Aspect-Oriented Design for Embedded Software

by

Natsuko NODA

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Tomoji Kishi

School of Information Science
Japan Advanced Institute of Science and Technology

September 2008

Abstract

In software design, it is important to encapsulate cross-cutting concerns, and the application of aspect-oriented technologies to design modeling is a significant challenge. We examined the design of software for embedded systems that exhibit complicated behavior and observed that aspect orientation is useful for designing such systems.

Aspect-orientation is one of the promising techniques not only for programming but also for analysis and design. For programming, we have a popular language, AspectJ, and the existence of this language facilitates the diffusion of aspect-oriented programming. Likewise, in order to actually utilize aspect-orientation in analysis and design phase, it is desirable to have good aspect-oriented modeling mechanism. Here, aspect-oriented modeling mechanism provides us the means for aspect-oriented modeling, in which the aspect-oriented concept such as "aspect" as well as conventional concepts such as "class" and "association" should be treated as the first class modeling elements.

In this thesis, we propose an aspect-oriented modeling mechanism for software development, especially for embedded software design. An aspect in this mechanism is a unit to modularize a concern, and it includes fragments of software model. In other words, it is a structure to realize a concern. It is similar to a "hyperslice" of Hyper/J [3]. In our modeling mechanism, an aspect contains one or more classes, each of which can have a state model that expresses its behavior.

We also examine issues in embedded software design, and point out that in modeling embedded software, we have to manage cross-cutting relationship. In order to avoid the issue, we examined the application of our aspect-oriented modeling mechanism to embedded software design. We introduce modeling of an example based on our mechanism, and show how this mechanism facilitates the embedded software design.

We also introduce an application of the mechanism to product-line development. As an aspect in our mechanism is independent each other, this characteristics makes it possible to design highly configurable product-lien architecture. We demonstrate the usefulness of our modeling mechanism based on an embedded system—vehicle illumination system.

Acknowledgments

The authors would like to thank Professor Tomoji Kishi for continuous supports and suggestions to the research. The members of Kishi laboratory give us helpful comments and support to the research. The case study of this research is based on a real project in a company. We appreciate the project members who provided us the useful example.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The advancement of embedded technologies has made our society increasingly dependent on embedded systems and increased the size and complexity of embedded software. In this scenario, embedded software developers must pay attention to not only performance and size but also extensibility and modifiability. Although most embedded software development has been implementation-centric, architecture design of embedded systems has assumed greater importance today.

One of the characteristics of embedded system is that they are context dependent, i.e., embedded systems react to changes in their context, and their behavior is also constrained by the context. In such systems, contexts are determined in terms of sensor values, and the behavior of the system is based on these contexts. Similarly, internal process controls actuator and assumes the context as the result of the control. Since sensors, actuators, contexts, and internal processing have cross-cutting relationships, we need to encapsulate them in order to make software modifiable and extensible.

In this thesis, we propose an aspect-oriented modeling approach for context-dependent embedded software and an aspect-oriented modeling mechanism that facilitates this modeling approach. The approach and mechanism are explained based on an embedded system—vehicle illumination system.

There are some applications of aspect-oriented techniques in the field of embedded software. Though there exist a few works that apply aspect-oriented techniques to architectural design [15], most other applications are the use of aspect-oriented programming [4] and the application of aspect-oriented modeling techniques in the context of model-driven development [9]. This thesis focuses on the application of aspect-oriented techniques to the architectural design of embedded systems.

As many embedded systems are developed in the context of product-line development (PLD), we apply our aspect-oriented modeling mechanism to variability management in PLD design. Thus far, several techniques for utilizing aspect-oriented technologies (AOTs) for PLD have been proposed. However, the application of AOTs to PLD is not simple and various issues related to the application, such as invasive change problem that prevents reusability, have been reported. In this study, we also show how our modeling mechanism works well in this area.

Chapter 2

Issues in Embedded Software Design

2.1 Characteristics and issues of embedded software design

One of the important characteristics of embedded systems is strong relation with the external world; they react to changes in the external world and they try to change the circumstances. For example, a system reacts to air temperature change, and if the temperature is too high, it works to cool down the air.

In order to function in such a way, embedded systems have sensors, internal processing, and actuators; they recognize the changes in the external world through the sensors, perform the internal processing that depends on the changes recognized by the sensors, and the internal processing may drive actuators to operate some entities in the environment.

These sensors and actuators have many variations. There are a variety of sensors and actuators different physically, even though they have a logically same function. Also, there can be multiple ways to recognize an outside change utilizing sensors. For example, to recognize vehicle speed, a velocity sensor can be used, or a GPS sensor may be used. Likewise, there may be multiple ways to drive actuators. For example, to reduce vehicle speed, the engine may be powered down, or the brake can be applied.

During the system's lifecycle, physical sensors and/or actuators and how to utilize them can be changed because of various reasons. For example, required accuracy cannot often be ensured, and which sensors and actuators are used and how to utilize them may be decided by trial and error; or, in concurrent development, sensors and/or actuators may be changed, regardless of software, due to hardware design. Also, modification and addition/deletion of system's function might be required and internal processing would be changed. That can cause changes of sensors/actuators to be utilized and usage of them.

In this situation, one of the biggest issues of embedded software design is that it is difficult to develop design model modifiable. If we directly couple sensors/actuators and internal processing, we cannot manage these changes mentioned above.

2.2 Context

To tackle the issue described in the previous section, contexts should be clearly recognized and considered. A context is a surrounding condition that the system needs clearly to capture and change. Typical conditions that the system needs clearly to capture are:

- context of outside environment; e.g., the temperature, humidity, brightness around the vehicle.
- context of entire system's condition; e.g., the position, velocity, acceleration of the vehicle.
- context of system element's condition; e.g., the engine status, door position, gear position.

These contexts are the outside conditions and the system recognizes them by means of sensor values. As implementation, some contexts may be sensor values themselves, or values obtained by data fusion of multiple sensors. We call these contexts *s-contexts*.

Conditions that the system needs to change are:

- context of outside environment; e.g., the temperature in the vehicle, brightness around the vehicle.
- context of entire system's condition; e.g., the position, velocity, acceleration of the vehicle.
- context of system element's condition; e.g., the engine status of the vehicle, the position of robot arm.

These contexts are the outside conditions that are supposed as a result of sending control values to actuators. We call these contexts *a-contexts*.

We observe that these contexts referred to in a specific domain are stable, as compared to sensors, actuators and internal processing. For example, in most automotive systems a door position context is necessary and recognized, however how to recognize this context and which sensors are utilized for the recognition of this context may widely vary. And without this context, a internal processing that reacts the change of the door position may vary and not be so stable, because it has to refer sensors directly.

Therefore, to avoid the problem described in 2.1, it is important to separate the surrounding conditions that the system needs clearly to capture (or change) and the means to capture (or changes) these conditions; namely, contexts and sensors/actuators. This separation will destroy direct coupling of sensors/actuators and internal processing.

2.3 Aspect-oriented context modeling

In 2.2, we pointed out the importance of separating contexts from sensors/actuators and internal processing. However, how to realize this separation in modeling is not straight forward. Contexts relate to sensors, actuators, and internal processing in crosscutting ways. For example, a context may be recognized by multiple sensors; a sensor may be utilized by multiple contexts; in addition, each of the contexts may require different interfaces from the same sensor, because different usages are needed to recognize each context. In other words, contexts crosscut, and are crosscut by, sensors/actuators and internal processing. Therefore, ordinary modularization and layering is not sufficient to model such systems.

We have been examining utilization of aspect-oriented technology to model such embedded software; each specific sensor/actuator, context, and internal processing is encapsulated as "*aspect*." Here, an aspect is like "hyperslice" [42][37] in Hyper/J [3]. That

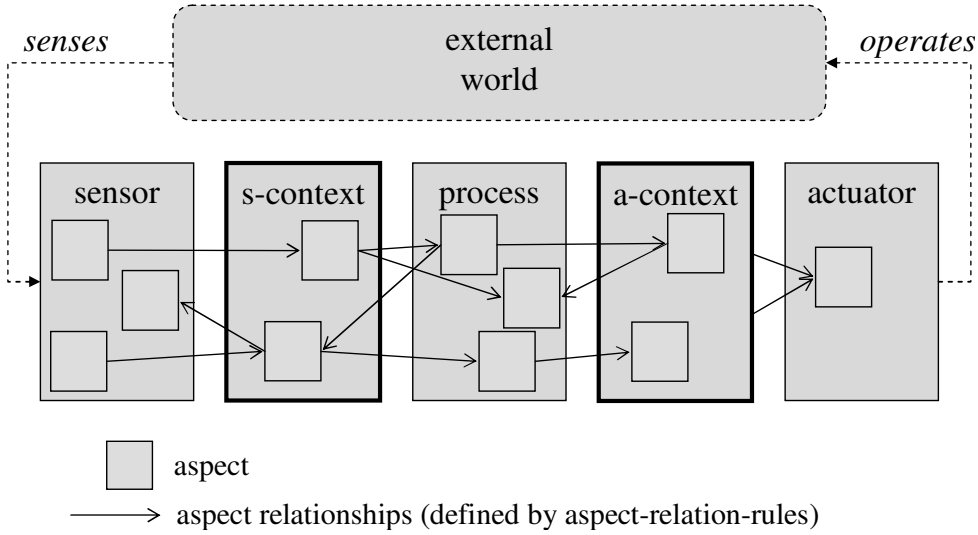


Figure 2.1: Overview of aspect-oriented context modeling

means an aspect encapsulates a concern by defining a (partial) class structure appropriate for that concern, might or might not cut across another, and can be understood in isolation; crosscutting behavior can be specified by composition and how to compose aspects are defined outside aspects; there is no base hierarchy and no aspect dominates the others.

We call this modeling *aspect-oriented context modeling*. An early sketch of this was illustrated in [17].

Figure 2.1 schematically presents an overview of the aspect-oriented context modeling.

- Components are categorized as sensors (components that handle sensor values), s-contexts (components that describe surrounding conditions to be captured), processes (components that realize internal processing), a-contexts (components that describe surrounding conditions to be changed), or actuators (components that control actuators). The sensors/actuators and processes are indirectly coupled.
- Each component is realized as an aspect. Aspects are independent and self-contained (similar to the "hyperslice").
- Aspects are related to each other by composition.

Chapter 3

Aspect-Oriented Modeling Mechanism

In Chapter 2, we pointed out issues in embedded software design. To facilitate the modeling for embedded software design considering the issues, we introduced the aspect-oriented context modeling. The introduced modeling provides a design policy for embedded software.

In order to model embedded software following the design policy, we need a modeling mechanism that provides us the means to model appropriately crosscutting concerns such as sensors, actuators, process and contexts. Especially, when we follow the design policy of aspect-oriented context modeling, it is important to separate relationships among aspects and make them independent of aspects. The required modeling mechanism is expected to realize this separation in modeling.

To facilitate the introduced aspect-oriented context modeling, we propose an aspect-oriented modeling mechanism. An early sketch of this mechanism was illustrated in [32].

Figure 3.1 shows a simple example of the aspect-oriented modeling utilizing the proposed modeling mechanism. In this example, we define three aspects—A, B, and C. Each aspect comprises a class diagram and state diagrams and is self-contained. The relationships among aspects are not defined within each aspect but are provided as aspect-relation-rules. These rules define the inter-locking among the state diagrams of different aspects. The relationships among aspects given by these rules are intuitively shown by dashed arrows. These rules are explained in detail in the following sections.

In the following sections, we will explain the modeling mechanism from the following viewpoints respectively.

- modeling elements
- execution semantics
- notation

3.1 Modeling elements

In this section, we will introduce new modeling elements used in this mechanism. Some "traditional" modeling elements, such as "class," are also used in the mechanism, without any change of their meaning. Therefore, these traditional elements will not re-defined

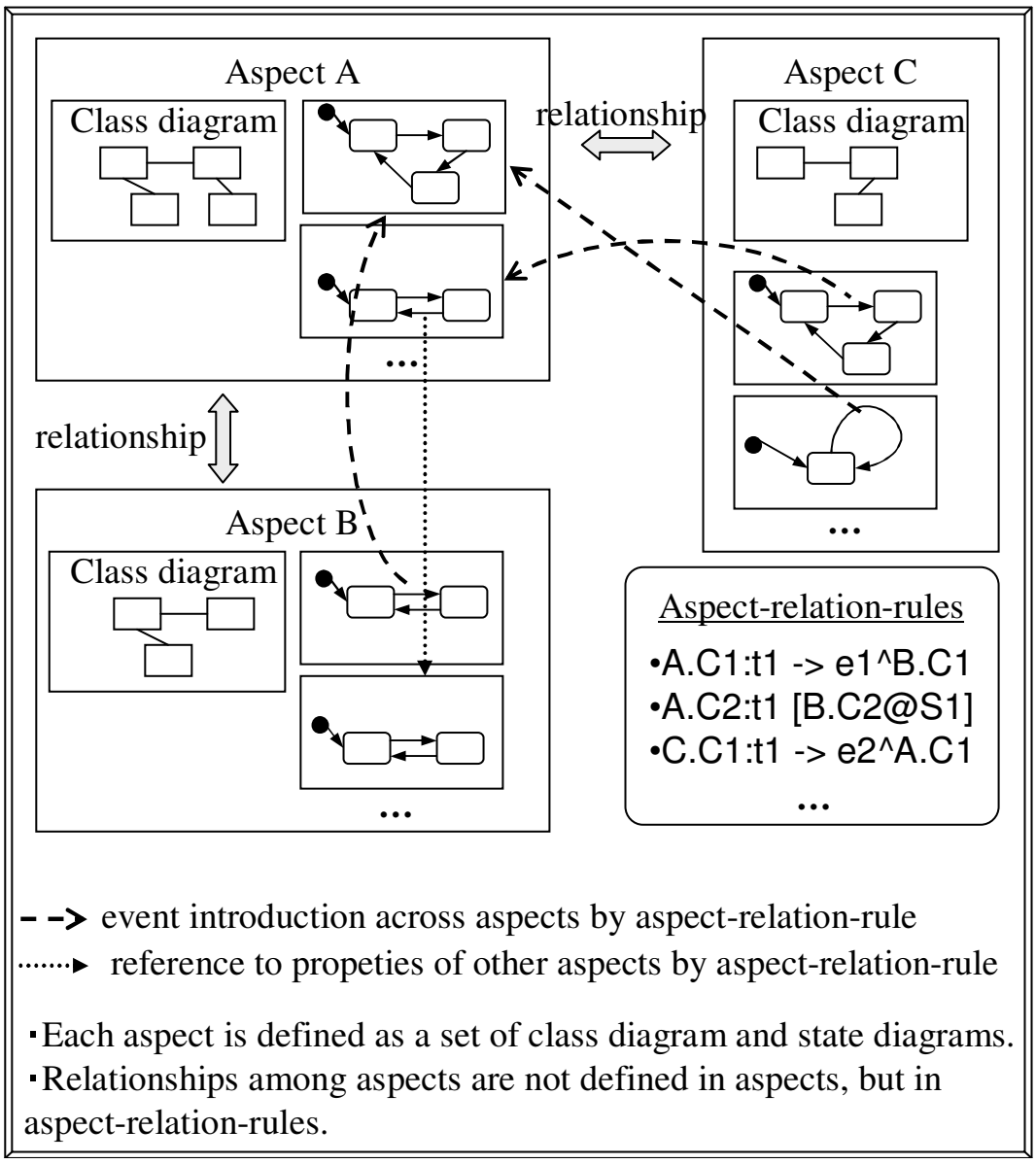


Figure 3.1: Overview of aspect-oriented modeling utilizing the proposed modeling mechanism

nor explained in this section; if we give some constraints on their usages, that will be explained. All the modeling elements, including newly introduced ones and traditional ones, will be described as a metamodel, which will be introduced in Appendix B.

3.1.1 Aspect

An aspect is a unit that modularizes a concern, and it is the projection of the entire software from the viewpoint of the corresponding concern. It is defined as a set including a static structure model and behavior models. The static structure model is defined as a class diagram and this diagram contains one or more classes. Each class of the static structure model has a behavior model. The behavior model is defined as a state diagram; That means behavior of objects are described by their states and transitions. Transitions are trigged by events; events are some occurrences that trigger transitions. Transitions may have guard conditions and actions.

Although concerns are related to each other, because they cross-cut, each aspect is defined such that it is self-contained. The relationships among aspects are not defined within the aspect but as aspect-relation-rules, which will be explained in 3.1.3.

In order to describe these rules, we allow the assignment of an identifier (such as a unique label) to a transition, if necessary.

In this thesis, we assume that each class has only one instance and instances of classes are not dynamically created, mainly because the ease of explanation. In embedded software field, it is a common way to utilize objects, and we believe that this restriction does not essentially narrow the applicability of our modeling mechanism.

3.1.2 Aspect-relation

Aspects have cross-cutting relations, when they are used to construct the entire software. Typical relationships are as follows:

- A particular behavior (e.g., state transition) of a class in aspect A triggers a certain behavior in a class of aspect B.
- A class in aspect A refers to the properties (e.g., state) of a class in aspect B to determine its behavior.

A set of such relationships between two aspects to realize a function of the entire software is called an aspect-relation. An aspect-relation is defined by a rule set and the rule set is a set of aspect-relation-rules that relate two aspects.

3.1.3 Aspect-relation-rule

We use aspect-relation-rules to define the details of aspect-relations. An aspect-relation-rule describes the relationships between different aspects in terms of events, transitions, and guard conditions for the transitions of state diagrams. The different types of rules are as follows:

- Event-introduction rule: An event-introduction rule introduces events to other aspects to trigger a certain behavior when a state transition occurs. For example, "A1.C1:t1 -> E1^A2.C2," which implies that "when transition t1 fires at class C1 of aspect A1, event E1 is introduced to class C2 of aspect A2."

- Condition-reference rule: A condition-reference rule provides a guard condition, with reference to the properties of an aspect, for a state transition in another aspect. For example, "A1.C1:t1 [A2.C2@S1]," which implies that "transition t1 fires at class C1 of aspect A1 only when class C2 of aspect A2 is in state S1."

For readability, we showed an example of rule description, such as "A1.C1:t1 -> E1^A2.C2" and "A1.C1:t1 [A2.C2@S1]." They are just examples of description, and there would be other possibilities. We place less importance on notations of the rules than the meanings of them. So far we have not found any problems in this notation we showed. The syntax of this notation will be explained in 3.3.3.

3.2 Execution semantics

The execution semantics for this mechanism are summarized below.

- Each object in an aspect runs concurrently.
- Each object has a queue for input events.
- Events to an object are stored in the queue. There exists no distinction between the events from objects in the same aspect and the events introduced by event-introduction rules.
- Each object can read an event from its queue based on a first-in-first-out policy. When an object receives an event from its queue, it triggers the corresponding state transition.
- When an event received from the queue triggers the corresponding state transition, each object refers to the condition-reference rules specified for the aspects they belongs to. If there exists a rule stating a guard condition for firing the corresponding state transition, the state of the specified object is referred to and the condition is evaluated; if the guard condition is evaluated to be true, the transition fires. There exists no distinction between the guards defined in state diagrams and those provided by condition-reference rules.

3.3 Notation

3.3.1 Policy for designing notation

In this modeling mechanism, an aspect is defined in the ordinary object-oriented way; it contains one or more classes and the behavior of each class is defined by the state transitions. Therefore, we do not need any special notation for it. We use class diagrams for its structure model and state diagrams for its behavior model.

In this study, we use state diagrams for simplicity. A state diagram is a subset of a state machine diagram; in this diagram, we do not allow the presence of composite states. In order to describe aspect-relation-rules (see below), we allow the assignment of an identifier (such as a unique label) to a transition, if necessary.

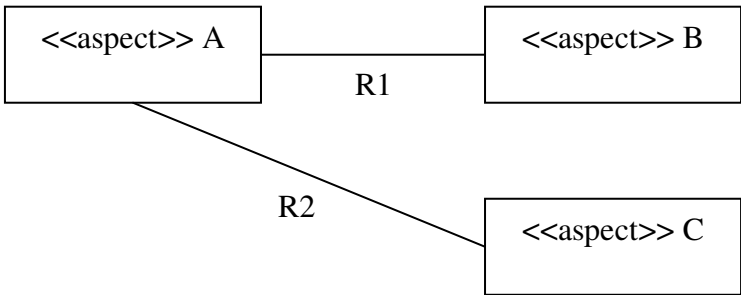


Figure 3.2: Aspect-relation overview diagram

Constructing entire software from multiple aspects using aspect-relations is new. To see and understand the entire software, we need a new notation. For this purpose, we will introduce an aspect-relation overview diagram in the next subsection.

3.3.2 Aspect-relation overview diagram

We use an aspect-relation overview diagram to understand the relations between aspects and to obtain an overview of the entire system comprising mutually related aspects. The aspect-relation overview diagram shows the aspects and the aspect-relations existing among them. An aspect is shown as a box with the stereotype "<<aspect>>." An aspect-relation is expressed as an association between two aspects with a label indicating a rule set. The definition of the aspect-relation is given by the corresponding rule set. Details of this rule set need not be described in this diagram; they can be defined somewhere.

Each rule in the rule set is written based on the syntax described in 3.1.3.

Figure 3.2 shows an example of the aspect-relation overview diagram.

3.3.3 Syntax of aspect-relation-rules

In 3.1.3, we introduced aspect-relation-rules. The syntax of the rules is shown in Figure 3.3. Note that rule description following the syntax shown here is one possibility; there would be other types of notation for the rules. In this thesis, we follow this syntax to describe the aspect-relation-rules.

Here, we allow arbitrariness (wildcards) in specifying class and transition names.

3.4 Simple example

Figure 3.4 shows a part of software design using our modeling mechanism. In this example, functionalities related to ensure the safety and functionalities related to an entertainment service are separated as aspect "SAFECTRL" and aspect "SRVC." Each aspect has a class diagram and state diagrams of the classes in the class diagram (in Figure 3.4, some state diagrams are omitted for the space limitation). These aspects are independent of each other. The relationships between these two aspects are defined as aspect-relation-rules. By these rules, the two aspects are related and provide an appropriate function; if in some dangerous situations, the service is stopped and cannot be started.

```
aspect_relation_rule
   ::= event_introduction_rule | condition_reference_rule
event_introduction_rule
   ::= source_aspect '.' source_class ':' transition_name
       '->' event_name '^' target_aspect '.' target_class
source_aspect ::= name
source_class ::= arbitrary | name
name ::= name_char+
name_char ::= [a-zA-Z0-9] | '_'
arbitrary ::= name? '*' name?
transition_name ::= arbitrary | name
event_name ::= name
target_aspect ::= name
target_class ::= arbitrary | name
condition_reference_rule
   ::= source_aspect '.' source_class ':' transition_name '[' condition ']'
condition ::= state_condition
state_condition ::= in_state | not_in_state
in_state ::= taget_aspect '.' target_class '@' state
state ::= name
not_in_state ::= '!' in_state
```

Figure 3.3: Syntax of aspect-relation-rules

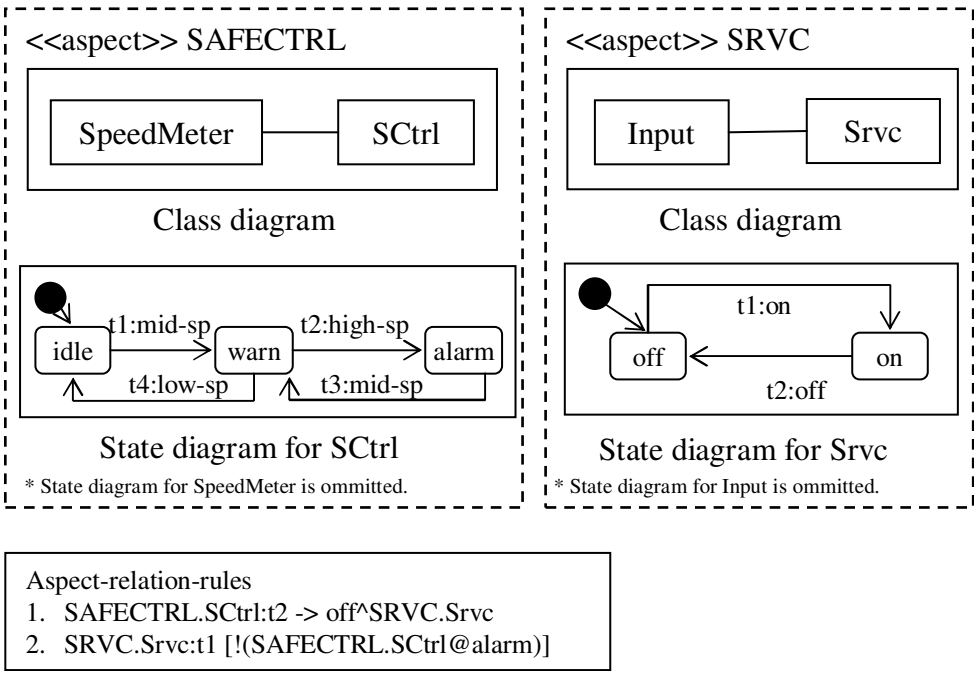


Figure 3.4: Design using aspect-oriented modeling mechanism

Chapter 4

Application of Mechanism to Architecture Design

In Chapter 2, we pointed out issues in embedded software design and described a desirable design policy to avoid the issues. In Chapter 3, we introduced a modeling mechanism that makes it possible to model based on the design policy. In this chapter, again we will describe the issue in order, and explain how the issue can be settled using the design policy and the modeling mechanism. The targeted phase for applying of the mechanism will be discussed as well.

