiple points of view using different types of diagrams, such as, class diagrams, activity diagrams, sequence diagrams, etc. Unfortunately, most of existing AOM approaches specify aspects within the same modeling view (e.g., structural, behavioral). In this research, we propose an AOM approach that covers both structural and behavioral views of a system. Notice that this does not mean that we cover all existing UML diagrams. Instead, we focus on those diagrams that are the most used by developers: class diagrams, sequence diagrams,

state machine diagrams, and activity diagrams. Figure 6.2 presents our specification of adaptations. We define two types of adaptations: structural and behavioral adaptations.

6.2.1.1 Structural Adaptations

Structural adaptations specify the modifications that affect structural diagrams. We focus on class diagrams since they are the most used structural diagrams in software design. A class diagram adaptation is similar to an introduction in AOP languages (e.g., AspectJ). A structural adaptation is modeled as an abstract meta-element named *StructuralAdaptation*. It is specialized by the meta-element *ClassAdaptation* used to specify class diagram adaptations, which contain adaptation rules for class diagram elements (see Sect. 6.2.2). Notice that the meta-element *StructuralAdaptation* can be specialized to model adaptations for other structural diagrams, such as, component diagrams, deployment diagrams, etc. As an example of a structural adaptation, *RoleAddition* in Fig. 6.3 is a class adaptation (stereotype ≪*ClassAdaptation*≫) used for the integration of a class named *Role* into a package, designated by the pointcut *SubscriberPackagePointcut*, as well as the adaptation rules that are required to the adoption of an RBAC solution. The definition and the specification of adaptation rules will be presented later in this section.

6.2.1.2 Behavioral Adaptations

Behavioral adaptations specify the modifications that affect behavioral diagrams. In our approach, we support the behavioral diagrams that are the most used for the specification of a system behavior, mainly, state machine diagrams, sequence diagrams, and activity diagrams. A behavioral adaptation is similar to an advice in AOP languages (e.g., AspectJ). A behavioral adaptation is modeled as an abstract meta-element named *BehavioralAdaptation*. We specialize the meta-element *BehavioralAdaptation* by three meta-elements: *StateMachineAdaptation*, *SequenceAdaptation*, and *ActivityAdaptation* that are used to specify adaptations for state machine diagrams, sequence diagrams, and activity diagrams respectively. As for the meta-element *StructuralAdaptation*, the meta-element *BehavioralAdaptation* can also be extended to model adaptations for other behavioral diagrams, such as, communication diagrams, interaction overview diagrams, etc. As an example of a behavioral adaptation, *CheckAccess* in Fig. 6.3 is a sequence adaptation (stereotype ≪*SequenceAdaptation*≫) defining the adaptation rules required to inject the behavior needed to check user permissions before any call to a sensitive method.

6.2.2 Aspect Adaptation Rules

An adaptation rule specifies the effect that an aspect performs on the base model elements. We support two types of adaptation rules: *Adding* a new element to the base model and *removing* an existing element from the base model. Figure 6.4 depicts our specified meta-model for adaptation rules.

6.2.2.1 Adding a New Element

The addition of a new diagram element to the base model is modeled as a special kind of operation, to which a stereotype ≪*Add*≫ is applied. We use the same specification for adding any kind of UML element, either structural or behavioral. Three tagged values are attached to the stereotype ≪*Add*≫:

- *Name*: The name of the element to be added to the base model.
- *Type*: The type of the element to be added to the base model. The values of this tag are provided in the enumerations *ClassElementType*, *StateMachineElementType*, *SequenceElementType*, and *ActivityElementType*.
- *Position*: The position where the new element needs to be added. The values of this tag are given in the enumeration *PositionType*. This tag is needed for some

