Date: November 2017.

UML Profile: Use Case Specification for SysML

Version 1.0

Short Title: UML Profile UCS4SysML

Author: Rajiv Murali and Andrew Ireland

Document Number: HW-MACS-TR-0013

Table of Contents

1 Scope 1
1.1 Introduction 1

2 References 1
2.1 Documents 1
2.1.1 OMG Documents 1
2.1.2 Other Documents 1

3 Additional Information 2
3.1 Architecture and Extension Mechanisms 2

4 Characterisation 3
4.1 Overview 3
4.2 UML Representation 4
4.2.1 Profile Diagram and Element Description 4
4.2.1.1 Biddable 4
4.2.1.2 Causal 4
4.2.1.3 Machine 4
4.2.2 Example Usage 5

5 Use Case Specification 6
5.1 Overview 6
5.2 UML Representation 8
5.2.1 Profile Diagram and Element Descriptions 8
5.2.1.1 UseCaseSpecification 8
5.2.1.2 UCSConstraint 9
5.2.1.3 UCSPrecondition, UCSPostcondition, UCSInvariant, UCSTrigger and UCSAssertion 10
5.2.1.4 UCSFlow 10
5.2.1.5 UCSBasicFlow 10
5.2.1.6 UCSAlternateFlow 10
5.2.1.7 UCSStep 11
5.2.1.8 UCSStartOfUseCase 11
5.2.1.9 UCSEndOfUseCase 11
5.2.1.10 UCSBehaviouralStep 12
5.2.1.11 UCSModularStep 12
5.2.1.12 UCSControlType 12
5.2.2 Diagram and Table Extensions 12
5.2.2.1 Use Case Specification Notation 12
5.2.2.2 Participant Table 13
5.2.2.3 Contract Table 13

6 Behavioural Step 15
6.1 Overview 15
6.2 UML Representation 16
6.2.1 Profile Diagram and Element Descriptions 16
6.2.1.1 UCSBehaviouralStep 16
6.2.1.2 UCSActionStep 17
6.2.1.3 UCSOperation 17

6.2.1.4 UCSInteractionStep 17
6.2.1.5 UCSSignal 18
6.2.1.6 UCSAssignedRequirement 18
6.2.1.7 UCSProofWarning 18
6.2.1.8 UCSProofType 19
6.2.1.9 UCSActionTag 19
6.2.1.10 UCSInteractionTag 19
6.2.2 Table Extensions 19
6.2.2.1 Flow Table 19
7 Modular Step 21
7.1 Overview 21
7.2 UML Representation 22
7.2.1 Profile Diagram and Element Descriptions 22
7.2.1.1 UCSIncludeStep 22
7.2.1.2 UCSExtensionStep 22
7.2.2 Table Extensions 23
7.2.2.1 Flow Table (extended with Include and Extension Step) 23
8 Scenarios and Test Cases 24
8.1 Overview 24
8.2 UML Representation 25
8.2.1 Profile Diagram and Element Descriptions 25
8.2.1.1 UseCaseSpecification (Extended with Scenarios and Test Cases) 25
8.2.1.2 UCSScenario 25
8.2.1.3 UCSTestCase 26
8.2.1.4 UCSVerdictKind 26
8.2.2 Table Extensions 26
8.2.2.1 Scenario Table 26

1 Scope

1.1 Introduction

This specification of a UML profile adds capabilities in UML/SysML to support a model-based description for use case specification. As part of use case driven development, use cases are expected to drive the analysis, design and testing of the system under consideration. In this style of development, use case specifications play an integral role by capturing use case narrative (flows) [Jacobson-2016] that describe the many ways in which the system is used to achieve the intent of the use case. These narratives cover scenarios that describe: (1) the ideal behaviour in the use of the system to achieve the intent of the use case (sunny day scenarios); (2) scenarios that handle any problems that may occur; and (3) also scenario that *do not* handle problems that may occur (better understand and promote discussion on the failures in the use of the system [Leveson-2011]).

Traditionally, these narratives are documented as flows in a textual case specification that are subject to the many limitations of a document-based approach: (1) no consistency between description of the use case specification and the UML/SysML model; (2) no methods to validate the construction of the use case specification; and (3) no potential for transformation of information in the use case specification (e.g. to sequence diagrams, state machines, or to other languages that could support execution and verification).

UCS4SysML provides the foundation for a model-based description of use case specifications by extending UML/SysML. It bridges the descriptions of the use case specification to other system model elements, e.g. structural elements (e.g. blocks and parts), behaviour and requirements to the use case narratives. The benefits of the model-based description for the use case specification are: (1) consistency between description of the use case specification and UML/SysML model; (2) validate construction of the use case specification and (3) support model-to-model transformations to down-stream UML diagrams (e.g. sequence diagram) and to languages that support execution of use cases and verification.

2 References

2.1 Documents

2.1.1 OMG Documents

The following normative documents contain provisions, which through reference in this text, constitute provisions of this specification. Subsequent amendments to, or revisions of, any of these publications do not apply.

- [UML] Unified Modelling Language, Version 2.5 (http://www.omg.org/spec/UML/2.5/)
- [SysML] Systems Modelling Language, Version 1.4 (http://www.omg.org/spec/SysML/1.4/)
- [OCL] Object Constraint Language, Version 2.3.1 (http://www.omg.org/spec/OCL/2.3.1/)

2.1.2 Other Documents

The following non-normative documents contain provisions constitute provisions of this specification.

- [Jacobson-2016] Jacobson, Ivar, Ian Spence, and Brian Kerr. Use-Case 2.0. Communications of the ACM, 2016.
- [Jackson-2001] Jackson, Michael. Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, 2001.
- [Leveson-2011] Leveson, Nancy. Engineering a Safer World: Systems thinking applied to Safety. MIT press, 2011.

3 Additional Information

3.1 Architecture and Extension Mechanisms

The relationship between UCS4SysML, SysML and the UML meta-model are shown in Figure 3.1.

UCS4SysML indirectly imports the UML 2 PrimitiveTypes library due to the transitivity of package import. In the remainder of this document, the references to Boolean, Integer, Real, and String.

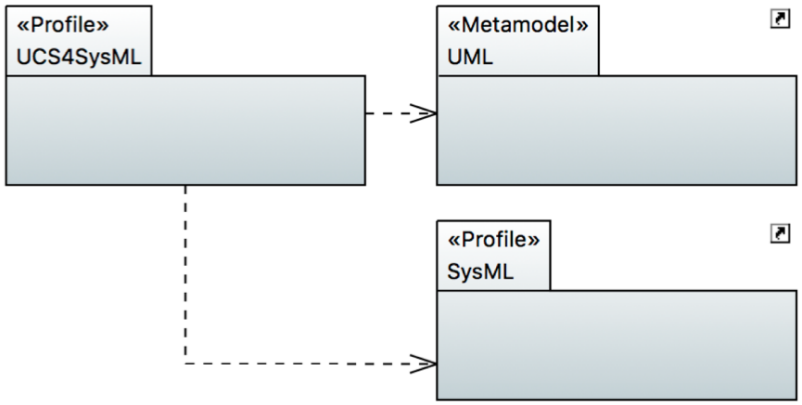

Figure 3.1 – Architecture of UCS4SysML Profile.

This specification uses the following mechanisms to define the SysML extensions:

• UML stereotypes

• UML diagram extensions

• Model libraries

UCS4SysML profile define new modelling constructs by extending existing UML meta-model and SysML constructs with new properties and constraints. UCS4SysML diagram extensions define new diagram notations that supplement diagram notations from SysML and UML.

The UCS4SysML user model is created by instantiating its meta-model and applying the stereotypes specified in the UCS4SysML profile. Each clause in this document describes how the profile and its stereotypes are and how they can be used to extend SysML.

4 Characterisation

4.1 Overview

This clause introduces the characterisation of domains from Problem Frames [Jackson-2001] for structure represented by Blocks in SysML. These characterisations are later used to fit Requirements modelled in SysML to sequence of steps in the flows of a use case specification (Section 6).