4.1 Design issues and solution

The following is the issue which is tackled in the design of embedded software, and which we try to settle using the introduced modeling mechanism:

- To increase reusability and modifiability of embedded software design, by localizing influences of various changes of sensors, internal processing, and actuators.

The background of this issue is increasing complexity of embedded software. In this situation, quality attributes at development time, such as reusability, is getting more and more important, as well as quality attributes at run-time, such as performance, which has been the top priority in the conventional embedded software development.

As described in 2.1, sensors and actuators may frequently be changed in embedded system development. Therefore, in the design of embedded software, it is necessary to localize the influences of these changes.

The desirable design policy for the issue is, as mentioned in 2.2 and 2.3, as follows:

1. In order to weaken the direct coupling between sensors/actuators and internal process, contexts are placed between sensors/actuators and internal process.

2. Relationships among modules that exist in crosscutting ways have to be separated from modules and be made independent of them.

To model embedded software based on the design policy, we can utilize the introduced modeling mechanism; we modularize each of sensors, actuators, internal processing, and contexts, as an aspect, and define relationships among them as aspect-relation-rules, separating from aspects.

We will explain this utilization in more detail.

For the number 1 in the policy

- Based on the aspect-oriented context modeling, described in 2.3, we recognize contexts (s-contexts and a-contexts).
- We model each of sensors—such as a door sensor, a power sensor, and a temperature sensor—as an aspect. Likewise, each of actuators, each of internal processing, each of s-contexts, and each of a-contexts is modeled as an aspect.
- In each aspect, we model only what is indispensable to the corresponding concern. For example, in a sensor aspect, the changes of the sensor value is modeled; in a s-context aspect, the changes of the context is modeled; relationships between the sensor and the s-context are not defined in the sensor aspect, nor in the s-context aspect.

For the number 2 in the policy

- We define relationships among aspects as aspect-relation-rules. For example, a relationship between a s-context aspect and a process aspect (aspect of internal processing), such as "when the context changes from A to B, it triggers a operation of the process", is defined as an aspect-relation-rule.

It is possible to separate sensor/actuator, context and internal processing utilizing conventional modeling techniques. For example, we could adopt the layered architecture and define sensor/actuator layers as the bottom layer, a context layer as the middle layer and a process layer as the top layer. In this case, the context layer depends on the sensor/actuator layers, and the process layer depends on the context layer, i.e. direct dependencies are defined among layers. Figure 4.1 shows it intuitively. On the contrary, in our modeling mechanism, each aspect is defined completely separately without depending on other aspects. This makes each module highly reusable and composable. In Figure 4.2, a model using our modeling mechanism is intuitively shown.

Furthermore, sometimes it is difficult to define a simple interface of each layer. For example, consider the situation in which two contexts "vehicle speed" context and "safety" context use the same sensor "velocity sensor." As each context has its own concern, the way in which the sensor is used may be different. In this case, the "vehicle speed" context requires the interface of the sensor to give the vehicle speed. On the other hand, the "safety" context requires the sensor to give the information whether the vehicle is stopped or moving. It may be possible to make the sensor to have these two interfaces, but it makes the role of the sensor ambiguous. On the other hand, in our modeling mechanism, relationships among aspects are not included in aspects, and each aspect can be defined more clearly and concisely.

4.2 Target phase

The introduced modeling mechanism is intended to be used for architecture design of embedded software. It is used as the basis of module construction technique. The mech-

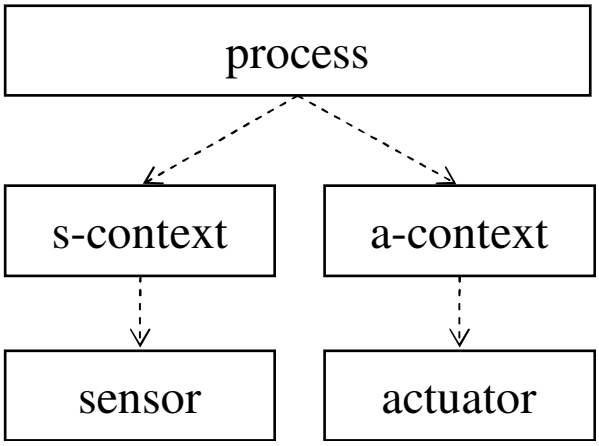


Figure 4.1: Model by the layered architecture

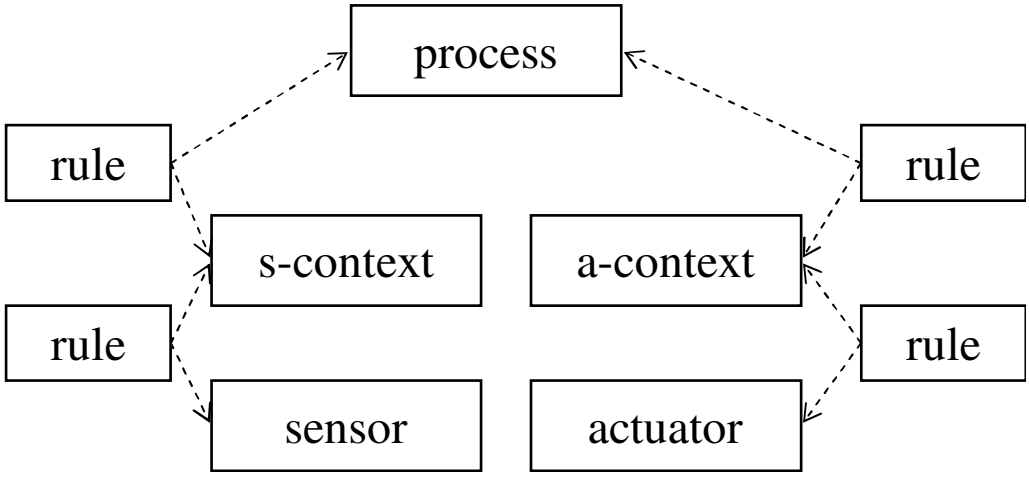


Figure 4.2: Model by the proposed mechanism

anism comes from the awareness of the issue that we need to increase the reusability and modifiability of embedded software design, with the increase of its complexity. To meet such quality attribute requirements, we have to consider them already at the architecture design phase, not to leave them until the detailed design phase. Therefore, our main target phase of the mechanism is the architecture design phase.

A typical development flow is, we make architectural design based on our modeling mechanism, and then implement the system based on the design. As there are gaps between the execution semantics of our modeling mechanism and that of actual software platforms, we need to examine implementation strategy.

There can be several approaches to implement software from the model.

- Approaches independent of implementation languages:

  Model Driven Development (MDD) approach An approach that places models as the center of the development [10][23][38] and in most cases implementation code is generated by tools internally.

  Explicitly weaving approach A weaved (composed) model is generated by the developers or tools. Weaving (composition) of aspects is done by extracting event introduction and guard condition reference described in rules on state diagrams in all aspects.

  Because the modeling mechanism is based on state transitions, code can be generated (by the developers or tools) in both approaches, using already known technologies of code generation (translation) from state transition models [39].

- Approaches dependent on aspect-oriented programming (AOP) languages: we could utilize AOP languages. For example, if we implement software with AspectJ, we describe aspect-relation-rules as "aspects" of AspectJ; with Hyper/J, we implement each aspect of the mechanism as a "hyperslice" and define a composition of Hyper/J based on aspect-relation-rules. However, we have to consider deeply whether AOP languages are appropriate to implement embedded software or not.

We believe, in the future, the MDD approach can be useful and promising.

Generally speaking, most MDD approaches [10][23][38] have merits on embedded software development, and the same merits seems to be valid using the proposed modeling mechanism.

And using the MDD approach, we can make use of the important characteristic of the modeling mechanism; the mechanism makes every aspect independent from each other. In the MDD approach, the model is the front end to software developers, and generated code is basically handled by tools internally, and the developers need not to mind it. Hence, the developers can fully enjoy the merits of our modeling mechanism. This scheme is similar to the programming language world, in which programmers look at source code and do not need to care about the machine language.

On the other hand, with the explicitly weaving approach, software developers have to look at weaved (configured) model and implement the system. In the weaved model, each part binds to each other and modularization of each aspect is collapsed. Based on the above reason, we believe the MDD approach is most promising.

Chapter 5

Case Study of Embedded Software

In this chapter, we explain the proposed approach using an example of embedded software. This example is based on an actual embedded software application provided by a company; however, in this thesis, it is simplified for explanation purposes.

5.1 System description

As the example, we use an embedded software application for vehicle illumination that controls the interior lights of an automobile based on the statuses of the front doors (the driver door and the front passenger door) and the power switch. The basic functionality of the system is illumination control—it turns on the interior light when a door is opened and turns it off when all doors are closed. Another functionality of the system is the "battery saver" that prevents the car battery from discharging; more specifically, when the power switch is turned off and a door is opened, the interior light is turned on at once but it is turned off after a certain period of time.

5.2 Modeling strategy

Before describing the example in detail, we explain the modeling objective and the modeling strategy. We also briefly outline the application of our aspect-oriented modeling mechanism for designing the example system.

Objective

The objective is to make the architecture of the system modifiable.

There are multiple types of door sensors (sensors for the doors), power sensor (sensor for the power switch) and light actuator (actuator for the interior light), and which of them are actually used may be changed. The system should be modifiable against these possible changes.

Strategy

In order to make the system modifiable, we apply our aspect-oriented context modeling for its modeling; "contexts" are recognized and separated from sensors and actuators,

and they are placed between the sensors/actuators and internal processing to loosen the coupling among them.

This system reacts the changes of the door status and the battery status, and tries to changes the status of the light. These statuses are modeled as the contexts; door context (statuses of the doors), battery context (status of the battery), and light context (status of the interior light).

Modeling overview

Based on the strategy described above, we model the system utilizing our aspect-oriented modeling mechanism. We define each sensor, context and processing as aspect, and relate them by aspect-relation rules.

For example, we define door sensors, door context and lighting control as aspects, and relate them in terms of aspect-relation rules.

Likewise, we define power sensor, light actuator, battery context, light context, and battery saver as aspects, and relate them in terms of aspect-relation rules.

5.3 Aspect-oriented modeling

In this section, we will explain the detail of the design that follows the modeling strategy mentioned above.

We explain the application of our modeling mechanism in the following order.

1. identify aspects
2. identify relationships among aspects
3. design aspects
4. define aspect-relation rules

1. identify aspects

sensor aspects

The system has five types of sensors related to the control of interior lights—power sensor, driver-door sensor, front-passenger-door sensor, driver-door-lock sensor, and front-passenger-door-lock sensor. Each of these sensors detects the statuses of their corresponding entities (e.g., the driver door sensor detects the open/closed status of the driver door). These sensors are modeled as two sensor aspects—"POWER" and "DOOR." The POWER aspect includes the "Power" class, and the DOOR aspect includes the following classes: "DriverDoor," "PassengerDoor," "DDLock" (driver-door-lock sensor), and "PDLock" (passenger-door-lock sensor). The Power class senses the power on/off status. In this example, not all the doors but only the driver door and front-passenger door influence the behavior of the system. Therefore, the DOOR aspect includes only these classes.

s-context aspects

Two contexts identified by sensors—s-contexts—are required to control the interior lights; the statuses of the doors and the battery. Because the interior light is turned on and off according to the status of the doors (whether all the doors are closed or not), the context of the door status must be captured. In addition, this system has a battery saver function for which the status of the battery (whether or not the battery is charged) must be captured in addition to the door status. These contexts are modeled as two s-context aspects—"DOOR-ST" and "BTRY-ST," which include the "DoorSt" and "BtrySt" classes, respectively.

process aspects

The internal processes of this system comprise illumination control and battery-saver functionalities. The illumination-control functionality is essentially employed to switch the interior lights on and off. The battery-saver functionality is employed to prevent the battery from discharging—it restricts the lights from remaining on continuously and prevents lights from being turned on if necessary. These internal processes are modeled as two process aspects—"LCONTROL" and "BTRYSVR," which include the "LControl" and "BSaver" classes, respectively.

a-context aspects

The light has one context that should be controlled by the system; that is whether the light is on or off. This is modeled as a-context aspect "ON-OFF," which includes the "OnOff" class.

actuator aspects

The interior light is modeled as an actuator aspect, which is a hardware wrapper for physical light. This actuator is modeled as the "LIGHT" actuator aspect, which includes the "Light" class.

2. identify relationships among aspects

Based on the aspects identified in the previous step, we analyze the system to identify the relationships among aspects. We examine how these aspects are configured to the system, and determine basic configuration for it.

The followings are relations for this illumination system. Relations are numbered for readability reason.

- RS1: between a-context aspect "ON-OFF" and actuator aspect "LIGHT." If the state of "ON-OFF" aspect changes, it may cause changes the state of "LIGHT," i.e. actually turn on/off the light.
- RS2: between process aspect "BTRYSVR" and a-context aspect "ON-OFF." If the state of "BTRYSVR" aspect changes, it may cause changes the state of "ON-OFF," i.e. the battery saver may turn off the light. Also, the change of the state of "ON-OFF" aspect may be affected by the state of "BTRYSVR" aspect, i.e. if the battery saver restricts lighting, the light status should not be "on."

- RS3: between process aspect "LCONTROL" and a-context aspect "ON-OFF". If the state of "LCONTROL" aspect changes, it may cause changes the state of "ON-OFF," i.e. the light control controls the on/off-status of the light.

- RS4: between s-context aspect "BTRY-ST" and process aspect "BTRYSVR." If the state of "BTRY-ST" aspect changes, it may cause changes the state of "BTRYSVR," i.e. charging status of battery affects the behavior of the battery saver. Also, the change of the "BTRYSVR" aspect may be affected by the state of "BTRY-ST" aspect, i.e. the battery saver refers the charging status of battery.

- RS5: between s-context aspect "DOOR-ST" and process aspect "BTRYSVR." If the state of "DOOR-ST" aspect changes, it may cause changes of the state of "BTRYSVR," i.e. if the door opens, the battery saver starts working.

- RS6: between s-context aspect "DOOR-ST" and process aspect "LCONTROL". If the state of "DOOR-ST" aspect changes, it may cause changes the state of "LCONTROL," i.e. the light control controls the light status depending on the door status.

- RS7: between sensor aspect "POWER" and s-context aspect "BTRY-ST." If the state of "POWER" aspect changes, it may cause changes the state of "BTRY-ST," i.e. sensor value changes may cause context changes.

- RS8: between sensor aspect "DOOR" and s-context aspect "DOOR-ST." If the state of "DOOR" aspect changes, it may cause changes the state of "DOOR-ST," i.e. sensor value changes may cause context changes. Also, the change of the state of "DOOR-ST" aspect may be affected by the state of "DOOR" aspect, i.e. value of door sensors are referred in deciding whether all doors close or not.

Figure 5.1 shows the overview of aspects and relationship among them in the form of the aspect-relation overview diagram introduced in 3.3.2.

3. design aspects

Next, we design each aspect separately. Figure 5.2 to Figure 5.7 show the results of the design. Figure 5.2 shows the DOOR aspect. The description of this aspect includes a class diagram that shows the four sensors and the relationships among them and state diagrams that depict the behavior of each class.

Figure 5.3 shows the POWER aspect.

Figure 5.4 shows the DOOR-ST and BTRY-ST aspects.

Figure 5.5 shows the LCONTROL and BTRYSVR aspects. The LControl class controls illumination—it switches the light on and off when its states are On and Off, respectively. It changes its state based on the on and off events. The BSaver class prevents the battery from discharging—it switches the light off after a certain period of time and prevents it from being turned on again. This class has three states—"Idle" (not activated), "Limiting" (activated and counting down), and "Forbidding" (lights off and prevents the light from being turned on). When the "save" event is introduced, the BSaver class is activated; then, after a certain period of time, the BSaver class enters the Forbidding state. Finally, upon the introduction of the "release" event, it resets itself.

Figure 5.6 shows the ON-OFF aspect.