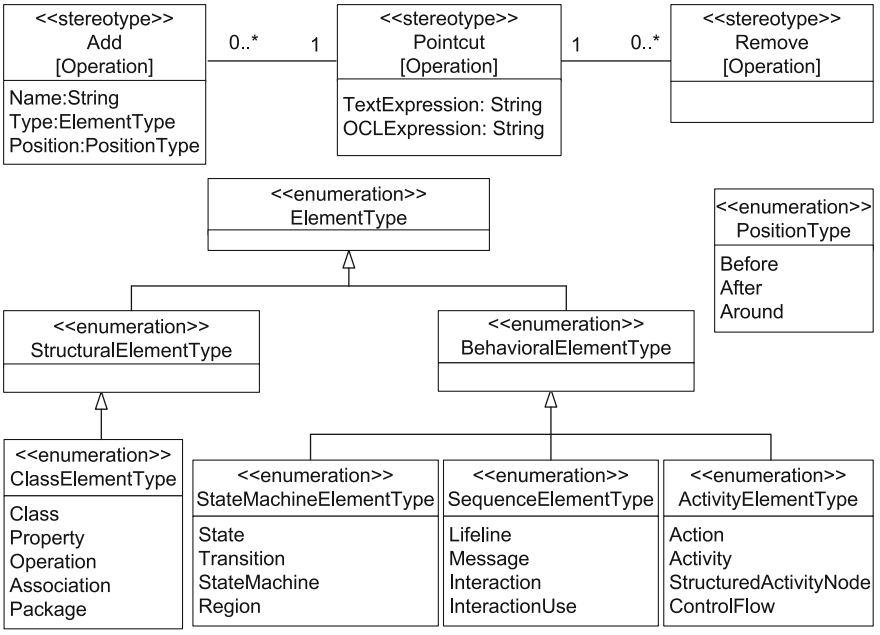

Fig. 6.4 Meta-model for specifying adaptation rules

elements (e.g., a message, an action) to state where exactly the new element should be added (e.g., *before/after* a join point). For some other elements (e.g., a class, an operation), this tag is optional since these kinds of elements are always added inside a join point.

The location where the new element should be added is specified by the meta-element *Pointcut* (see Sect. 6.2.3). For example, in Fig. 6.3, the operation *AddRole()* stereotyped ≪*Add*≫ is an adaptation rule belonging to the class adaptation *RoleAddition*. It adds a new class, named *Role*, to the package *SubscriberPackage*, matched by the pointcut *SubsriberPackagePointcut*. The class *Role* is defined inside the RBAC aspect.

6.2.2.2 Removing an Existing Element

The deletion of an existing element from the base model is modeled as a special kind of operation stereotyped ≪*Remove*≫. The set of elements that should be removed are given by a pointcut expression specified by the meta-element *Pointcut* (see Sect. 6.2.3). The same specification is used for removing any kind of UML element, either structural or behavioral. No tagged value is required for the specification of a *Remove* adaptation rule; the pointcut specification is enough to select the elements that should be removed.

The proposed profile for the specification of adaptations and their adaptation rules is expressive enough to cover the common AOP adaptations; i.e., introductions and *before/after/around* advices. For example, the profile allows to specify the addition of a new class to an existing package, a new attribute or an operation to an existing class, or a new association between two existing classes. In addition, we can remove an existing class, an attribute or an operation from an existing class, or an association between two existing classes. As for behavioral modifications, the profile allows to specify the injection of any UML behavior *before*, *after*, or *around* any behavioral UML element matched by the concerned pointcut. For example, the profile allows to specify the addition of an interaction fragment *before/after/around* a specific message in a sequence diagram, or an action *before/after/around* a specific action in an activity diagram. Moreover, the proposed adaptation rules are generic; they can be used to specify any security solution for any design. Table 6.1 summarizes the main adaptation rules that are supported by our approach.