Jackson introduces four characterisations to distinguish broad types of domain in the system, these are: machine, causal, lexical and biddable.

- **Machine**: The machine is a specialisation of a general-purpose computer; the specialisation being achieved by programming. It represents the machine to be built (or software under consideration). A crucial characteristic of the machine domain is its very high reliability.
- **Causal**: The causal domains other than the machine is common in many applications. For example, equipment such as buttons, lights, doors, sensors, motors and winding gear all constitutes one or more causal domains. These domains react in a deterministic, predictable way to events, but in general are often much less reliable than the machine.
- **Lexical**: A lexical domain is, physically, a causal domain. For example, it may be a database held on one or more disk drives, or an object structure inside a machine. The physical causal domain provides the infrastructure, but the significance of the lexical domain is in its data.
- **Biddable**: Many problems involve human beings as users or operators, or as originators or recipients of information. These domains can be bid, or asked, to respond to events, but cannot be expected always to react to events in any predictable, deterministic way.

Figure 4.1 is an example of an interlock system where the domains Power Controller, Human Operator and Power Source are represented by blocks in SysML. This clause should enable the block Power Controller to be characterised as Machine, the Human Operator characterised as Biddable and Power Source as Causal. These characterisations are later reflected in the steps representing interaction between these domains (in Section 6).

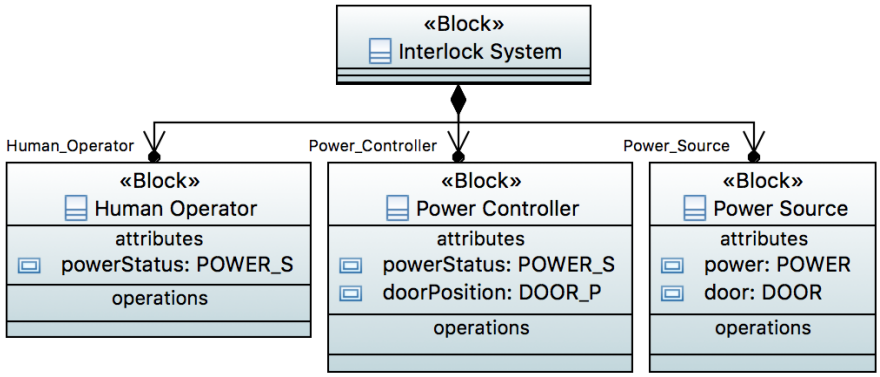

Figure 4.1 – Domain of an Interlock System represented by blocks in SysML.

4.2 UML Representation

4.2.1 Profile Diagram and Element Description

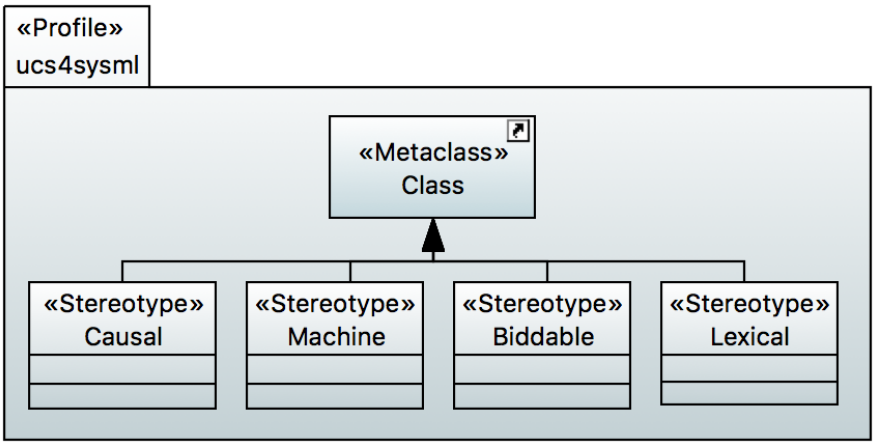

Figure 4.2 - UCS4SysML profile on characterisation.

4.2.1.1 Biddable

This stereotype is used to characterise the SysML structural element Block as biddable. The biddable characterisation conveys a domain that can be bid, or asked, to respond to events, but cannot be expected always to react to events in any predictable, deterministic way [Jackson-2001]. A biddable domain usually represents people.

The stereotype extends the UML metaclass Class. The constraint ensures the extended Class is expected to have the SysML stereotype Block.

Extensions

- Class (from UML)

Constraints

[1] Classes stereotyped «Biddable» must also have the stereotype «Block» (from SysML::Blocks::Block).

[2] Classes stereotypes « Biddable » must not have the stereotypes «Machine», «Causal» or «Lexical».

4.2.1.2 Causal

This stereotype is used to characterise a structural element Block as causal. This is characterisation represent domains that react in a deterministic, predictable way to events, but in general are often much less reliable than the machine.

Extensions

- Class (from UML)

Constraints

[1] Classes stereotyped «Causal» must also have the stereotype «Block» (from SysML::Blocks::Block).

[2] Classes stereotypes «Causal» must not have the stereotypes «Machine», «Biddable» or «Lexical».

4.2.1.3 Machine

This stereotype is used to characterise a structure element as machine. This is characterisation is derived from the Problem Frames approach. In conveys a domain that represents the software under consideration or the software to be built. This block is often introduced as the subject in the use case diagram that contains the use cases.

Extensions

- Class (from UML)

Constraints

[1] Classes stereotyped «Machine» must also have the stereotype «Block» (from SysML::Blocks:Block).

[2] Classes stereotypes «Machine» must not have the stereotypes «Biddable», «Causal» or «Biddable».

4.2.2 Example Usage

…

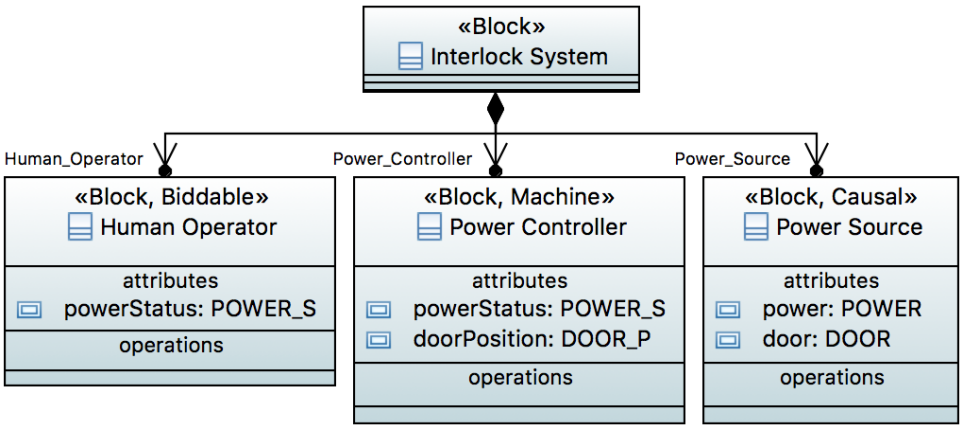

Figure 4.3 – Blocks with Characterisation from Problem Frames.

5 Use Case Specification

5.1 Overview

A **use case** is all the ways of using a system to achieve a particular goal for a particular user [Jacobson-2016].

The purpose of a **use case specification** is to capture the *contract* (constraints), *narratives* (flows), *scenarios* (stories) and test cases for a use case.

The **contract** expresses constraints on the execution of the use case that are represented via *pre-conditions*, *post-conditions* and *invariants*, described as follows:

- *Pre-conditions*: a collection of constraints required to be true *before* the execution of the use case.
- *Post-conditions*: a collection of constraints required to be true *after* the execution of the use case.
- *Invariants*: a collection of constraints required to be true *throughout* the execution of the use case.

