The UML4SOA Profile

Philip Mayer, Nora Koch, Andreas Schroeder, Alexander Knapp
Technical Report, LMU Muenchen, 2010

Abstract

This document describes UML4SOA, a profile for the Unified Modeling Language (UML) [OMG10b] which extends the UML by providing the possibility for behavioural specifications of services, focussing on service orchestrations. UML4SOA is an easy-to-use, conservative extension to the UML for modelling service orchestrations on a high level of abstraction, and allows a fully automated, model-driven approach for transforming orchestrations down to code. It is based on the upcoming OMG standard *SoaML* [OMG09].

Metadata

- Current Version: 3.0
- Date: 2010-10-02
- Authors: Philip Mayer, Nora Koch, Andreas Schroeder, Alexander Knapp

License

Acknowledgements

This work has been partially supported by the EC project Sensoria, IST-2005-016004.

Contents

Chapter 1

Extending UML for Service Behaviour

The Unified Modeling Language (UML) [OMG10b] is a well-known and mature language for modelling software systems with support ranging from requirement modelling to structural overviews of a system down to behavioural specifications of individual components. However, UML has been designed with object-oriented systems in mind, thus native support and top-level constructs for service-oriented computing such as participants in a SOA, modelling service communication, and compensation support are not included. As a consequence, modelling SOA systems with plain UML requires the introduction of technical helper constructs, which degrades usability and readability of the models.

In this chapter, we therefore introduce a UML extension for SOAs — called the *UML4SOA profile* — which is a high-level domain-specific language for modelling the *behaviour* of services, service orchestrations, and service protocols. For modelling the structural aspects of services, we build on the upcoming OMG standard *SoaML* [OMG09]. One of the main goals of UML4SOA is minimalism and conciseness: service engineers should have to provide only as much information as necessary for the generation of code, and at the same time as little as possible in order to keep diagrams readable.

The Unified Modeling Language Infrastructure [OMG10a] describes several ways of extending the UML for specific modelling purposes, among them being *profiles*, a lightweight mechanism of defining domain-specific modelling languages on top of the UML, which we will adopt here. In the following, we describe why a profile for behavioural SOA modelling on top of the UML is desirable before moving on to the description of the profile in the next chapter.

1.1 Motivation

Behavioural modelling of services and service orchestrations has several requirements on a (graphical) modelling language; in particular, the following three

concepts — first-level citizens of a SOA system — should be supported:

- *Communication and Partners.* Services are inherently based on a networked architecture, i.e. communication between services is a key requirement for a working SOA-based system. Communication primitives for sending and receiving calls must thus be supported in a domain-specific SOA language; furthermore, specification of communication partners should be possible in a straightforward way.
- *Long-Running Transactions.* A service, and in particular a service orchestration may represent a business transaction and thus potentially run for a long time. This has various requirements for the modelling language: It must be possible to query the transaction for status updates; it must be possible to handle problems occurring during the transactions, and finally successfully completed transactions might need to be undone due to later failures.
- *Self-Descriptions.* Part of the appeal of service-oriented computing is the focus on a clear self-description of each component in the SOA. Regarding the behaviour, a specification of the *protocol* a service provides or requires is key to enabling quick and confident assembly of SOAs.

UML already includes several ways of specifying the behaviour of software systems which we can extend for modelling service behaviour. In particular, the following two model elements and accompanying diagram types are a good match for modelling SOA behaviour:

- *Activities.* In UML, activities are used for modelling the behaviour of a software component based on a workflow-like paradigm. Workflows are a concept which is also common outside of software modelling; in particular, an interesting area are business processes as they closely match the abstraction level of SOAs. We use activities as the basic mechanism for specifying service and service orchestration behaviour.
- *State Charts.* The UML distinguishes between behavioural and protocol state machines; the first being used to model element behaviour, the second for describing the behaviour of a protocol. As communication is a key aspect of services and service orchestrations, modelling the protocol of a service is important to be able to identify matching service implementations. We therefore use protocol state machines, with a minimal extension to be able to model observed operation calls, in addition to activity modelling for specifying the required or provided protocol of a service.

Attempting to model services using these two UML elements and diagrams while considering the three key aspects of SOA systems discussed above shows several important shortcomings of the UML. We consider a case study from the domain of computer-assisted university management as an example for these problems.

1.2 Case Study

The administration of a university is a complicated task. Student applications, enrolment, course management, theses, and examination management all pose individual problems and, in general, a lot of paperwork. Nowadays, many of these tasks can be and are being automated. As universities are often large organisations with autonomous sub-organisations, a promising approach for this is the use of SOA-based software, in which the individual parts of a university as well as (external) students can work together with respective back- and front-ends of a web-based system.

To investigate the problem of developing SOA-based university management systems, the SENSORIA project includes a case study based on a set of university scenarios that make use of the specific features of SOAs [HÖ7]. In particular, we consider eUniversities, i.e., universities in which at least all of the paperwork, if not the courses themselves, are handled online.

The chosen scenario here is the problem of *Thesis Management*. In this scenario, we have considered the management of a student thesis (bachelor, master, or diploma) from the initial announcement to the final grading.

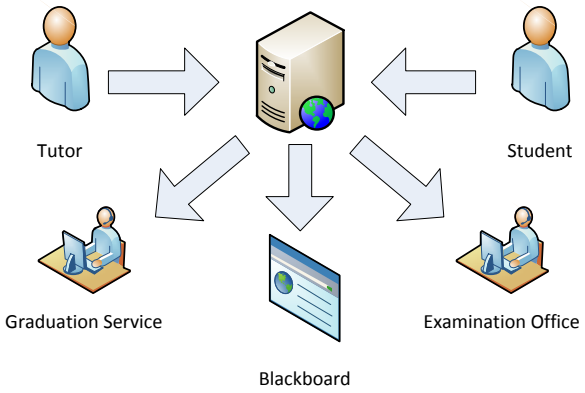


Figure 1.1: eUniversity Case Study: Overview

The scenario consists of six partners and computing systems working together. As shown in figure 1.1, a tutor (left) first provides a new thesis proposal — bachelor, master, or diploma — to a central server (middle), which distributes it to a university-wide blackboard (bottom centre) to inform students of this opportunity. Once a student (right) decides to pick up a thesis, the central server starts the thesis, informing both the examination office (bottom right), and registering the student for the graduation ceremony (bottom left).

While the thesis is in progress, the student may provide updates, which the tutor may retrieve. Once the thesis is declared to be finished, an assessment is

requested from the tutor. If the assessment is positive, the examination office is instructed to issue the corresponding certificates. If not, the examination office is informed of the problem, and the graduation service needs to unregister the student from the graduation ceremony.

1.3 Modelling the Case Study

The structural aspects of the case study, modelled using UML and SoaML, can be seen in figure 1.2. As noted above, UML4SOA is concerned with behaviour. There are several entities in figure 1.2 for which we might want to specify behaviour. First of all, there is the central participant `ThesisManagement`, which is a service orchestration for which the workflow might be specified. Secondly, the participant contains service and request ports at which services are provided and required, their interfaces being specified as $\ll$*ServiceInterface*$\gg$s. For these interfaces, the protocol may be specified as a state chart. Finally, the services required by the orchestration have their own behaviour which may be implemented by external means, or may be modelled using UML or UML4SOA as well.

1.4 Requirements for UML4SOA

As an example for the three requirements of modelling SOA systems given above, we model the behaviour of the participant `ThesisManagement` and its required and provided protocols. This example will also be used in the remainder of this document.

1.4.1 Communication Actions

The first requirement is the ability to specify communications in-between services. The `ThesisManagement` orchestration, for example, is contacted by a student accepting a thesis, subsequently registering this thesis with the examination office. Modelling this sequence as an activity looks like the diagram shown in figure 1.3.

The first action in the figure is an UML `AcceptEventAction` or subclass thereof. It identifies a point in the workflow where the process waits for an incoming event or operation call. The result of the call is placed in an output pin, which is denoted with an arrow leading away from the action. The second action in the figure is an UML `InvocationAction` or subclass, and shows that the process sends out an event or operation call. There are several problems with modelling service communication in this style:

- In UML, there is no graphical distinction between the various subclasses of `AcceptEventAction`, and — even worse — no distinction between an `InvocationAction` and a generic action. This requires a description such as the one given above to precisely define the semantics of the diagram. A

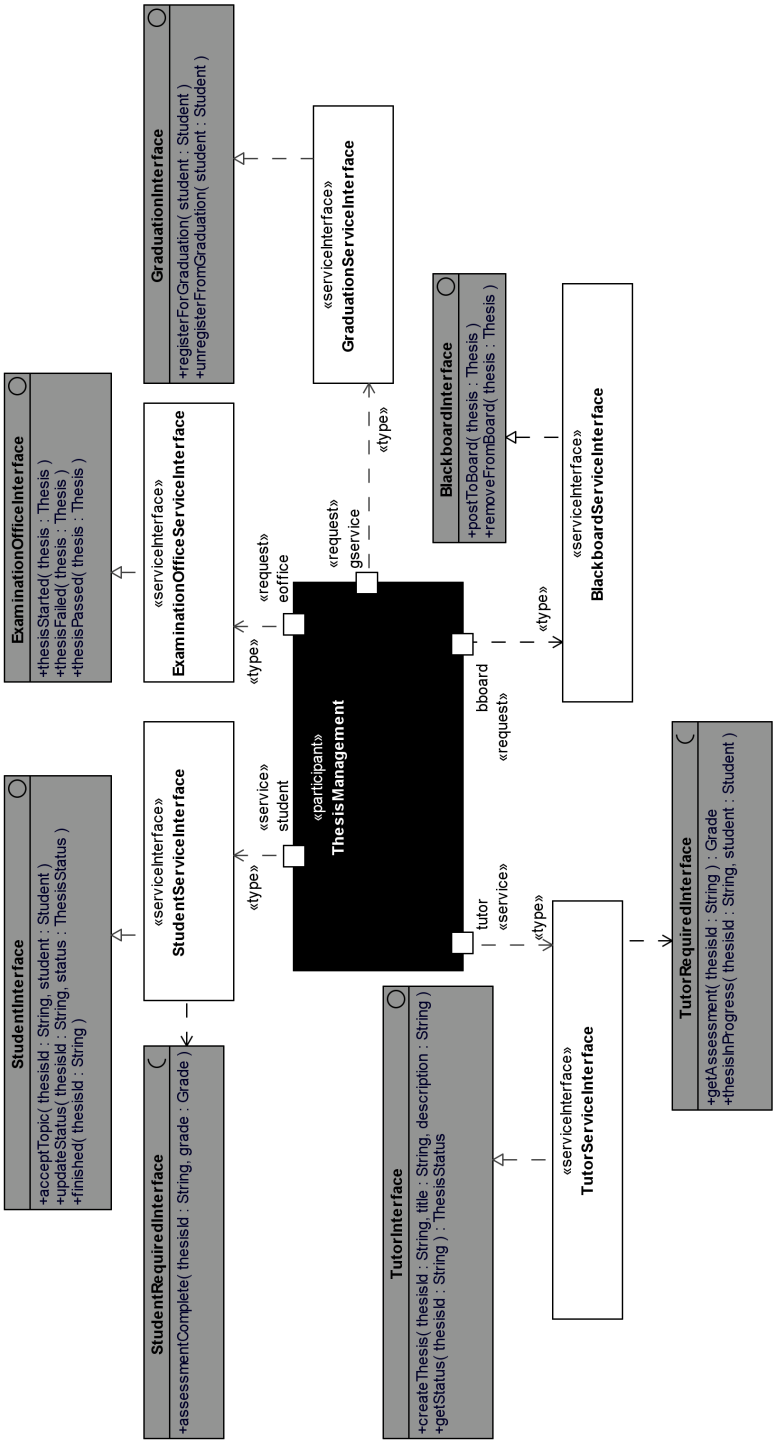


Figure 1.2: eUniversity Case Study: Static Model

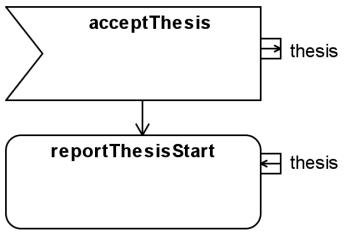


Figure 1.3: eUniversity Case Study: Communications

better way would be to use precisely specified symbols or tags to identify a receiving, sending, or replying action in the sense of a service communication.

- Each of the operations referred to by an action (and denoted in the body) are part of a `ServiceInterface` attached to a service- or request port, specifying the port on which an event or operation is expected or sent out. This way of specifying a port is rather indirect and not visible in the diagram, which does not lay well with the fact that partner services are a first-level concept in a SOA. Furthermore, there is no way of specifying *which* port is to be used if several ports share one `ServiceInterface`. Thus, specifying the port as part of an action would greatly increase readability of the model and also allow for more precise specifications.

- Finally, standard input- and output-pins are used to denote data to be received or sent. In the case of an `AcceptEventAction`, an output pin is used as the data is a result of the action; in the case of an `InvocationAction`, an input is used as the data is used in the action. When considering service calls, however, another intuition is possible: Data received in the process is *input* data, and data sent is *output* data. A different notation for data sent and received in a service context can clarify the intuition in use.

We shall come back to these three problems in the next chapter, where we define our UML profile.

1.4.2 Long-Running Transactions

The second requirement discussed above is support for *long-running transactions*. Services and, in particular, service orchestrations may be used to specify business or technical processes which potentially run a long time. There should be specific support for such processes in the modelling language; in particular, a long-running transaction may run into problems which need to be handled; it may need to be queried for status updates, and it may need to be rolled back in case of subsequent errors.