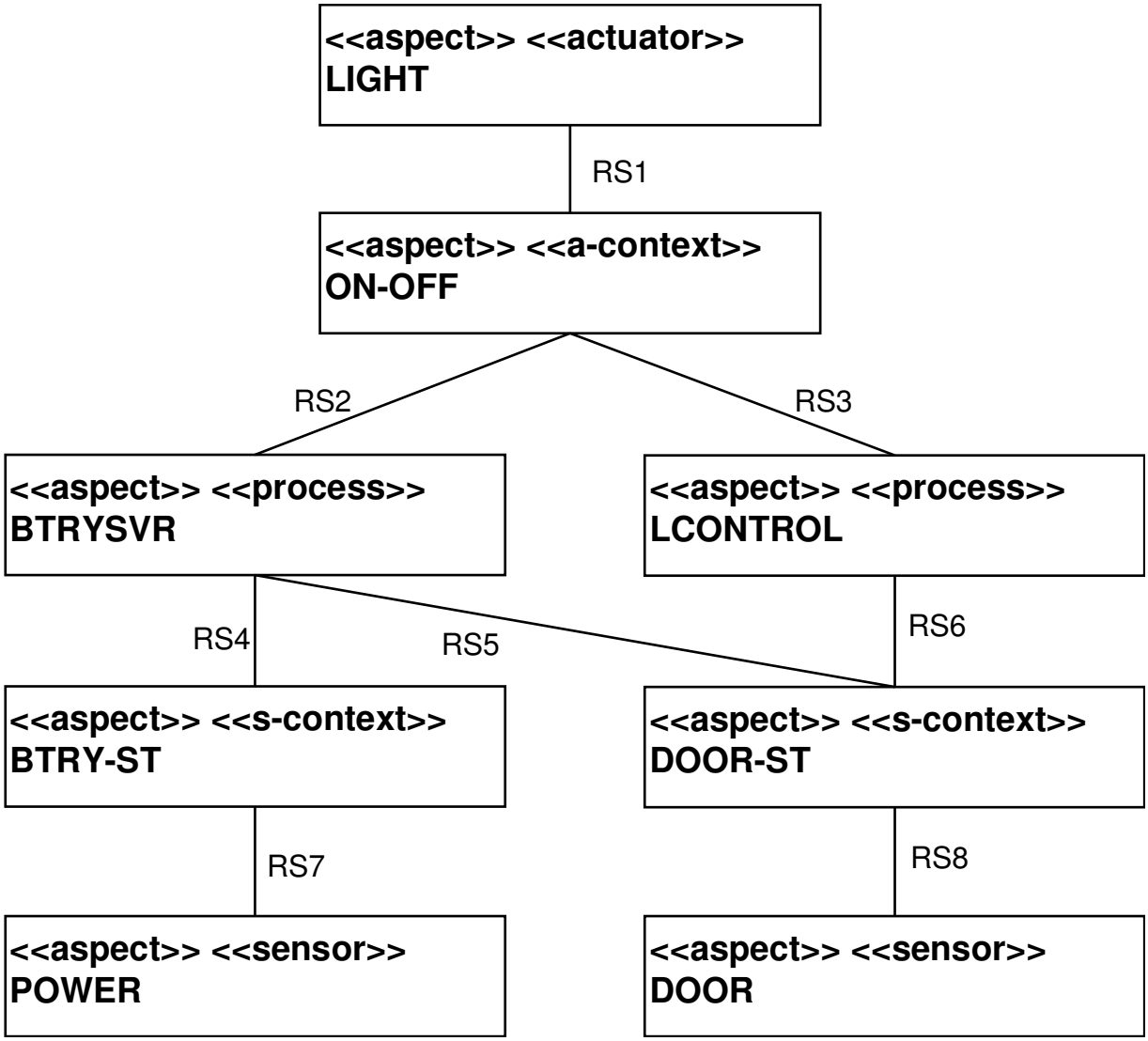


Figure 5.1: Aspect-relation overview diagram for the example

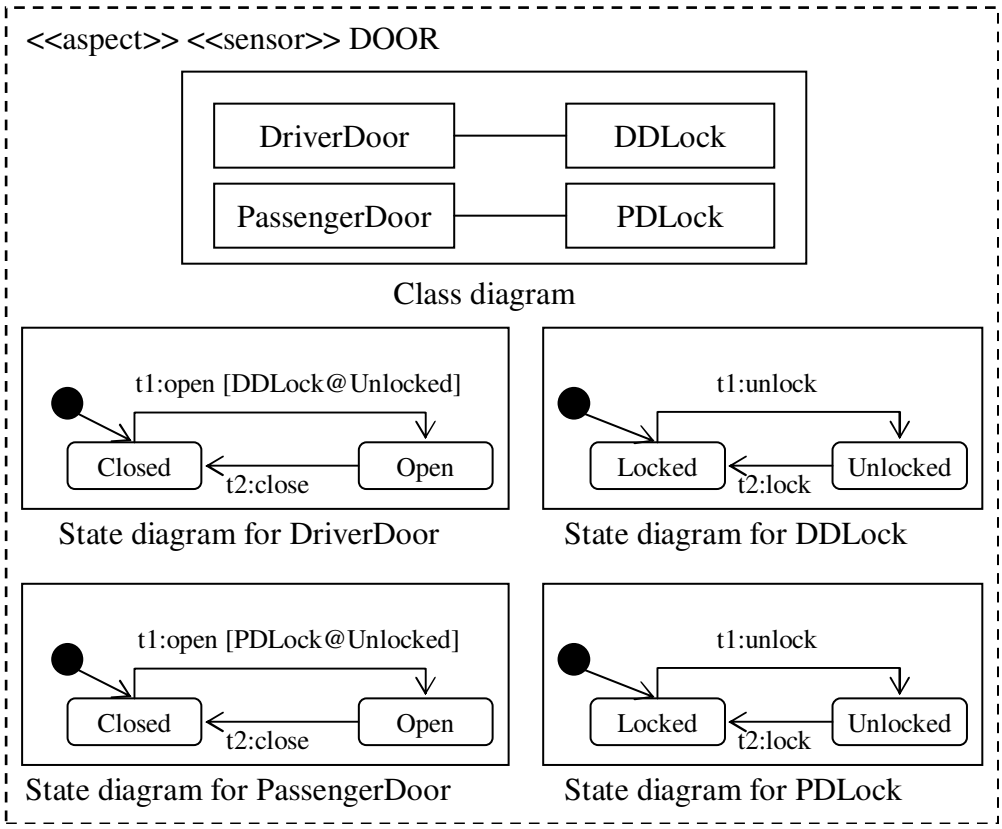


Figure 5.2: Sensor aspect—DOOR

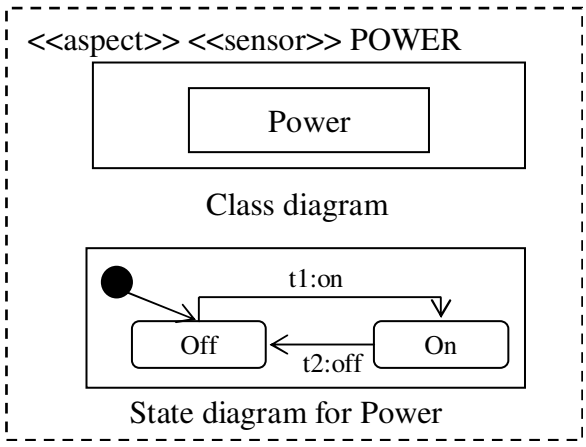


Figure 5.3: Sensor aspect—POWER

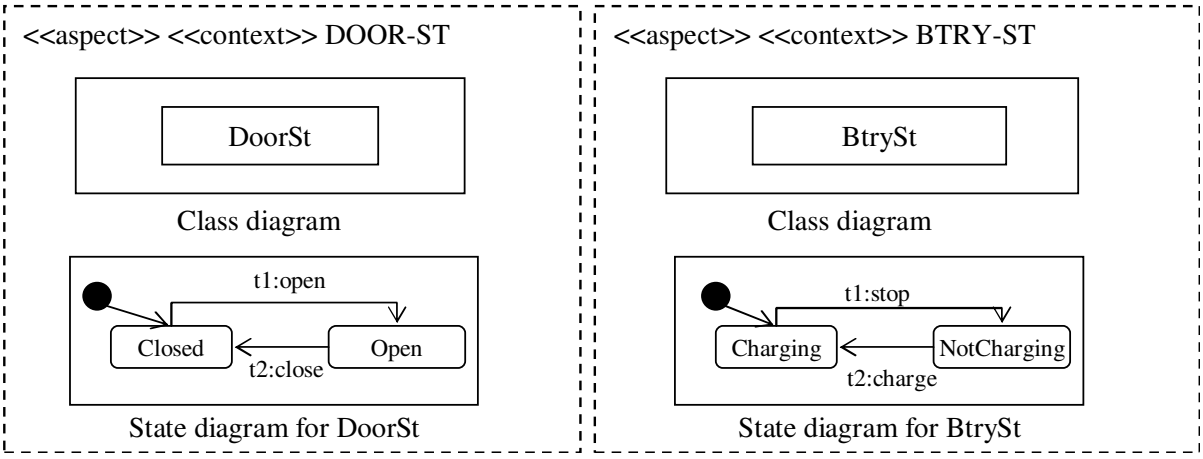


Figure 5.4: S-context aspects—DOOR-ST and BTRY-ST

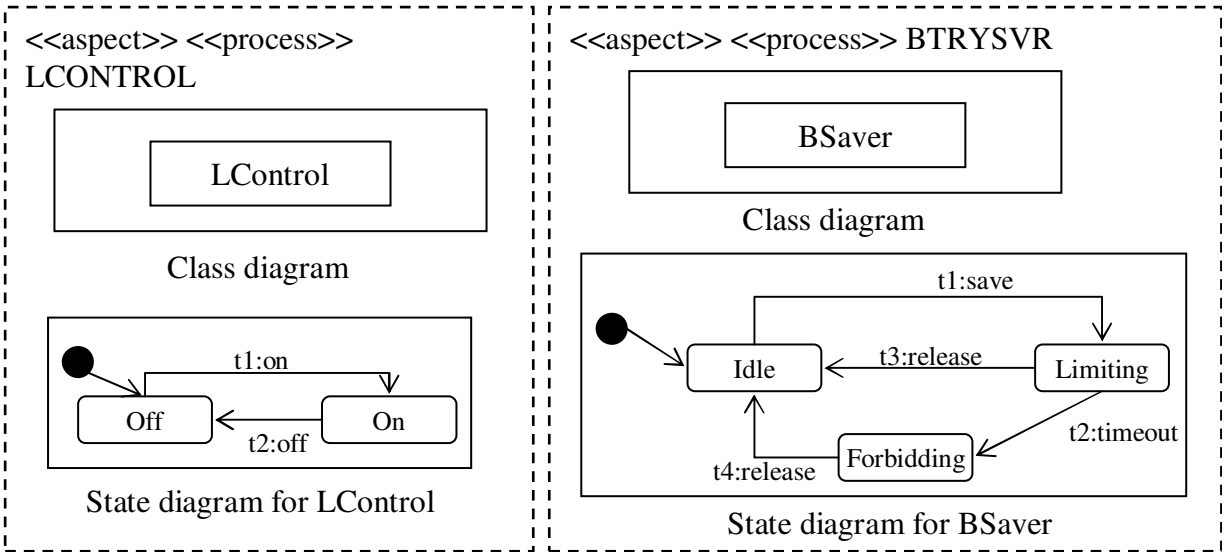


Figure 5.5: Process aspects—LCONTROL and BTRYSVR

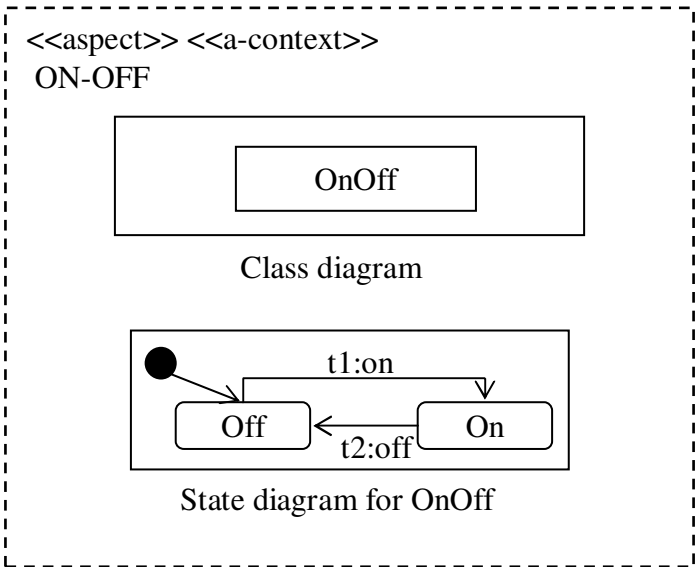


Figure 5.6: A-context aspects—ON-OFF

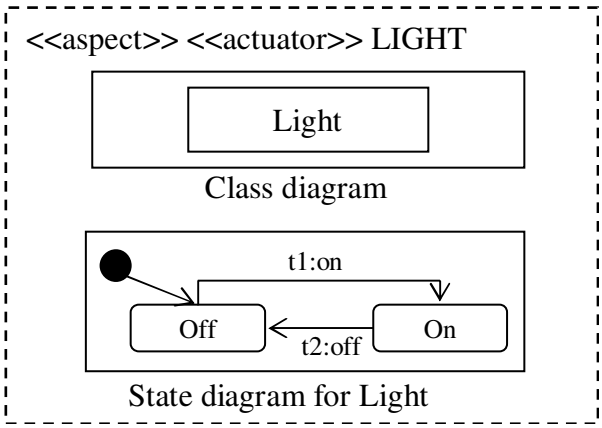


Figure 5.7: Actuator aspect—LIGHT

Figure 5.7 shows the LIGHT aspect. The Light class is a hardware wrapper for physical light. It has On/Off states that correspond to the actual light status.

4. define aspect-relation rules

To compose these aspects into the entire system, we define the details of aspect-relations in terms of the aspect-relation-rules introduced in Chapter 3. Figure 5.8 shows the aspect-relation-rules for this system.

The name of each rule is the same as that are used in the aspect-relation overview diagram (Figure 5.1).

These rules realize the behavior of the entire system. For example, when the open event occurs under the condition that DDLock is unlocked and transition t1 fires at DriverDoor in DOOR, the open event is introduced to DoorSt in DOOR-ST (by rule 1 of RS8); this causes transition t1 at DoorSt in DOOR-ST (this state transition is shown by the state

RS1

1. ON-OFF.OnOff:t1 -> on^LIGHT.Light
2. ON-OFF.OnOff:t2 -> off^LIGHT.Light

RS2

1. BTRYSVR.BSaver:t2 -> off^ON-OFF.OnOff
2. ON-OFF.OnOff:t1 [BTRYSVR.BSaver@Idle]

RS3

1. LCONTROL.LControl:t1 -> on^ON-OFF.OnOff
2. LCONTROL.LControl:t2 -> off^ON-OFF.OnOff

RS4

1. BTRY-ST.BtrySt:t1 -> save^BTRYSVR.BSaver
2. BTRY-ST.BtrySt:t2 -> release^BTRYSVR.BSaver
3. BTRYSVR.BSaver:t1 [BTRY-ST.BtrySt@NotCharging]

RS5

1. DOOR-ST.DoorSt:t1 -> save^BTRYSVR.BSaver
2. DOOR-ST.DoorSt:t2 -> release^BTRYSVR.BSaver

RS6

1. DOOR-ST.DoorSt:t1 -> on^LCONTROL.LControl
2. DOOR-ST.DoorSt:t2 -> off^LCONTROL.LControl

RS7

1. POWER.Power:t1 -> charge^BTRY-ST.BtrySt
2. POWER.Power:t2 -> stop^BTRY-ST.BtrySt

RS8

1. DOOR.*Door:t1 -> open^DOOR-ST.DoorSt
2. DOOR.*Door:t2 -> close^DOOR-ST.DoorSt
3. DOOR-ST.DoorSt:t2 [DOOR.*@Closed]

Figure 5.8: Aspect-relation-rules

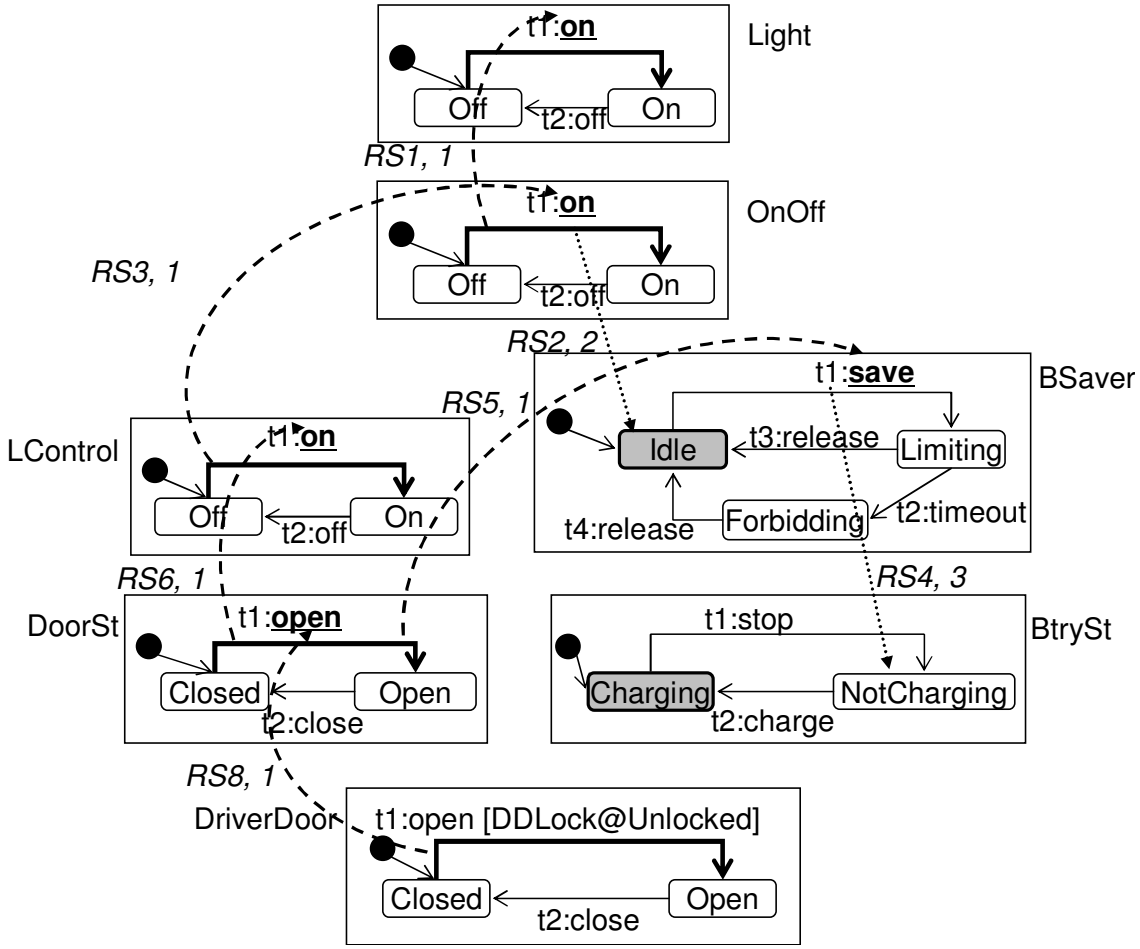


Figure 5.9: Schematic of system behavior

diagram for DoorSt in Figure 5.4). This, in turn, introduces the on event to LControl (by rule 1 of RS6) and it results in the triggering of state transition t1 of LControl (see Figure 5.5). This transition at LControl introduces the on event to OnOff (by rule 1 of RS3); simultaneously, transition t1 at DoorSt introduces the save event to BSaver (by rule 1 of RS5). If BtrySt is in the NotCharging state, the save event triggers transition t1 at BSaver (by rule 3 of RS4), thereby changing its state to Limiting (see Figure 5.5). Thus, OnOff in ON-OFF context keeps in the state of Off, because transition t1 fires only when BSaver is in the Idle state (by rule 2 of RS2); on the other hand, if BtrySt is in the Charging state, transition t1 fires, and the on event to Light (by rule 1 of RS1) and Light is lit. Figure 5.9 shows this sequence schematically as an aide for understanding.

Chapter 6

Application to Software Product Line Architecture Design

6.1 Architecture design of software product line

In PLD, it is important to make architecture and components flexibly configurable. AOTs are considered to be promising techniques for the development of such reusable core assets [20][24] [36]. The addition or deletion of a feature may cause crosscutting changes in the design and implementation of core assets, and AOTs are expected to effectively handle these crosscutting changes. However, the application of AOTs to PLD is not that simple.

One of the problems in applying AOTs to PLD is the management of crosscutting relationships [8][19]. In most of the existing AOTs, a crosscutting concern is modeled or implemented by a module in which functionalities and crosscutting relationships are encapsulated together. This strong dependency between the functionalities and crosscutting relationships causes a problem because the same functionalities may have different crosscutting relationships, depending on conditions.

As we have introduced, one of the major characteristics of our aspect-oriented modeling mechanism is that we separate aspect-relation-rules (crosscutting relationships) from aspects (functionalities). This characteristic facilitates the management of crosscutting relationships mentioned above, and we can effectively apply AOT to develop architecture and components highly configurable. In other words, we can effectively manage variabilities in design model utilizing AOT.

In this chapter, we propose a variability management technique in which the aspect-oriented modeling mechanism proposed in Chapter 3 is used to design the product line architecture (PLA).

6.2 Separating functionalities and crosscutting relationships

One of the problems in applying AOTs to PLD is the management of crosscutting relationships. This problem addresses the situation in which the same concern has different crosscutting relationships, depending on conditions. For example, it has been pointed out that the addition or deletion of a feature may cause a change in the parts of assets that depend on the feature as these parts include dependencies related to the variable feature

[19]. In the example of an elevator in [19], the cancellation feature is realized as an aspect whose pointcut designates every call services. However, if an emergency call feature is added, the pointcut must be changed so as to avoid cancelling the emergency call.

In most AOTs, a crosscutting concern is modeled or implemented by a module in which the functionalities (that are necessary to model/implement the concern) and the crosscutting relationships are encapsulated together. In AspectJ, for example, *advices* (that realize the functionalities) and *pointcuts* (that describe the relationships with others) are encapsulated in an aspect. In this type of AOTs, the abovementioned problem easily arises. Several implementation techniques (applicable at the programming level) have been proposed in order to prevent this problem. However, solutions implemented at the programming level are not sufficient. From the viewpoint of variability management, it is important to examine the crosscutting relationships among the functionalities and to design a PLA that presents a strategy on how functionalities and crosscutting relationships are realized.

The aspect-oriented modeling mechanism proposed in Chapter 3 encapsulates crosscutting concerns by using a mechanism similar to a "hyperslice" of Hyper/J [3]. This modeling mechanism enables us to separate functionalities and crosscutting relationships. We propose a variability management technique in which our modeling mechanism is used to design the PLA. In the following sections, we explain our approach.

6.3 Application of our mechanism to PLD

In this section, we briefly outline the application of our modeling mechanism to PLD.

In the PLA, generally, a feature corresponds to some modules, which realize the necessary functionalities for the feature.

In our modeling mechanism, the functionalities and crosscutting relationships among them are defined separately: aspects realize functionalities and aspect-relation-rules define the relationships among the aspects. We use these aspects and their relationships as the building blocks of the PLA. Therefore, in our PLA, a feature corresponds to some aspects and their relationships with other aspects.

In order to relate two aspects, an aspect-relation, that is defined by (in general) multiple aspect-relation-rules, is required. These multiple aspect-relation-rules are managed as a unit of *rule set*.

A feature can be optional or alternative; accordingly, optional or alternative aspects and rule sets are defined in the PLA. In the next section, we will explain how to describe these variants.

6.4 Product line architecture diagram

To depict the PLA, we introduce a PLA diagram. In our PLA diagrams, variants appear as variant aspects and variant rule sets. We explain the basic notation of the PLA diagram in 6.4.1, variant aspects in 6.4.2, and variant rule sets in 6.4.3.

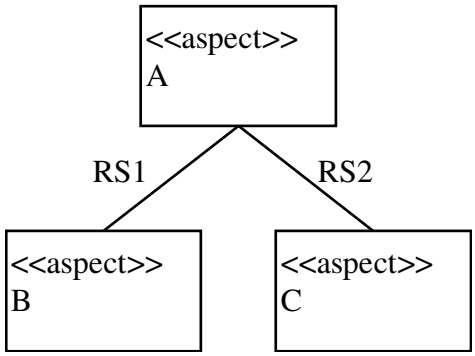


Figure 6.1: Example of PLA diagram

6.4.1 Basic notation

In the PLA diagrams, an aspect is represented as a box with the stereotype "<<aspect>>." A rule set is represented as a line between the aspects. Each rule set has an identifier that can be indicated near the rule set line. Figure 6.1 shows an example of a PLA diagram.

6.4.2 Variant aspect

There are *optional* aspects and *alternative* aspects.

Optional aspect: An optional aspect may or may not be used for the selected product architecture. The stereotype <<optional>> is added for the optional aspect in a PLA diagram.

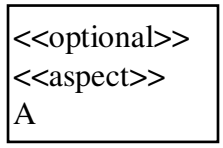
Alternative aspect: Among a group of alternative aspects, only one aspect must be chosen for the selected product architecture. The aspects that are grouped are shown by constraint notation. If two aspects are present in the group, both the aspects are connected by a dashed line with arrowheads at both the ends, and the text string "alternative" is placed near the line. Further, if three or more aspects are present in the group, a note containing the string "alternative" is attached to each aspect of the group with a dashed line.

Figure 6.2 (a) and (b) show the example of optional and alternative aspects, respectively.

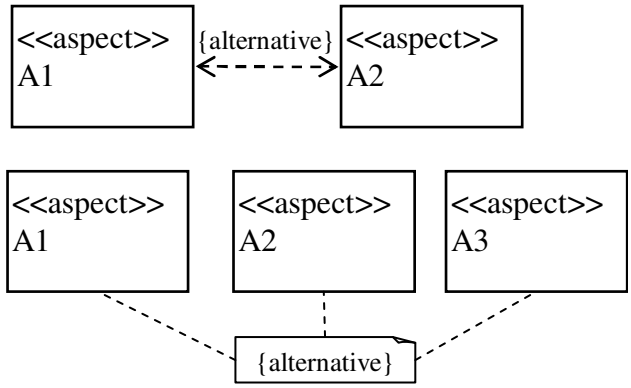
6.4.3 Variant rule set

There are *optional* rule sets and *alternative* rule sets.

Optional rule set: An optional rule set may or may not be used for the selected product architecture. The stereotype <<optional>> is added for the optional rule set in a PLA diagram. A rule set related to an optional aspect is optional. Therefore, the stereotype <<optional>> can be omitted in this case. There can be optional rule sets among the mandatory (neither optional nor alternative) aspects; in this case, the stereotype <<optional>> cannot be omitted. Figure 6.3 (a) shows the examples of optional rule sets.



(a) optional aspect



(b) alternative aspect

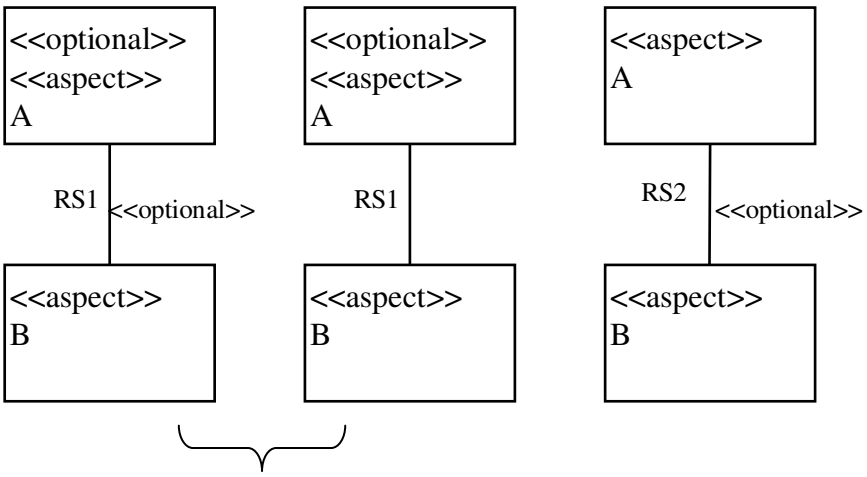
Figure 6.2: Optional and alternative aspects

Alternative rule set: Among a group of alternative rule sets, only one rule set must be chosen for the selected product architecture. The rule sets that are grouped are shown by the constraint notation. If two rule sets are present in the group, both the rule sets are connected by a dashed line with arrowheads at both the ends, and the text string "alternative" is placed near the line. Further, if three or more rule sets are in the group, a note containing the string "alternative" is attached to each rule set of the group with a dashed line. Figure 6.3 (b) shows the examples of alternative rule sets. A rule set related to an alternative aspect is alternative. There can be alternative rule sets among non-alternative aspects.