The **use case narratives** encompass the many ways in which the system may be used that are described via *flows*. There are two types of flows *basic flow* and *alternative flows*:

- *Basic Flows*: The description of the normal, expected path through the use case. This is the path taken by most of the participants most of the time; it is the most important part of the use-case narrative.
- *Alternate Flows*: Description of variant or optional behaviour as part of a use-case narrative. Alternative flows are defined relative to the use case's basic flow. An alternate flow specifies an optional trigger condition that is required to be true in order for the alternate flow to begin execution

A **flow** captures a sequential collection of steps. Figure 5.1 illustrates the steps of the basic flow and alternate flow. The basic flow has two steps that denote the START OF USE CASE (SoU) and END OF USE CASE (EoU), and a collection of steps 1 to 7 between them. There are three alternate flows ALT 1, ALT 2 and ALT 3 that deviate-from and rejoin-at the steps of the basic flow. For example, ALT1 introduced a deviation from STEP 1 and rejoin-at STEP 3.

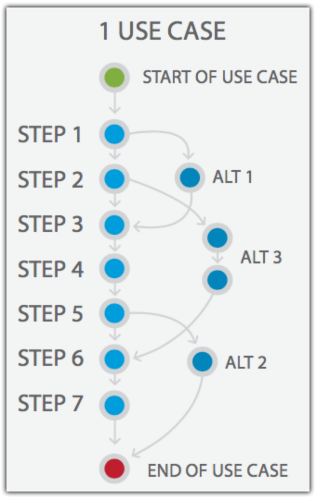

Figure 5.1 - Use case narratives [Jacobson-2016].

Steps in the use case specification are used to convey the behaviour of the participants and also take into account the include and extension-point relationships on the specified use case. Apart from steps representing the SoU and EoU, we distinguish two types of steps: Behavioural Steps and Modular Step. These two types of steps are introduced and further explained in Sections 6 and 7, respectively.

The steps in the use case specification aim to capture the interaction between **participants** of the use case. In SysML, an internal block diagram is created where the enclosing frame corresponds to the *system context*, and the **participating actors** correspond to **parts** in the internal block diagram. To support this technique, the actors associated to the use case

(via communication lines) are allocated to blocks using the *allocation* relationship. This enables the parts representing actors can be typed by the block.

In SysML, the subject of the use case also referred to as *system under consideration* is represented by a Block (SysML structural construct).

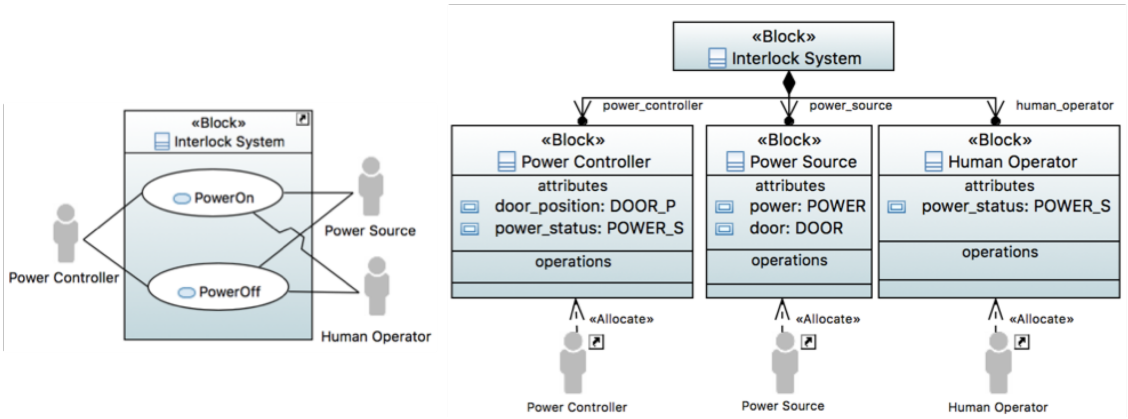

Figure 5.2 – Actors allocated to blocks in SysML.

Use cases are effective for capturing the functional requirements[1]. Use cases are used to support requirements analysis in many of the model-based approaches using UML and SysML. The incorporation of text-based requirements into SysML effectively accommodates a broad range of requirements.

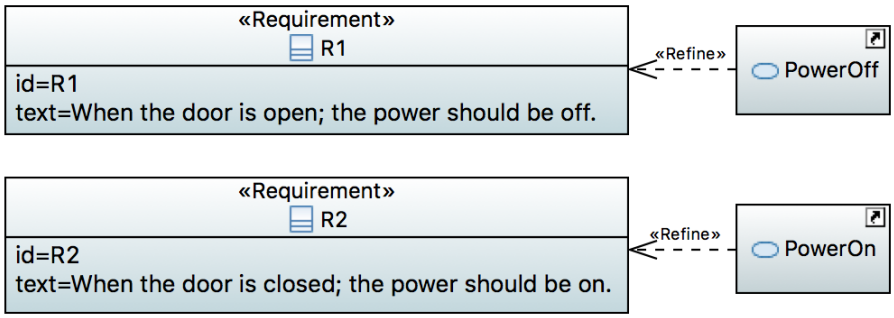

Figure 5.3 – Requirement relationship <<Refine>> (from SysML).

[1] Not as well suited for capturing a wide array of other requirements, such as physical requirements (e.g., weight, size, vibration); availability requirements; or other so-called non=functional requirements

5.2 UML Representation

5.2.1 Profile Diagram and Element Descriptions

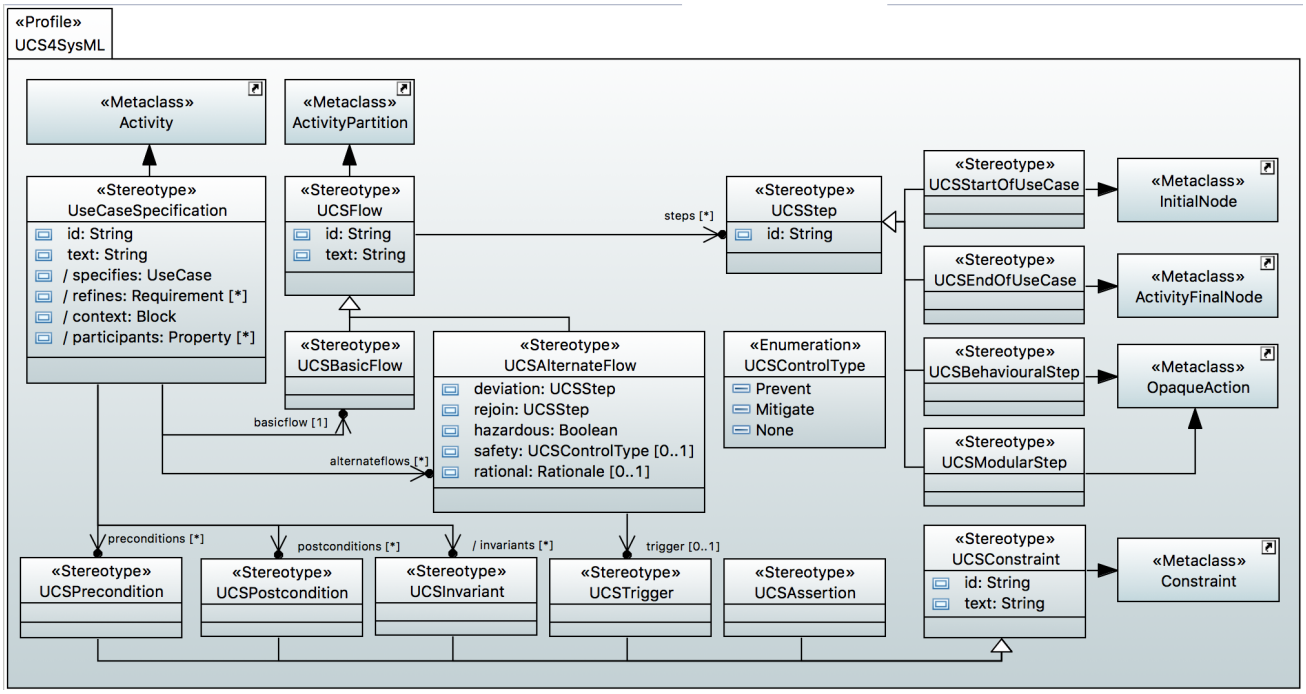

Figure 5.4 - Profile diagram on use case specification.