6.2.3 Pointcuts

A pointcut is an expression that designates a set of join points. To specify pointcuts, we propose a pointcut language in a textual representation rather than using UML notations. This choice is motivated by the expressiveness and the easiness of the textual representation comparing to UML. For example, expressing logical pointcuts

Table 6.1 Supported adaptation rules

UML diagram	Supported adaptation rules
Class diagram	Adding/removing a class
	Adding/removing an attribute
	Adding/removing an operation
	Adding/removing an association
	Adding/removing a package
State machine diagram	Adding/removing a state machine
	Adding/removing a state
	Adding/removing a transition
	Adding/removing a region
Sequence diagram	Adding/removing an interaction
	Adding/removing an interaction use
	Adding/removing a lifeline
	Adding/removing a message
Activity diagram	Adding/removing an activity
	Adding/removing an action
	Adding/removing a structured activity node
	Adding/removing a control flow

in a textual way is more readable than expressing them in UML. In our approach, a pointcut is modeled as a meta-element stereotyped ≪*Pointcut*≫ with two tagged values (Fig. 6.4):

- *TextExpression*: The pointcut expression specified in our proposed textual pointcut language.
- *OCLExpression*: An OCL expression equivalent to the textual one, which will be automatically generated during the weaving process as we will see in Chap. 7.

The textual pointcuts are high-level and easy to write and understand. However, they cannot be directly used to query UML elements and select the appropriate join points. Thus, in our framework, we translate the textual pointcut expressions into OCL expressions to query UML elements. By doing so, we benefit from the expressiveness of the OCL language and, at the same time, we eliminate the overhead of writing such complex expressions from the developers. More details about the generation of OCL expressions from the textual ones are provided in Chap. 7.

Since the targeted join points are UML elements, pointcuts should be defined based on designators that are specific to UML. To this end, we define a pointcut language that provides UML-specific pointcut designators needed to select UML join points. The proposed pointcut language covers all the kinds of join points where our supported adaptations are performed. In the following, we present the primitive pointcut designators for the main UML diagrams that are supported by our approach, i.e., class diagrams, state machine diagrams, sequence diagrams, and activity diagrams.

Those primitives can be composed with logical operators (AND, OR, and NOT) to build other pointcuts.

6.2.3.1 Class Diagram Pointcuts

Table 6.2 presents the pointcut primitives that are proposed to designate class diagram elements. We choose the main elements that are usually used in class diagrams, i.e., class, attribute, operation, association, and package. Class diagram elements are

Table 6.2 Class diagram pointcuts—part 1

Join point	Pointcut designator	Description
Class	Class(NamePattern)	Selects a class based on its name
	Inside_Package(PackagePointcut)	Selects a class that belongs to a specific package matched by *PackagePointcut*
	Contains_Attribute(AttributePointcut)	Selects a class that contains a specific attribute matched by *AttributePointcut*
	Contains_Operation(Operation-Pointcut)	Selects a class that contains a specific operation matched by *OperationPointcut*
	Associated_With(ClassPointcut)	Selects a class that is associated with a specific class matched by *ClassPointcut*
Attribute	Attribute(NamePattern)	Selects an attribute based on its name
	Inside_Class(ClassPointcut)	Selects an attribute that belongs to a specific class matched by *ClassPointcut*
	Of_Type(TypePattern)	Selects an attribute that is of a certain type
	Of_Visibility(VisibilityKind)	Selects an attribute that is of a certain visibility (e.g., public, private)
Operation	Operation(NamePattern)	Selects an operation based on its name
	Inside_Class(ClassPointcut)	Selects an operation that belongs to a specific class matched by *ClassPointcut*
	Args(TypePattern1, TypePattern2,…)	Selects an operation based on the type of its arguments
	Of_Visibility(VisibilityKind)	Selects an operation that is of a certain visibility (e.g., public, private)
Association	Association(NamePattern)	Selects an association based on its name
	Between(ClassPointcut, ClassPointcut)	Selects an association that is between certain classes
	Member_Ends(AttributePointcut, AttributePointcut)	Selects an association based on its member ends
	Aggregation_Kind(AggregationKind)	Selects an association based on its aggregation kind (e.g., composite)
Package	Package(NamePattern)	Selects a package based on its name
	Inside_Package(PackagePointcut)	Selects a package that belongs to a specific package matched by *PackagePointcut*
	Contains_Class(ClassPointcut)	Selects a package that contains a specific class matched by *ClassPointcut*

designated either by their main properties, e.g., name, type, visibility, container, and owned elements, or by other associated elements. For example, the following pointcut expression designates a class, named *c1*, that is inside a package *p1*, and contains an operation *op1*:

```
Class(c1) && Inside_Package(p1) && Contains_Operation(op1)
```