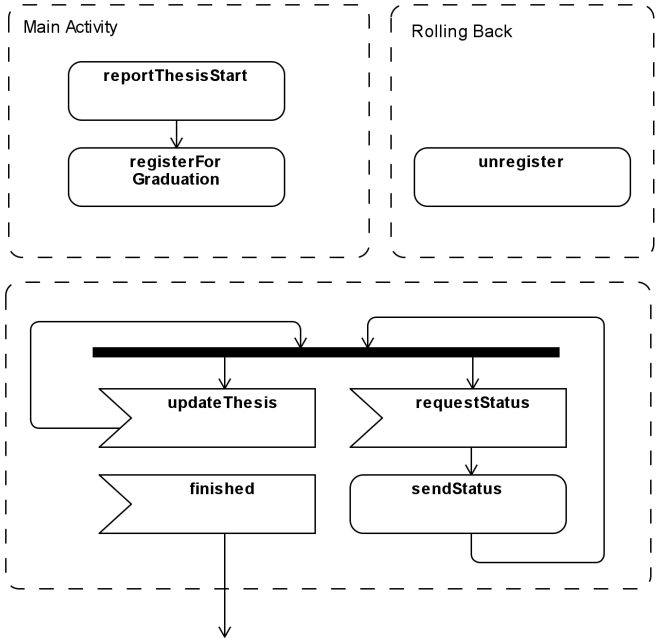


Figure 1.4: eUniversity Case Study: Long-Running Transactions

Consider figure 1.4 which contains a plain UML activity diagram which models two different part of the eUniversity case study. In the upper part, the bootstrapping of the thesis is modelled, in which a student has already accepted a thesis topic and the necessary messages are sent out to register this information. This initialisation might later need to be rolled back (compensated), for example if the student has already been registered for a graduation ceremony. In the lower part, the thesis is in progress. The student provides updates until he is finished, and additionally — and concurrently — the tutor might ask for the current status. Again, there are several problems involved in the figure.

- Firstly, the behaviour for rolling back the main activity later cannot be attached to the activity itself — it must be placed at the point where the rollback(s) takes place and therefore a different place than expected in the diagram. A better way would be associating rollback actions directly with the element to be undone.

- Secondly, modelling concurrent behaviour which might occur multiple times — such as the tutor requesting the status — is difficult to model in UML. The status requests are in fact optional, which is enabled by the interrupting edge leaving the interruptible activity region. Needless

to say, this is not very intuitive to write and read; a better separation between the *main* behaviour and events such as the status updates might be appropriate.

UML4SOA addresses these concerns as discussed in the next chapter.

1.4.3 Self-Descriptions

Most of the basic definitions of services include a notion of self-description, i.e. the ability of a service to describe, more or less completely, how it can be invoked. The SoaML model for the eUniversity case study shown in figure 1.2 (page 11) already addresses the static aspect of such self-description: The main participant provides two services through the ≪*Service*≫ ports whose operations are given as part of the `StudentServiceInterface` and the `TutorServiceInterface`, and requires two services through its ≪*Request*≫ ports whose operations are given in their respective interfaces. While these descriptions are required as the basis for service interactions, the actual protocol of a provided or required service is not yet specified.

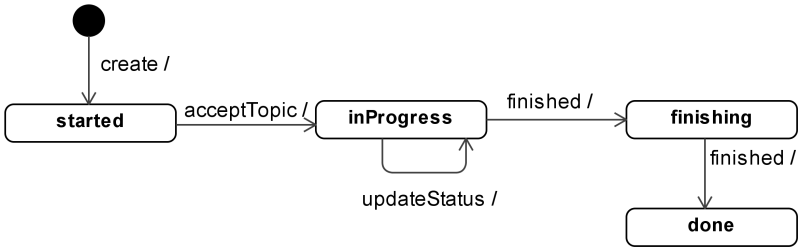


Figure 1.5: eUniversity Case Study: Protocol

UML Protocol State Machines (PrSMs) can be used for this purpose; in our case, they are attached to the types of the SoaML service and request ports, i.e., the `ServiceInterface`s. A protocol state machine in UML may contain states and (protocol) transitions; the latter of which may contain triggers. An example for a standard UML protocol state machine for the `Tutor ServiceInterface` is shown in figure 1.5.

In this figure, two types of events are used: `acceptTopic`, `updateStatus` and the first `finished` transition are based on `ReceiveOperationEvent` triggers, while the second `finished` transition is based on a `SendOperationEvent` trigger. There are two problems associated with this diagram.

First, the fact that the first three transitions actually use a `ReceiveOperationEvent`, and the last an `SendOperationEvent` is not visible in the figure, degrading readability and hampering the comparison with the actual service behaviour (which needs to fulfil this protocol).

Second, the `finished` transition is not legal in the standard definition of protocol state machines. Although it only *observes* an event, this event is not targeted at an operation *implemented* by the classifier the PrSM is attached to, but rather by a required interface of the classifier. However, as we believe that the observation of calls to external services are an important aspect of service specifications, this ability should be added.

Finally, in the interest of ease of modelling and readability, it is again desirable to have a special notation for service-related communication.

Due to the shortcomings discussed above, modelling service behaviour and protocols with plain UML is a cumbersome task. At the same time, the resulting UML models are difficult to read and translate into executable code. Thus, we have developed the UML4SOA profile and meta-model which adds specific support for services, service orchestrations and service protocols to the UML.

Chapter 2

The UML4SOA Profile

This chapter introduces the *UML4SOA* profile, a domain-specific, graphical notation for modelling service behaviour and service protocols. Extending the UML is possible via several mechanisms, among them MOF meta-model extensions and UML profiles. Both mechanisms can also be combined, an approach which has been used for UML4SOA as well.

In section 2.1, we discuss the design decisions behind the profile, introducing — on a high level — the concepts UML4SOA contributes to the UML. In section 2.2, we define the meta-model in a MOF modelling style. Mapping of this meta-model to a UML profile takes place in section 2.3. Data handling in UML-4SOA is discussed in 2.4. We discuss the difference between UML4SOA/Open and /Strict in section 2.6, and finally talk about life cycle management in section 2.7.

2.1 Design Considerations

The design of a meta-model for behavioural service specifications requires specific support for the three concepts *communication and partners*, *long-running transactions*, and *self-descriptions* already introduced in chapter 1, which we shall revisit in this section, discussing how UML4SOA addresses these issues.

Furthermore, an important point in specifying service behaviour is data handling, in particular in service orchestrations: Data must be received, might need to be manipulated, and then passed on. Although the UML defines a set of actions for explicitly dealing with data, a textual DSL for data handling greatly simplifies this task for developers.

We begin with defining special support for service communication in activities in section 2.1.1, discuss long-running transactions in section 2.1.2, introduce self-descriptions as protocols in section 2.1.3, and finally discuss data handling in section 2.1.4.

2.1.1 Service Interactions and Partners

In section 1.4.1, we have noted that although using subclasses of the UML `InvocationAction` or `AcceptEventAction` actions in activities is the preferred way of modelling communication, using this approach suffers from several problems. As a remedy, UML4SOA adds specific support for the requirements of service communication: Firstly, actions are explicitly marked as *sending*, *receiving*, or *replying*. Secondly, we add specialised pins for specifying input and output data. These pins are stereotyped with new icons which precisely show whether data is sent or received. Finally, the partner service an action relates to is attached to an action in a new pin. An UML4SOA diagram replacing the pure UML diagram from section 1.4.1 is shown in figure 2.1.

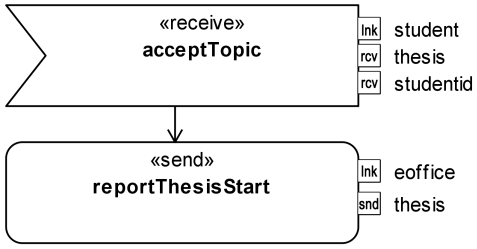


Figure 2.1: eUniversity Case Study: Communications in UML4SOA

As usual in the UML, we employ operation specifications for referencing which functionality is invoked in an interaction. In the case of services, these operations are specified in interfaces or classes used as types of the ≪*Service*≫ or ≪*Request*≫ ports of SoaML. We distinguish:

- service invocations (e.g. `reportThesisStart`), for which we define the new stereotypes ≪*Send*≫ and ≪*Send&Receive*≫ for invoking an action without or with an expected return information. A service invocation is an interaction with a named partner, in which an operation is called, which may, as usual, have parameters and return types.
- service receives (e.g. `acceptTopic`), for which we define the new stereotype ≪*Receive*≫. A service receive is a point where a behaviour waits for an incoming call from a partner, receiving an operation invocation which may, again, have parameters.
- Finally, we add the notion of service replies, which are used to answer a call previously received from a certain partner, and add the new stereotype ≪*Reply*≫ for this notion. As usual in UML, a reply ends a previous invocation.

Each of the service actions may have associated pins which are again stereotyped with UML4SOA stereotypes. They are used to specify the following information:

- a ≪*Lnk*≫ (link) pin specifies the partner for an operation (i.e., the port through which messages are sent or received, for example `student` or `eoffice` in the example above),
- a ≪*Rcv*≫ (receive) pin denotes where received information is stored. In general, this will be a variable (`studentId` and `thesis` in the example above),
- a ≪*Snd*≫ (send) pin denotes the information to be sent as part of a call (the variable contents of the `thesis` variable in the example above). Besides the contents of variables, such data may also be generated on-the-fly (for example, by string concatenation).

More information about the data handling syntax is given in 2.1.4.

2.1.2 Events and Compensation

Handling long-running transactions in the SOA world has two major requirements. First of all, it should be possible to query a long-running service for its status or other information, which is additional work the service has to carry out in addition to the normal behaviour. Secondly, successfully completed work might need to be undone in a customised way (transaction rollback).

The first issue is handled in UML4SOA by means of *event handlers*, which allow the specification of optional behaviour occurring in parallel to the main work of a service. The second issue is handled by means of *compensation handlers*, which allow attaching roll-back behaviour to a certain action or set of actions of a process definition. An example of both is shown in figure 2.2, the counterpart of figure 1.4 in UML4SOA.

As the figure shows, UML4SOA introduces a grouping concept — the ≪*ServiceActivity*≫ — to which specialised edges for event and compensation handling can be attached.

Firstly, the figure shows how to attach compensation handling to an area in UML4SOA (top half). We use a specialised edge ≪*Compensation*≫, indicating that compensation handling is available for a certain area. In the example, the complete `Registration` activity can be rolled back by executing the action in the `CompensationHandler` activity. By attaching these actions to the area to be compensated, we only need to specify them once, and they are defined in close context to their counterpart.

With regard to events, UML4SOA includes a specialised ≪*Event*≫ edge to attach an event handler to a certain area. In the example, an event handler is attached to the `InProgress` activity. This means that during the waiting time for any number of `updateStatus` calls or a final `finished` call, a `getStatus` call might come in and is answered. In general terms, an event is a message which might be accepted during the run of a certain element in the workflow, asking — for example — for status information or for cancellation. The ≪*Event*≫ edge allows us to specify such events in an easy and readable way.

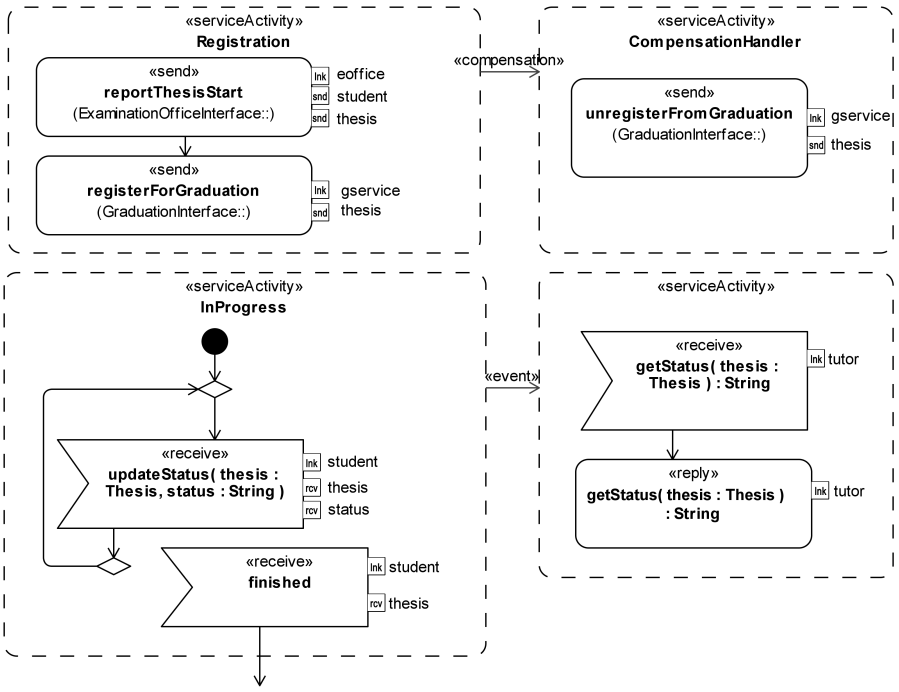


Figure 2.2: eUniversity Case Study: Long-Running Transactions in UML4SOA

2.1.3 Self-Describing Protocols

UML4SOA orchestration specifications are complemented by protocols assigned to the ports of the SoaML participant. A port protocol always describes actions of the ≪*Participant*≫ — either actions provided to clients or actions invoked on or expected from partners.

As noted in the previous section, the definition of protocol state machines in UML [OMG10b] allows referencing operations *implemented* by the context classifier. An important part of service behaviour, on the other hand, is sending out calls to partner services; these operations are *used* by context classifiers. UML4SOA thus extends the ability of PrSMs to include triggers with a `SendOperationEvent` event. It is important to note that this event is not an *effect* of a transition; rather; it is an *observed operation call* of the participant the classifier of the PrSM is attached to.

As already discussed in section 2.1.3, it is again beneficial to tag the individual parts of a PrSM to clarify the semantics of UML4SOA protocols. The following actions may be observed in a service protocol:

- The receipt of an invocation is observed as a (UML) `ReceiveOperationEvent`. UML4SOA adds a special transition with this constraint with the

≪*Receive*≫ stereotype.