5.2.1.1 UseCaseSpecification

Use case specification is used to establish the contract, flows, scenarios and test cases of the use case it specifies.

Use case specification is a stereotype of Activity. The activity extended by the use case specification must be owned by a Use Case. This enables the use case specification to be modelled as an owned behaviour of the use case. The capability of activities to define partitions and nodes are used by the use case specification to define flows and steps, respectively. Deleting the container use case specification deletes the contained flows and their contained steps, a functionality inherited from UML.

Alternatively, the actors can be allocated to blocks using the allocation relationship described in Chapter 14, and then the parts representing actors can be typed by the block.

Extensions

- Activity (from UML)

Attributes

- id: String
  - A unique id for the use case specification.
- text: String
  - A brief textual representation on the intent of the use case being specified.
- /specifies: UseCase
  - Explicitly states the use case specified by this use case specification. The value is derived from the owner of the extended Activity metaclass.
- /context: Block (from SysML::Blocks::Block)
  - A block that represents the context of the use case. The participants and constraints of the use case are defined in this context.
- /participants: Property [*]

    - The actors associated to the *specified* use case who have been allocated to the participants in the context.
- /refines: Requirements [*] (from SysML::Requirements::Requirement)
    - Derived from the requirements that are suppliers of a «Refine» relationship for which the use case is a client and the value of the property /specify in the use case specification.
- /invariants: UCSInvariant [*]
    - A collection of constraint required to be true during the execution of the use case. The invariants are derived from any constraint defined in the /context of the use case that has the stereotype «UCSInvariant».

Associations

- preconditions: UCSPrecondition [*]
    - A collection of constraints required to be true before the execution of the use case.
- postconditions: UCSPostcondition [*]
    - A collection of constraints required to be true after the execution of the use case.
- basicflow: UCSBasicFlow
    - Basic flow captures an ideal sequence of actions required to achieve the goal of the use case being specified.
- alternateflow: UCSAlternateFlow [*]
    - Alternate flow captures a variant or optional sequence of actions that aim to achieve the goal of the use case. An incomplete alternate flow is represented as a accident flow.

Constraints

[1] *ucs_owned_by_usecase* – Base activity of the UCS must be owned by a use case.

```
base_Activity.owner.oclIsTypeOf(UseCase)
```

[2] *owned_usecase_subject_is _block* - The subject of the owned use case must be by Block (SysML).

```
base_Activity.owner.oclAsType(UseCase) -> subject.oclIsKindOf(Block)
```

Operations

[1] getSpecifies: UseCase

```
getSpecifies = base_Activity.owner
```

[2] getContext: Block

```
getContext = owner.oclAsType(UseCase).subject.getStereotypeApplications() -> any(e |
e.oclIsTypeOf(SysML::Blocks::Block))
```

[3] getParticipants: Set (Property)

```
TO DO (an unsuitable short cut is used for not : context.getParts())
```

[4] getRefines: Set (Requirement)

```
getRefines = (SysML::Requirements::Refine).allInstances() -> select(e | (e.base_Abstraction.client -> one(i | i
= base_Activity.owner))) -> collect(base_Abstraction.supplier)
```

[4] getInvariants: Set (UCSInvariant)

```
(UCS4SysML::UCSInvariant).allInstances() -> select(e | (owner.oclAsType(UseCase).subject) -> one(i | i =
a.base_Constraint.owner))
```

5.2.1.2 UCSConstraint

This stereotype represents a constraint in the use case specification. It extends the UML metaclass Constraint.

It specifies a unique id and an informal description for the constraint.

Extension

- Constraint (from UML)

Attributes

- id: String
    - A unique id for the constraint.

- text: String
    - A natural language description of the constraint.

5.2.1.3 UCSPrecondition, UCSPostcondition, UCSInvariant, UCSTrigger and UCSAssertion

These stereotypes serve as an annotation to help distinguish the different types of UML Constraints referenced by the use case specification.

They are a generalisation of the stereotype UCSConstraint.

Generalisation

- UCSConstraint

5.2.1.4 UCSFlow

This stereotype represents a flow in the use case specification. A flow captures a sequence of steps.

ActivityPartition are used to represent flows, i.e. basic flow and alternate flows, in the use case specification. UCSFlow extends the metaclass ActivityPartition. The constraint ensures that the ActivityPartition to which the UCSFlow stereotype is applied to is owned by an Activity with the stereotype UseCaseSpecificaiton.

Each flow captures a collection of steps.

Extension

- ActivityPartition (from UML)

Attributes

- id: String
    - A unique id for the flow in the use case specification.
- text: String
    - A brief description of the flow.

Associations

- steps: UCSStep [*]

5.2.1.5 UCSBasicFlow

This stereotype represents the basic flow in the use case specification. It captures the normal or expected path through the use case, i.e. the path taken by most of the users most of the time. It is the most important part of the use case narrative.

Every use case specification must have at least one basic flow.

It is a generalisation of UCSFlow. It inherits a unique id, brief textual description and a sequence of steps.

Generalization

- UCSFlow

5.2.1.6 UCSAlternateFlow

UCSAlternateFlow is a flow in the use case specification that represents an alternate flow, i.e. a variant sequence of actions to achieve the goal of the use case. Description of variant or optional behaviour as part of a use-case narrative. Alternative flows are de ned relative to the use case's basic flow.

Generalization

- UCSFlow

Attributes

- deviation: UCSStep
  - A step before which the deviation to this alternate flow may take place.
- rejoin: UCSStep
  - A step before which the rejoin from this alternate flow may take place.
- hazardous: Boolean = false
  - When true denotes the alternate flow is incomplete and represents an accident flow that should violate the constraints of the use case.
- safety: UCSControlType [0..1]
  - The type of control mechanism used to control the hazard.
- rational: Rationale [0..1]
  - A rationale provided for the type of \safety control mechanism for the hazard.

Association

- trigger: UCSTrigger [0..1]
  - An optional constraint that when provided must be true in order for the alternate flow to being execution.

5.2.1.7 UCSStep

This stereotype represents a step in the flow of a use case specification.

A step captures a unique id. This stereotype does not extend a UML metaclass.

Attributes

- id: String
  - A unique id for the flow in the use case specification.

5.2.1.8 UCSStartOfUseCase

This stereotype represents the start of the use case (short SoU). It is a generalisation of the stereotype UCSStep and extends the UML metaclass InitialNode.

Every use case specification must have one UCSStartOfUseCase. This must be the first step in the UCSBasicFlow.

Generalisation

- UCSStep

Extension

- InitialNode

5.2.1.9 UCSEndOfUseCase

This stereotype represents the end of the use case (short EoU).

It is a generalisation of the stereotype UCSStep and extends the UML metaclass AcitivtyFinalNode.

Every use case specification must have one UCSStartOfUseCase. This must be the last step in the UCSBasicFlow.

Generalisation

- UCSStep

Extension

- ActivityFinalNode

5.2.1.10 UCSBehaviouralStep

This stereotype represents a step in the flow of a use case specification that capture the type of behaviour performed a participant of the use case.

It is a generalisation of the UCSStep and extends the UML metaclass OpaqueAction. This stereotype is further defined in Section 6.

Generalisation

- UCSStep

Extension

- OpaqueAction

5.2.1.11 UCSModularStep

This stereotype represents a step in the flow of a use case specification that how either the includes or extends relationship are used.

It is a generalisation of the UCSStep and extends the UML metaclass OpaqueAction. This stereotype is further defined in Section 7.