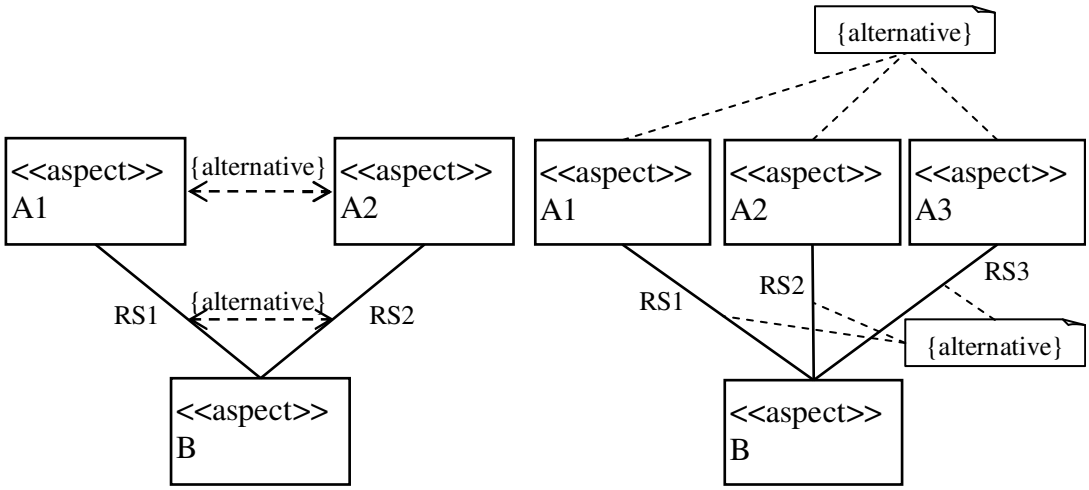
Each aspect design (class diagram and state diagrams) and the details of each rule in the rule sets are described apart from the PLA diagram.

6.4.4 Example of PLA diagram

Figure 6.4 shows a part of an example of a PLA diagram and a rule set description. The aspect SAFECTRL and SRVC are the same as those in Figure 3.4. These aspects are mandatory used for all products in the PL, but how to relate these aspects can be different by the chosen product, because there are two alternative rule sets RS1 and RS2 between them. The aspect-relation-rules in the rule set RS1 are as the same as those shown in Figure 3.4. By RS2, an "off" event is introduced from SAFECTRL to SRVC at a different timing, and the service can be started only when the class SCtrl of aspect SAFECTRL is in the "idle" state. This behavior is safer than that realized by RS1.

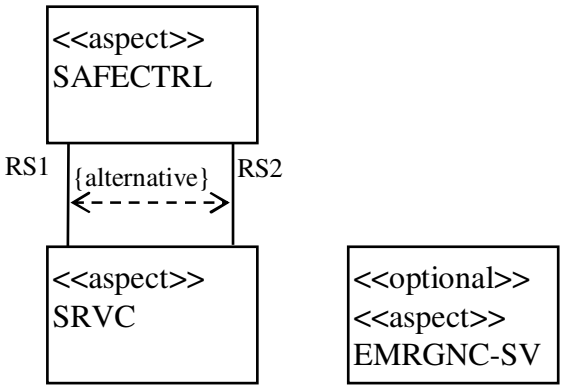


(a) optional rule set



(b) alternative rule set

Figure 6.3: Optional and alternative rule sets



(A part of) PLA diagram

RS1
...
RS2
1. SAFECTRL.SCtrl:t1 -> off^SRVC.Srvc
2. SRVC.Srvc:t1 [SAFECTRL.SCtrl@idle]

Rule set description

Figure 6.4: Example of PLA design

There is an optional aspect EMRGNC-SV in the PLA diagram. In some products, this aspect is chosen and provides an emergency-related service. Further, this aspect has no relationships with the aspect SAFECTRL; therefore, this service is not interrupted by SAFECTRL.

6.5 Case study: vehicle illumination product line

In this section, we explain the proposed approach using an example of embedded software product line (PL). The example we use is an embedded software PL for vehicle illumination that controls the interior lights of an automobile based on the statuses of the doors. Some products in this PL may have the "battery saver" function that prevents the car battery from discharging. And some products may have the "fade control" function; the light is faded in and out gradually, instead of turned on and off immediately. Also, there is a sensor variation in this PL; different sensors may be used to realize same functionality. The software introduced in Chapter 5 is the software of one product included in this PL. This example is based on an actual embedded software PL provided by a company; however, in this thesis, it is simplified for explanation purposes.

Before describing the example in detail, we explain the modeling objective and modeling strategy. We also briefly outline the modeling of this PL.

- **Objective:** The objective is to define PLA so as to make components highly configurable, i.e. to realize effective product derivation.

There are sensor variations in this PL. The possible changes caused by these variations should be localized. Furthermore, functionalities are different in each product. Sensors/actuators and functionalities cross cut each other; a function may use multiple sensors, and a sensor/actuator may be used by multiple functions. For example, the battery saver uses door sensors and a power sensor. and the door sensors are used by the basic lighting control functionality and the battery saver. PLA should manage these relationships in order to attain flexible configurability.

- **Strategy:** We apply our aspect-oriented modeling mechanism to manage the crosscutting relationships. Furthermore, we analyze commonalities and variabilities of products in this PL, and systematically manage components (aspects and rule sets) utilizing the PLA diagram.

  Products in this PL react the changes of the door status and the battery status, and tries to changes the status of the light. These statuses are modeled as the contexts; door context (statuses of the doors), battery context (status of the battery), and light context (status of the interior light). Following the aspect-oriented context modeling, we can model each product in terms of aspects (sensors, actuators, contexts, and processes) and relationships among them.

  Though some aspects are shared by every products in the PL, some aspects are only used by certain products; these variable aspects are expected flexibly added or removed. In most existing AOTs, crosscutting relationships are not separated from functionalities, and this lowers configurability. Utilizing our aspect-oriented modeling mechanism, aspects do not depend on other aspects and aspect-relation-rules encapsulate crosscutting relationship, i.e. we can effectively manage crosscutting relationship. We configure various products in this PL utilizing aspect-relation-rules and identify rule sets that are used as units when we configure each product.

  For example, some products have the battery saver, but others do not. We define a battery saver aspect, and also identify rule sets each of which relates the battery saver aspect with other aspects (aspects for the battery context, the door context, and the light context, respectively). If we need the battery saver, we add the battery saver aspect with these rule sets. If we do not need it, we just remove the battery saver aspect and these rule sets. Note that we do not have to modify any aspects and rule sets depending on system configurations. We use the PLA diagram to depict PLA in which mandatory, optional and alternative components (aspects and rule sets) are defined.

  Likewise, we can manage other variabilities such as various sensor types.

- **Modeling overview:** Based on the strategy described above, we model PLA for the PL. By analyzing products in PL, we identify necessary sensors, contexts and processing. Then we define each of them as aspect, and also define aspect-relation-rules.

  In this PL, there are two types of door sensors; we define each of them as aspect, and define aspect-relation-rules between the door context aspect and the door sensor aspects.

  This PL has two ways of lighting; simple on/off and fade-in/fade-out. We prepare two contexts, one for on/off and one for fade-in/fade-out, and define aspect-relation-

rules that relate these aspects with other aspects (the light actuator, a battery saver process, and a lighting control process).

Also, we define a battery saver aspect for the battery saver process, and define aspect-relation-rules that relate the battery saver aspect with other aspects (the battery context, the door context, the on-off context and the fade-in/fade-out context).

These aspects and aspect-relation-rules are used to configure each product in PL. We analyze commonalities and variabilities and how these aspects and aspect-relation-rules are used to configure each product. We group aspect-relation-rules that are used together when we configure products in terms of rule set. Some aspects and rule sets are mandatory, i.e. they are used by every products. Other aspects and rule sets are optional or alternatively used. Based on this examination we define PLA and depict it in terms of the PLA diagram.

The detail of the design is explained in the following order:

1. identify features
2. identify aspects and design each aspect
3. design PLA

1. identify features

At first, we analyze what features the PL has and describe them in a feature model.

The basic and common functionality of the system is illumination control—it turns on the interior light when a door is opened and turns it off when all doors are closed.

An optional feature of the PL is the "battery saver" that prevents the car battery from discharging; more specifically, when the power switch is turned off and a door is opened, the interior light is turned on at once but it is turned off after a certain period of time.

In some products for luxurious cars, a "fade control" feature is necessary. This feature fades the light in and out gradually, instead of turning it on and off immediately.

There are variations in door sensors. In some products, each door has its own door sensor and each door sensor does not know the status of other doors; other products use a different type of a door sensor that directly senses the situation wherein all doors are closed.

The software introduced in Chapter 5, one of the software of one product included in this PL, has the battery saver feature, but does not have the fade control feature; the statuses of each door is sensed by its own door sensor and the sensor that directly senses the situation wherein all doors are closed is not equipped.

We describe these features in the form of feature model [14]. A feature model of this PL is shown in Figure 6.5.

2. identify aspects and design each aspect

Based on the analysis of the features, we identify aspects of this PL design.

For this identification, we adopt the aspect-oriented context modeling introduced in 2.3 as the basis of categorizing aspects. In the aspect-oriented context modeling, aspects

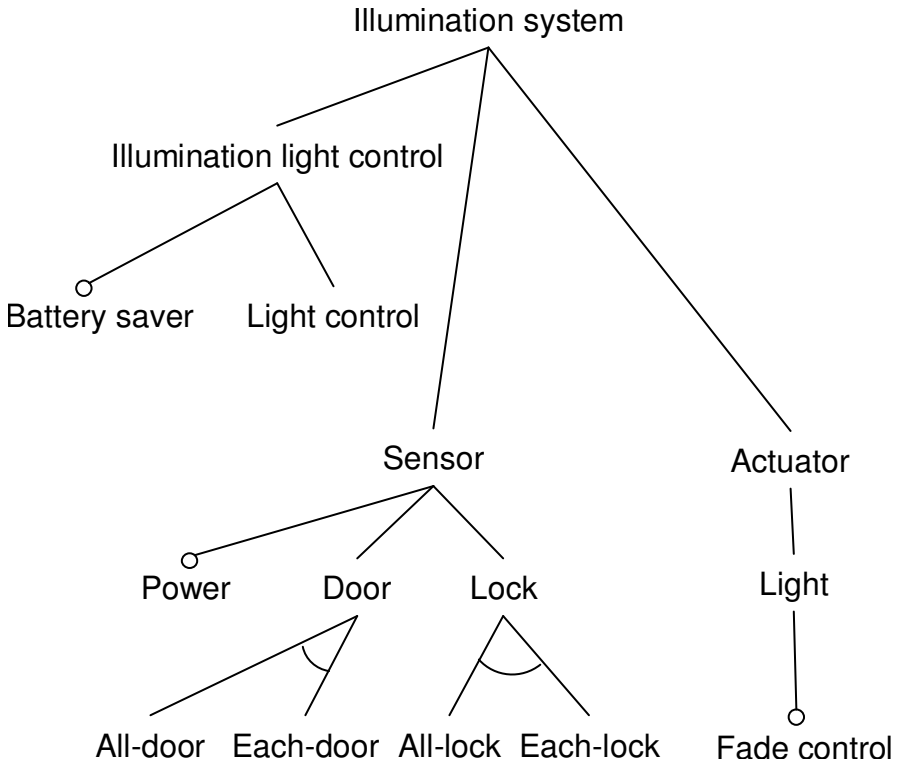


Figure 6.5: Feature model of vehicle illumination PL

are categorized as sensors, contexts (s-contexts and a-contexts), processes, or actuators. This categorization is shown by the stereotypes attached to aspects; <<sensors>>, <<s-context>>, <<process>>, <<a-context>>, and <<actuator>>.

The followings are identified aspects and the design of each aspect.

Actuator aspect: In this PL, the interior light is modeled as an actuator aspect, which is a hardware wrapper for physical light. This actuator is modeled as the "LIGHT" actuator aspect, which includes the "Light" class. Figure 6.6 shows the design of "LIGHT" actuator aspect.

A-context aspect: The light has two contexts that should be controlled by the systems of the PL; one context is whether the light is on or off, and the other context is whether the light is fading in or fading out. These are modeled as a-context aspects "ON-OFF" and "FADE-IN-OUT," which include the "OnOff" and "Fade" classes, respectively. Figure 6.7 shows the design of these aspects.

Process aspect: As process aspects, "LCONTROL" and "BTRYSVR" are designed. The LCONTROL aspect provides the common functionality of illumination control, that is, it turns the interior light on and off. This aspect has the class "LControl." Some products in the PL provides a battery saver function. In order to realize this functionality, the aspect BTRYSVR is designed. This aspect has the class "BSaver." Figure 6.8 shows the design of these process aspects.

S-context aspect: In this PL, because the interior light is turned on and off according to the status of the doors (whether all the doors are closed or not), the door status

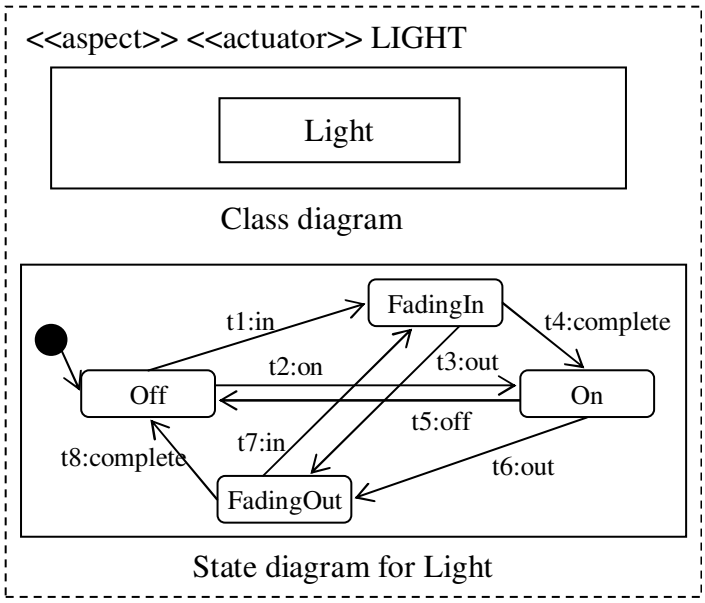


Figure 6.6: Actuator aspect—LIGHT

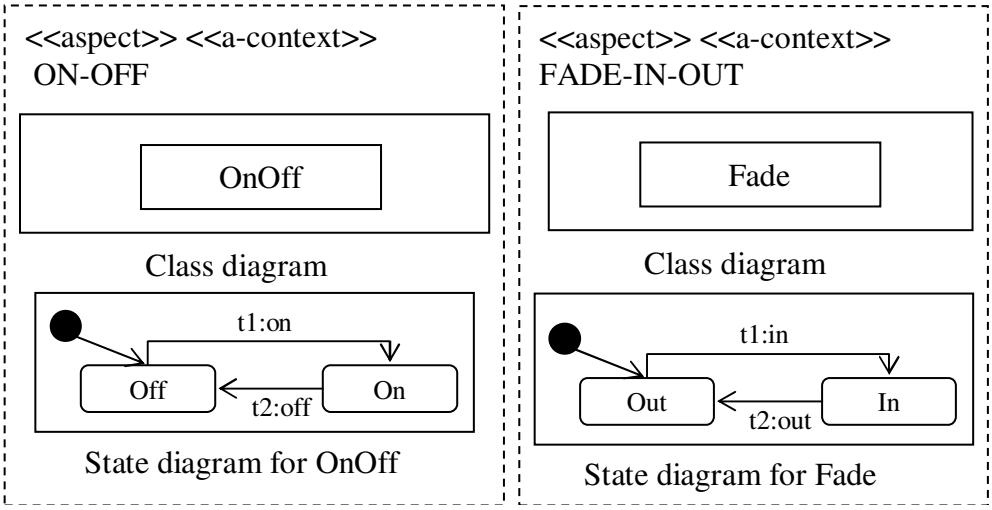


Figure 6.7: A-context aspect—ON-OFF and FADE-IN-OUT

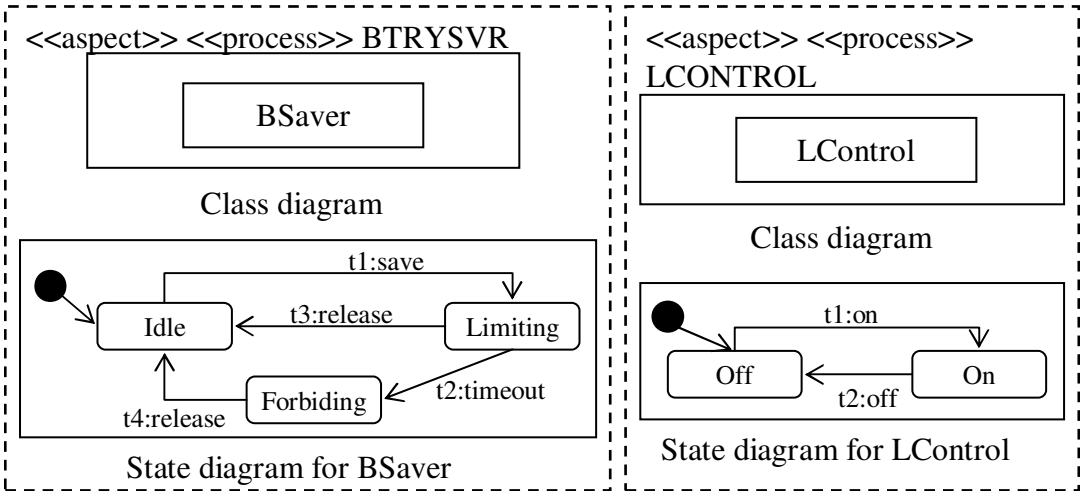


Figure 6.8: Process aspect—LCONTROL and BTRYSVR

must be captured. In addition, some products have a battery saver function for which the status of the battery (whether or not the battery is charged) must be captured in addition to the door status. These are the contexts recognized through the sensors; therefore, they are modeled as the s-context aspects; "DOOR-ST" (the status of the doors) and "BTRY-ST" (the status of the battery), which include the "DoorSt" and "BtrySt" classes, respectively. Figure 6.9 shows the design of these aspects.

Sensor aspect: There are three types of sensors: power sensors, door sensors, and lock sensors. Each of these sensors detects the status of its corresponding entity. In this PL, there is one type of a power sensor. This is modeled as a sensor aspect "POWER," which includes the "Power" class. The door sensor has variations in this PL: a dedicated sensor for each door, and a sensor that directly senses the situation wherein all doors are closed. They are separated as the different aspects "DOOR" and "ALL-DOOR." The lock status is not directly used, but it influences the behavior of the door sensors. Therefore, in this PL, we include lock sensors in the aspects of doors. As a result, the sensor aspect DOOR includes the following classes: "DriverDoor" (driver door sensor), "DLock" (driver-door-lock sensor), "Passenger-Door" (passenger door sensor), and "PDLock" (passenger-door-lock sensor). The sensor aspect ALL-DOOR includes the classes of "AllDoor" and "AllLock." Figure 6.10 shows the design of the sensor aspects.

It is important that all these aspects are designed to be independent and self-contained. For example, because the aspect LCONTROL provides the function that turns the illumination lights on and off according to the on/off status of the doors, it has a potential relationship with the DOOR-ST aspect. However, this relationship is included in neither LCONTROL nor DOOR-ST. Further, nothing dependent on DOOR-ST is included in the design of LCONTROL.

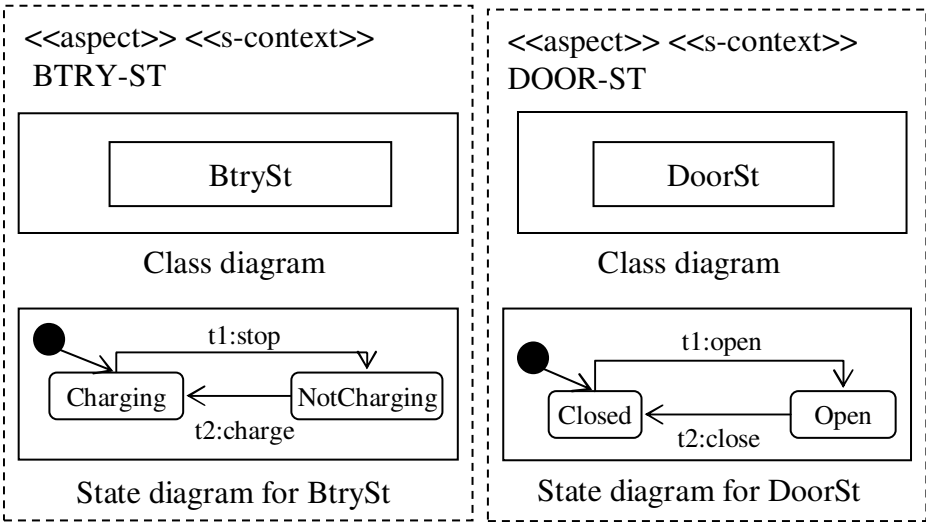


Figure 6.9: S-context aspect—BTRY-ST and DOOR-ST

3. design PLA

Using the identified aspects, we design the PLA.

Figure 6.11 shows the PLA of this vehicle illumination PL. Rule sets are required for relating the aspects. Figure 6.12 shows the required rule sets. RS1, RS2, ..., RS12 are the identifiers of the rule sets and correspond to those in Figure 6.11. From RS1 to RS12 in Figure 6.11 are rule sets. Though they are numbered sequentially, the order is not essential. Some rules are as same as those are defined in Chapter 5, and we put the same number for these rules for readability reason.

Sensor variation: Because the all-door sensor (the sensor that directly senses the situation wherein all doors are closed) and the each-door sensor (the sensor that senses the corresponding door) are alternatively used, as the feature model (Figure 6.5) shows, the ALL-DOOR and the DOOR aspects are alternative. Further, the rule set RS12 that relates the aspects ALL-DOOR and DOOR-ST and the rule set RS8 that relates aspects DOOR and DOOR-ST are alternative.

Battery saver: The battery saver is the optional feature; hence, the aspect BTRYSVR is optional. The rule sets related to it (RS2, RS4, RS5, and RS10) are also optional.

Fade control: In the feature model, the fade control feature is described as optional. In the PLA, corresponding to this feature, the FADE-IN-OUT aspect is designed to be alternative to the ON-OFF aspect because we found that the light fades in (out) or turns on (off) in a specific product. Therefore, the rule sets related to the FADE-IN-OUT aspect are also alternative to the rule sets related to the ON-OFF aspect.

Table 6.1 shows the correspondence between the variant features and the PLA of this PL.