- A service invocation is observed as a (UML) `SendOperationEvent`. UML-4SOA adds a special transition with this constraint with the ≪*Send*≫ stereotype.
- As noted above, a service invocation might be a reply to a previous ≪*Receive*≫. For clarity, this is modelled separately in UML4SOA, although we observe again a (UML) `SendOperationEvent`. The UML4SOA stereotype for a service reply transition is ≪*Reply*≫.
- Finally, a protocol may also need to specify the fact that a reply is *expected* from a partner in response to a previous ≪*Send*≫. The UML4SOA stereotype for an expected reply transition is ≪*ReceiveReply*≫; the event used is again a `ReceiveOperationEvent`.

A matching UML4SOA diagram for figure 1.3 is shown in figure 2.3.

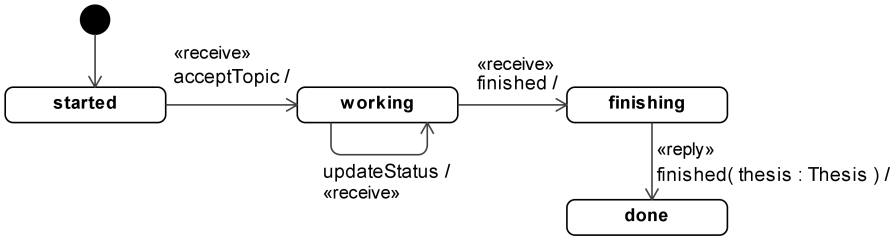


Figure 2.3: eUniversity Case Study: Protocols in UML4SOA

The protocols specified at the ports of a ≪*Participant*≫ must *match* the behaviour of the orchestration. Furthermore, clients and partners must be *compatible* with the given protocol for the system to work correctly.

2.1.4 Data Handling

Services in a real-world environment are about data: Whether from user input, databases, or as a result of a lengthy calculation, data needs to be handled within services and received via or sent over the network. This holds especially true for service orchestrations, part of whose job is the distribution of data.

The SoaML profile [OMG09] already defines the static aspects of data handling with the ≪*MessageType*≫ stereotype, which tags data classes without behaviour to be used in operation calls. In UML activities, certain actions such as `ReadVariableAction` or `AddVariableValueAction` are available for dealing with data. In the interest of both readability and usefulness, UML4SOA replaces these with a simple data manipulation language on top of message types, which allows assignments of (parts of) data and manipulations such as simple mathematical operations or string concatenation. It is important to note that

this language is not an *action language*; its sole purpose is the specification of data manipulation statements.

A syntax for data handling is relevant in three parts of UML4SOA specification:

- *Variables.* (UML) variables hold the data of the service. A variable is referred to in UML4SOA receive pins (for storing data) and in send pins (for data to be sent out).

- *Guards.* A guard (for example on an outgoing edge from a decision node) may contain a boolean condition which might reference data from one of the variables.

- *Explicit Data Operations.* Finally, sometimes using on-the-fly data handling is not enough; for example, when performing complex copy operations between input and output operations. UML4SOA introduces a specific action for these transactions.

The UML4SOA data handling language is a strongly typed language built on primitive data types and the message types defined in the SoaML model part. Inspired by the Java syntax, the language reads like pseudo code and matches the overall abstraction layer of UML4SOA. An example for three different expressions in this language is shown in listing 1.

Listing 1: UML4SOA Data Handling Example

```
String currencies;
currencies= convert.from + "-" + convert.to
::Main.conversionInfo= currencies
```

The first line declares a variable `currencies` with the well-known UML type String. The second line assigns a field of a variable `convert` concatenated with the constant string "-" concatenated with another field of `convert` to `currencies`. The third line assigns `currencies` to an existing variable `conversionInfo` which resides in an enclosing element called `Main`.

In general, a UML4SOA $\ll Rcv \gg$ pin contains what is usually regarded as the left-hand side of an expression, i.e. the specification of where to store data. This will normally be some variable which, if it does not exist, is implicitly created in UML4SOA. A $\ll Snd \gg$ pin may contain what is usually regarded as the right-hand side of an expression, i.e. a complete statement including mathematical or string operations.

Regarding more complex data operations, UML4SOA adds a new action for data handling with the stereotype $\ll Data \gg$, which may contain statements for declaring variables and manipulating data.

2.2 The UML4SOA Meta-Model

In this section, we define the UML4SOA meta-model which forms the basis for the UML4SOA profile. The meta-model is closely based on the UML meta-model and in particular, UML activities and protocol state machines.

The two figures 2.4 and 2.5 on pages 24 and 25 show the complete meta-model. Grey classes are taken from the UML while white classes are newly defined in UML4SOA. Note that some classes appear twice for layouting purposes.

The two figures can be grouped into four areas. In the first, the top shows the main structuring element of UML4SOA (`ServiceActivityNode`) along with actions and edges for specifying compensation and data. We shall discuss these in section 2.2.1. The bottom of this figure shows the protocol extensions, which we discuss last (section 2.2.4).

In the second figure, the service communication actions are shown, which are linked to data pins on the bottom. We shall discuss the pins first in section 2.2.2, afterwards using them in the definition of the communication actions in section 2.2.3.

2.2.1 Structuring Elements

Structuring service behaviour diagrams is important not only for readability, but for being able to handle events and compensation in a straightforward way. As discussed in the last section, UML4SOA introduces the structuring concept of `ServiceActivityNode`s, to which event and compensation handlers can be attached. Compensation handlers can later be invoked by using specialised actions.

ServiceActivityNode

Description

A `ServiceActivityNode` represents either

1. a special `Activity` for service behaviour, or
2. a grouping element for actions and other `ServiceActivityNode`s (top-level, and nested)

A `ServiceActivityNode` may have control edges connected to it, and pins when merged with CompleteActivities or on specialisations in CompleteStructuredActivities. The execution of any embedded actions may not begin until the `ServiceActivityNode` has received its object and control tokens. The availability of output tokens from the structured activity node does not occur until all embedded actions have completed execution. Note that completion waits for already running event handlers.

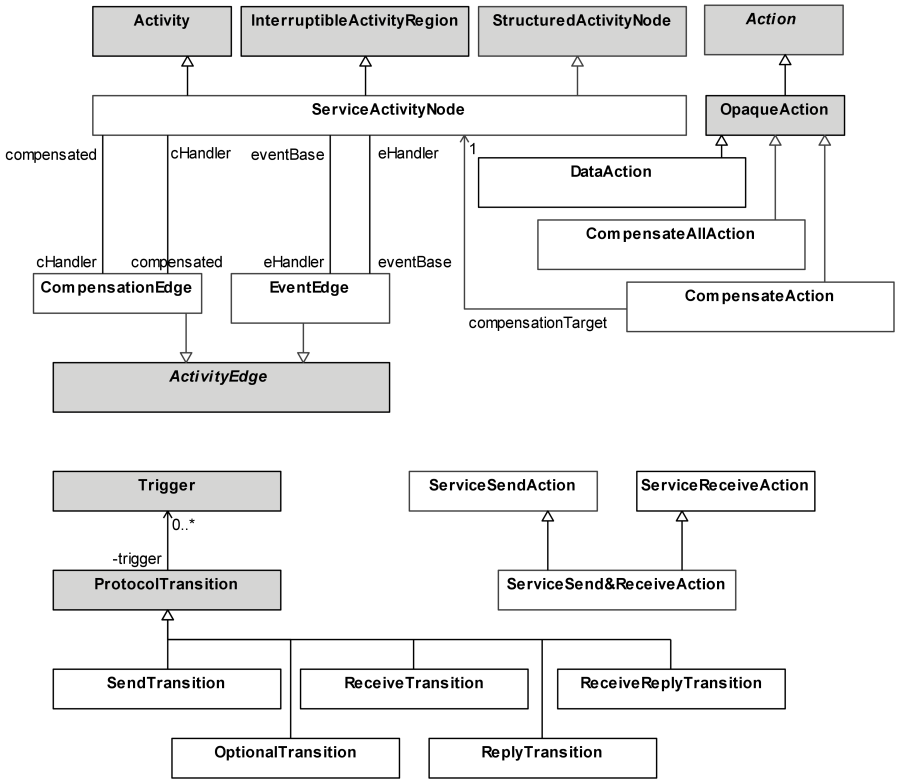


Figure 2.4: UML4SOA Meta-Model (Structures and Protocols)

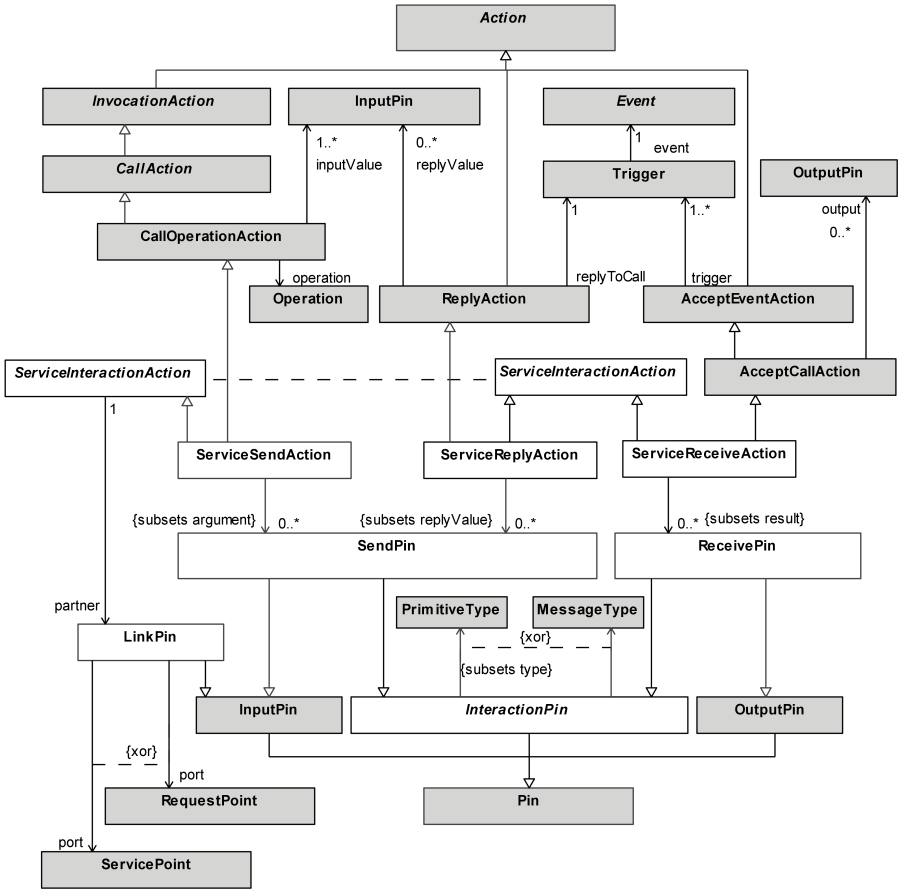


Figure 2.5: UML4SOA Meta-Model (Actions and Pins)

In addition to both `Activity` and `StructuredActivityNode`, a `ServiceActivityNode` node may have attached event and compensation handlers. An event handler may be executed at any time during the execution of the `ServiceActivityNode`, running in parallel to the `ServiceActivityNode`. Event handlers may be invoked multiple times. A compensation handler, on the other hand, defines behaviour to be executed to undo the work of a successfully completed `ServiceActivityNode`. Note that if no compensation handler is defined for a `ServiceActivityNode`, a default handler with a ≪*CompensateAll*≫ action is assumed.

Furthermore, interrupting edges may halt execution at any time (as already defined in the UML class `InterruptibleActivityRegion`).

A top-level service activity is attached to a SoaML ≪*Participant*≫ and defines the behaviour of the ≪*Participant*≫ across all service- and request ports.

Generalisations

- `StructuredActivityNode`
- `InterruptibleActivityRegion`
- `Activity`

Associations

- `eHandler` : `EventEdge`[0..*]
  An event edge leading to an event handler for this activity.
  {subsets `outgoing`}
- `eventBase` : `EventEdge`[0..*]
  An event edge leading to another activity for which this activity is an event handler.
  {subsets `incoming`}
- `cHandler` : `CompensationEdge`[0..1]
  A compensation edge leading to a compensation handler for this activity.
  {subsets `outgoing`}
- `compensated` : `CompensationEdge`[0..1]
  A compensation edge leading to another activity for which this activity is the compensation handler.
  {subsets `incoming`}

Constraints

1. If a compensation handler is specified, the target element must have this element as the `compensated` element.
2. If event handlers are specified, each of them must have this element as the `eventBase` element.

Notation

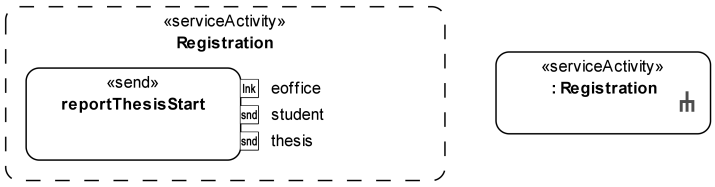
As a `ServiceActivityNode` comes in two versions, there are two notations.

1. `StructuredActivityNode` notation: the `ServiceActivityNode` is drawn with a dashed round cornered rectangle enclosing its nodes and edges, with the stereotype notation ≪*ServiceActivity*≫ at the top. Also see children of `StructuredActivityNode`.

2. `Activity` notation: Same notation as for activities applies; as before, the ≪*ServiceActivity*≫ stereotype must be used.

Examples

The following examples show the use of a service activity. The activity on the left contains one action with the stereotype ≪*Send*≫, which in turn contains three pins. The service activity is annotated with the ≪*ServiceActivity*≫ stereotype, and carries a name (`Registration`). On the right, an action for invoking the activity without displaying the internals is shown.