Moreover, if we want to designate all classes that contain either private attributes or private operations, then the following pointcut is an example of such expression:

```
Class(*) && (Contains_Attribute(Of_Visibility(Private)) ||
   Contains_Operation(Of_Visibility(Private)))
```

Note that the symbol "∗" is used to designate all the elements of a particular type regardless of their names, as it is used in AspectJ [113].

6.2.3.2 State Machine Diagram Pointcuts

Table 6.3 presents the pointcut primitives proposed to designate the elements of state machine diagrams. We choose the main elements that are usually used in state machine diagrams, i.e., state machine, region, state, and transition. A state machine diagram element is designated either by its name, container, owned elements, specified elements (in case of a state machine), incoming/outgoing transitions (in case of a state), or source/target states (in case of a transition). For example, the following pointcut expression designates a state, named *s1*, with an incoming transition *t1*, and that belongs to a state machine *sm1*:

```
State(s1) && Incoming(t1) && Inside_State_Machine(sm1).
```

6.2.3.3 Sequence Diagram Pointcuts

Table 6.4 presents the primitives proposed to designate sequence diagram elements. We choose the main elements that are commonly used in sequence diagrams, i.e., interaction, message, and lifeline. A sequence diagram element is designated either by its name, type, container, owned elements, specified elements (in case of an interaction), or source/target lifelines (in case of a message). For example, the pointcut *SensitiveMethodPointcut* in Fig. 6.3 is a conjunction of three pointcuts: (1) *Message_Call(SensitiveMethod)* selects any message that calls *SensitiveMethod()*, (2) *Message_Source(User)* selects any message whose source is of type *User*, and (3) *Message_Target(Resource)* selects any message whose target is of type *Resource*. The conjunction of these three pointcuts allows the selection of all message calls to *SensitiveMethod()* from a *User* instance to a *Resource* instance.

Table 6.3 State machine diagram pointcuts

Join point	Pointcut designator	Description
State machine	State_Machine(NamePattern)	Selects a state machine diagram based on its name
	Contains_Region(RegionPointcut)	Selects a state machine that contains a specific region matched by *RegionPointcut*
	Contains_State(StatePointcut)	Selects a state machine that contains a specific state matched by *StatePointcut*
	Contains_Transition(TransitionPointcut)	Selects a state machine that contains a specific transition matched by *TransitionPointcut*
	Specifies_Class(ClassPointcut)	Selects a state machine that specifies a specific class matched by *ClassPointcut*
Region	Region(NamePattern)	Selects a region based on its name
	Inside_State_Machine(StateMachinePointcut)	Selects a region that belongs to a specific state machine matched by *StateMachinePointcut*
	Inside_State(StatePointcut)	Selects a region that belongs to a specific state matched by *StatePointcut*
	Contains_State(StatePointcut)	Selects a region that contains a specific state matched by *StatePointcut*
	Contains_Transition(TransitionPointcut)	Selects a region that contains a specific transition matched by *TransitionPointcut*
State	State(NamePattern)	Selects a state based on its name
	Inside_Region(RegionPointcut)	Selects a state that belongs to a specific region matched by *RegionPointcut*
	Inside_State(StatePointcut)	Selects a state that belongs to a specific state matched by *StatePointcut*
	Inside_State_Machine(StateMachinePointcut)	Selects a state that belongs to a specific state machine matched by *StateMachinePointcut*
	Incoming(TransitionPointcut)	Selects a state that has a specific incoming transition matched by *TransitionPointcut*
	Outgoing(TransitionPointcut)	Selects a state that has a specific outgoing transition matched by *TransitionPointcut*
	Contains_State(StatePointcut)	Selects a state that contains a specific state matched by *StatePointcut*
	Contains_Transition(TransitionPointcut)	Selects a state that contains a specific transition matched by *TransitionPointcut*
Transition	Transition(NamePattern)	Selects a transition based on its name
	Inside_Region(RegionPointcut)	Selects a transition that belongs to a specific region matched by *RegionPointcut*
	Inside_State(StatePointcut)	Selects a transition that belongs to a specific state matched by *StatePointcut*
	Inside_State_Machine(StateMachinePointcut)	Selects a transition that belongs to a specific state machine matched by *StateMachinePointcut*
	Source_State(StatePointcut)	Selects a transition that has a specific source state matched by *StatePointcut*
	Target_State(StatePointcut)	Selects a transition that has a specific target state matched by *StatePointcut*