Generalisation

- UCSStep

Extension

- OpaqueAction

5.2.1.12 UCSControlType

The type of control used in the hazardous alternate flow i.e. an accident flow to control the hazardous action.

Literal Values

- prevent
  - The type of control provided in the alternate flow prevents the hazardous action.
- mitigate
  - The type of control provided in the alternate flow mitigates the hazardous action.
- none
  - No control of the hazardous action is provided.

5.2.2 Diagram and Table Extensions

5.2.2.1 Use Case Specification Notation

The use case specification is represented in Table 5.1. The «UseCaseSpecification» compartment label for the stereotype properties compartment (e.g. text, specify, context and refines) can be elided.

Table 5.1 – Diagram Elements for Specify and UseCaseSpecification.

Node Name	Concrete Syntax	Abstract Syntax Reference

UseCaseSpecification	«UseCaseSpecification» MaintainH_UCS «UseCaseSpecification» text=The intent of this use case is to maintain the water level in the water tank below high (H) limit. specify=MaintainH context=Water Tank System Context refines=[R1, R2]	UCS4SysML::UseCaseSpecification

5.2.2.2 Participant Table

The tabular format is used to represent the value properties of a participant in the use case specification. This includes:

- The participant who owns the value property.
- The name of the property.
- The type of the property.
- The default value of the property, i.e. represents initialisation.
- The read-only value of the property, either true or false. This indicates if the

Table 5.2 – Table for properties of Participant.

table [UseCaseSpecification] PowerOnUCS [Participant associated to PowerOn and its value properties.]

Participant	Name	Type	Default Value	IsReadOnly
Power_Source	power	POWER	On	false
Power_Source	door	DOOR	Closed	false
Human_Operator	powerStatus	POWER_S	Unknown	false
Power_Controller	powerStatus	POWER_S	Unknown	false
Power_Controller	doorPosition	DOOR_P	Unknown	false

5.2.2.3 Contract Table

The tabular format is used to represent the constraints in the contract of the use case specification, and may include:

- A column for the type of constraint in contract, i.e. pre-condition, post-condition or invariant.
- A column for the unique name id provided to the constraint.
- A column for the natural language description of the contract.
- A column for the OCL description of the constraint.

Table 5.3 – Table showing constraints in the contract of the use case specification.

table [UseCaseSpecification] PowerOnUCS [Constraints on the execution of use case PowerOn.]

Kind	Id	Text	OCL
Precondition	POff_Pre_1	Power source's power is Off.	power_source.power = POWER::Off
Precondition	POff_Pre_2	The process state of the power status for the human operator and power controller is the same.	human_operator.power_status = power_controller.power_status
Postcondition	POff_Post_1	Power Source's power is On.	power_source.power = POWER::On

Postcondition	POff_Post_2	The process state of the power status for the human operator and power controller is the same.	human_operator.power_status = power_controller.power_status
Invariant	SC_1_1	Power Source's power and door not must not be On and Open.	not(power_source.power = POWER::On and power_source.door = DOOR::Open)

6 Behavioural Step

6.1 Overview

A flow in a use case specification is composed of a collection of steps [Jacobson 2016]. Figure 6.1 provides the steps in a basic flow of a Withdrawal use case from UseCase 2.0 [Jacobson 2016]. The steps 1 to 7 capture behaviour performed by actors that play a role in the use case. The description of these steps are kept minimal, e.g. Insert Card, Card Valid Sucessful[2], for step 1 and 2. This enables us to investigate an extensions to this concept by our profile.

In UCS4SysML, the steps 1 to 7 are distinguished as behavioural step (UCSBehaviouralStep). A behavioural step must have one actors (or in SysML, participant) who initiates the step. For example, in step 1 'Insert Card' would be initiated by a participant representing an account holder, while step 2 'Card Valid' would be initiated by a participant representing the ATM.

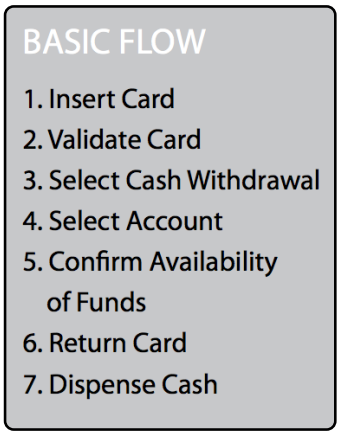

Figure 6.1 – An example of steps of a basic flow for a Withdraw use case [Jacobson-2016].

The behavioural step is further distinguished into two types: interaction (UCSInteractionStep) and action step (UCSActionStep) in this section of the UCS4SysML profile.

An *interaction step* captures a communication of *send request* (signal) between two participants of the use case. The initiator of this step is referred to as the *sender* and a participant who is expected to receive the signal is referred to as the *receiver*. The description of the communication is denoted by a UML signal. For example, in step 1 'Insert Card' in UCS4SysML is introduced as an interaction step where the participant account holder is the *sender* and the participant ATM is the *receiver*. The communication of this step refers to a signal Card_Inserted.

An *action step* captures a change to the state (operation) of the participant. In an action step, the participant that initiates the step is referred as a *performer* and refers to an operation in the type of the participant. For example, step 2 'Card Valid' in UCS4SysML is introduced as an action step. Here, the participant ATM is the performer and it refers to an operation in the type.

As part of requirements-guided development and to support traceability of requirements, the *requirements*[3] that are refined by the use case can be *assigned* to the behavioural steps of the use case specification. The problem classes Commanded Behaviour, Required Behaviour and Information Display, are used to assign a functional requirement to the behavioural steps of the use case (paper).

UCS4SysML aims to use formal methods to enable model execution of steps in the flows of a use case specification. The intent is to be able to support verification of behaviour performed by the use case against the constraints in the contract of the use case. The failure of proofs from the formal method are introduced as proof warnings.

[2] The step 2 'Validate Card' should be written as 'Card Validation Successful' as an alternate flow from this step is 'Card Validation Unsuccessful.

[3] We expect the Requirements refined to the use case are functional requirements.