CompensationEdge

Description

A `CompensationEdge` is an edge connecting a `ServiceActivityNode` to be compensated with the one specifying a compensation. It does not model a normal control flow — instead, it indicates an association between a (main) service element and a compensation handler. Execution of a compensation handler is triggered with a `CompensateAction` or a `CompensateAllAction`.

Exceptions thrown during a compensation handler must be handled in the invoking `ServiceActivityNode`, or in a handler attached to the compensation handler.

Generalisations

- `ActivityEdge`

Associations

- `compensated` : `ServiceActivityNode`[1..1]
  The service activity which is compensated.

{subsets `source`}

- `cHandler` : `ServiceActivityNode`[1..1]
  The service activity specifying the compensation actions.
  {subsets `target`}

Constraints

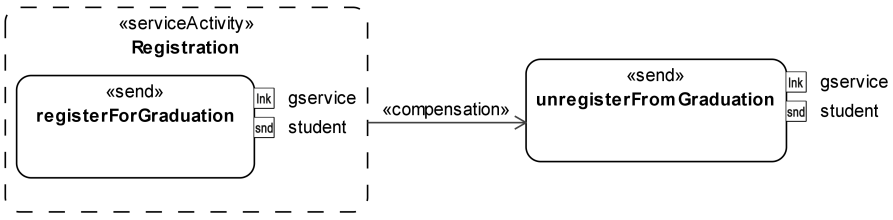
The compensated element must have this element as the compensation handler. A compensation edge may only be attached to a service activity.

Notation

The edge is annotated with the stereotype ≪*Compensation*≫.

Examples

This example shows the use of a ≪*ServiceActivity*≫-typed compensation handler. An ordinary ≪*ServiceActivity*≫ registers a student for a graduation celebration event. Later on, if the student fails to graduate, the compensation handler is invoked to unregister the student.



EventEdge

Description

An `EventEdge` is an edge connecting event handlers with a `ServiceActivityNode` during which the event may occur. It does not model a normal control flow — instead, it indicates an association between a (main) service element and an event handler.

Execution of an event handler is triggered externally by means of a call, or a timed event. An event handler may be executed zero, one, or multiple times in parallel to the service element it is attached to.

Note that only one instance of a specified event handler is active at the same time.

Generalisations

- `ActivityEdge`

Associations

- `eventBase` : `ServiceActivityNode`[1..1]
  The service activity to which an event handler is attached.
  {subsets `source`}
- `eHandler` : `ServiceActivityNode`[1..1]
  The service activity specifying an event handler.
  {subsets `target`}

Constraints

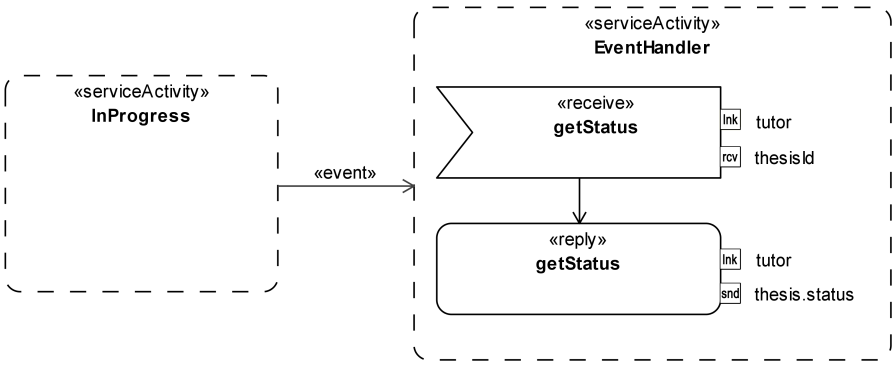
The event base element must have this element as an event handler. An event edge may only be attached to a service activity.

Notation

The edge is annotated with the stereotype ≪*Event*≫.

Examples

This example shows the use of a ≪*ServiceActivity*≫-typed event handler. The event handler is installed in parallel to the `InProgress` activity, and allows to retrieve the status of the service with a call (`getStatus`). This happens in parallel to the `InProgress` activity.



CompensateAction

Description

The `CompensateAction` invokes the compensation handler for a particular `ServiceActivityNode`, whose name is given in the body of the action and which must be nested inside the service element the handler in which the `CompensateAction` is specified in is attached to.

A `CompensateAction` may only be invoked from an exception or compensation handler. After the compensation handler of the given `ServiceActivityNode` has been executed, the instance is removed (*uninstalled*) from the referenced node, and the execution resumes normally after the `CompensateAction`.

Generalisations

- `OpaqueAction`

Associations

- `compensationTarget` : `ServiceActivityNode`[1..1]
  The `ServiceActivityNode` to be compensated.

Constraints

- The `CompensateAction` may only be used within a compensation or exception handler.

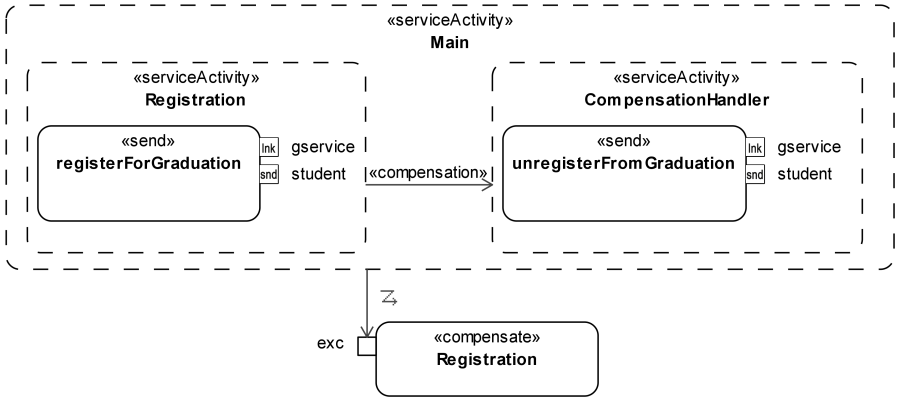
- The `compensationTarget` must be a `ServiceActivityNode` which has a compensation handler, and that `ServiceActivityNode` must be nested within the `ServiceActivityNode` in which the compensation action is invoked.

Notation

Annotation with stereotype ≪*Compensate*≫. The target name is given inside the body of the action.

Examples

This example shows the use of the compensate action. In this example, the compensation handler of `Registration` is invoked by means of a ≪*Compensate*≫ action.



CompensateAllAction

Description

The `CompensateAllAction` invokes all installed compensation handlers which are nested in the `ServiceActivityNode` to which the handler the `CompensateAllAction` is specified in is attached to.

A `CompensateAllAction` may only be invoked from an exception or compensation handler. It starts compensation of all inner `ServiceActivityNode`s of the `ServiceActivityNode` the exception- or compensation handler the action is defined in is attached to.

The inner `ServiceActivityNode`s with compensation handlers are compensated in reverse order of their completion, i.e. the last completed `ServiceActivityNode` first. However, this applies only if the `ServiceActivityNode`s are on the same level; inside the compensation handlers which are started in reverse order, the inner compensated `ServiceActivityNode`s compensation handlers might not necessarily run in (global) reverse order (they do in local reverse order).

After the compensation handlers have been executed, the instances are removed (*uninstalled*) from their respective `ServiceActivityNode`s, and the execution resumes normally after the `CompensateAllAction`.

Generalisations

- `OpaqueAction`

Associations

None.

Constraints

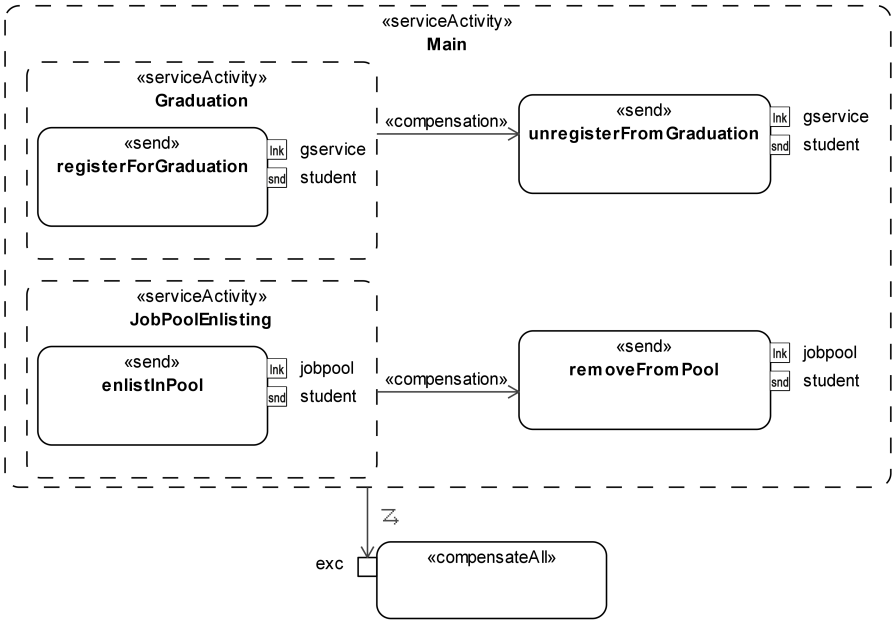
The `CompensateAllAction` may only be used within a compensation or exception handler.

Notation

Annotation with stereotype ≪*CompensateAll*≫.

Examples

In this example, two service activities are present. Each has an attached compensation handler. The first is installed after the `Graduation` activity completes; the second after `JobPoolEnlisting`. Both can potentially be invoked with the ≪*CompensateAll*≫ call if an exception is caught in the `Main` scope.



DataAction

Description

A `DataAction` is an action for data manipulation, for example, declaring variables and manipulating them (assignments, calculations, etc.). The `DataAction` allows the specification of arbitrarily many statements, written in the domain-specific UML4SOA expression language (see Sect. 2.4).

Generalisations

- `OpaqueAction`

Associations

None.

Constraints

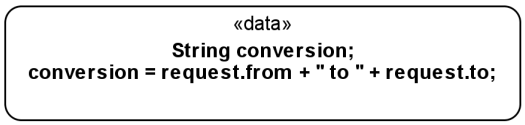
No additional constraints.

Notation

A `DataAction` is stereotyped with ≪*Data*≫. The statements to be executed are given inside the body.

Examples

This example shows a data action. In the action, a string-typed variable is declared (`conversion`). Afterwards, conversion is assigned by using two fields of the `request` variable, a string (" `to` "), and the string concatenation operator "+".



2.2.2 Pins

This section lists the pin classes of UML4SOA, which are used in service interactions for denoting partners as well as received and sent calls. The pin meta-classes are shown at the bottom of figure 2.5.

LinkPin

Description

A `LinkPin` is used to indicate the partner service for the service interaction. As a partner service is indicated through the ports of the participant to which the main `ServiceActivityNode` is attached to, the `LinkPin` is bound to a port. At runtime, an instance of the port is dynamically provided at `LinkPin`s.

Note that for `LinkPin`s referencing `Request` ports, a partner must be bound before execution by external means. For `Service` ports, incoming calls trigger creation of a new port instance which is given in the `LinkPin`.

Generalisations

- `InputPin`

Associations

- `port` : `Service` *xor* `Request`[1..1]

Constraints

- The `port` must be attached to the class which the root `ServiceActivityNode` of this behavioural specification belongs to.

Notation

The pin is stereotyped with ≪*Lnk*≫, or with the corresponding icon ("lnk"). The port name is specified along with the pin.

Examples

This example shows the use of a `LinkPin`. In all partner-related actions, for example in this ≪*Send*≫, the port at which the operation is requested or received must be specified.

In the example, the port is a ≪*Service*≫ port of the corresponding participant which carries the name `eoffice`.

InteractionPin

Description

An `InteractionPin` serves as the common abstract base class of `SendPin` and `ReceivePin`, restricting their type to either `MessageType` or `PrimitiveType`.

Generalisations

- `Pin`

Associations

- (inherited association from supertype) : `MessageType` *xor* `PrimitiveType`[1..1]
  {subsets `type`}

Constraints

The type must be a subtype of either `MessageType` or `PrimitiveType`.

Notation

None.

SendPin

Description

A `SendPin` is used in send actions to denote the data to be sent to an external service. A `SendPin` specifies data to be transmitted. Arbitrary right-hand side expressions specified in the UML4SOA expression language may be used.

Generalisations

- `InputPin`
- `InteractionPin`

Associations

No additional associations.

Constraints

The type must be a subtype of either `MessageType` or `PrimitiveType`. Also, the `SendPin` must have the correct type for the operation and partner invoked.

Notation

The `SendPin` must be stereotyped with $\ll Snd \gg$, or with the corresponding icon ("snd"). Furthermore, it needs to be annotated with the information about data to be sent. In UML, pins are ordered, which cannot directly be shown in the diagram. As a convention, UML4SOA send pins should be denoted on the right-hand side of an action from top to bottom.

Examples

This example shows the use of two `SendPin`s; this means that the operation used (`reportThesisStart`) requires two parameters. The first send pin specifies the variable `student`; the second the variable `thesis`.

ReceivePin

Description

A `ReceivePin` is used in receive actions to denote the place where the data received from an external service is stored (i.e., a variable, or a part of a variable).

Generalisations

- `OutputPin`
- `InteractionPin`

Associations

No additional associations.

Constraints

The type must be a subtype of either `MessageType` or `PrimitiveType`. Also, the `ReceivePin` must have the correct type for the operation and partner invoked.

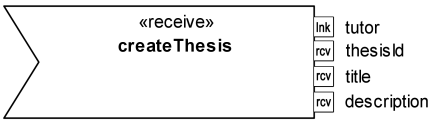
Notation

The `ReceivePin` must be stereotyped with $\ll Rcv \gg$, or with the corresponding icon ("rcv"). Furthermore, it needs to be annotated with the information about where to store the received data.

In UML, pins are ordered, which cannot directly be shown in the diagram. As a convention, UML4SOA send pins should be denoted on the right-hand side of an action from top to bottom.