Based on the PLA, various product design can be derived. For example, the architecture of the product intended for most luxurious vehicles includes the aspects FADE-IN-

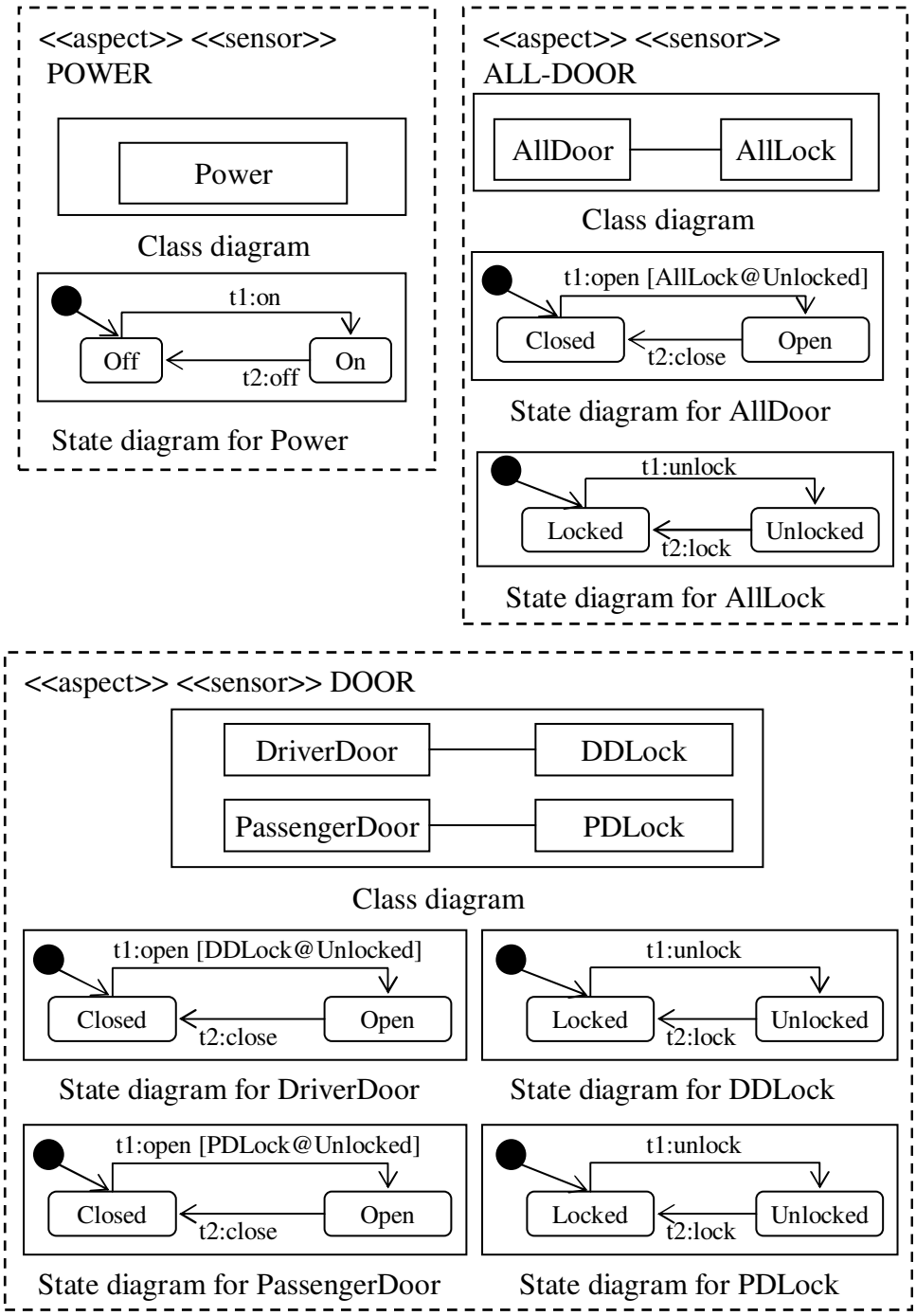


Figure 6.10: Sensor aspect—POWER, ALL-DOOR, and DOOR

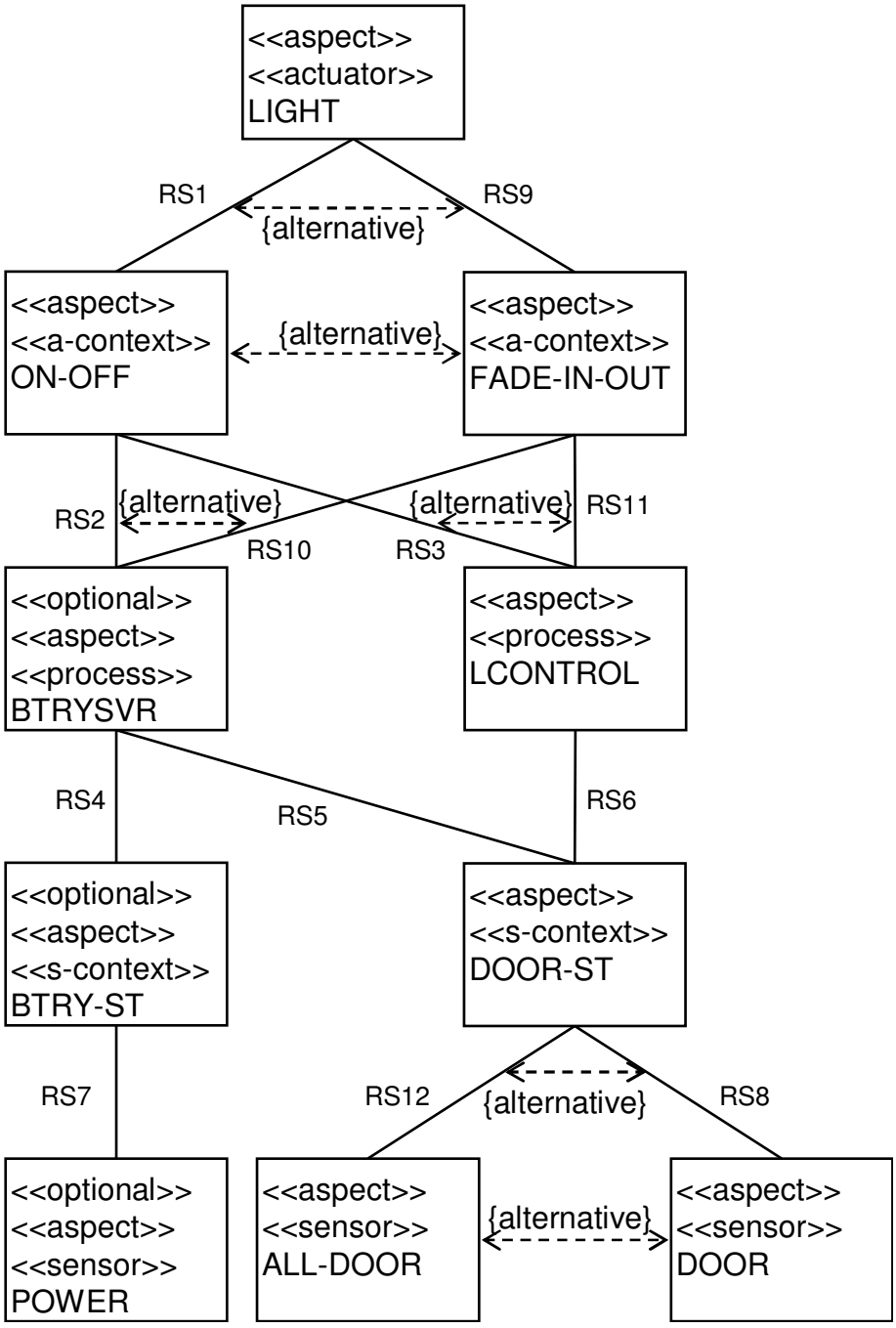


Figure 6.11: PLA diagram of vehicle illumination PL

Table 6.1: Correspondence between variant feature and PLA

Feature	Aspect	rule set
Battery saver	BTRYSVR, BTRY-ST	RS2, RS4, RS5 RS7, RS10
Power sensor	POWER	RS7
All-door sensor	ALL-DOOR	RS12
All-lock sensor	ALL-DOOR	RS12
Each-door sensor	DOOR	RS8
Each-lock sensor	DOOR	RS82
Fade control	FADE-IN-OUT	RS9, RS10, RS11

OUT, BTRYSVR, BTRY-ST, POWER, and ALL-DOOR, and the rule sets RS2, RS4, RS6, RS7, RS8, RS10, and RS11, along with the mandatory aspects and rule sets.

RS1
1. ON-OFF.OnOff:t1 -> on^LIGHT.Light
2. ON-OFF.OnOff:t2 -> off^LIGHT.Light

RS9
1. FADE-IN-OUT.Fade:t1 -> in^LIGHT.Light
2. FADE-IN-OUT.Fade:t2 -> out^LIGHT.Light

RS2
1. BTRYSVR.BSaver:t2 -> off^ON-OFF.OnOff
2. ON-OFF.OnOff:t1 [BTRYSVR.BSaver@Idle]

RS10
1. BTRYSVR.BSaver:t2 -> out^FADE-IN-OUT.Fade
2. FADE-IN-OUT.Fade:t1 [BTRYSVR.BSaver@Idle]

RS3
1. LCONTROL.LControl:t1 -> on^ON-OFF.OnOff
2. LCONTROL.LControl:t2 -> off^ON-OFF.OnOff

RS11
1. LCONTROL.LControl:t1 -> in^FADE-IN-OUT.Fade
2. LCONTROL.LControl:t2 -> out^FADE-IN-OUT.Fade

RS4
1. BTRY-ST.BtrySt:t1 -> save^BTRYSVR.BSaver
2. BTRY-ST.BtrySt:t2 -> release^BTRYSVR.BSaver
3. BTRYSVR.BSaver:t1 [BTRY-ST.BtrySt@NotCharging]

RS5
1. DOOR-ST.DoorSt:t1 -> save^BTRYSVR.BSaver
2. DOOR-ST.DoorSt:t2 -> release^BTRYSVR.BSaver

RS6
1. DOOR-ST.DoorSt:t1 -> on^LCONTROL.LControl
2. DOOR-ST.DoorSt:t2 -> off^LCONTROL.LControl

RS7
1. POWER.Power:t1 -> charge^BTRY-ST.BtrySt
2. POWER.Power:t2 -> stop^BTRY-ST.BtrySt

RS12
1. ALL-DOOR.AllDoor:t1 -> open^DOOR-ST.DoorSt
2. ALL-DOOR.AllDoor:t2 -> close^DOOR-ST.DoorSt

RS8
1. DOOR.*Door:t1 -> open^DOOR-ST.DoorSt
2. DOOR.*Door:t2 -> close^DOOR-ST.DoorSt
3. DOOR-ST.DoorSt:t2 [DOOR.*Door@Closed]

Figure 6.12: Rule sets for vehicle illumination PL

Chapter 7

Related Works

7.1 Aspect-oriented modeling

In some studies, not only those on aspect-oriented programming but also those on aspect-oriented modeling and design [40] [41], an aspect is considered as a module of an additional local characteristic or behavior to the "core concerns," as typified by the notion of an aspect in AspectJ [1] . This type of aspect may be useful in certain situations, but it may be difficult to express design level concerns using these aspects.

Our study is influenced by the concept of the hyperslice in Hyper/J and multi-dimensional separation of concerns [42]. A hyperslice is a set of conventional modules that contain all, and only, those units that pertain to, or address, a given concern; hyperslices can overlap and are composed to form the complete system [42]. The concept of an aspect in the proposed mechanism is similar to that of a hyperslice. Based on our experience in embedded system development, this type of modularization is useful in the design phase, especially in the initial phase of design, e.g., architectural design.

There exist a few aspect-oriented design approaches that focus on not only the static structure but also the behavioral structure. These researches are categorized into three groups.

The first group uses sequence diagrams to denote cross-cutting behaviors (which are similar to those shown by "join points" and "advices"), for example, Theme/UML [6] and [18]. We prefer state diagrams because they can define the general behavior of classes, while sequence diagrams essentially depict only examples of a behavior. Moreover, in actual embedded software development, state diagrams are widely used to design applications. For this reason, the proposed modeling mechanism is based on state diagrams, instead of sequence diagrams.

The second group uses Specification and Description Languages (SDL) [13], for example, WEAVR [9]. SDL can be seen as a special kind of state diagram, but it is suitable for the design of more procedural sides and it may depict more precise and finer-grained behavior. It seems that WEAVR uses SDL because its purpose is automated code generation. Modeling elements in WEAVR are again similar to the elements in AspectJ.

The last group uses state diagrams, similar to us. The approach introduced in [11], models core and cross-cutting concerns as orthogonal regions in the state of a state diagram. These cross-cutting regions are integrated by specifying which events in one region shall be reinterpreted to have significance in another. With regard to the behavior of classes in each aspect being defined in terms of a state model and the relationships among

them being defined as event introductions, the approach in [11] is similar to our proposed mechanism. However, there is an important dissimilarity between the two. In our approach, the coupling between aspects is weakened by introducing aspect-relation-rules. Event introductions realizing aspect collaborations are defined as rules separated from aspect definitions. On the other hand, in [11], an event introduced to another aspect (depicted as another region) is directly defined in an aspect as an action of a transition. This causes a strong coupling between aspects.

There are few researches that focus on aspect-oriented modeling and/or design for embedded software. Most researches on aspect-orientation in the embedded software domain are implementation- and programming-centric, for example, how to use aspect-oriented languages in embedded software [4]. One interesting research is described in [15]. The architectural view concept in this research provides a general guideline of aspect-oriented system decomposition and refinement. Our proposed aspect-oriented context modeling, on the other hand, focuses on embedded systems that capture contexts in the real world and react to them, and it gives more concrete and specific guidelines for those types of software.

7.2 Application of AOTs to PLD

In software development, we are required to manage various concerns, and separation of concerns is one of the most important disciplines. AOTs are technologies of advanced separation of concerns that handle crosscutting concerns, and are expected to be effectively applicable to PLD.

However, various problems in the application of AOTs to PLD have been reported. For example, Mezini et al. compared the feature-oriented programming and the aspect-oriented programming and pointed out the weakness of AspectJ in terms of variability management [24]. They also showed how Casar, their aspect-oriented programming environment that has more expressive development mechanisms, addresses this variability problem. Lee et al. examined AspectJ in terms of feature oriented analysis, and claimed the necessity of guidelines and techniques to handle variability and dependency [19]. Colyer et al. reported their observation through refactoring large scale middleware using AOT, in which they identified two types of crosscutting concerns, homogeneous and heterogeneous [8]. Heterogeneous crosscutting concerns are more difficult to handle, as they cannot be designated in a homogeneous way. All these researches and experiences indicate the difficulties of the management of crosscutting relationships, and this thesis addresses such difficulties.

Though most researches in this field are at the programming level, there are approaches applying aspect-oriented modeling to PLD. Nyssen et al. showed an application of AOT to PLD, and claimed the importance of architecture as a means to examine how features and architectural components should relate [36]. We also believe in the importance of architecture in applying AOTs, though there is a difference between their assumption of aspect and ours; what they call “aspect” in modeling is similar to the notion of aspect in AspectJ, and our modeling is based on our mechanism. We believe our mechanism facilitates designing architecture.

There are researches on aspect-oriented modeling related to model-driven development (MDD). Liu et al. examined how aspect-oriented modeling benefits the MDD of PLD [20].

Voelter et al. discussed the variability management by combining AOT and MDD [43]. We believe that our modeling mechanism could be the basis of MDD, but we need further examination on this point.

Chapter 8

Discussion

8.1 Characteristics of our mechanism

In this section, we discuss the characteristics of the proposed aspect-oriented modeling mechanism.

An aspect is independent and there is no dominant aspect.

In our modeling mechanism, an aspect has a static structure which includes at least one class and each class of the static structure has its behavior model. These classes and their behavior are defined independently on other aspects. That means an aspect is self-contained. Because each aspect is defined independently, no base hierarchy (dominant structure) is needed. Each aspect is not dominant to the others.

In some research, on the other hand, an aspect is a module of an additional local characteristic or behavior to the "core concern" in a dominant base hierarchy, as typified by the notion of aspect in AspectJ.

The concept of aspect of our mechanism is similar to "hyperslice" of Hyper/J. From our experience in embedded system development, this kind of modularization is useful in the design phase, especially in the initial phase of design, e.g., architectural design.

Relationships among aspects are separated from aspects.

An aspects is related to others at composition to consist an entire system. This relation is described as aspect-relation-rules, separately from aspects. It makes aspects highly reusable.

In AspectJ and some modeling approaches conceptually similar to AspectJ, some concerns are encapsulated in base hierarchy and others are encapsulated in aspects. In these approaches, relationships among concerns basically become relationships between base hierarchy and aspects. (To relate aspects themselves is very difficult.) And these relationships are embedded in aspects. Those embedded relationships make aspects lose their independence and may hamper their reusability.

Relationships among aspects are defined based on their behavior.

The above mentioned aspect-relation-rules do not describe correspondence between elements in different aspects (e.g., class C in aspect A and class C' in aspect A' are the same, or method m of aspect A is overwritten by method m' of aspect A'), but

instead describe event introduction to different aspects triggered by transitions at some aspects, and reference to states of other aspects when transitions are triggered.

The correspondence between elements in different aspects would be relatively complicated; there may be many types of correspondence, and it would be difficult to model software with these many correspondence.

In our modeling mechanism, there are just two types of aspect-relation-rules, and this keeps relationships among aspects relatively simple. There would be more types of rules, but so far, from our design experience of embedded software, we do not need any other types of rules.

The mechanism is state transition based.

There are some approaches of aspect-oriented design using UML notation. Many of them use sequence diagrams to denote cross-cutting behaviors (which are like those shown by "join points" and "advices"), e.g., in Theme/UML [6]. In our mechanism, we use state diagrams and aspect-relation-rules, instead of sequence diagrams. We prefer state diagrams, because they can define general behavior of classes, but sequence diagrams basically show examples of behavior. And in real embedded software development, state diagrams are widely used to design software.

We introduced the aspect-oriented modeling mechanism to utilize it in embedded software architecture design. The mechanism could be used for design of other domain's software. To confirm it, we need design experience in other domains and might need extensions to the mechanism. One of these issue will be explained in Appendix C.

As we have shown in this thesis, our modeling mechanism has merits on modeling of embedded software. However, the mechanism has demerits as well. First, it can be difficult to understand behavior of the entire software; local behavior (of aspects) could be more understandable, but the entire behavior cannot be understood easily. This is claimed not only to our mechanism, but also to any aspect-oriented technology generally. This claim is reasonable, but to understand the entire software can be difficult, whenever we design it with any modularization technique. For this problem, appropriate granularity of modules, aspects in our mechanism, is important. Also, tools that show the developers the entire behavior of the model may help. We will show a prototype of such tools in Appendix D. Also, the original motivation of introducing AOT is that modules have crosscutting relationships. AOT gives us a strategy to relate these modules; without such a strategy, the model becomes more complex and more difficult to be understood.

Second, aspect-relation-rules may become so many that the entire model of the software can be complicated. This can be happened, however, we believe the concept of our "aspect-relation" and "rule set" can help. Instead of defining each aspect-relation-rule separately, we define an aspect-relation described by a group of aspect-relation-rules, and these rules work together for certain purpose (such as realizing certain function) as a rule set.

Also, we believe it is important to analyze the trade-off between design with our mechanism and that with conventional technique such as the layering. If we do not need higher reusability, or each module and its usage is stable, design with our mechanism is too much; but when the relationships among modules are complicated and tend to be easily changed, our mechanism works effectively.

8.2 Aspect-oriented technology and product-line development

We proposed a variability management technique using our aspect-oriented modeling mechanism as a means of modeling PLA. We discuss why, how and in what situation our approach improves the reusability and contributes to making core assets flexibly configurable.

Modularization of crosscutting concerns:

In software engineering, the separation of concerns has been one of the important disciplines, and many techniques have been developed to support it. However, there are various concerns that are difficult to separate, such as crosscutting concerns, and approaches to advanced separation of concerns, such as subject-oriented technologies, AOT, and Hyperspaces, have been proposed. In PLD, we are also required to manage crosscutting concerns. For example, the addition or deletion of a feature may cause crosscutting changes in the design and implementation of core assets. The basic advantage of our approach is that it makes crosscutting concerns to be neatly modularized by adopting AOT.

Management of crosscutting relationships:

Though there are multiple approaches to apply AOT to PLD, there are some difficulties. One of the serious problems is the management of crosscutting relationships, which we discussed in this thesis.

This problem occurs from various reasons. One of them is feature interaction [14]; in this case, crosscutting relationships differ depending on the combination of features. Other situation is in handling heterogeneous crosscutting concerns [8]. In this case also, we cannot designate crosscutting relationships in a homogeneous way. One of our motivations is to solve these problems in the management of crosscutting relationships. In our approach, we separate functionalities (aspects in our mechanism) and crosscutting relationships (aspect-relation-rules). This mechanism makes it possible to flexibly change crosscutting relationships without changing functionalities. As functionalities are steadier than crosscutting relationships, our aspects are defined independently from aspect-relation-rules. From this viewpoint, our approach has an advantage over most existing aspect-oriented modeling approaches that do not separate functionalities and crosscutting relationships (such as AspectJ).

Aspectual product line architecture:

There are approaches that utilize AOTs in early phases of software development, but most AOT approaches are at the programming level. Needless to say, techniques at the programming level are important. However, we strongly believe that in order to use AOT effectively, it is indispensable to examine how to apply AOT at the architectural level. This is because, first, there is a big gap between the concepts at the requirement level (such as features) and those at the implementation level (such as classes or aspects in programming languages). PLA plays an important role in bridging this gap. In PLA, we decide the strategy of design and implementation considering how features are modularized by means of architectural components. Second, the aspectual architecture design provides the support for

managing crosscutting relationships. As discussed above, some important reasons for the problems in the management of crosscutting relationships depend on the semantics of applications. Feature interaction problems, for example, must be resolved by understanding the requirements and the architecture of the system. Our approach provides the means for modeling PLA in an aspectual way, and this feature contributes to resolve the problem.

Context modeling as a reference model:

Reusability cannot be improved by merely using an aspect-oriented modeling technique. In order to use the technique effectively, we must apply it rightly to the right place. Our context modeling plays a role of reference model that gives directions on how to utilize the mechanism.

In this thesis, we focus on embedded software, and most embedded software has context dependencies. We have observation that, in this domain, contexts are steadier than sensors (the means to capture contexts), actuators (the means to control the external world), and processes (applications). Further, contexts, and, sensors, actuators, and processes, have crosscutting relationships. For example, a sensor may be used to capture multiple contexts (speed sensor may be used to decide the speed context and the safety context), and a context may be determined by multiple sensors (the safety context is determined by fusing the data from speed sensors, gear-position sensors, sheet- and door-status sensors). Based on our approaches, contexts, sensors, actuators, and processes are modeled as aspects (of our mechanism), and crosscutting relationships among them are modeled as aspect-relation-rules. With that, we can develop steady architecture against changes in sensors, actuators, and applications.

Chapter 9

Conclusion

In this thesis, we examine issues in embedded software design. One of the characteristics of embedded software is strong relation with the external world through sensors and actuators which have many variations. This characteristic introduces crosscutting relationships into design model and makes it difficult to develop reusable model.

In order to manage the problem, we introduced the aspect-oriented context modeling, and also proposed an aspect-oriented modeling mechanism to facilitate the modeling. This mechanism has unique characteristics; each aspect is highly independent, and we define cross-cutting relationships in terms of aspect-relation-rules. This makes embedded software design model modifiable and composable.

We also proposed a variability management technique utilizing our aspect-oriented modeling mechanism. Though there are many applications, it is a challenge to apply AOT to PLD. One of the difficulties is the management of crosscutting relationships. We showed how our modeling mechanism can effectively manage the crosscutting relationships and can be applied to PLA design. For this purpose, we proposed a PLA diagram for aspectual PLA design. We demonstrated how our technique improves reusability of core assets.

Our future works include the refinement of the mechanism, and provision of modeling methods and environment for embedded system design.

Bibliography

[1] http://www.aspectj.org

[2] http://www.early-aspects.net/

[3] http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[4] F. Afonso et al. : "Applying aspects to a real-time embedded operating system", Proc. 6th workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'07), 2007.

[5] N. M. Ali and A. Rashid : "A State-based Join Point Model for AOP. Workshop on Views, Aspects and Roles (2005)

[6] S. Clarke, and R.J. Walker : "Towards a standard design language for AOSD", Proc. 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD'02), April 2002.

[7] S. Clarke and E. Baniassad : "Aspect-Oriented Analysis And Design: The Theme Approach", Addison-Wesley, 2005.

[8] A. Colyer, and A. Clement : "Large-scale AOSD for middleware", Proc. 3rd International Conference on Aspect-oriented Software Development (AOSD'04), March 2004.