Table 6.4 Sequence diagram pointcuts

Join point	Pointcut designator	Description
Interaction	Interaction(NamePattern)	Selects an interaction based on its name
	Contains_Message(Message-Pointcut)	Selects an interaction that contains a specific message matched by *MessagePointcut*
	Contains_Lifeline(Lifeline-Pointcut)	Selects an interaction that contains a specific lifeline matched by *LifelinePointcut*
	Specifies_Operation(Operation-Pointcut)	Selects an interaction that specifies the behavior of a specific operation matched by *OperationPointcut*
Message	Message_Call(NamePattern)	Selects a message call, either synchronous or asynchronous, based on its name
	Message_Syn_Call(NamePattern)	Selects a message that specifies a synchronous call
	Message_Asyn_Call(Name-Pattern)	Selects a message that specifies an asynchronous call
	Reply_Message(NamePattern)	Selects a reply message based on its name
	Create_Message(NamePattern)	Selects a message that creates an object
	Destroy_Message(NamePattern)	Selects a message that destroys an object
	Message_Source(TypePattern)	Selects a message whose source is of a certain type
	Message_Target(TypePattern)	Selects a message whose target is of a certain type
	Inside_Interaction(Interaction-Pointcut)	Selects a message that belongs to a specific interaction matched by *InteractionPointcut*
Lifeline	Lifeline(NamePattern)	Selects a lifeline based on its name
	Inside_Interaction(Interaction-Pointcut)	Selects a lifeline that belongs to a specific interaction matched by *InteractionPointcut*
	Covered_By_Fragment(Name-Pattern)	Selects a lifeline that is covered by a specific interaction fragment
	Contains_Execution(NamePattern)	Selects a lifeline that contains a specific execution specification

6.2.3.4 Activity Diagram Pointcuts

Table 6.5 presents the primitives proposed to designate the elements of activity diagrams. We choose the main elements that are commonly used in activity diagrams, i.e., activity, action, and edge. An activity diagram element is designated either by its name, type, container, owned elements, specified elements (in case of an activity), incoming/outgoing edges (in case of an action), or source/target actions (in case of an edge). For example, the following pointcut expression designates a call operation