Examples

This example shows the use of `ReceivePin`s. There are three receive pins; each for one of the parameters of the `createThesis` call. Each pin contains the target where the data will be stored; in this case, these are all variable names.



2.2.3 Communication Actions

Having introduced structuring elements and pins for data handling, we can now discuss the specialised actions for communication in UML4SOA. These actions are displayed on the top of figure 2.5.

ServiceInteractionAction

Description

`ServiceInteractionAction` is the common base class of all service interaction actions which have an associated `LinkPin`. The interaction is linked to a partner (i.e. a certain `port`) of the behaviour via the link pin (see section 2.2.2 for more information on `LinkPin`s). The operation is specified in the actions themselves.

Generalisations

None.

Associations

- `partner` : `LinkPin`[1..1]
  Specifies the partner of this `ServiceInteractionAction`. In case of a `ServiceSendAction`, this association subsets `target`.

Constraints

No additional constraints.

Notation

No notation.

ServiceSendAction

Description

A `ServiceSendAction` is an action that invokes an operation of a target service without expecting a return value. The argument values are data to be transmitted as parameters of the operation call. `CallOperationAction` contains the operation directly.

`ServiceSendAction` inherits argument from `InvocationAction`. We restrict this to `SendPin`s which contain the data to be sent.

Generalisations

- `CallOperationAction`
- `ServiceInteractionAction`

Associations

- (inherited association from supertype) : `SendPin`[0..*]
  {subsets `argument`}

Constraints

- `ServiceSendAction` constrains `argument` (inherited from `InvocationAction`) to pins of type `SendPin`.
- `target` is constrained to instances of `LinkPin`.

Notation

A `ServiceSendAction` is stereotyped with ≪*Send*≫. The operation name is given inside the action body.

Examples

This example shows a send. An operation call is sent to the partner attached to the port `eoffice` (specified in the link pin). The data to be sent is stored in two variables: `student` and `thesis` (specified in the send pins). There is no return value.

ServiceReceiveAction

Description

A `ServiceReceiveAction` is an accept call action representing the receipt of an operation call from an external partner. No answer is given to the external partner.

A `ServiceReceiveAction` blocks until the specified operation call is received. It requires a trigger (with a `CallEvent` event), which contains the operation. According to the operation, appropriate `ReceivePin`s must be given which contain the variables in which the incoming data is stored.

Note that there is a caveat involved with attaching, through the superclass `ServiceInteractionAction`, a `LinkPin` to a `ServiceReceiveAction`. The former is an `InputPin`, while the second is an `AcceptCallAction`. Unfortunately, the UML superstructure defines a constraint on `AcceptEventAction` (the direct superclass of `AcceptCallAction`, prohibiting the use of `InputPin`s on this class. This will be further discussed in section 2.5.

Generalisations

- `ServiceInteractionAction`
- `AcceptCallAction`

Associations

- (inherited association from supertype) : `ReceivePin`[0..*]
  {subsets `result`}

Constraints

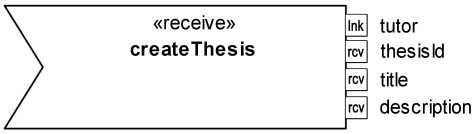
The result pins must be `ReceivePins`. This ensures that the data received has value or message types. The trigger must be a `CallEvent`.

Notation

A `ServiceReceiveAction` is stereotyped with ≪*Receive*≫. The operation name (from `trigger` → `CallEvent`) is given inside the action body.

Examples

This example shows a receive. A call is received from a partner (called `tutor`, specified in the link pin). The data is stored in three variables (`thesisId`, `title`, and `description` (specified in the receive pins). The operation invoked is called `createThesis`.



ServiceReplyAction

Description

`ServiceReplyAction` is an action that accepts a return value and a value containing return information produced by a previous `ServiceReceiveAction`. The reply action returns the values to the request port of the previous call, completing execution of the call.

`ServiceReplyAction` is a specialised version of `ReplyAction` for the service-oriented context. The inherited attribute `replyValue` is subset to point to instances of `SendPin`, instead of a generic input pin, thereby ensuring the data can be interpreted as value data. Thus, a `ServiceReplyAction` sends back data to a request port for which previous data was received.

Generalisations

- `ReplyAction`
- `ServiceInteractionAction`

Associations

- (inherited association from supertype) : `SendPin`[0..*]
  {subsets `replyValue`}

Constraints

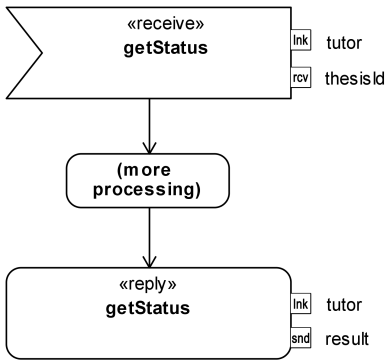
The `replyValue` pins must be of type `SendPin`.

Notation

A `ServiceReplyAction` is stereotyped with ≪*Reply*≫. The operation name is given inside the action body (corresponding to the operation inside the attached trigger).

Examples

This example shows a reply. A reply is always an answer to a previous receive, and carries the same partner and operation name as the receive. In this example, a `getStatus` call is received from partner `tutor`, and the single parameter is stored in the variable `thesisId`. Now, some processing takes place. Afterwards, the data in the variable `result` is sent as a reply to the `tutor` partner.



ServiceSend&ReceiveAction

Description

A `ServiceSend&ReceiveAction` action is a complete operation call execution with a partner. Some data (stored in the `SendPin`s) is sent, then the action waits for data to be sent back, which is stored in the `ReceivePin`s.

Generalisations

- `ServiceSendAction`
- `ServiceReceiveAction`

Associations

None.

Constraints

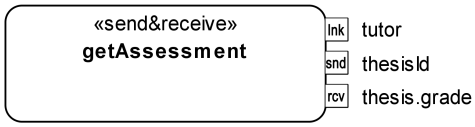
No additional constraints.

Notation

A `ServiceSend&ReceiveAction` is stereotyped with ≪*Send&Receive*≫. The operation name is given inside the action body.

Examples

This example shows a ≪*Send&Receive*≫. An operation is invoked on the partner `tutor` (specified in the link pin). The data itself is stored in the variable `thesisId` (specified in the send pin) and must be initialised before the action. The return value from the service is stored in the element `grade` of the variable `thesis` (specified in the receive pin).



2.2.4 Protocols

This section lists specialised transitions for denoting send, receive, and reply operations of a participant a UML4SOA protocol state machines belongs to.

ReceiveTransition

Description

A specialised transition indicating that an operation call is received by the participant to which the protocol state machine is attached to.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

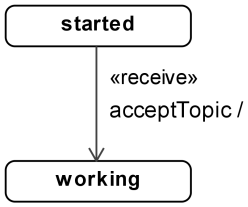
The trigger of this transition must be a `ReceiveOperationEvent`. Furthermore, the event must reference an operation *implemented* in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*Receive*≫.

Examples

This example shows a ≪*Receive*≫ in a protocol state machine. The example contains two states, `started` and `working`. In the `started` state, the operation call `acceptTopic` is expected, which leads the PrSM to the `working` state.



SendTransition

Description

A specialised transition indicating that an operation is invoked without returning information by the participant to which the protocol state machine is attached to. The operation invoked must be specified in a required interface of the classifier the protocol state machine is attached to.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

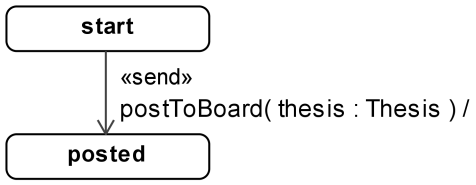
The trigger of this transition must be a `SendOperationEvent`. Furthermore, the event must reference an operation implemented in an interface *used* in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*Send*≫.

Examples

This is an example for using a send transition. Two states are used: `start` and `posted`. In the `start` state, the participant may choose to send out the `postToBoard` call; in this case, the PrSM is advanced to the `posted` state.



ReplyTransition

Description

A specialised transition indicating that a previous operation call is being replied to by the participant to which the protocol state machine is attached to.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

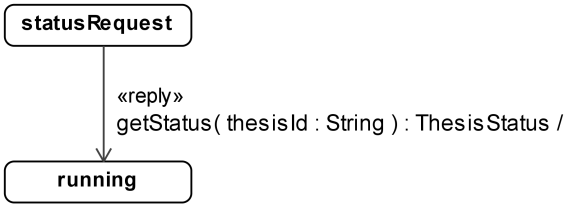
The trigger of this transition must be a `SendOperationEvent`. The event must reference an operation implemented in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*Reply*≫.

Examples

This is an example for using a reply transition. At the beginning, the PrSM is in the `statusRequest` state. Here, the participant may choose to reply to the `getStatus` call. The PrSM is advanced to the `running` state.



ReceiveReplyTransition

Description

A specialised transition indicating that an operation call is received by the participant to which the protocol state machine is attached to, and that this receive is in response to a previous send originating from this participant.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

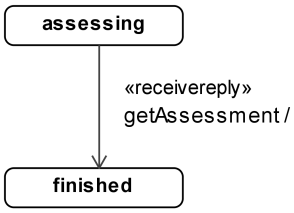
The trigger of this transition must be a `ReceiveOperationEvent`. The event must reference an operation implemented in the port type the PrSM is attached to.

Notation

Annotation with stereotype ≪*ReceiveReply*≫.

Examples

This example shows a ≪*ReceiveReply*≫ in a protocol state machine. The example contains two states, `assessing` and `finished`. In the `assessing` state, a reply to the operation `getAssessment` is expected, which leads the PrSM to the `finished` state.



OptionalTransition

Description

`OptionalTransition` is a specialised transition indicating that the operation given as part of this transition (specified with ≪*Send*≫, ≪*Receive*≫, ≪*Receive-reply*≫ or ≪*Reply*≫) is optional, i.e. may or may not be supported by an implementation of this protocol.

Generalisations

- `ProtocolTransition`

Associations

None.

Constraints

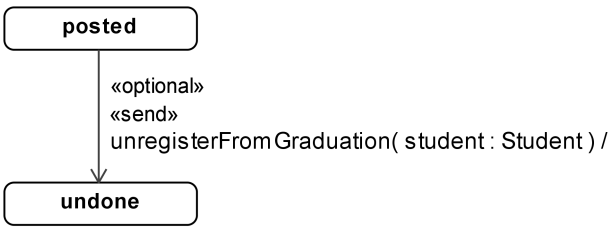
The transition must also be annotated with ≪*Send*≫, ≪*Receive*≫, ≪*Receive-reply*≫, or ≪*Reply*≫.

Notation

Annotation with stereotype ≪*Optional*≫.

Examples

This is an example for using an optional send transition. If the PrSM is in the state `posted`, the participant may choose to send the `unregisterFromGraduation` call, leading to the state `undone`.



2.3 From Meta-Model to Profile

As indicated at the beginning of chapter 2, the aim is defining a lightweight extension of the UML in the form of a profile. In the previous section, we have defined a meta-model for UML4SOA; we now map the meta-classes and attributes of this meta-model to stereotypes and tag definitions.

As a UML profile, UML4SOA defines a profile package whose meta-model reference-element is the UML, and which additionally imports the stereotypes from the SoaML profile (see figure 2.6).

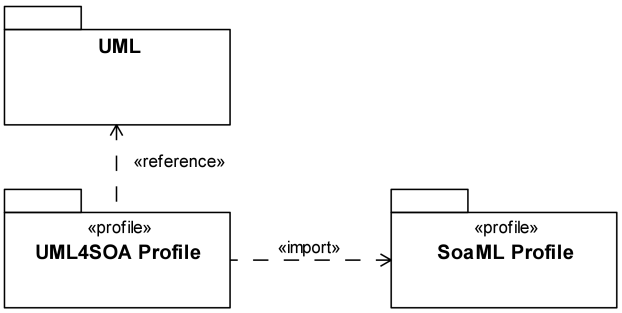


Figure 2.6: UML4SOA Profile Package

The following figures will each show several meta-classes and their mapping to stereotypes. The notation is as follows:

- UML meta-classes, which are extended by the stereotypes of the UML4-SOA profile, are shown in gray.
- Stereotypes of the UML4SOA profile are shown in yellow. Note that these may only *extend* UML meta-classes, which is shown by the extension arrows.
- Finally, as a reference, the UML4SOA meta-classes are shown in white. A $\ll mapsTo \gg$ relation between a stereotype and an UML4SOA meta-class gives the intuition of which stereotype represents which meta-class; this notation is not defined in the UML.

We attach a specific semantic meaning to the $\ll mapsTo \gg$ relationship: If a stereotype *maps to* a UML4SOA meta-class, it is subject to the same constraints regarding inherited associations. For example, the meta-class `ServiceSend-Action` requires that all arguments must be `SendPin`s; this is transferred to the $\ll Send \gg$ stereotype.

Note that the stereotypes are *defined* with an uppercase letter; by contrast, the *application* of a stereotype uses a lowercase letter. This style is in line with section 18.3.8 (Stereotypes) of the UML superstructure (see [OMG10b]).