6.2 UML Representation

6.2.1 Profile Diagram and Element Descriptions

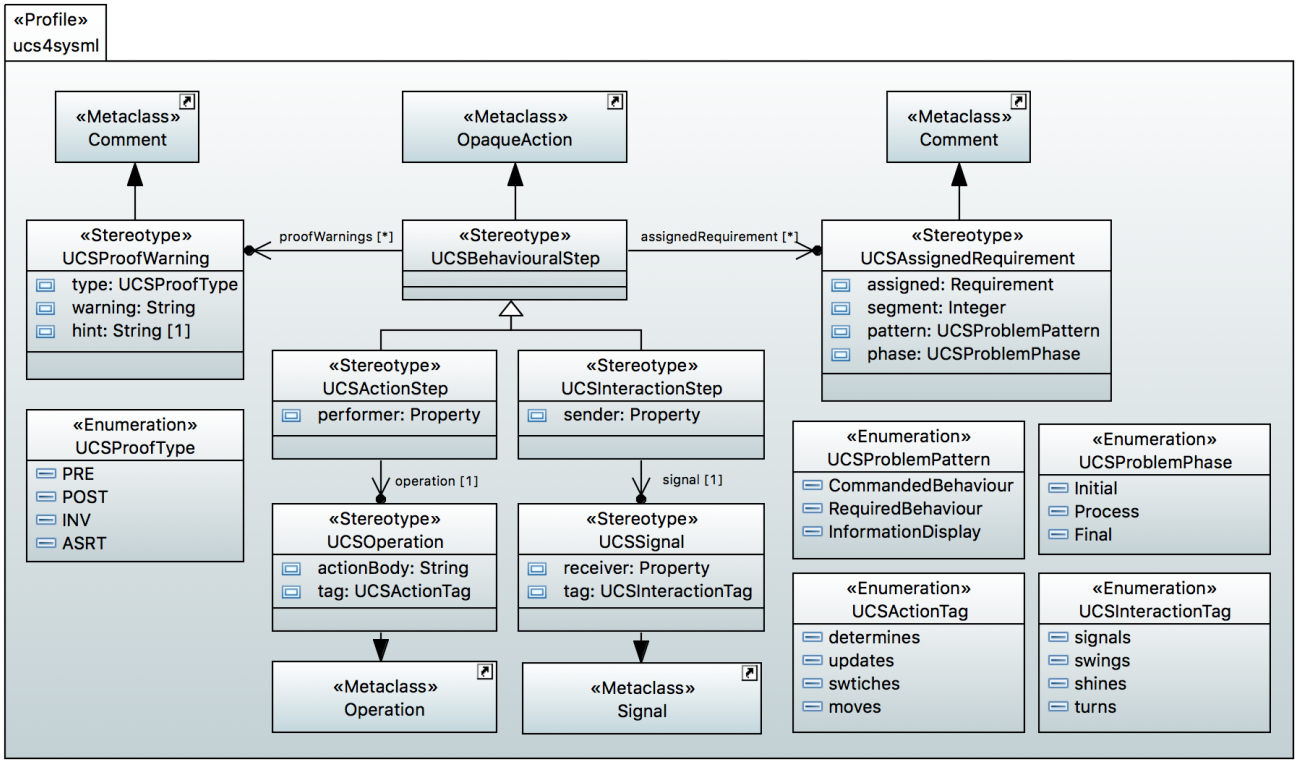

Figure 6.2 - UCS4SysML profile types of behavioural steps.

6.2.1.1 UCSBehaviouralStep

A behavioural step in the use case specification. The behavioural step is further distinguished as either an action step or interaction step in Sections 6.2.1.2 and 6.2.1.4, respectively.

A behavioural is a generalisation of UCSStep. It is an extension of OpaqueAction. This OpaqueAction is owned by the base class of the use case specification and referenced to by the partition representing the flow. This provides a suitable mechanism, where deleting the use case specification (Activity) automatically deletes the steps (OpaqueAction).

Generalization

- UCSStep

Extension

- OpaqueAction

Association

- proofWarnings: UCSProofWarnings [*]
  - A warning of proof failure assinged to this behavioural step.
- assignedRequirement: UCSAssignedRequirement [*]
  - A requirement that is assigned to this behavioural step.
- assignedRequirement: UCSAssertion [0..1]
  - An optional assertion that is assigned to this behavioural. This denotes the constraint is true before the execution of this behavioural step.

6.2.1.2 UCSActionStep

In an action step, the participant that is expected to initiate the behaviour is denoted as the performer. This step captures a change to the state of the participant via reference to an operation.

It is a generalisation of the stereotype UCSBehaviouralStep.

Generalisations

- UCSBehaviouralStep

Attributes

- performer: Property
  - Represents the performer of the action step. The property is a participant in the use case.

Association

- operation: UCSOperation [*]
  - An operation that changes the state of the /performer participant.

Constraints

[1] The performer must be a participant property from the context Block described in the use case specification.

6.2.1.3 UCSOperation

This stereotype extends a UML Operation. It captures an action describing a modification to the state of the participant and tag describing the type of operation.

Extension

- Operation

Attributes

- actionBody: String
  - Action describing a modification to the state of the participant.
- actionTag: ActionTag
  - A tag that represents the type of action.

6.2.1.4 UCSInteractionStep

An *interaction step* captures a communication of *send request* (signal) between two participants of the use case. The initiator of this step is referred to as the *sender* and a participant who is expected to receive the signal is referred to as the *receiver*. The description of the communication is denoted by a UML signal.

It is a generalisation of the UCSBehaviouralStep.

Generalisations

- UCSBehaviouralStep

Attributes

- sender: Property
  - Represents the sender in the interaction step. The property is a participant in the use case.
- tag: InteractionTag
  - A tag that represents the type of interaction.
- receiver: Property
  - Represents the receiver in the interaction step. The property is a participant in the use case.

Association

- signal: UCSSignal [*]
    - A signal that captures the communication between the sender and receiver.

6.2.1.5 UCSSignal

This stereotype is an extension to UML Signal. It captures additional information of the type of communication via a tag and which participant is the receiver.

Extension

- Signal

Attributes

- interactionTag: UCSInteractionTag
    - Type of communication between participants.
- receiver: Property
    - The participant who is the receiver of the signal.

6.2.1.6 UCSAssignedRequirement

A requirement assigned to the behavioural step. This requirement

Extension

- Comment

Attributes

- type: UCSProofType
    - The type of failed proof warning.
- warning: String
    - An informal description of the failed proof.
- hint: String [0..1]
    - An optional hint to resolve to the failed proof.

6.2.1.7 UCSProofWarning

A warning identified by a failed proof that is associated to a behavioural step. It captures the type of proof failure, an informal description of the warning and may or may not provide a hint to resolve the failed proof.

This stereotype is an extension of Comment meta-class. The comment is owned by the OpaqueAction that is extended by the UCSBehaviouralStep stereotype.

Extension

- Comment

Attributes

- assigned: Requirement
    - The requirement that is assigned to this behavioural step.
- segment: Integer
    - A numerical value that denotes which segment of the requirement is related to this behavioural step
- pattern: UCSProblemPattern
    - The type of problem pattern that is associated to this assigned requirement. This can be either a Commanded Behaviour, Information Display, Required Behaviour of Work Piece pattern of interaction.
- phase: UCSProblemPhase

- The phase in the problem pattern at which the assigned requirement refers to. This is either Initial, Process or Final.

6.2.1.8 UCSProofType

An enumeration with literals that describe the type of proof failure.

6.2.1.9 UCSActionTag

An enumeration that captures a growing collection of reusable tags to describe the type of action step.

6.2.1.10 UCSInteractionTag

An enumeration that captures a growing collection of reusable tags to describe the type of interaction step.

6.2.2 Table Extensions

6.2.2.1 Flow Table

The tabular format is used to represent the steps in the flow of the use case specification, and may include:

- A column for the kind of constraint in step, e.g. either start of use case, end of use case, behavioural step or modular step.
- A column for the unique id provided to the step.
- A column for the description of the step.
  - <interaction step> ::= <sender> <interaction tag> <signal> 'to' <receiver>
  - <action step> ::= <performer> <action tag> <operation>
  - <start of use case> ::= 'Start of use case.'
  - <end of use case> ::= 'End of use case.'
- A column for the natural language description of the step.
  - <assigned requirement> ::= <requirement id> '_' < pattern> '_' <phase> ':' <requirement segment>

Table 6.1 – Table showing steps in the basic flow of the use case specification.

table [UCSBasicFlow] POff_BasicFlow [Steps in the basic flow of Power Off.]

Kind	Id	Description	Requirement
SoU	POff_SoU	Start of use case.	
I	POff_1	**human_operator** *swings* door_Open to **power_source**	R1_CB_Initial**:** When the Human Operator opens door to the Power Source
A	POff_2	**power_source** *moves* doorToOpening()	
I	POff_3	**power_source** *signals* doorSensor_Open to **power_controller**	
A	POff_4	**power_controller** *updates* doorPositionToOpenOrPartiallyOpen()	
A	POff_5	**power_controller** *determines* powerStatusToOff()	
I	POff_6	**power_controller** *signals* power_Off to **power_source**	
A	POff_7	**power_source** *swtiches* powerToOff()	
I	POff_8	**human_operator** *swings* door_Open to **power_source**	

A	POff_9	**power_source** *swings* doorToOpen()	R1_CB_Final**:** Power Source should switch power off.
EoU	POff_EoU	End of use case.	

7 Modular Step

7.1 Overview

This section of profile introduces the modular step (UCSModularStep) as part of a flow in the use case specification. This step helps take into account (and provide a meaning) to the *includes* and *extends* use case relationships that are defined in the use case diagram. Figure 7.1 provides examples of these relationships between the use cases of a Braking System. These relationships help reduce complexity in the use case flows by factoring out common and optional behaviour via the include and extends relationships, respectively.