Table 6.5 Activity diagram pointcuts

Join point	Pointcut designator	Description
Activity	Activity(NamePattern)	Selects an activity based on its name
	Contains_Action(ActionPointcut)	Selects an activity that contains a specific action matched by *ActionPointcut*
	Contains_Edge(EdgePointcut)	Selects an activity that contains a specific activity edge matched by *EdgePointcut*
	Specifies_Operation(Operation-Pointcut)	Selects an activity that specifies the behavior of a specific operation matched by *OperationPointcut*
Action	Action(NamePattern)	Selects an action based on its name
	Call_Operation_Action(Name-Pattern)	Selects an action that performs an operation call
	Call_Behavior_Action(Name-Pattern)	Selects an action that performs a behavior call
	Create_Action(NamePattern)	Selects an action that creates an object
	Destroy_Action(NamePattern)	Selects an action that destroys an object
	Read_Action(NamePattern)	Selects an action that reads the value(s) of a structural feature
	Write_Action(NamePattern)	Selects an action that updates the value(s) of a structural feature
	Inside_Activity(ActivityPointcut)	Selects an action that belongs to a specific activity
	Input(TypePattern, …)	Selects an action based on the type of its input pins
	Output(TypePattern, …)	Selects an action based on the type of its output pins
Control Node	Initial(NamePattern)	Selects an initial node based on its name
	Final(NamePattern)	Selects an activity final node based on its name
	Flowfinal(NamePattern)	Selects a flow final node based on its name
	Fork(NamePattern)	Selects a fork node based on its name
	Join(NamePattern)	Selects a join node based on its name
	Decision(NamePattern)	Selects a decision node based on its name
	Merge(NamePattern)	Selects a merge node based on its name
Activity Edge	Edge(NamePattern)	Selects an edge based on its name
	Inside_Activity(ActivityPointcut)	Selects an edge that belongs to a specific activity
	Source_Action(ActionPointcut)	Selects an edge that has a specific source
	Target_Action(ActionPointcut)	Selects an edge that has a specific target

action, named *a1*, that belongs to an activity *act1*: *Call_Operation_Action(a1)* && *Inside_Activity(act1)*.

6.3 Related Work on AOM

During the last decade, AOM has become the center of many research activities. Following the success of AOP techniques in modularizing crosscutting concerns at the implementation level, considerable number of contributions worked on abstracting AOP concepts and adopting them at different specification and design languages. An overview and a comparison of the existing approaches are presented in [31, 170, 182]. In the following, we provide a summary of the main approaches.

Kienzle et al. [116, 117] have proposed Reusable Aspect Models (RAM); an AOM approach that specifies a concern using class, state machine, and sequence diagrams. One of the goals of the RAM approach is to support aspect reusability, i.e., build aspects with complex functionalities by reusing simple ones, by means of aspect dependency chains. A weaver is implemented using Kompose [85] for weaving class diagrams and Geko [138] for weaving state machine diagrams and sequence diagrams.

The High-Level Aspects (HiLA) approach [212] extends UML state machines for specifying history-dependent and concurrent behaviors. Join points in HiLA capture points when a transition is being fired. Pointcuts may also contain constraints, i.e., advices are only executed when the constraints are satisfied. To increase reusability, aspects are specified as UML templates, which are then specialized to the designer's application. HiLA also allows transformational aspects, i.e., aspects that can match a sub-structure of the base state machine and replace them by the advice.

Klein et al. [118] have proposed various formal definitions of join points in sequence diagrams. Aspects are specified as pairs of UML 2.0 sequence diagrams: One sequence diagram for pointcuts and the other one for advice specification. Join points can be either a single element or a collection of elements. This approach also provides a formal definition of a new composition operator for sequence diagrams, called an amalgamated sum, and describes its implementation using Kermeta.[2]

Tkatchenko and Kiczales [196] have added a join point model (JPM) to UML metamodel. They have covered three UML diagrams, namely, class diagrams, state machine diagrams, and sequence diagrams. For class diagrams, the considered join points are class and operation elements. For sequence diagrams, they have considered messages and lifelines as join points. For state machine diagrams, states and call triggers have been considered as join points. Comparing with our approach, we cover a wider range of diagrams and UML elements as join points. In addition, the matching process in this approach is based only on direct name matching or on signature comparison.

[2] http://www.kermeta.org/.

Clark et al. [59] have proposed an AOM approach called Theme/UML. This approach is a symmetric one, i.e., there is no distinction between the base model and the crosscutting concerns. It is a general-purpose AOM language. Aspects are modeled as templates that are bound to base elements through binding relationships. Package and class diagrams are used for modeling structural adaptations and sequence diagrams are used for modeling behavioral adaptations. This approach is possibly the most mature and the most well-engineered approach to AOM. However, its main intent is the identification of aspects in the requirements analysis phase and mapping those aspects to the design.