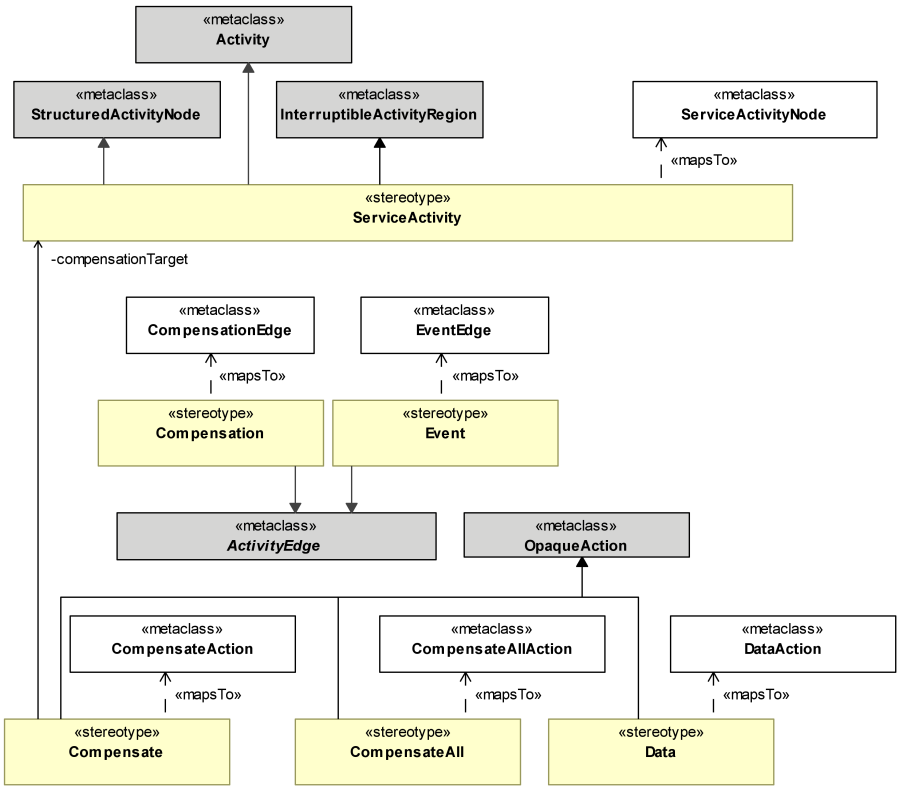


Figure 2.7: UML4SOA Stereotypes for Structuring Classes

Structuring Classes

We begin with the structuring classes of the UML4SOA meta-model. Figure 2.7 shows the mapping of these classes of the UML4SOA meta-model to the stereotypes of the UML4SOA profile.

The most important stereotype is $\ll ServiceActivity \gg$, which corresponds to the `ServiceActivityNode` meta-class. The $\ll Compensation \gg$ and $\ll Event \gg$ stereotypes are based on `ActivityEdge` and correspond to the `CompensationEdge` and `EventEdge` meta-classes of UML4SOA. Note that we do not define tags here; the relationship between base elements and compensation/event handlers is given through the standard meta-attributes of `ActivityEdge`; constraints apply as per definition in the UML4SOA meta-classes `ServiceActivityNode` and `ActivityEdge`.

In the lower section of the figure, three stereotypes for actions are defined, namely $\ll Compensate \gg$, $\ll CompensateAll \gg$ and $\ll Data \gg$. They correspond to the meta-classes `CompensateAction`, `CompensateAllAction` and `DataAc-`

`tion`, respectively. The first of these needs a tag definition: The ≪*Compensate*≫ stereotype must be tagged with the target service activity to be compensated.

Communication Classes

We continue with classes used for communication, i.e. the various sending and receiving actions as well as pins for specifying partners and data. Figure 2.8 shows the mapping of the corresponding meta-classes to stereotypes.

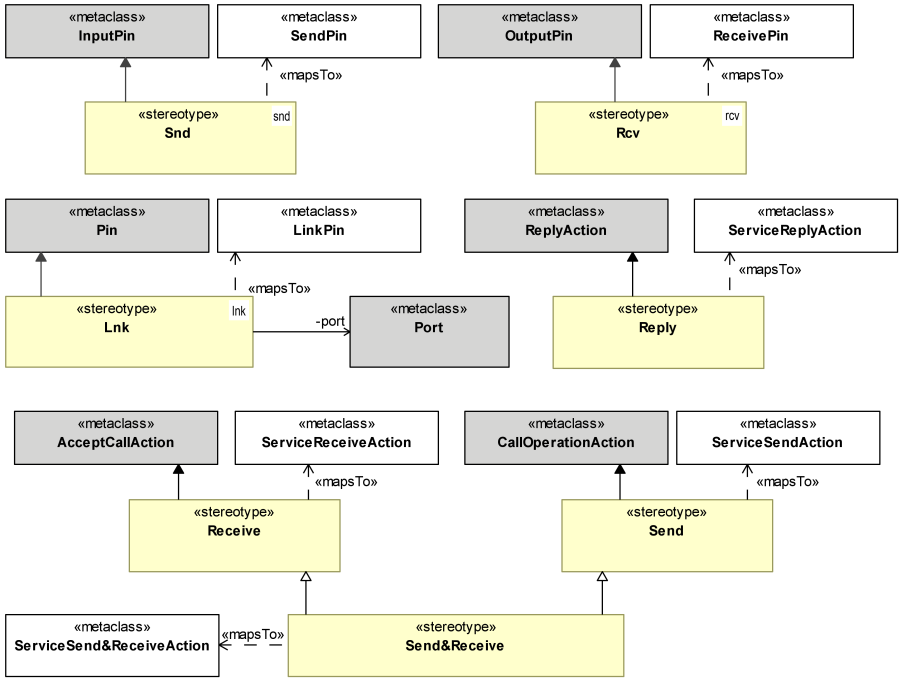


Figure 2.8: UML4SOA Stereotypes for Communication Classes

On the top, the stereotypes ≪*Snd*≫ and ≪*Rcv*≫ are defined; both with an optional icon for displaying a pin graphically. The first stereotype maps to the `SendPin` meta-class, thus inheriting the requirement that the type of the pin must be either a `MessageType` or an `PrimitiveType`; the second stereotype likewise maps to the `ReceivePin` meta-class.

In the second row, the stereotype ≪*Lnk*≫ is mapped to the meta-class `LinkPin`. Note that we need to define an additional tag here, specifying which port of the corresponding participant is referenced.

The ≪*Reply*≫ stereotype in the second row and the ≪*Receive*≫, ≪*Send*≫, and ≪*Send&Receive*≫ stereotypes in the third row are used for tagging communicating actions. They correspond to the `ServiceReplyAction`, `Service-`

`ReceiveAction`, `ServiceSendAction` and `ServiceSend&ReceiveAction` meta-classes, respectively. No tag definitions are required, however, once again, constraints apply.

Protocol Classes

Finally, we get to the last five stereotypes of the UML4SOA profile, which concern the specification of service protocols and are shown in figure 2.9.

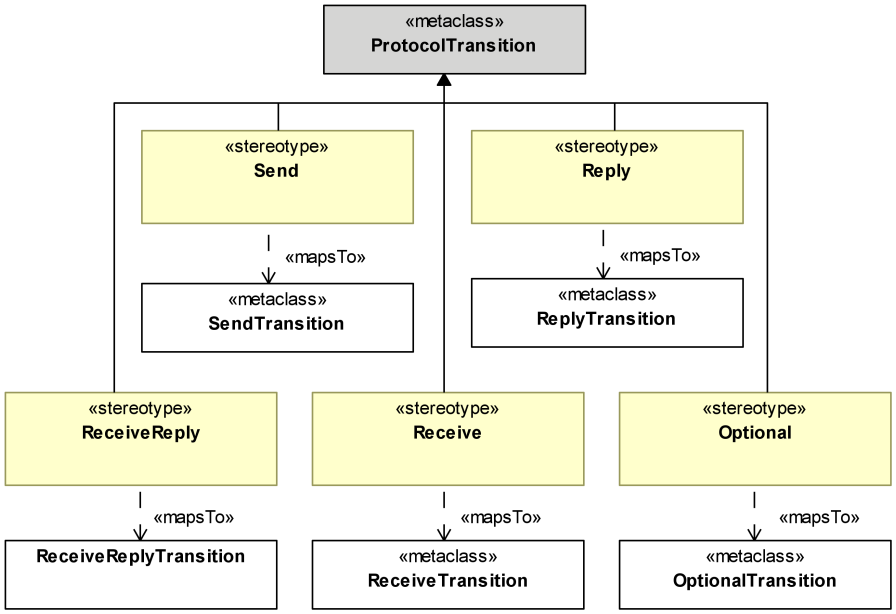


Figure 2.9: UML4SOA Stereotypes for Protocol Specification

We have seen three of the stereotypes listed here before; they are used for both activities and protocol state machines. As expected, the ≪*Send*≫ stereotype maps to the `SendTransition` meta-class, ≪*Receive*≫ maps to the `ReceiveTransition` meta-class, and ≪*Reply*≫ maps to the `ReplyTransition` meta-class. Furthermore, the ≪*ReceiveReply*≫ stereotype maps to the `ReceiveReplyTransition` meta-class and the ≪*Optional*≫ stereotype maps to the `OptionalTransition` meta-class.

No further tag definitions are required, though it is worth noting that the communicating stereotypes inherit a constraint on the trigger allowed on a stereotyped transition. By contrast, the ≪*Optional*≫ stereotype is only used for tagging the transition.

The UML4SOA profile is thus complete and can be used in arbitrary, profile-enabled UML modelling tools.

2.4 Data Handling

An important point in modelling service behaviour and service orchestrations is data handling. Data is received by services, manipulated, and then sent on or back to another service. We have devised a declarative, textual language for this purpose, which aims to closely match the level of detail of UML4SOA. A major goal of the UML4SOA data handling language was to be generic enough to be understandable on the modelling level, yet contain enough information to allow transformation to more lower-level languages for execution. The main requirements for a UML4SOA data handling language is support for data in messages sent in-between services, and variables for storing such data, which requires:

- Support for primitive and complex (composite) data types.
- Typing of and access to variables, including assignments and partial assignments.
- Basic operations for manipulation data.

The UML4SOA data handling language is strongly typed and based on UML primitive types as well as classes annotated with the $\ll$*MessageType*$\gg$ stereotype from the SoaML profile, which are in effect data types (i.e., classes without behaviour). Data is modified by imperative statements which may be used in three distinct areas within UML4SOA models:

- *Pins.* While receive pins may only hold variable references or (implicit) variable declarations, send pins may also be used to construct new data on-the-fly.
- *Guards.* A guard may contain a boolean-typed UML4SOA expression.
- *Data Handling Actions.* When inline data handling in pins or guards is not possible, data handling statements can also be added explicitly with a $\ll$*Data*$\gg$ action.

Usually, services and service orchestrations directly work on structured data, i.e. $\ll$*MessageType*$\gg$-typed classes which carry the business-relevant information. The UML4SOA data handling language provides built-in support for these data types, although some restrictions apply:

- A data type may only be assembled from primitive types or other structured data types.
- Inheritance is allowed, but again only amongst structured data types.
- Associations between structured data types is possible with the exception of bidirectional associations.

The data manipulation language supports both sets and lists (unordered and ordered associations). Furthermore, operations on basic data types is supported (mathematical operation on integers and reals; logical operations on booleans, and concatenation on strings).

2.4.1 Syntax Used

Whenever a concrete syntax is described in this document, we display it the same manner as in the Java Language Specification [GJSB05]. We use a context-free grammar, i.e. a number of productions with a nonterminal symbol on the left and both terminals and non-terminals on the right:

Listing 2: Context-Free Grammar Example

Element:
    execute(*AnotherElement***)**

We denote nonterminal symbols with *italics*, and terminal symbols with **bold** font. A definition of a nonterminal is given by the nonterminal suffixed with a colon (:), as shown for *Element* in the example. For *Element*, the right-hand side consists of the terminal symbol **execute** followed by the terminal symbol for an opening brace (**(**), the non-terminal *AnotherElement*, and another terminal symbol, the closing brace (**)**).

In general, the right-hand side of a non-terminal definition consists of one or more lines which form the possible alternatives. An example is the following definition:

Listing 3: Denoting Alternatives

Elements:
    Element
    Elements Element

This definition of *Elements* introduces two alternatives: Either the non-terminal *Element*, or *Elements* again, followed by a single *Element*. This definition is recursive, as *Elements* occurs both left and right of the colon. Note that if a line is too long to fit on the page, we let it continue indented below.

We introducing a special suffix (*opt*) for specifying an optional element. Consider the following example:

Listing 4: Optional Elements (1)

Element:
    execute(*AnotherElement***)** $OtherElements_{opt}$

In this definition of *Element*, *OtherElements* may optionally be specified after the main **execute** definition. This is a shortcut for

Listing 5: Optional Elements (2)

Element:
  execute(*AnotherElement***)**
  execute(*AnotherElement***)** *OtherElements*

Finally, we define three special non-terminals:

- *String*, which identifies a sequence of arbitrary characters,
- *Number*, which identifies a sequence of characters in the range of `[0-9]`,
- and *VarName*, which identifies a sequence of letters and numbers, where the first character must be a letter.

This concludes the introduction of the syntax notation.

2.4.2 Grammar

We start with the declaration of a statement and preliminaries:

Listing 6: UML4SOA Data Handling Syntax

DataHandling:
  Statement

Statement:
  Declaration
  Assignment

Declaration is used to declare the type of a variable. To denote an ordered or unordered list, the corresponding brackets ([] or {}) can be appended to the type:

Listing 7: UML4SOA Data Handling Syntax: Declarations

Declaration:
  *Type Identifier***;**

Type:
  VarName $([]_{opt} \mid \{\}_{opt})$

Identifier:
  VarName

An *Assignment* is an expression for assigning a value to a variable. It is split between a left-hand-side (left of the assignment operator) and a right-hand-side (right of the assignment operator):

Listing 8: UML4SOA Data Handling Syntax: Assignments

Assignment:
LeftHandSideExpression **:=** *RightHandSideExpression***;**

Left-hand sides are, in effect, references to variables or elements within variable types. A variable in UML4SOA has a declaring scope, which is the service activity it was first used or declared in. If the scope of a variable is not the current one, it may be given with the **::***ServiceActivityName* syntax. Furthermore, not only a variable can be referenced but also a part within the variable, which must be a publicly accessible field of a ≪*MessageType*≫.

Listing 9: UML4SOA Data Handling Syntax: Left-Hand Sides

LeftHandSideExpression:
(**::***VarName***.**)$_{opt}$ *VarAccess*

VarAccess:
VarName (**.***VarAccess*)$_{opt}$

Right-hand sides are more complex as they can be used not only in assignments, but also in conditional statements; furthermore, they contain the complete syntax for data manipulation and calculations. *RightHandSideExpression* is defined by starting with a conditional or, which has the least precedence, and continuing until we reach the basic literals and qualifiers.