[9] T. Cottenier et al. : "The Motorola WEAVR: model weaving in a large industrial context", Proc. 6th Int'l Conf. on Aspect-Oriented Software Development (AOSD'07), March 2007.

[10] B. P. Douglass : "Real-Time Uml: Developing Efficient Objects for Embedded Systems", Addison-Wesley, 1999.

[11] T. Elrad et al. : "Expressing aspects using UML behavioral and structural diagrams", Aspect-Oriented Software Development, Pearson Education, Inc., Boston, MA, September 2004, pp. 459-478.

[12] W. Harrison and H. Ossher : "Subject-oriented programming (a critique of pure objects)", Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, Washington,D.C., September 1993.

[13] ITU, Z. 100: Specification and Description Language (SDL), International Telecommunication Union, 2000

[14] K. C. Kang et.al. : "Feature-Oriented Product Line Engineering", IEEE Software, July/August 2002.

[15] M. Katara, and S. Katz : "Architectural views of aspects", Proc. 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03), Boston, MA, March 2003.

[16] G. Kiczales et al. : "Getting Started with AspectJ", Communications of the ACM, vol. 44, no. 10, pp.59-65, 2001.

[17] T. Kishi, and N. Noda : "Aspect-oriented context modeling for embedded systems", Early-Aspects 2004, 2004.

[18] J. Klein et al. : "Semantic-based weaving of scenarios", Proc. 5th Int'l Conf. on Aspect-Oriented software development (AOSD'06), ACM Press, Bonn, Germany, 2006.

[19] K. Lee, et.al. : "Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development", Proc. 10th International Software Product Line Conference (SPLC'06), Aug. 2006.

[20] J. Liu, et.al. : "The Role of Aspects in Modeling Product Line Variabilities", Proc. First Workshop on Aspect-oriented Product Line Engineering (AOPLE-1), Oct. 2006.

[21] M. Mahoney, et.al. : "Using Aspects to Abstract and Modularize Statecharts", The 5th Aspect-Oriented Modeling Workshop (2004)

[22] M. Mahoney, and T. Elrad : "Modeling platform specific attributes of a system as crosscutting concerns using aspect-oriented statecharts and virtual finite state machines", The 6th Aspect-Oriented Modeling Workshop, 2005.

[23] S. J. Mellor and M. J. Balcer : "Executable UML: A Foundation for Model-Driven Architecture", Addison-Wesley, 2002.

[24] M. Mezini and K. Osternann : "Variability Management with Feature-Oriented Programming and Aspects", ACM SIGSOFT Software Engineering Notes, Vol.29, Issue 6, Nov. 2004.

[25] N. Noda and T. Kishi : "On Aspect-Oriented Design - An Approach to Designing Quality Attributes -", The 6th Asia-Pacific Software Engineering Conference (APSEC'99), pp.230-237, 1999.

[26] N. Noda and T. Kishi : "Aspect-Oriented Design for Quality Attributes" FOSE'99, pp.52-59, 1999 (in Japanese).

[27] N. Noda and T. Kishi : "On Aspect Oriented Design - Basic Idea -", SE123-06, 1999 (in Japanese).

[28] N. Noda and T. Kishi : "Design Pattern Concerns for Software Evolution", International Workshop on Principles of Software Evolution (IWPSE), pp158-161, 2001.

[29] N. Noda and T. Kishi : "about Evaluation Methods of Software Architecture", SE138-17, pp.121-128, 2002 (in Japanese) .

[30] N. Noda and T. Kishi : "On Aspect-Oriented Design Model", FOSE 2003, pp181-184, 2003 (in Japanese).

[31] N. Noda and T. Kishi : "A Proposal of Aspect-Oriented Design Model", SE146-6, pp47-54, 2004 (in Japanese)

[32] N. Noda and T. Kishi : "An aspect-oriented modeling mechanism based on state diagrams", 9th Int'l Workshop on Aspect-Oriented Modeling (AOM'06), October 2006.

[33] N. Noda and T. Kishi : "Design Verification Tool for Product Line Development, tool demonstration", 11th Software Product Line Conference (SPLC 2007), the second proceedings, pp147-148, 2007

[34] N. Noda and T. Kishi : "Aspect-Oriented Modeling for Embedded Software Design", Proc. 14th Asia-Pacific Software Engineering Conference (APSEC 2007), Dec. 2007.

[35] N. Noda and T. Kishi : "Aspect-oriented Modeling for Variability Management", Proc. 12th Software Product Line Conference (SPLC 2008) to appear.

[36] A. Nyssen et.al. : "Are Aspects useful for Managing Variability in Software Product Line?", Aspects and Software Product Lines: An Early Aspects Workshop at SPLC-Europe 2005, Sep. 2005.

[37] H. Ossher and P. Tarr : "Using multidimensional separation of concerns to (re)shape evolving software", Communications of the ACM, vol. 44, no. 10, pp.43-50, 2001.

[38] B. Selic et.al. : "Real-Time Object-Oriented Modeling", John Wiley&Sons,Inc., 1994

[39] M. Samek : "Practical Statecharts in C/C++: Quantum Programming for Embedded Systems", Cmp Books, 2002.

[40] D. Stein et.al. : "A UML-based Aspect-Oriented Design Notation", Proc. 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD 2002), Enschede, The Netherlands, April 2002.

[41] J. Suzuki, and Y. Yamamoto : "Extending UML with aspects: aspect support in the design phase", Proc. of AOP Workshop at ECOOP'99, June 1999.

[42] P. Tarr et al. : "N degrees of separation: multi-dimensional separation of concerns", Proc. 21st Int'l Conf. on Software Engineering (ICSE'99), 1999.

[43] M. Voelter and I. Groher : "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development", Proc. 11th International Software Product Line Conference (SPLC'07), Sep, 2007.

Publications

[1] N. Noda and T. Kishi: "An aspect-oriented modeling mechanism based on state diagrams," 9th Int'l Workshop on Aspect-Oriented Modeling (AOM '06), October 2006.

[2] N. Noda and T. Kishi: "Design Verification Tool for Product Line Development," 11th Software Product Line Conference (SPLC 2007), the second proceedings, pp147-148, 2007.

[3] N. Noda and T. Kishi: "Aspect-Oriented Modeling for Embedded Software Design," Proc. 14th Asia-Pacific Software Engineering Conference (APSEC 2007) Dec. 2007.

[4] N. Noda and T. Kishi: "Aspect-oriented Modeling for Variability Management," Proc. 12th Software Product Line Conference (SPLC 2008) to appear.

Appendix A

Aspect-Orientedness

Aspect-oriented technologies (AOTs) are considered to be promising techniques in software development today. Software has potentially crosscutting requirements, e.g. functional requirements and non-functional requirements, its functionalities may crosscut each other, and software modules realizing these functionalities and requirements can crosscut each other. To manage such crosscutting issues, any technology that decouples software in just one dimension, e.g. object-oriented technology, is not sufficient; AOTs can handle this issue.

First AOTs were aspect-oriented programming languages, e.g. AspectJ. Gradually the concept of aspect has been spreading on entire software development processes; there are some aspect-oriented design methodologies [6][7], and nowadays many researches about "early aspects"(aspect-oriented requirements engineering and architecture design) [2] are carried out.

There are two contrasting modeling paradigms; one is a paradigm as represented by AspectJ [1] and the other as represented by Hyper/J [3]. In this appendix, first we briefly explain the general definition of aspects in A.1, and then introduce each of the two paradigms in A.2 and A.3 respectively.

A.1 General definition

An aspect is an encapsulation of a crosscutting concern.

By traditional modularization mechanisms, such as an "object", a certain kind of concern cannot be localized and cuts across multiple modules; this is the crosscutting concern. Typical crosscutting concerns are:

- concerns needed in development time: some concerns are not included in final products, but may be necessary in development time. E.g, logging, tracing, and profiling.
- concerns related to products' quality attributes: software has many requirements about quality attributes, and some of them, especially run-time quality attribute requirements, are often considered to be met with concerns. E.g., synchronization, security check, and error handling.
- concerns related to to products' features: software has many features, and they often considered concerns. They may be related to "role" and the same entity has different roles in different concerns. E.g., "personnel" feature and "payroll" feature.

If these crosscutting concerns are left across modules, it hampers readability, understandability, reusability, modifiability, and adaptability.

A.2 Aspect in AspectJ

AspectJ is an aspect-oriented extension to Java [1].

Typical crosscutting concerns handled as aspects in AspectJ are as follows:

- logging
- error handling
- synchronization
- platform dependent process

The AspectJ language has three critical elements [16]:

- A join point model; it describes the "hooks" where enhancements may be added. AspectJ employs dynamic join points, in which join points are certain well-defined points in the execution flow of the program. An example of dynamic join points is a method call join point that is the point when a method is called.
- A means of identifying join points; pointcut designators identify particular join points, using method signatures or method properties.
- A means of specifying behavior at join points; advice declaration. They define additional code that runs at join points.

Basic structure of an aspect is pointcut designators and advice declarations. (Some additional attributes can be defined in aspects.)

Characteristics of this paradigm is as follows:

- One base hierarchy is necessary. It is a structure defined by ordinary classes. This is the base and dominant to aspects.
- An aspect must be defined based on the base hierarchy. In other words, it is dependent on classes in the base.
- An aspect tends to be small and additive concern. Because it defines additional behavior to the base classes' behavior, it can be difficult to express a "big" concern constructs software, such as a "feature".

Figure A.1 provides an intuitive explanation of aspect in AspectJ.

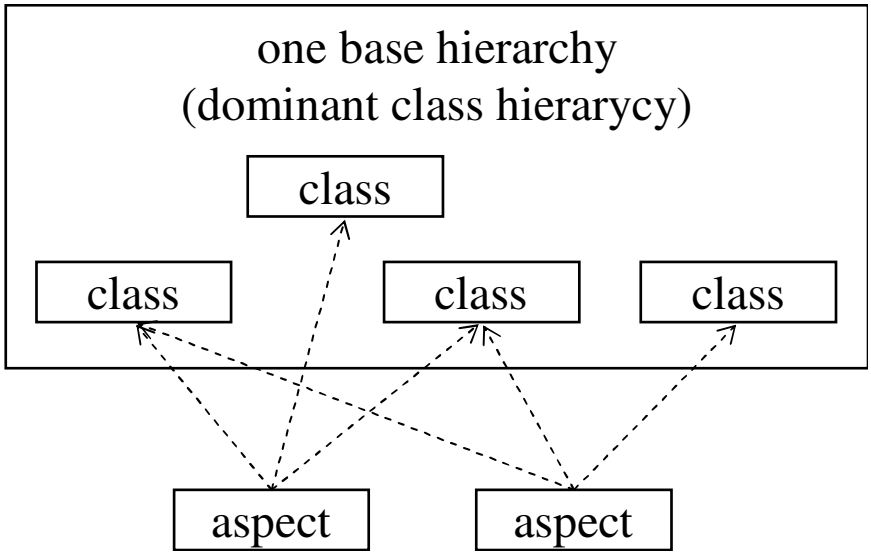


Figure A.1: Aspect in AspectJ

A.3 Aspect in Hyper/J

Hyper/J supports multi-dimensional separation of concerns (MDSOC) [42] for Java.

MDSOC is one approach to separation of concerns, supporting construction, evolution and integration of software. Concerns to be considered and handled in software development may widely vary. There are multiple kinds of concerns; for example, "data concern", "feature concern", "business concern", "nonfunctional concern", and so on. It is almost impossible to deal with these variety of concerns in one dimension. Therefore, MDSOC separates concerns multi-dimensionally. The goals of MDSOC are to enable encapsulation of all kinds of concerns in a software system simultaneously, overlapping and interacting concerns, and on-demand re-modularization. Some of the ideas come from subject-oriented programming [12].

The above mentioned concerns (e.g. "feature concern" and "nonfunctional concerns") are handled as aspects[1].

In Hyper/J, each concern is encapsulated in a separated model, which defines and implements a (partial) class hierarchy appropriate for that concern. This separated model is an aspect[2]. These models are defined and implemented independent of the others. Therefore, they can be understood in isolation; they might or might not be overlapped with the others. To compose entire software of these aspects, a developer writes a declaration, which indicates how the aspects are related, and how the composition is to be carried out. Because this declaration is separately written from the definition of aspects, the aspects do not lose their independence.

Characteristics of this paradigm is as follows:

- No base hierarchy is needed. There is no dominant model to aspects.

[1] In Hyper/J, the term of "aspect" is not used. However, in this appendix, "aspect" is used to express a general concept, in order to show the similar concept in each language in an easily understandable way. The corresponding original term in Hyper/J is "hyperslice."

[2] "hyperslice" in the term of Hyper/J.

(no base hierarchy, and no dominant aspect)

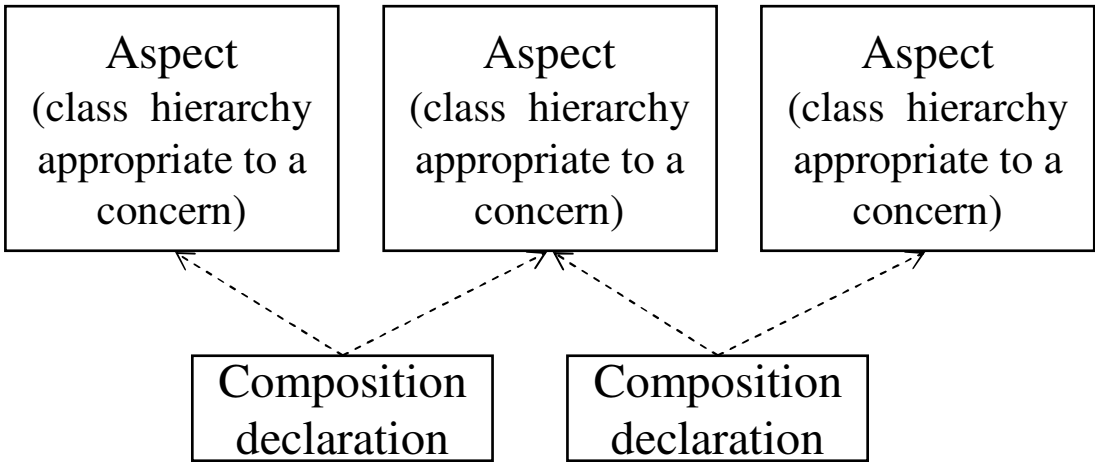


Figure A.2: Aspect in Hyper/J

- Each aspect is independent and self-contained. Potential relationship (to make up the entire software) between aspects are not defined in aspects. No aspect dominates the others.
- An aspect can be easily describe a "big" concern, such as a "feature," because each aspect can includes a (partial) class hierarchy appropriate for that concern.

Figure A.2 provides an intuitive explanation of aspect in Hyper/J.

Appendix B

Metamodel of Aspect-Oriented Modeling

This appendix describes the MOF based metamodel of our aspect-oriented model. This metamodel provides the clear meanings of the modeling elements in our aspect-oriented modeling (AOM) and their relationships.

B.1 Overview

The metamodel contains three packages: AOM_StaticStructure, AOM_Behavior, AOM_AspectRelation. The AOM_StaticStructure package holds the elements used to show the static structure, which can be described in the form of class diagrams. The AOM_Behavior package contains the elements used to describe the behavior of each aspect, which can be described in the form of statemachine diagrams. The AOM_AspectRelation package describes the elements necessary for providing the relationships among aspects, which is given in the aspect-relation-rules form. Figure B.1 shows these packages and their relationships.

In the following three sections, elements of each package can be explained. Classes with the names used in UML2 have same characteristics of the corresponding UML2 classes and are consistent with them. Therefore, the explanation of those classes may be omitted in the following sections.

B.2 AOM_StaticStructure package

Figure B.2 shows the AOM_StaticStructure package. Each element of this package is described from B.2.1 to B.2.6.

B.2.1 Classifier (from UML)

A classifier is a classification of instances, it describes a set of instances that have features in common.

Description Classifier is an abstract metaclass. It is defined in UML. To define an aspect, it is included into the metamodel of the AOM.

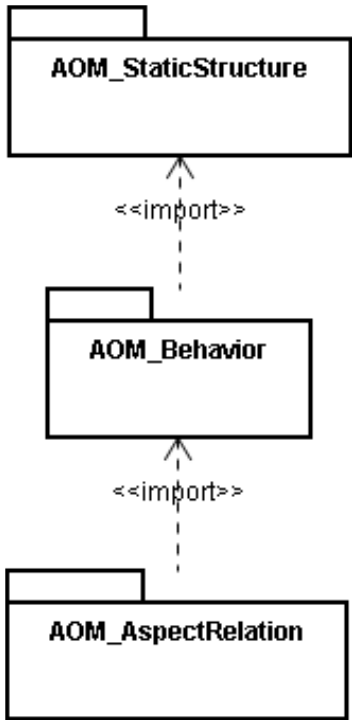


Figure B.1: Packages

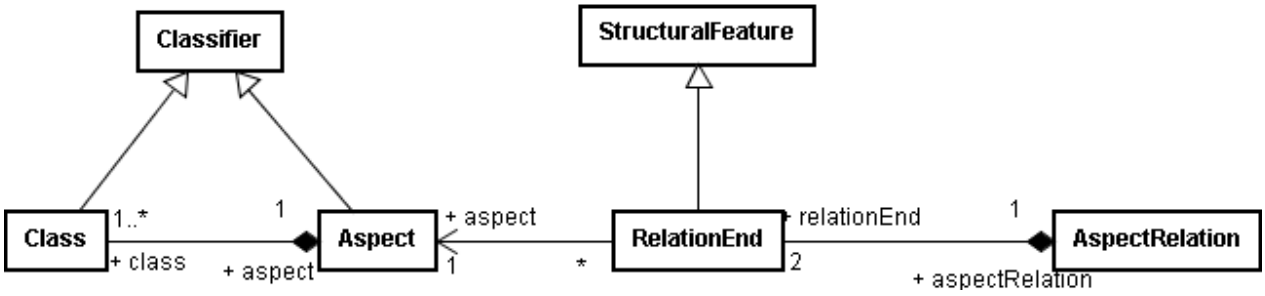


Figure B.2: AOM_StaticStructure

B.2.2 StructuralFeature (from UML)

A structural feature is a typed feature of a classifier that specifies the structure of instances of the classifier.

Description StructuralFeature is an abstract metaclass. It is defined in UML. To define an end of relation between aspects, it is included into the metamodel of the AOM.

B.2.3 Class (from UML)

A class describes a set of objects that share the same specification of features, constraints, and semantics.

Description Class is a kind of classifier. It is defined in UML. To define an aspect, it is included into the metamodel of the AOM.

B.2.4 Aspect

An aspect is a unit that modularizes a concern, and it is the projection of the entire software from the viewpoint of the corresponding concern.

Generalizations

- "Classifier (from UML)"

Description Aspect is a kind of classifier. Aspect is the projection of the entire software from the viewpoint of a specific concern, therefore, Aspect contains multiple Classes. Aspect may be seen as a special Class having composite aggregation with other Classes. However, inherited Aspect nor nested Aspect is not allowed. Aspect is related to other Aspect by AspectRelation. RelationEnd is not navigable from Aspect. That means Aspect does not know to which Aspect it is related.

Associations

- class : Class [*]

  Classes that are contained in this Aspect.

B.2.5 AspectRelation

An aspect-relation is an association between aspects. To compose a system with aspects, aspects are related to each other. This relationship is defined as an aspect-relation.

Description AspectRelation is a special kind of association. It relates two Aspects.

Associations

- relationEnd : RelationEnd [2]

  RelationEnds that are at the both end of the AspectRelation.

B.2.6 RelationEnd

An relation end is the end of aspect-relation. An aspect-relation has two end. It holds the value of multiplicity defined for corresponding aspect-relation.

Generalizations

- "StructuralFeature (from UML)"

Description RelationEnd is the end of AspectRelation. AspectRelation has two RelationEnd. RelationEnd is a kind of StructuralFeature, which is a subclass of MultiplicityElement (from UML). Therefore, RelationEnd holds the value of multiplicity defined for corresponding AspectRelation.

Associations

- aspectRelation : AspectRelation

  AspectRelation that holds the RelationEnd.

- aspect : Aspect

  AspectRelation that holds the RelationEnd.

B.3 AOM_Behavior package

Figure B.3 shows the AOM_Behavior package.

This package contains no new elements extending UML2. All elements are defined in UML2. Relationships among the elements are simplified in this package, being in consistency with UML2.

Each element of this package is described briefly from B.3.1 to B.3.7.

B.3.1 Class (from AOM_StaticStructure)

A class has behavior that may be defined by a statemachine.

Description Class may be associated with one StateMachine that defines its behavior.

Associations

- stateMachine : StateMachine [0..1]

  StateMachine that defines the behavior of the Class.

B.3.2 StateMachine (from UML)

A statemachine describes the state transition of an entity.

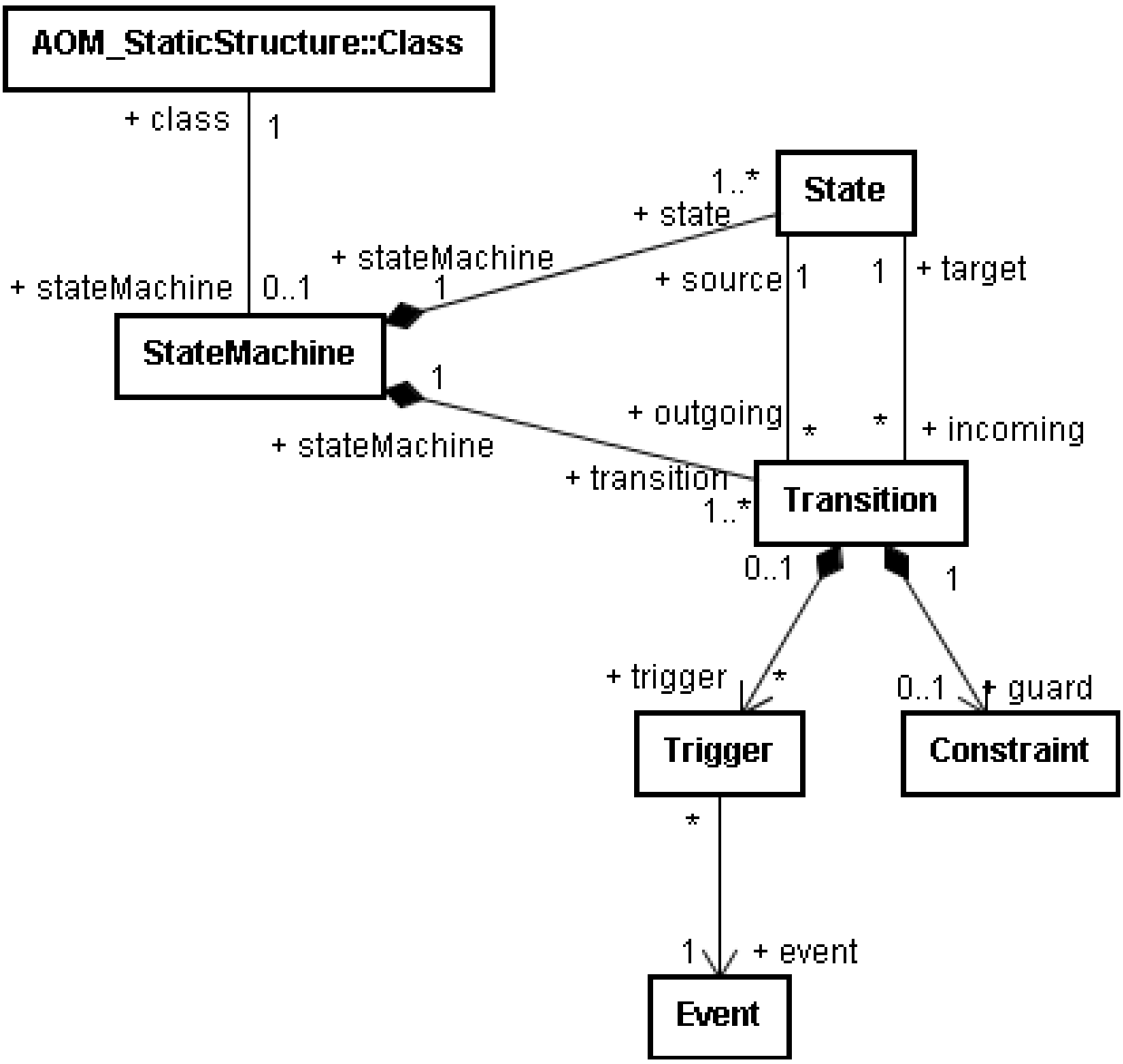


Figure B.3: AOM_Behavior

Description Statemachine may describes the state transition of Class. It has multiple States and multiple Transitions. It is defined in UML.