Some contributions have focused on abstracting AspectJ [113] into the modeling level [79, 191, 208]. Evermann [79] has proposed a UML profile for AspectJ based on the existing UML metamodel. An aspect is specified as a stereotyped class. Pointcuts are modeled as stereotyped attributes, while advices are modeled as stereotyped operations. In contrast to previous work on AspectJ profiles, this is possibly the most complete specification so far. Stein et al. [191] have proposed one of the earlier profiles for AspectJ. Pointcuts and advices are specified as stereotyped operations. Join points are considered as messages in collaboration diagrams. The introduction of new class elements or associations is specified using UML diagram templates. Weaving of advices and introductions into base models is modeled as relationships in collaboration diagrams denoting the crosscutting effects of aspects on their base classes.

Yan et al. [208] have adopted the extension of UML metamodel by introducing an AspectJ metamodel in order to support AspectJ software modeling. First, a metamodel for Java was designed by tailoring UML meta-classes to Java. Then, the Java metamodel was extended into AspectJ metamodel. This work aims at narrowing the gap between conceptual modeling of aspects and their concrete implementation in AspectJ. The same approach of extending UML metamodel for aspect specification was also proposed by Chavez and Lucena [55]. However, the main limitation of such an approach is the fact that extending UML metamodel requires either modifying existing UML case tools, or implementing new ones in order to provide support for the newly defined meta-classes.

One of the initial proposals in this field is the one of Aldawud et al. [28]. It provides a UML profile for aspect specification by applying stereotypes on classes. Later, it has been extended to support pointcut and advice specification [29]. Crosscutting associations are used to show how aspect elements relate to base model elements. This profile is very generic and captures only few concepts of AOP. Other contributions in this area [43, 44, 91, 112, 139, 165] have provided extensions of the UML language for modeling aspects using standard UML extension mechanisms. However, the majority of these approaches are programming language dependent and specify only few concepts of AOP.

6.4 Conclusion

In this chapter, we have presented an AOM approach for specifying and weaving security aspects into UML design models. This approach is well suited for job separation: security experts provide high-level security solutions including the details on how to apply them in UML diagrams and the designers apply them in their design by adapting them to the design context. With our approach, even the designers with limited security knowledge can use the security solutions to enforce the needed security requirements in a systematic way in their design. As another result of our contribution, security solutions can be integrated into software from the early phases of the development life cycle. This in turn helps accelerating the development of secure applications and reducing errors and costs.

Different mechanisms can be used to specify aspects at the model level. Some contributions suggest extending UML metamodel by adding new meta-classes or creating new meta-models to specify aspect-oriented concepts. These techniques suffer from implementation and interoperability issues, as UML case tools need to be extended to support the newly specified meta-classes. The other technique, i.e., using standard UML extension mechanisms, is a better solution as it overcomes the limitations identified in the previous approaches.

In this setting, we have developed a UML profile for the specification of aspects at the design level. The proposed profile allows the specification of common aspect-oriented primitives, i.e., adding new elements *before/after/around* join points and removing existing elements. In addition, the proposed profile supports both structural and behavioral adaptations and covers the main diagrams that are used in UML design. Furthermore, we have defined a high-level and user-friendly pointcut language that can be used by security experts to designate UML elements. We have seen that the proposed pointcut language is expressive enough to designate the main elements that are used in a software design. In the next chapter, we will present our approach for systematically weaving the security aspects, specified using our AOM profile, into UML design models.

http://www.springer.com/978-3-319-16105-1

Aspect-Oriented Security Hardening of UML Design Models

Mouheb, D.; Debbabi, M.; Pourzandi, M.; Wang, L.; Nouh, M.; Ziarati, R.; Alhadidi, D.; Talhi, C.; Lima, V.

2015, XVIII, 237 p. 123 illus., Hardcover
ISBN: 978-3-319-16105-1