Listing 10: UML4SOA Data Handling Syntax: Right-Hand Sides (1)

RightHandSideExpression:
ConditionalOrOperation

ConditionalOr:
ConditionalAnd (|| *ConditionalAnd*)$_{opt}$

ConditionalAnd:
Equality (&& *Equality*)$_{opt}$

Equality:
Relational ((== | **!=**) *Relational*)$_{opt}$

Listing 11: UML4SOA Data Handling Syntax: Right-Hand Sides (2)

Relational:
Additive ((> | >= | <= | <) *Additive*)$_{opt}$

Multiplicative:
PrefixedUnary ((∗ | / | %) *PrefixedUnary*)$_{opt}$

PrefixedUnary:
(− | !) *Unary*

Evaluating right-hand sides starts from the bottom to the top, i.e. the prefixed unary literals **-** and **!** have the highest precedence, while the conditional or || has the lowest.

Before we can define the literals, we have to take care of the unary elements referenced above. A unary element is either a literal, a left-hand-side expression, or a right-hand-side expression in parenthesis.

Listing 12: UML4SOA Data Handling Syntax: Unary Elements

Unary:
Literal | *LeftHandSideExpression* | *ParenthesisExpression*

ParenthesisExpression:
(*RightHandSideExpression*)

Finally, we can define the literals, which are simple numbers, string constants, boolean constants, or the special value **null**.

Listing 13: UML4SOA Data Handling Syntax: Literals

Literal:
StringLiteral | *NumberLiteral* | *BooleanLiteral* | **null**

StringLiteral:
"*String*"

NumberLiteral:
Number (. *Number*)$_{opt}$

BooleanLiteral:
true | **false**

As usual, this grammar allows a few constructs which are not legitimate from a semantic point of view. For example, comparing a string with a boolean using the *Equality* construct makes no sense in a strongly typed language. We

believe, however, that these cases are intuitively clear and thus do not require further lengthy discussion.

2.4.3 Using the data language

Three of the non-terminal elements of the UML4SOA data language can be used as *top-level* elements in expressions inside of UML4SOA models:

- The *DataHandling* statement is intended to be used in ≪*Data*≫ actions. A ≪*Data*≫ action may contain an arbitrary number of data handling statements.
- *LeftHandSideExpression*s and *Declaration*s may be used in ≪*Rcv*≫ pins. The first is used to specify an already existing variable in which the data received is to be placed. The second can be used as a shortcut for the modeller; it both declares the variable and specifies it for the received data.
- In ≪*Snd*≫-Pins, entire *RightHandSideExpression*s can be used. This allows creating data on-the-fly. In this case, it is convenient to think of a remote message invocation as a distributed assignment.

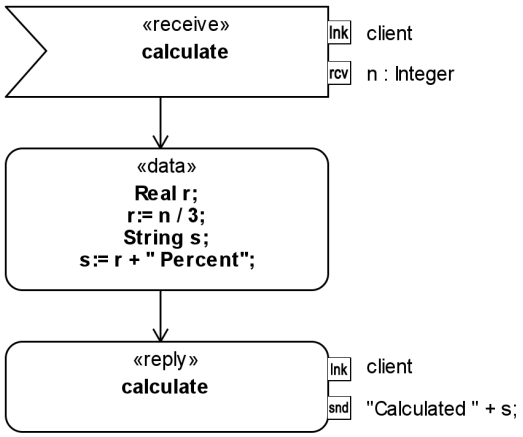


Figure 2.10: UML4SOA Data Manipulation: Simple Example

An example of using simple data types, assignment operations, and basic operation on numbers and strings is shown in figure 2.10. First, a variable is declared on-the-fly in a ≪*Rcv*≫ pin of the ≪*Receive*≫ action `calculate`; the variable is called `n` and is typed with the well-known UML type `Integer`. Second, a data handling action is used which executes four statements:

- The variable `r` is declared with the UML type `Real`,

- `r` is assigned the expression `n / 3`,
- the variable `s` is declared with the UML type `String`,
- `s` is assigned the expression `r + " Percent"`.

Finally, the reply action uses the on-the-fly right-hand expression `"Calculated " + s` to add an additional string before `r`, the result of which is then sent back to the invoker.

As indicated above, the UML4SOA data language also provides extensive support for dealing with structured data types, which are tagged with ≪*MessageType*≫ in SoaML. An example of three structured data types is shown in figure 2.11: a `Thesis` object may reference a `Student` and a `Tutor`.

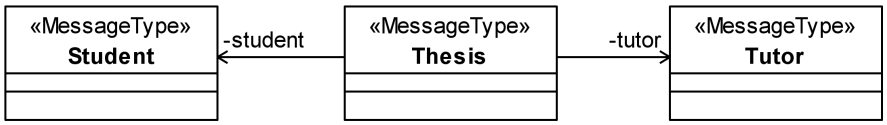


Figure 2.11: UML4SOA Data Manipulation: Structure Types

Working with the instances of the `Thesis` class in UML4SOA can take advantage of these associations. Figure 2.12 shows an example where both the tutor and the student association ends are set in a single data action.

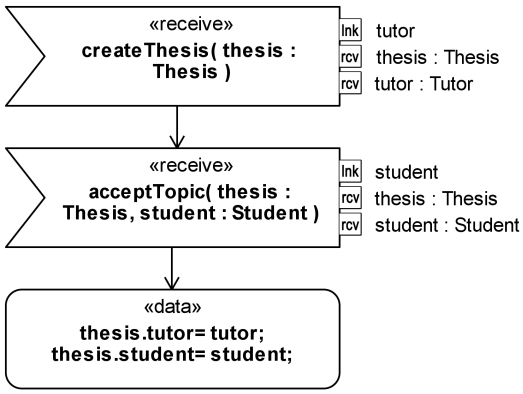


Figure 2.12: UML4SOA Data Manipulation: Structure Example

To sum up, adding a data handling language to UML4SOA has the benefits of being able to specify data operations on the UML level of abstraction. With its low complexity and easy-to-use syntax, the language is a good match for the UML4SOA graphical language and enables transformation to code.

2.5 Changes to the UML

Specifying service behaviour in UML introduces some key new requirements for a modelling language which was originally designed with object-oriented systems in mind. The SoaML profile [OMG09] has already shown how to mould the UML to include SOA concepts, which has led to the proposal of adding the concept of *conjugation* to the UML.

Within UML4SOA, we have identified the need for two additional changes to the UML to enable developers to model SOA behaviour in a natural and straightforward way. In the following, we revisit these changes already introduced in the previous sections.

Adding an InputPin to AcceptCallAction

UML4SOA uses the UML meta-class `AcceptCallAction` with the stereotype $\ll Receive \gg$ for denoting a place where service behaviour waits for an incoming call through a port of the corresponding participant. As has been discussed above, the concrete port must be specified as part of the action to indicate the partners from which a call is to be expected. UML4SOA has introduced the $\ll Lnk \gg$ stereotype for this purpose, which is attached to the UML meta-class `InputPin`, as the partner information is an input to the receiving action.

Unfortunately, the UML superstructure [OMG10b] contains a restriction on `AcceptEventAction`, which is a superclass of `AcceptCallAction`, which prevents the use of input pins on instances of this class.

UML4SOA requires that this restriction is relaxed to allow pins which carry additional information for the receiving action, which, in our case, is a $\ll Lnk \gg$-stereotyped `InputPin` specifying the port the operation attached to the trigger of the action is received on.

Allowing Call Observations in PrSMs

Transitions in UML Protocol State Machines [OMG10b] are based on the meta-class `ProtocolTransition`. This class contains two important restrictions. First, the `effect` association must be empty, i.e. a protocol transition may not have associated actions. Second, as a subclass of `Transition`, a protocol transition may include a trigger. There are two restrictions on this trigger:

- First, the specification of `ProtocolTransition` includes the requirement that if a call trigger is used, the operation referenced *should apply* to the context classifier of the state machine of the protocol transition.
- Second, the specification states that non-call events may be used on protocol transitions, but again refers to *incoming* events whose target is the context classifier.

We believe that in the context of service protocol specification, this restriction should be lifted to be able to observe events which *originate* from the

context classifier instead of using it as a target. In fact, a corresponding UML meta-class for this concept exists: `SendOperationEvent` specifies that a call invocation request is sent to an object (at which it may result in the occurrence of a call event).

As sending out calls to partner services requires being able to note this fact in a protocol, we believe that it should be possible to also reference operations which are *used* by the context classifier of a protocol state machine. UML4SOA thus extends the ability of PrSMs to include triggers with a `SendOperationEvent` event. It is important to note that this event is not an *effect* of a transition; rather; it is an *observed operation call* of the participant the classifier of the PrSM is attached to.

This ability is restricted in UML4SOA to transitions stereotyped with $\ll Send \gg$ or $\ll Reply \gg$.

2.6 UML4SOA/Open and UML4SOA/Strict

In this section, we introduce two *dialects* of UML4SOA: One serves modellers interested in having maximum freedom in applying UML4SOA in combination with the UML, while the other serves modellers interested in code generation and formal analysis.

On the one hand, UML as a graphical language is great for communication between people. For this use case, the focus lies on readable diagrams, which tend to focus on the overall architecture of a system and ignoring low-level details. Some of the diagram types of UML, for example use case diagrams, are explicitly geared towards this usage, but with a sufficient level of abstraction this method is applicable to all modelling elements. UML4SOA can be used for this purpose: UML4SOA/Open defines a dialect which contains no restrictions on how UML elements and UML4SOA elements may be used and combined in models, only requiring the constraints in chapter 2 to hold.

On the other hand, model-driven software approaches build on generating code from models. To enable such code generation, the semantics of the models must be specified more precisely, which in general requires more detail and stricter rules for placing elements in the model. Again, some diagram types in UML are better suited for this purpose, for example state machines and activity diagrams, but once more there are also methods for generating i.e. tests from use case diagrams. UML4SOA can be used for this purpose as well: UML4SOA/Strict defines a set of rules for modellers to follow which enables formal analysis and code generation.

UML4SOA/Open

The purpose of UML4SOA/Open is to give maximum freedom to software modellers. For this reason, no additional constraints apply — UML4SOA elements may be freely mixed with UML activity and state machine model elements, fully exploiting the means of specifying models with UML.

UML4SOA/Strict

By contrast, UML4SOA/Strict defines a set of rules which must be followed to create compliant UML4SOA activity and state machine models usable for generation of code and the specification of a formal semantics.

- A UML4SOA/Strict model must be based on a SoaML ≪*Participant*≫ with ≪*Service*≫ or ≪*Request*≫ ports. Each port must have a port type stereotyped with ≪*ServiceInterface*≫ which may include operations (either directly or inherited) and declare usage relationships to other types. Each operation may have multiple `in` and `return` parameters; `out` and `inout` parameters are not allowed.

- All UML4SOA activities must be stereotyped with ≪*ServiceActivity*≫ and attached to a ≪*Participant*≫. The only actions allowed in an UML-4SOA activity diagram are the actions stereotyped with UML4SOA stereotypes with the one exception of `RaiseExceptionAction`. All communicating actions must reference an operation (either directly or through a trigger) from one of the port types of the corresponding participant.

- For controlling the workflow, decision and merge nodes as well as fork and join nodes may be used. However, the resulting model must be well-nested, i.e. all paths from a decision node not ending in a flow-final or activity-final node must end in a merge node. The same goes for fork and join nodes. Loops can (as usual) be modelled using fork and join nodes; however, the looping (back) link may not carry any additional actions. Each service activity must have an identifiable start node, i.e. either an action without incoming links or a dedicated start pseudo node.

- The only grouping constructs allowed are UML4SOA service activities. Handlers (again, service activities) may be attached as usual to service activities, but interrupting edges are restricted to non-handler service activities. Handlers may only be attached to non-handler service activities.

- Each event handler must start with a ≪*Receive*≫ action and end with a ≪*Reply*≫ action or a `RaiseExceptionAction` to ensure the event handler termination is either communicated to the partner, or exception handling is triggered.

- All data handling statements in ≪*Snd*≫ and ≪*Rcv*≫ pins, guards, and data actions must follow the syntax and semantics of the UML4SOA data manipulation language. The only UML primitive types allowed are `Integer`, `Boolean`, and `String`; additionally, floating point values may be declared using a (custom) data type `Double`, and dates with a (custom) data type `Date`.

- The root behaviour of a participant must start with a ≪*Receive*≫ action to ensure that it can be started from the outside.

For protocol state machines, the following rules apply:

- Transitions not annotated with a UML4SOA communication stereotype are allowed, but are assumed to be internal to the protocol and the corresponding implementation. They do not follow the usual completion semantics.
- States may not be nested, i.e. the state and transition structure must be flat.
- An explicit start pseudo node is required to be present. The usual restrictions apply for the start transition.

We believe that besides being a requirement for code generation and formal analysis, these requirements also lead to more readable diagrams and easier implementation, and thus recommend following the constraints given here regardless of the use of code generation.

2.7 Lifecycle Management

A SoaML participant with its accompanying service and request ports and UML4SOA activities models a single instance of an orchestration execution. The owned behaviours (UML4SOA service activities) each model one execution of the orchestration; the port protocols (UML4SOA PrSMs) each model an observation of the interaction with a partner during the lifetime of the orchestration (provided or requested).

However, in client/server and SOA computing, an orchestration is normally executed multiple times, often in parallel. In UML4SOA, we do not require the developer to model this bootstrapping mechanisms as it is normally not of interest to the modeller. Handling of multiple workflows, including instance matching, is done implicitly in UML4SOA designs as follows (cf. figure 2.13).