Associations

- class : Class [0..1]

  Class whose behavior is define by the StateMachine.

- state : State [1..*]

  States included in the StateMachine.

- transition : Transition [1..*]

  Transitions included in the StateMachine.

B.3.3 State (from UML)

A state models a situation during which some invariant condition holds.

Description State models a situation during which some invariant condition holds. It is defined in UML. In the AOM metamodel, State is a simple state that does not have substates.

Associations

- outgoing : Transition [*]

  Transition that goes out from the State.

- incoming : Transition [*]

  Transition that comes into the State.

B.3.4 Transition (from UML)

A transition is a directed relationship between states.

Description Transition is a directed relationship between States. It is triggered by Event and may be constrained by Constraint. It is defined in UML.

Associations

- source : State

  State that is the source of the Transition.

- target : State

  State that is the target of the Transition.

- trigger : Trigger [*]

  Triggers that trigger firing of the Transition.

- guard : Constraint [0..1]

  A guard is a constraint that provides a control over the firing of the transition.

B.3.5 Trigger (from UML)

A trigger relates an event to execution of transitions.

Description Trigger relates Event to the execution of Transition. It is defined in UML.

Associations

- event : Event

  Event that causes the Trigger.

B.3.6 Event (from UML)

An event is the description of some occurrence that may potentially trigger some effects, especially state transitions.

Description Event works as Trigger. Event is associated to Trigger, but Trigger is not navigable from Event. Event is defined in UML.

B.3.7 Constraint (from UML)

A constraint is a condition or restriction for the purpose of declaring some of the semantics. In this package, it is used to express a guard condition for firing transitions.

Description Constraint specifies a condition for firing Transition. It is defined in UML.

B.4 AOM_AspectRelation package

Figure B.4 shows the AOM_AspectRelation package. Each element of this package is described from B.4.1 to B.4.9.

B.4.1 AspectRelation (from AOM_StaticStructure)

An aspect-relation is associated to a set of aspect-relation-rules.

Description AspectRelation is associated with RuleSet that is a set of aspect-relation-rules.

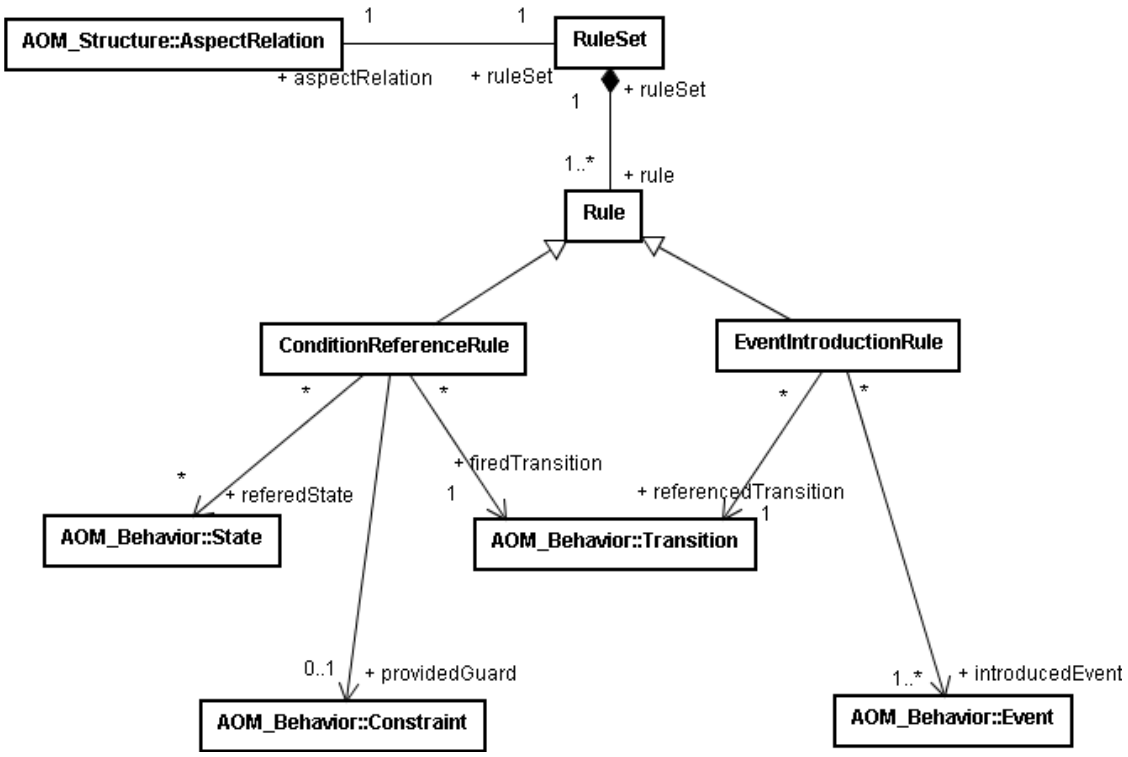


Figure B.4: AOM_AspectRelation

Associations

- ruleSet : RuleSet

  RuleSet that is associated to the AspectRelation.

B.4.2 RuleSet

A rule set is a set of aspect-relation-rules. It describes the detail of an aspect-relation.

Description RuleSet is associated with AspectRelation, and consists of one or more Rules. RuleSet gives a behavioral detail of the corresponding AspectRelation.

Associations

- aspectRelation : AspectRelation

  AspectRelation that is associated to the RuleSet.

- rule : Rule [1..*]

  Rules that are composed in the RuleSet.

B.4.3 Rule

A rule—aspect-relation-rule—describes the detail of relationships between aspects.

Description Rule describes the detail of AspectRelation between Aspects. It is an abstract metaclass.

Associations

- ruleSet : RuleSet

  RuleSet that includes the Rule.

B.4.4 ConditionReferenceRule

A condition-reference rule is a kind of aspect-relation-rule and provides a guard condition, with reference to the properties of an aspect, for a state transition in another aspect.

Generalizations

- "Rule"

Description ConditionReferenceRule indicates a condition-reference rule. It is a kind of Rule. ConditionReferenceRule has associations with State (from AOM_Behavior), Transition (from AOM_Behavior), and Constraint (from AOM_Behavior). These associations indicate relation occurred dynamically; when a rule is evaluated, these relationships are observed. ConditionReferenceRule refers the State of the object described in the rule, and that provide the guard in the form of Constraint, and if the guard is true, Transition is fired.

Associations

- referedState : State [*]

  States that are referred when the ConditionReferenceRule is evaluated.

- providedGuard : Constraint [0..1]

  Constraint that is provided as a guard condition to Transition.

- firedTransition : Transition

  Transition that may be fired after the ConditionReferenceRule is evaluated.

B.4.5 EventIntroductionRule

An event-introduction rule is a kind of aspect-relation-rule and introduces events to other aspects to trigger a certain behavior when a state transition occurs.

Generalizations

- "Rule"

Description EventIntroductionRule indicates a event-introduction rule. It is a kind of Rule. EventIntroductionRule has associations with Transition (from AOM_Behavior) and Event (from AOM_Behavior). These associations indicate relation occurred dynamically; when a rule is evaluated, these relationships are observed. EventIntroductionRule refers the Transition described in the rule, and introduce Event.

Associations

- referencedTransition : Transition

  Transition referenced by the EventIntroductionRule.

- introducedEvent : Event [1..*]

  Events that are introduced when the EventIntroductionRule is evaluated.

B.4.6 State (from AOM_Behavior)

A state is referred by an condition-reference rule.

Description State models a situation during which some invariant condition holds. State in the AOM metamodel is defined in the AOM_Behavior package. State is associated to ConditionReferenceRule, but ConditionReferenceRule is not navigable from State. That means State does not know to which ConditionReferenceRule it is related.

B.4.7 Constraint (from AOM_Behavior)

A constraint is provided by an condition-reference rule to a transition as a guard condition.

Description Constraint specifies a condition for firing Transition. Constraint in the AOM metamodel is defined in the AOM_Behavior package. Constraint is associated to ConditionReferenceRule, but ConditionReferenceRule is not navigable from Constraint. That means Constraint does not know to which ConditionReferenceRule it is related.

B.4.8 Transition (from AOM_Behavior)

A transition may be fired, as a result of evaluating a condition-reference rule. Or it observed by an event-introduction rule.

Description Transition is a directed relationship between States. Transition in the AOM metamodel is defined in the AOM_Behavior package. Transition is associated to ConditionReferenceRule, but ConditionReferenceRule is not navigable from Transition. That means Transition does not know to which ConditionReferenceRule it is related. Likewise, Transition is associated to EventIntroductionRule, but Transition does not know to which EventIntroductionRule it is related.

B.4.9 Event (from AOM_Behavior)

An event is introduced by an event-introduction-rule.

Description Event works as Trigger. Event in the AOM metamodel is defined in the AOM_Behavior package. Event is associated to EventIntroductionRule, but EventIntroductionRule is not navigable from Event. That means Event does not know to which EventIntroductionRule it is related.

Appendix C

Rule Extension

In this thesis, we assume that each class has only one instance, because it is a common way to utilize objects in embedded software field. However, if we remove this restriction, it may make our modeling mechanism more applicable. Basically, this restriction does not affect the mechanism except the aspect-relation-rules. In this appendix, we examine the extension of the rules.

C.1 Generalization of aspect-relation-rules

We remove the restriction, namely, we assume that each class has one or more instances. The other assumption, that instances of classes are not dynamically created, is kept.

For this new assumption, we have to identify different instances of the same class. To identify the instances, we use index numbers which are serial in the class. These indexes are given at the initial time.

Our event-introduction rule is, e.g., "when transition t1 fires at class C1 of aspect A1, event E1 is introduced to class C2 of aspect A2." If classes can have multiple instances, the correspondence between objects of class C1 and objects of class C2 has to be clarified. Using the indexes, the aspect-relation-rules can be extended in the following two ways:

- Corresponding the instances with the same index number.
  - "when transition t1 fires at object O(i) of class C1 of aspect A1, event E1 is introduced to object O'(i) of class C2 of aspect A2."
- Corresponding an arbitrary instance of C1 to all instances of C2.
  - "when transition t1 fires at object O(i) of class C1 of aspect A1, event E1 is introduced to all objects of class C2 of aspect A2."

Here, O(i) means the number i object of the class, where i is an index number.

Our condition-reference rule is, e.g., 'transition t1 fires at class C1 of aspect A1 only when class C2 of aspect A2 is in state S1." In the same way as the event-introduction rule extension, the correspondence between objects of class C1 and objects of class C2 has to be clarified. The rule can be extended in the following ways:

- Corresponding the instances with the same index number.

  - "transition t1 fires at object O(i) of class C1 of aspect A1 only when object O'(i) of class C2 of aspect A2 is in state S1."

- Corresponding an arbitrary instance of C1 to all instances of C2.
  - "transition t1 fires at object O(i) of class C1 of aspect A1 only when all objects class C2 of aspect A2 is in state S1."
  - "transition t1 fires at object O(i) of class C1 of aspect A1 only when any one object of class C2 of aspect A2 is in state S1."

Based on the above, we extend our aspect-relation-rules as follows.

Event-introduction rule:

- A1.C1:t1 -> E1^A2.C2
  - implies that "when transition t1 fires at an arbitrary object of class C1 of aspect A1, event E1 is introduced to an object with the corresponding index of class C2 of aspect A2."
  - note: if class C2 only has the unique instance, event E1 is introduced to that unique instance.
- A1.C1:t1 -> E1^A2.C2.__all
  - implies that "when transition t1 fires at an arbitrary object of class C1 of aspect A1, event E1 is introduced to all objects of class C2 of aspect A2."

Condition-reference rule:

- A1.C1:t1 [A2.C2@S1]
  - implies that "transition t1 fires at any arbitrary object class C1 of aspect A1, only when the object with the corresponding index of class C2 of aspect A2 is in state S1."
  - note: if class C2 only has the unique instance, the state of that unique instance is referred.
- A1.C1:t1 [A2.C2.__all@S1]
  - implies that "transition t1 fires at any arbitrary object class C1 of aspect A1, only when all of the objects of class C2 of aspect A2 is in state S1."
- A1.C1:t1 [A2.C2.__any@S1]
  - implies that "transition t1 fires at any arbitrary object class C1 of aspect A1, only when any one arbitrary object of class C2 of aspect A2 is in state S1."

The syntax of the extended rules is shown in Figure C.1.

```
aspect_relation_rule
   ::= event_introduction_rule | condition_reference_rule
event_introduction_rule
   ::= source_aspect '.' source_class ':' transition_name
       '->' event_name '^' target_aspect '.' target_class all_obj_disignation?
source_aspect ::= name
source_class ::= arbitrary | name
name ::= name_char+
name_char ::= [a-zA-Z0-9] | '_'
arbitrary ::= name? '*' name?
transition_name ::= arbitrary | name
event_name ::= name
target_aspect ::= name
target_class ::= arbitrary | name
all_obj_disignation ::= '.' all_obj
all_obj ::= '__all'
condition_reference_rule
   ::= source_aspect '.' source_class ':' transition_name '[' condition ']'
condition ::= state_condition
state_condition ::= in_state | not_in_state
in_state ::= taget_aspect '.' target_class obj_disignation? '@' state
obj_disignation ::= '.' any_or_all_obj
any_or_all_obj ::= any_obj | all_obj
any_obj ::= '__any'
state ::= name
not_in_state ::= '!' in_state
```

Figure C.1: Syntax of extended aspect-relation-rules

C.2 Application of extended aspect-relation-rules

We explain the usage of the extended aspect-relation-rules with examples. The same example described in Chapter 5 is used with a little modification.

First, we change the example so that the system controls multiple interior lights of the very same type. These lights are equipped in a vehicle (maybe in different places) and turned on and off at the same time. In this case, the lights are modeled as multiple instances of the same Light class in the LIGHT actuator aspect. Although the number of instances of the Light class is multiple, that of the OnOff class in ON-OFF a-context aspect can be one. Because all the light instances are turned on (off) at the same time, only one on-off context is sufficient. Figure C.2 shows the aspects of LIGHT and ON-OFF, along with an object diagram of the static structure of each aspect, respectively. The aspect-relation between these two aspects, RS1 in Figure 5.1 in 5.3, is also changed. Figure C.3 shows the modified aspect-relation-rules. These rules mean: when the state of the unique object of OnOff in ON-OFF changes from Off to On (from On to Off), the on (off) event is introduced to all instances of Light in LIGHT.

Second, we change door sensors of the example. In the example in Chapter 5, the DriverDoor and PassengerDoor sensors are used. It can be a typical model of embedded software. However, if the very same sensor is attached to each door, we may model one Door class with multiple objects. Figure C.4 shows this situation. The context the system captures is the same; whether all doors are closed or not. Therefore, the DOOR-ST s-context aspect is as same as the example in Chapter 5. The aspect-relation between these two aspects, RS8 in Figure 5.1 in 5.3, is also changed. Figure C.5 shows the modified aspect-relation-rules. The rule 1 (of RS8) means: when the state of any object of Door in DOOR changes from Closed to Open, the open event is introduced to the unique object of DoorSt in DoorSt. Although the basic meaning of this rule is that the event is introduced to the one corresponding object, the event is always introduced to the same object in this case, because the DoorSt class has a unique instance only. The rule is understood likewise. The rule 3 means: the transition t2 of unique object DoorSt in DOOR-ST fires (its state changes from Open to Closed) only when all objects of Door in DOOR are in the state of Closed.

This extension can make the aspect-relation-rules to be used in wider areas. When modularization with the concept of aspects in our mechanism is possible and useful in the design of software, our mechanism can work well for the software, even if it is not for embedded system. We had studied and confirmed that this kind of modularization is possible and useful other than in embedded software domain. One example is the encapsulation of "design pattern concern" [28]. We need further study to clarify the effective use of our mechanism in non-embedded software domain.

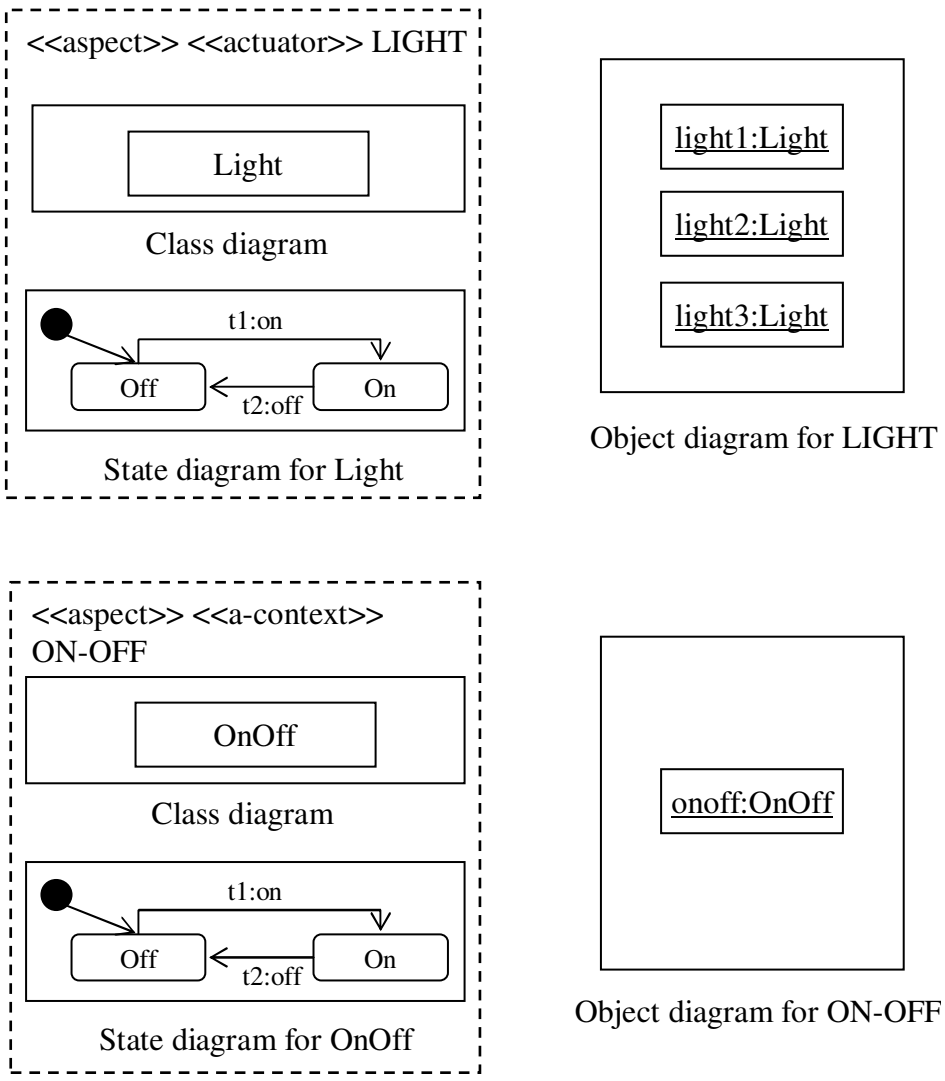


Figure C.2: LIGHT and ON-OFF aspects of modified example

RS1

1. ON-OFF.OnOff:t1 -> on^LIGHT.Light.__all
2. ON-OFF.OnOff:t2 -> off^LIGHT.Light.__all

...

Figure C.3: Modified aspect-relation-rules (1)

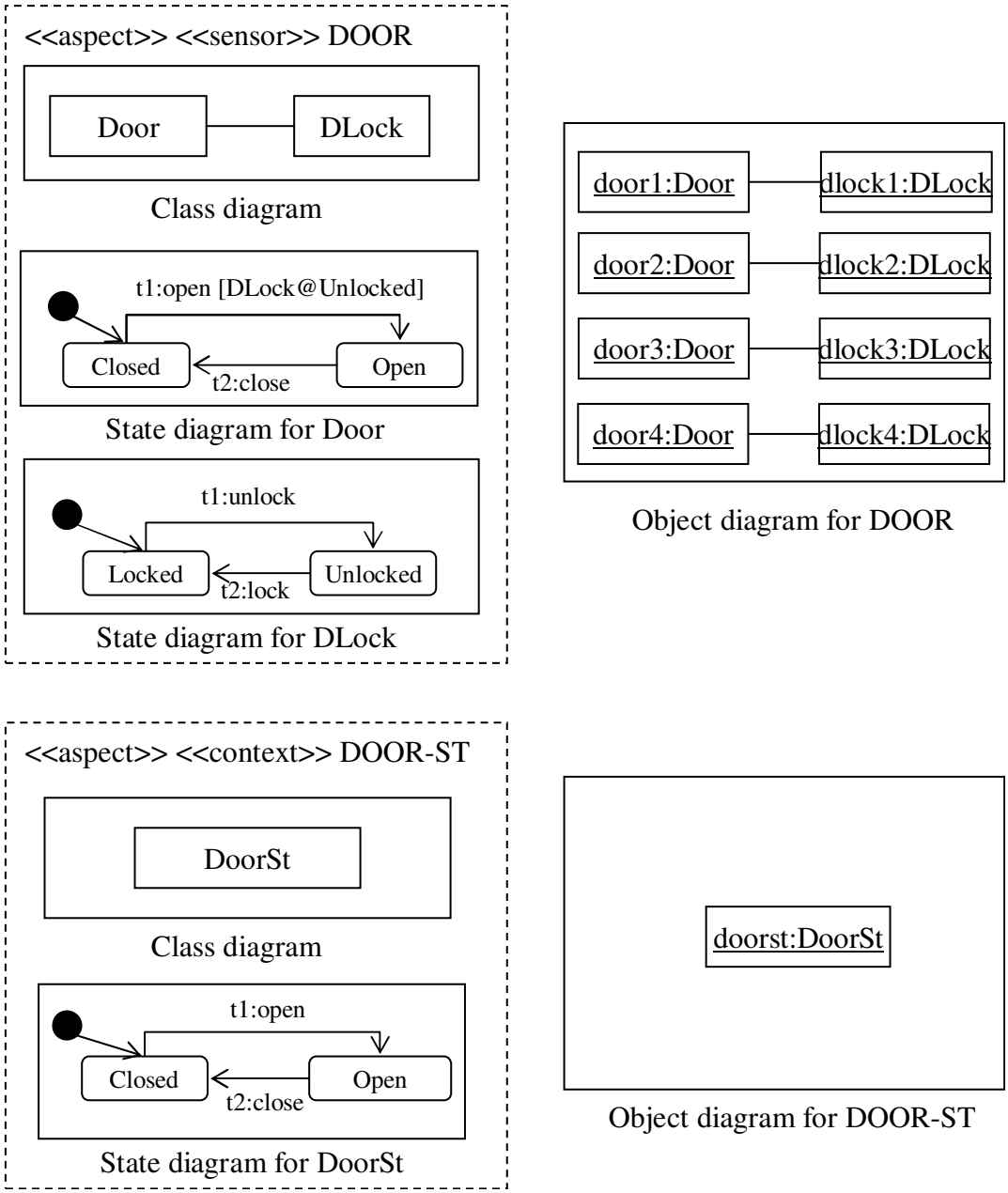


Figure C.4: DOOR and DOOR-ST aspects of modified example

```
...
RS8
1. DOOR.Door:t1 -> open^DOOR-ST.DoorSt
2. DOOR.Door:t2 -> close^DOOR-ST.DoorSt
3. DOOR-ST.DoorSt:t2 [DOOR.Door.__all@Closed]
```

Figure C.5: Modified aspect-relation-rules (2)

Appendix D

Modeling Example by Prototype System

This appendix describes a modeling example by a prototype system that supports the modeling mechanism proposed in this thesis.