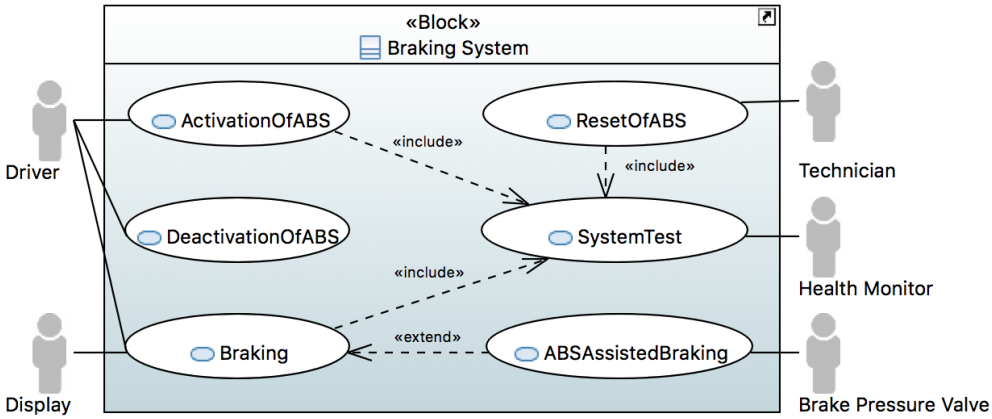

Figure 7.1 – An example of includes and extends relationships between use cases of a Braking System.

The *include* relationship provides a mechanism for factoring out common functionality that is shared among multiple use cases, and is required for the goals of the actor of the base use case to be met. For example, in the Braking System a system test is required to be performed in three different occasion: (1) activation of ABS (begin vehicle journey); (2) reset of ABS (after technician repair); and (3) each time braking is initiated. Hence, the use cases ActionOfABS, ResetOfABS and Braking all have an includes relationship that is directed towards the SystemTest[4] use case. This denotes that the behaviour of the included use case (SystemTest) is ***inserted*** into the behaviour of the including use case (ActionOfABS, ResetOfABS and Braking).

The includes step enables the use case specification for the including use cases to specify the step at which the steps of the included use case are introduced.

The *extend* relationship provides optional functionality, which extends the base use case as defined by *extension points* under specified *conditions*. For example, in the Braking System the use case ABSAssistedBraking introduces a supplementary functionality to the Braking use case. This is optionally executed where under the condition where deceleration rate is above a threshold limit. An extension point is a feature of a use case which identifies (reference) a point in the behaviour of the use case where that behaviour can be extended by some other (extending) use case, as specified by extend relationship.

The extends step enables the use case specification for the base use case to state the extension point and the extending use case that refers to this extension point.

[4] Include use case is incomplete without the behaviour of the included use. That is, the functionality of the SystemTest cannot be viewed independent of the including use cases.

7.2 UML Representation

7.2.1 Profile Diagram and Element Descriptions

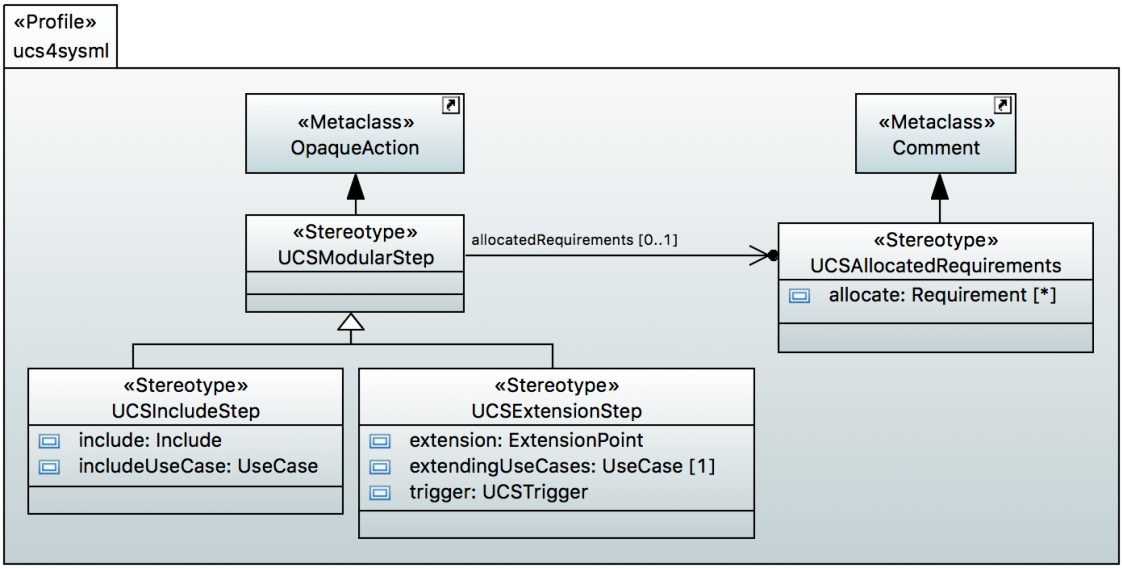

Figure 7.2 - UCS4SysML profile on types of modular steps.

7.2.1.1 UCSIncludeStep

A step in the use case specification that refers to an includes relationship where the base use case is the source and included use case is the target. The included use case is derived value.

The steps from the use case specification of the included use case is introduced at the include step.

Generalization

- UCSModularStep

Attributes

- include: Include
  - A reference to an include relationship to the use case.
- /includedUseCase: UseCase
  - A derived value of the use case to be included.

7.2.1.2 UCSExtensionStep

A step in the use case specification that refers to an extension point and a use case that refers to this extension point. The steps of the extension use case is included.

Generalization

- UCSModularStep

Attributes

- extensionpoint: ExtensionPoint
  - A reference to an include relationship to the use case.
- extensionUseCase: UseCase
  - A derived value of the use case to be included.

7.2.2 Table Extensions

7.2.2.1 Flow Table (extended with Include and Extension Step)

The tabular format is used to represent the steps in the flow of the use case specification, and may include:

- A column for the description for the modular step.
    - <include step> ::= 'Include Use Case:' <included use case>
    - <extension step> ::= 'Extension Use Case:' <extension point> '; Condition:' <extension use case>

Table 6.1 provides an example of an includes step in ABSOn_5. The basic and alternate flows of the use case Health Monitor is introduced between step ABSOn_5 and ABSOn_6.

Table 7.1 – Table showing include step at ABSOn_5.

table [UCSFlow] ABSOn_BasicFlow [Basic flow for ActivationOfABS]

Kind	Id	Description	Requirements
SoU	ABSOn_SoU	Start of Use Case	
I	ABSOn _1	**Driver** *turns* ignition_On to **Vehicle**	
A	ABSOn _2	**Vehicle** *switches* powerToOn()	
I	ABSOn _3	**Vechile** *signals* vehiclePowerStatus_On to **ABS_Controller**	
A	ABSOn _4	**ABS_Controller** *updates* testStatusToCheckForIgnition()	
In	ABSOn _5	Include Use Case: **Health Monitor**	Allocated: R2, R3
…	…	…	
EoU	ABSOn_EoU	End of Use Case	

Table 6.2 provides an example of an extension step in BR_5. The step refers to the use case ABSAssistedBraking (via extension point ImminentWheelLockUp). The basic flow and alternate flows of the extension use case is introduced back.

Table 7.2 – Table showing extension step at BR_5.

table [flow] ABSBR_BasicFlow [Basic flow for ActivationOfABS]

Kind	Id	Text
SoU	BR_SoU	Start of Use Case
…	…	…
Ex	BR_5	Extension: **ABSAssistedBraking**: Condition: Deceleration rate above threshold limit.
…	…	…
EoU	BR_EoU	End of Use Case

8 Scenarios and Test Cases

8.1 Overview

Figure 8.1 illustrates the relationship between the flows of a use-case narrative and the scenarios (or stories as Jacobson refers to them) it describes [Jacobson-2016]. Jacobson describes how the network of flows can be thought of as a map that summarizes all the scenarios needed to describe the use case.