D.1 Objective

The objective of introducing a modeling example is to demonstrate the usefulness of the modeling mechanism based on an actual working prototype.

One of the important characteristics of our modeling mechanism is that aspects are defined without depending on other aspects, and aspect-relation-rules relate aspects so as to configure them as a system. This characteristic increases the reusability of aspects, i.e. we can define different system configuration by switching aspect-relation-rules without modifying aspects. We demonstrate this modeling features utilizing a prototype system.

D.2 Prototype system

In this section, we introduce a prototype system we have developed.

Figure D.1 shows an overview of the prototype system.

- The prototype consists of modeler, configurater and execution engine.
- Modeler has capability to define aspects and aspect-relation-rules. Aspects are defined in terms of class diagram and state diagram. A set of aspect-relation-rules is a set of rules written in textural form. Modeler can define multiple sets of aspect-relation-rules, and each set of them corresponds to a system configuration.
- Configurater configures a system from defined aspects and aspect-relation-rules and outputs configured model. If there defined multiple sets of aspect-relation-rules, users select one set to specify a system configuration.
- Execution engine runs the configured model and shows an execution result in terms of sequence diagram.

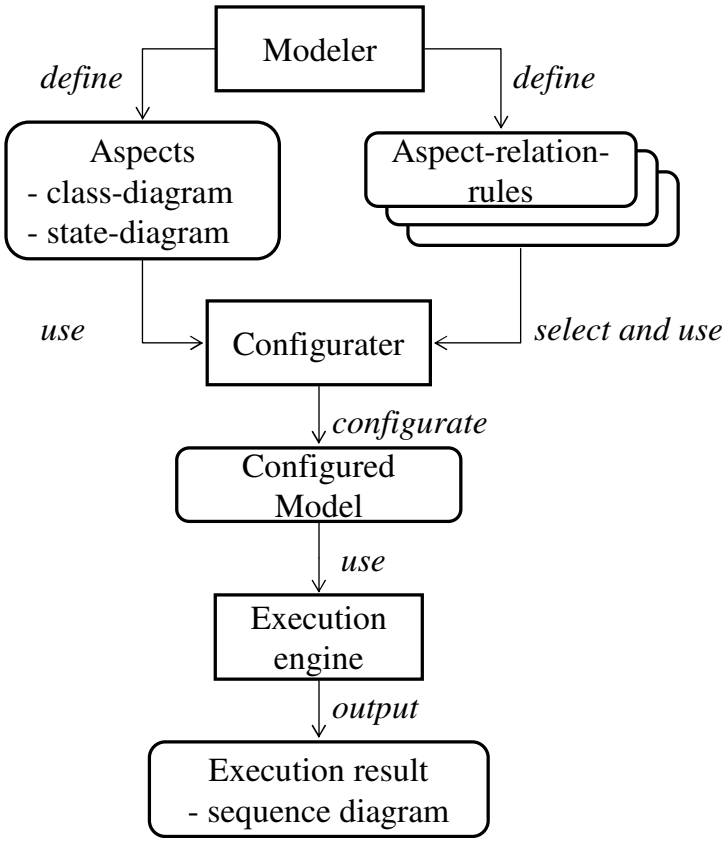


Figure D.1: Overview of prototype system

Concretely speaking, the prototype is developed on Eclipse platform, and utilizes UML plug-in as a modeler and SPIN as a simulation engine.

The prototype does not fully implement the modeling mechanism described in the thesis. The followings are major limitations of this prototype.

- It supports only event-introduction-rule, and does not support condition-reference rule.
- The notation of aspect-relation-rules is different from that is introduced in this thesis.

D.3 Modeling example

We have defined model of vehicle illumination system introduced in Chapter 5. The model is basically the same as that introduced in Chapter 5, but due to the limitation of the prototype system mentioned above, some parts of the model are simplified. Also, we omit some parts of model that are not used in the model execution described in the next section.

D.3.1 Aspects

Overview

Figure D.2 shows the top level class diagram. Each aspect is depicted as UML package stereotyped as "aspect." There defined four aspects "sensor," "context," "process" and "actuator." Here, "context" is s-context; we omit a-context for simplification of this example.

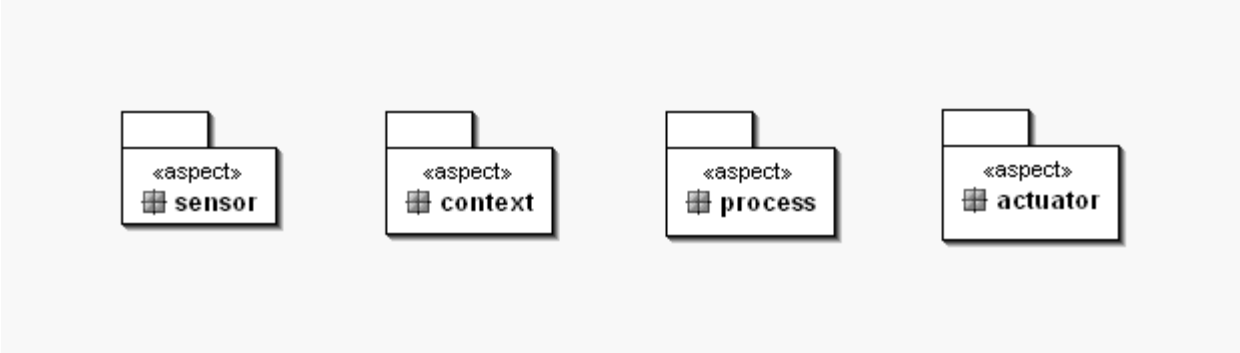


Figure D.2: Aspects

Sensor aspect

Figure D.3 shows the class diagram of "sensor" aspect. In this aspect, class "DriverDoor" is defined.

Figure D.4 shows the state diagram of "DriverDoor" class. There defined two states "Close" and "Open," and also defined transitions from "Close" state to "Open" state

Figure D.3: Sensor aspect

(labeled as “2open”), and from “Open” state to “Close” state. As guard conditions are “true,” state transition can always occur.

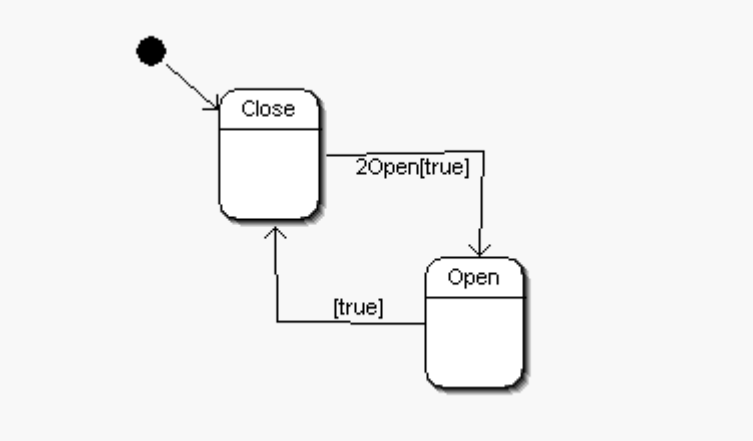


Figure D.4: DriverDoor class (state diagram)

Context aspect

Figure D.5 shows the class diagram of “context” aspect. In this aspect, class “DoorStatus” is defined.

Figure D.6 shows the state diagram of “DoorStatus” class. There defined two states “Close” and “Open,” and also defined transitions from “Close” state to “Open” state (labeled as “2open”), and from “Open” state to “Close” state.

Each state has an entry action which read input from message queue (named “ds”) to temporary variable “x.” The transition from “Close” state to “Open” state is fired by message “open,” and the transition from “Open” state to “Close” state is fired by message “close.”

Process aspect

Figure D.7 shows the class diagram of “process” aspect. In this aspect, classes “Lighting-Control,” “BatterySaver” and “Timer” are defined.

Figure D.8 shows the state diagram of “LightingControl” class. There defined two states “Off” and “On,” and also defined transitions from “Off” state to “On” state (labeled

Figure D.5: Context aspects

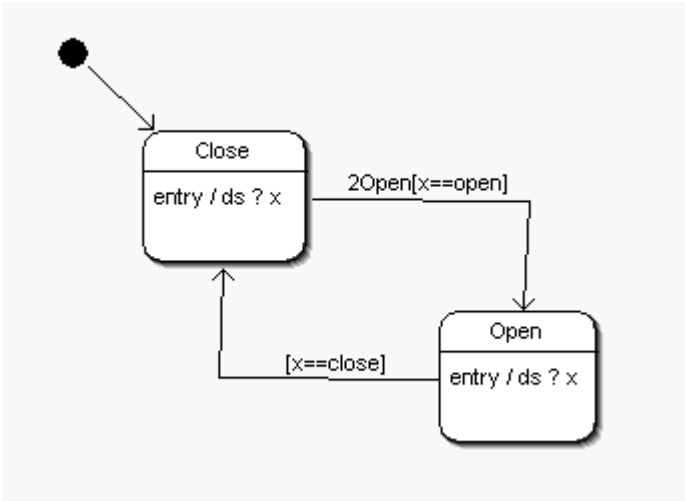


Figure D.6: DoorStatus class (state diagram)

as "2On"), and from "On" state to "Off" state.

Each state has an entry action which reads input from message queue (named "lc") to temporary variable "x." The transition from "Off" state to "On" state is fired by message "open," and the transition from "On" state to "Off" state is fired by message "close."

Figure D.9 shows the state diagram of "BatterySaver" class. There defined three states "Idle," "Lighting" and "Forbidding," and also defined transitions from "Idle" state to "Lighting" state, from "Lighting" state to "Idle" state, from "Lighting" state to "Forbidding" state, and from "Forbidding" state to "Idle" state.

Each state has an entry action which reads input from message queue (named "bs") to temporary variable "x." The transition from "Idle" state to "Lighting" state is fired by message "save," the transition from "Lighting" state to "Idle" state is fired by message "release," the transition from "Lighting" state to "Forbidding" state is fired by message "tout," and the transition from "Forbidding" state to "Idle" state is fired by message "release."

In entering "Lighting" state, a variable "tset" is set to true, and "Timer" class starts counting time. In exiting "Lighting" state, this variable is set to false.

Figure D.10 shows the state diagram of "Timer" class. There defined two states "idle" and "done," and also defined transitions from "idle" state to "done" state and "done"

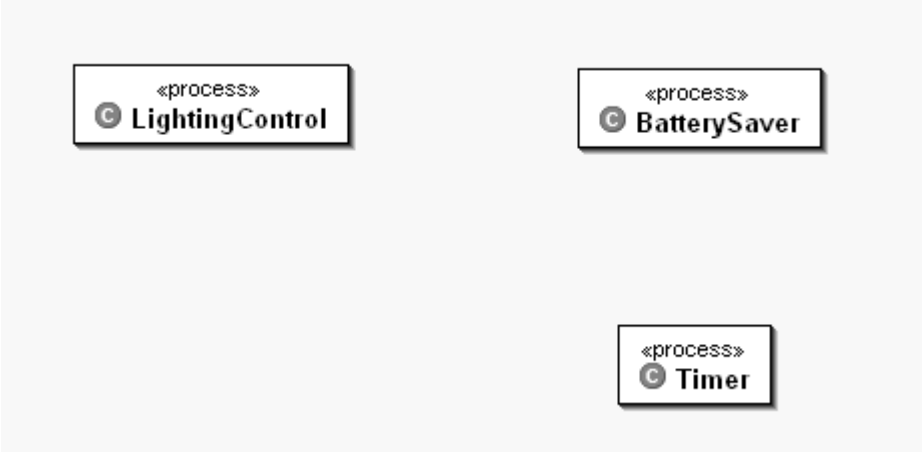


Figure D.7: Process aspect

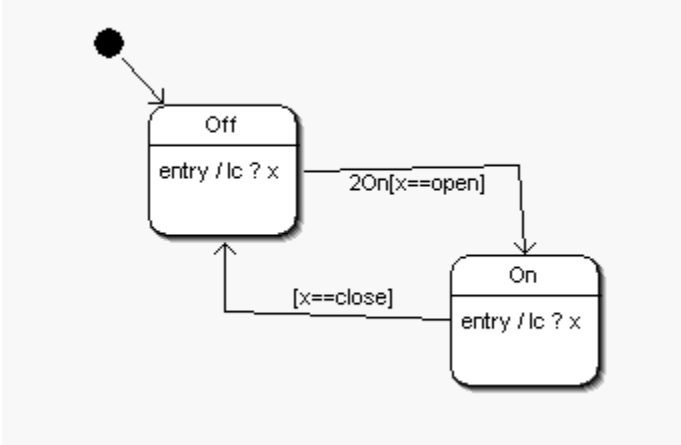


Figure D.8: LightingControl class (state diagram)

state to "idle" state.

This class is not defined in the model in Chapter 5, but in this example we model it in order to realize timeout; i.e. if a variable "tset" is true, it may send "tout" message to message queue "bs" (or before sending the message, "tset" turned back to false.)

Actuator aspect

Figure D.11 shows the class diagram of "actuator" aspect. In this aspect, class "Light" is defined.

Figure D.12 shows the state diagram of "Light" class. There defined two states "Off" and "On," and also defined transitions from "Off" state to "On" state and from "On" state to "Off" state.

Each state has an entry action which reads input from message queue (named "lt") to temporary variable "x." The transition from "Off" state to "On" state is fired by message "on," and the transition from "On" state to "Off" state is fired by message "off."

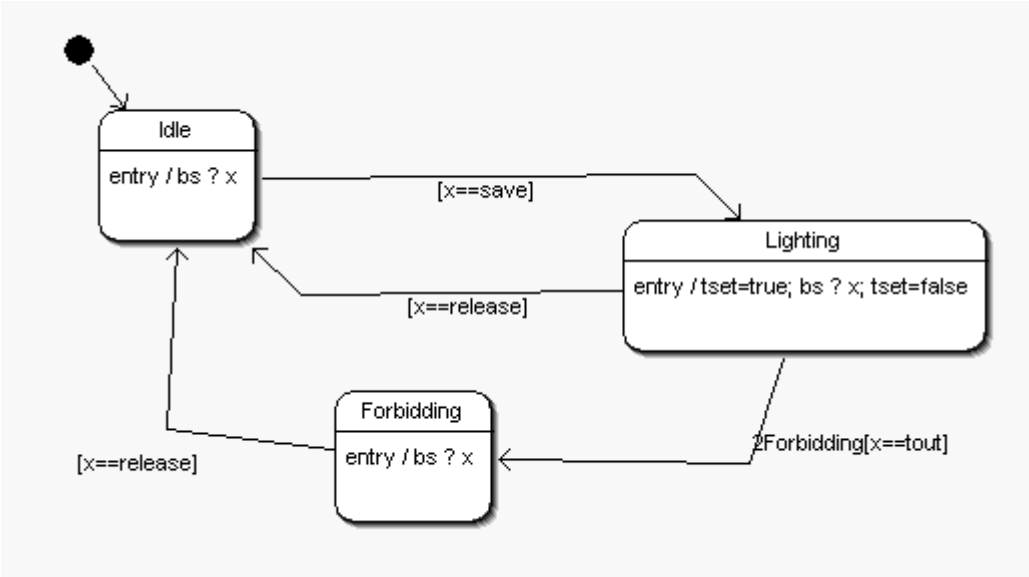


Figure D.9: BatterySaver class (state diagram)

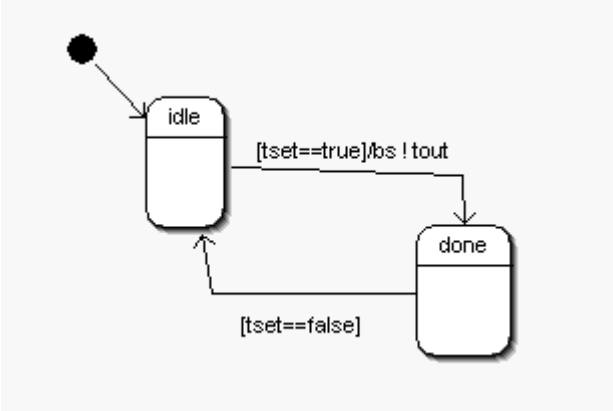


Figure D.10: Timer class (state diagram)

D.3.2 Aspect-relation-rules definition

We have defined two system configurations. One is illumination system without battery saver, and the other is that with battery saver. In order to realize these different configurations, we define the following sets of aspect-relation-rules.

As we have mentioned, our prototype supports event-introduction rule. Though the notation is slightly different, we use the notation introduced in this thesis in oder to avoid confusion.

We do not implement condition-reference rule in our prototype, and we do not use this rule in the example.

Figure D.11: Actuator aspect

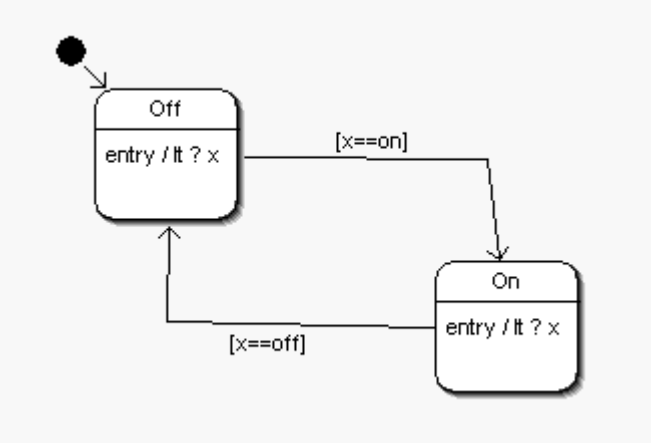


Figure D.12: Light class (state diagram)

Illumination system without battery saver

Figure D.13 shows a set of aspect-relation-rules for a system configuration without battery saver. Each rule has the following meaning.

- If transition labeled "2Open" in the state model of class "DriverDoor" in "sensor" aspect fires, then event "open" is introduced to class "DoorStatus" in "context" aspect.
- If transition labeled "2Open" in the state model of class "DoorStatus" in "context" aspect fires, then event "open" is introduced to class "LightingControl" in "process" aspect.
- If transition labeled "2On" in the state model of class "LightingControl" in "process" aspect fires, then event "on" is introduced to class "Light" in "actuator" aspect.

```
sensor.DriverDoor:2Open -> open^context.DoorStatus
context.DoorStatus:2Open -> open^process.LightingControl
process.LightingControl:2On -> on^actuator.Light
```

Figure D.13: Rule sets (no BatterySaver)

Illumination system with battery saver

Figure D.14 shows a set of aspect-relation-rules for a system configuration with battery saver. This set includes the same three rules that are included in the aspect-relation-rules for illumination system without battery saver, and also includes the following rules.

- If transition labeled "2Open" in the state model of class "DoorStatus" in "context" aspect fires, then event "save" is introduced to class "BatterySaver" in "process" aspect.
- If transition labeled "2Forbidding" in the state model of class "BatterySaver" in "process" aspect fires, then event "off" is introduced to class "Light" in "actuator" aspect.

```
sensor.DriverDoor:2Open -> open^context.DoorStatus
context.DoorStatus:2Open -> open^process.LightingControl
context.DoorStatus:2Open -> save^process.BatterySaver
process.LightingControl:2On -> on^actuator.Light
process.BatterySaver:2Forbidding -> off^actuator.Light
```

Figure D.14: Rule sets (with BatterySaver)

D.4 Result

We configure a system with the aspect-relation-rules in Figure D.13, and execute the configured system.

Figure D.15 shows a snapshot of the execution depicted in sequence diagram. This diagram shows the following behavior:

1. "DriverDoor" class in "sensor" aspect goes into state "Open."
2. "DoorStatus" class in "context" aspect goes into state "Open."
3. "LightingControl" class in "process" aspect goes into state "On."
4. "Light" class in "actuator" aspect goes into state "On."

As rules do not relate "BatterySaver" class with classes in other aspects, "BatterySaver" class (and consequently "Timer" class) does not work. Namely, the Light keeps on lighting.

Then we configure a system with the aspect-relation-rules in Figure D.14, and execute the configured system.

Figure D.16 shows a snapshot of the execution depicted in sequence diagram. This diagram shows the following behavior:

1. "DriverDoor" class in "sensor" aspect goes into state "Open."

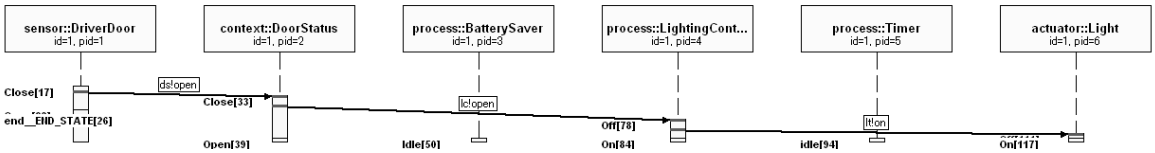


Figure D.15: Execution result (no BatterySaver)

2. "DoorStatus" class in "context" aspect goes into state "Open."
3. "LightingControl" class in "process" aspect goes into state "On."
4. "BatterySaver" class in "process" aspect goes into state "Lighting."
5. "Light" class in "actuator" aspect goes into state "On."
6. "Timer" class in "process" aspect start counting time, sends "tout" message to "BatterySaver" and goes into state "Done."
7. "BatterySaver" class in "process" aspect goes into state "Forbidding."
8. "Light" class in "actuator" aspect goes into state "Off."

As rules relate "BatterySaver" to other classes, it works. Namely, the Light is once turned on, but after that battery saver works and the Light is turned off. Namely, the Light does not keep on lighting, and saves a battery from discharging.

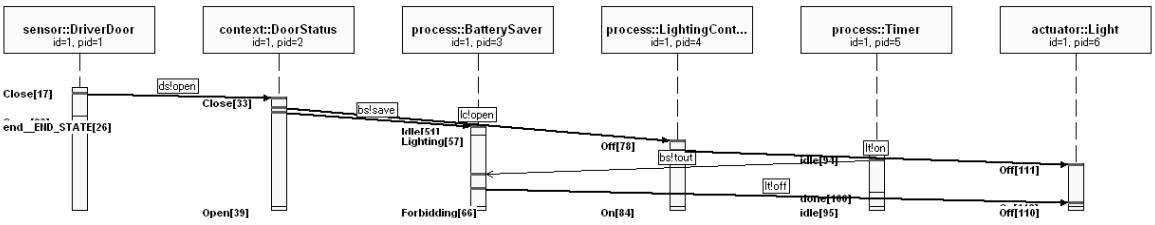


Figure D.16: Execution result (with BatterySaver)

As we have demonstrated above, we can configure two different systems by switching sets of aspect-relation-rules, without modifying aspects.