On the left of the figure the basic flow is shown as a linear sequence of steps and the alternative flows are shown as detours from this set of steps. The alternative flows are always de ned as variations on the basic. On the right of the diagram some of the stories covered by the flows are shown.

Each story traverses one or more flows starting with the use case at the start of the basic flow and terminating with the use case at the end of the basic flow. This ensures that all the stories are related to the achievement of the same goal, are complete and meaningful, and are complementary as they all build upon the simple story described by the basic flow.

Scenarios help support the exploration of the use cases with stakeholders (often represent non-technical users). Each scenario is a *complete path* through a use case and should be of some value to the users and other stakeholders.

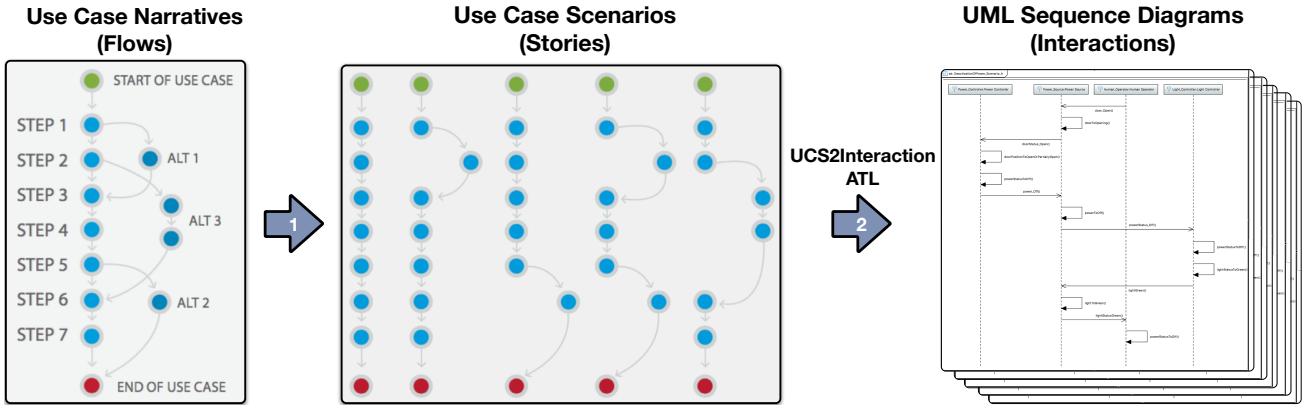

Figure 8.1 – Relation between use case narratives and scenarios (stories)

For each scenario will be one or more test cases [Jacobson-2016]. The purpose of a test case is to provide a clear definition of what it means to complete a slice of the requirements. A test case defines a set of test inputs and expected results for the purpose of evaluating whether or not a system works correctly. Test cases generated are used to provide a mechanism to complete and verify the requirements (ongoing implementation).

8.2 UML Representation

8.2.1 Profile Diagram and Element Descriptions

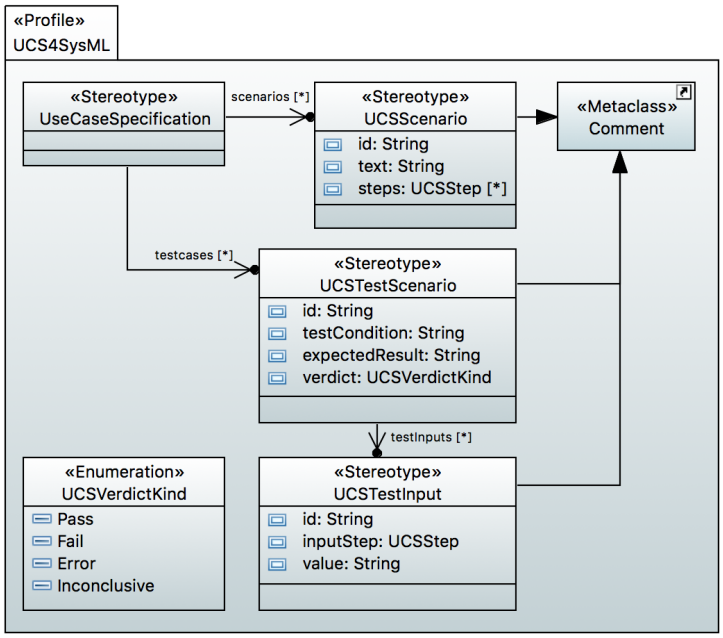

Figure 8.2 - UCS4SysML profile on Scenarios and Test Cases.

8.2.1.1 UseCaseSpecification (Extended with Scenarios and Test Cases)

The use case specification stereotype is further (from Section 5) with the associations *scenarios* and *test cases*. Each use case specification is allowed to contain a collection of scenarios that describe complete paths through the use case's flow. The test cases are derived through model execution (Event-B).

Associations

- scenarios: UCSScenario [*]
  - A collection of scenario that covers a complete path through the use case's flows. There may be one or more scenarios depending on the number of alternate flows.
- testcases: UCSTestScenario [*]
  - A derived value of the use case to be included.

8.2.1.2 UCSScenario

This stereotype represents a scenario in the use case specification. Each scenario captures a unique id, a brief textual description of the scenario, and a collection of steps that represent a complete path through the execution of the use case..

It extends the UML comment model element. This base comment owned by the base activity of the UseCaseSpecification stereotype.

Extensions

- Comment (UML)

Attributes

- id: String
  - A unique id for the scenario.
- text: String

    - A brief textual description of the scenario.
- steps: UCSSteps [*]
    - A collection of steps in the use case scenario that represents a complete path through the use case.

8.2.1.3 UCSTestCase

For each scenario can have one or more test cases. The purpose of a test case is to provide a clear definition of what it means to complete a slice of the requirements. A test case defines a set of test inputs and expected results for the purpose of evaluating whether or not a system works correctly.

Extensions

- Comment (UML)

Attributes

- id: String
    - A reference to an include relationship to the use case.
- testCondition: String
    - A brief textual description of the condition for the test case.
- expectedResult: String
    - A brief textual description on the expected result for the test case.
- verdict: UCSVerdictKind
    - A verdict provided by the test engineer on the result of performing the test case.
- \verifies: Requirement [*]
    - A collection of requirements that the test case is expected to verify.

8.2.1.4 UCSVerdictKind

Each test case is provided with a verdict by the test engineer. The UCSVerdictKind is an enumeration with four literal values Pass, Fail, Error or Inconclusive.

Literal Values

- Pass
    - Test case successful.
- Fail
    - Test case failure.
- Inconclusive
    - Test case incomplete.

8.2.2 Table Extensions

8.2.2.1 Scenario Table

The tabular format is used to represent the scenario in the he use case specification, and includes:

- The kind of step in the scenario.
- The Id of the step in the scenario.
- The description of the step in the scenario (descriptions same as the flow table).

Table 8.1 – Table showing a scenario of power off use case.

table [UCSScenario] POff_Scenario_1 [Scenario covering basic flow of Power Off Use Case.]

Kind	Id	Description
SoU	POff_SoU	Start of use case.
I	POff_1	**human_operator** *swings* door_Open to **power_source**
A	POff_2	**power_source** *moves* doorToOpening()
I	POff_3	**power_source** *signals* doorSensor_Open to **power_controller**
A	POff_4	**power_controller** *updates* doorPositionToOpenOrPartiallyOpen()
A	POff_5	**power_controller** *determines* powerStatusToOff()
I	POff_6	**power_controller** *signals* power_Off to **power_source**
A	POff_7	**power_source** *swtiches* powerToOff()
I	POff_8	**human_operator** *swings* door_Open to **power_source**
A	POff_9	**power_source** *swings* doorToOpen()
EoU	POff_EoU	End of use case.