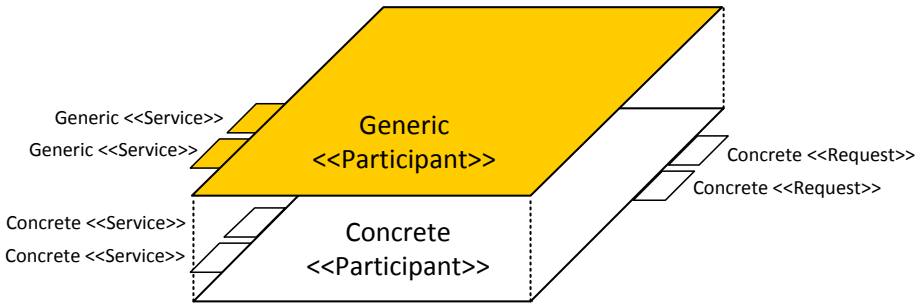


Figure 2.13: Generic and Concrete Participants

A concrete UML4SOA participant (i.e. the version modelled by a user) is not instantiated directly on system startup. Instead, one can imagine that a

generic version of the participant is implicitly created and instantiated. The generic version has as many ≪*Service*≫ ports as the concrete one — these are *generic* ≪*Service*≫ ports, which each provide and accept only one operation again and again, which corresponds to the first accepted operation in the concrete ≪*Service*≫ port PrSM. On startup of the generic participant, the generic ports wait for incoming calls. Once a message is received, a new instance of the concrete participant along with its ports is instantiated by the generic participant and the startup message is passed to the corresponding *concrete* port.

All non-startup communication is done directly with the *concrete* instances of the participant and its ports. To ensure instance matching, the port instances are provided to the workflow via the ≪*Lnk*≫ pins.

Finally, instance matching, i.e. routing messages to the appropriate port and thus instances of UML4SOA activities, requires some information (like an ID) as part of the message with which the system can route a message to the appropriate port (and thus workflow) instance. This information is assumed to be added transparently to the incoming and outgoing messages of the workflow based on the unique IDs associated with each port instance.

Each service action in UML4SOA service activities has a ≪*Lnk*≫ pin which carries the information about the instance of the port which is in use for this particular orchestration instance. This information is used to match calls to and from the correct workflow instance.

Chapter 3

Modelling Examples

In this chapter, we will detail how the thesis management scenario from the eUniversity case study from the SENSORIA project has been modelled with UML-4SOA, and give some pointers to other examples.

3.1 Modelling the eUniversity Case Study

In chapter 1, we have introduced the static (SoaML) model of the eUniversity case study (Figure 1.2 on page 11). Now, after having discussed the UML4-SOA profile and its extensions for activities and protocol state machines, we can add the behaviour of the `ThesisManagement` participant and the protocols of its ports. In the following, we use the UML4SOA/Strict dialect for both the activity and the PrSM models.

The UML4SOA activity describing the behaviour of the `ThesisManagement` participant is shown in figure 3.1. From top to bottom, the behaviour is as follows:

- The orchestration begins with a ≪*Receive*≫ action, requiring a client to send the `createThesis` call via the service port `tutor`. A thesis object is expected and placed in the newly declared `thesis` variable.

- Afterwards, the process starts with its `Main` activity and another receive operation: `acceptTopic` allows a student to start working on the thesis. Attached to `Main` is an exception handler which catches the `ThesisFailedException`. It contains one action, namely a ≪*Compensate*≫ call with the target activity `Registration`.

- Having completed `acceptTopic`, the `Registration` activity is entered. Here, we first inform the examination office about the newly started thesis, and secondly register the student for a seat in the graduation gala. This is a classic example of how certain parts of a workflow complete successfully but may need to be undone later on; therefore, the `Registration` scope

has an attached compensation handler `CompensateRegistration` which undoes the reservation of a seat in the gala (this happens if the thesis is not accepted).

- Finally, the tutor is notified that a student has accepted the thesis and is now working on it.

- Now that the thesis is in progress, we enter the `InProgress` activity, which starts with a loop. In this loop, the student is allowed to send updates by using the `updateStatus` call via the `student` port until the thesis is complete, in which case he sends a `finished` call.

- During this time, which is potentially quite long, the tutor might want to request updates. Therefore, the `InProgress` activity has an event handler, `StatusInformation`, which contains the receive action `getStatus` to be used by the tutor for retrieving the current status of the thesis.

- Once the `finished` call has been received, the assessment is requested from the tutor using `getAssessment`, which is reported to the student. In case the thesis was finished successfully, this information is reported to the examination office and the process ends. If not, a failure is reported and an exception thrown, which leads, via the compensate activity in the `ExceptionHandler`, to unregistering the student from the gala.

As the `ThesisManagement` participant uses four ports — two service port and two request ports — we define four UML4SOA protocol state machines for specifying the externally visible protocol of the participant.

Figure 3.2 shows the protocols of the `ThesisManagement` participant. From top left to bottom right, these are the `student` protocol, the `eoffice` protocol, the `bboard` protocol, and the `tutor` protocol.

Student Protocol

The protocol provided to the student allows receipt of the `acceptTopic` call. Once received, the protocol is in `working` state, allowing the receipt of either the `updateStatus` call, which leads back to `working`, and `finished`, which requests the process to finished (being replied to with $\ll$*Reply*$\gg$ `finished`).

EOffice Protocol

The protocol of the examination office begins with the ability to send a `reportThesisStarted` call, indicating that a student has started a thesis. In the subsequent state `working`, the thesis is either reported as being successfully completed with `reportThesisSuccess` leading to the `success` state, or having failed, in which case we receive a `reportThesisFailure` which leads to the `failed` state.

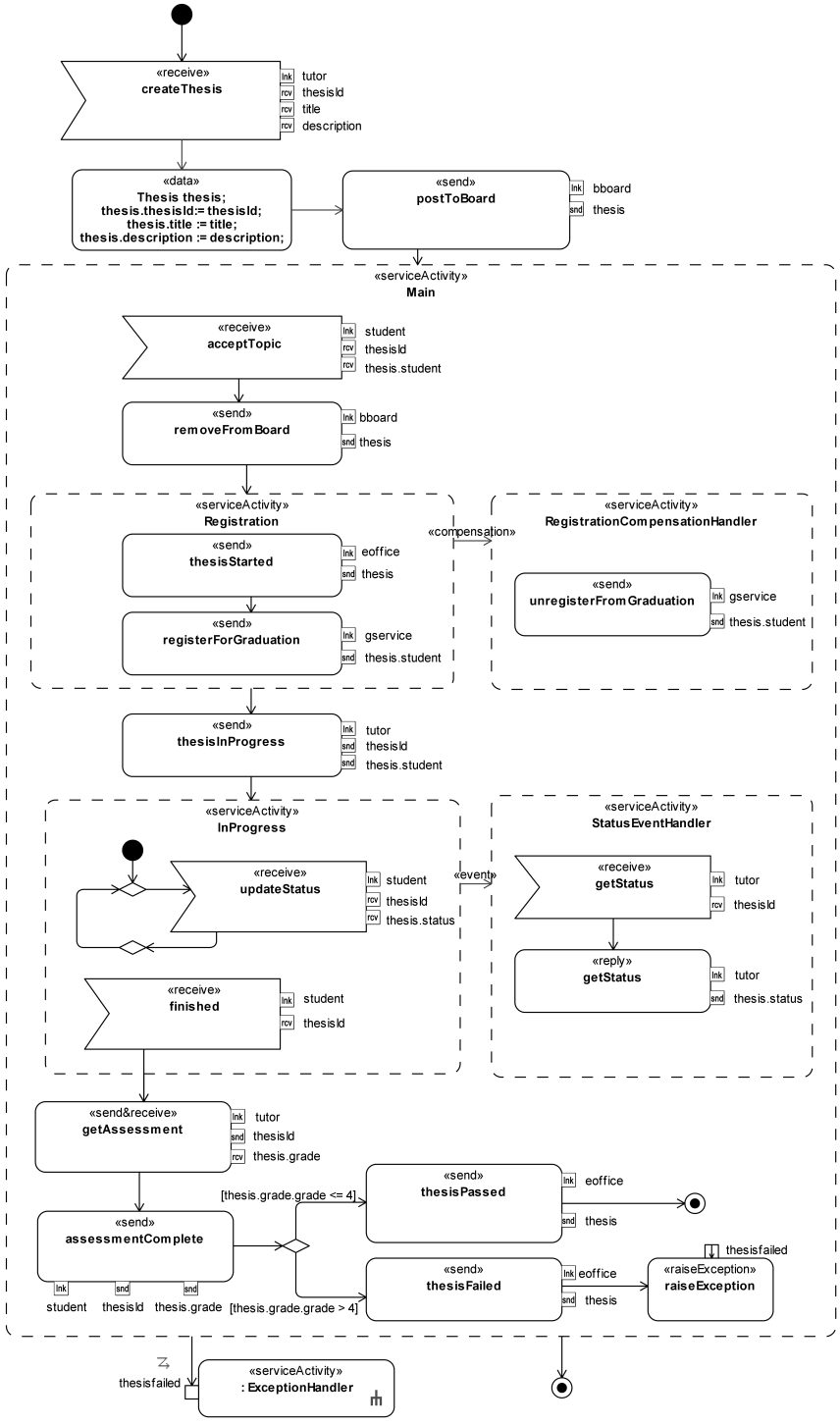


Figure 3.1: eUniversity Case Study: Thesis Manager Activity

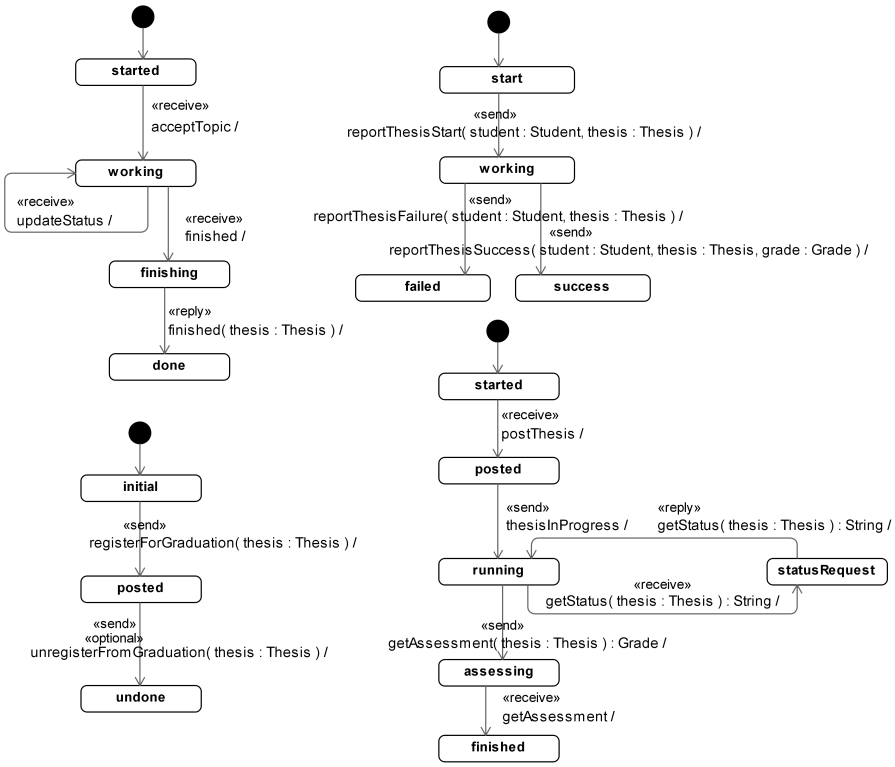


Figure 3.2: eUniversity Case Study: Protocol Specification

Graduation Protocol

The graduation service protocol is rather simple — the `TutorManagement` participant expects to be able to send a `registerForGraduation` call to the graduation service, and it might — using the ≪*Optional*≫ stereotype — also need to unregister the student with `unregisterFromGraduation`.

Tutor Protocol

Finally, the tutor protocol is another protocol provided by the `ThesisManager`. The participant first expects a `postThesis` message from the tutor. Once a student has chosen the thesis, the tutor is informed with the `thesisInProgress` call. Now, the tutor may send `getStatus` calls for retrieving the status of the thesis, for which he must be able to receive a reply. Finally, once the `getAssessment` call is sent to the tutor, he must send back the assessment with the `getAssessment` call.

3.2 Other Examples

Besides the thesis management scenario of the eUniversity case study, UML4-SOA has also been used for several other scenarios from different case studies within the SENSORIA project.

- In the context of the eUniversity case study, another scenario has been modelled with UML4SOA, which revolves around a *student application* to an online university.
- From the automotive case from the SENSORIA project, several scenarios have been modelled in UML4SOA, the most elaborate of which is the *roadside assistance* scenario.
- Finally, the finance case study and its *credit request* scenario have been modelled with UML4SOA.

An overview of these scenarios is given in [EGK+10]. Furthermore, the SENSORIA web site `www.sensoria-ist.eu` contains tutorials and downloads for each of these case studies.

Chapter 4

Summary

This document has introduced the UML4SOA profile, a lightweight extension of the UML for modelling the behaviour and the protocols provided and required of participants in service-oriented architectures.

In chapter 1, we have discussed the need for an extension for service behavioural modelling in the UML due to insufficient representation of key service concepts such as communicating actions, long-running transactions, and self-descriptions in activities and protocol state machines.

Chapter 2 has then introduced the UML4SOA meta-model and, subsequently, the profile. According to the main aim of the definition of the UML4-SOA profile — minimalism and conciseness — we have defined additions to the UML for both activities and PrSMs, while reusing existing UML constructs such as structured activities, actions, and control structures such as fork or decision nodes.

This chapter has also discussed a lightweight data manipulation language for guards, actions, and pins in UML4SOA, which enables the modeller to stay on the same level of abstraction as in the rest of UML4SOA. Finally, in order to accommodate different usage scenarios of UML4SOA, the two *dialects* UML4-SOA/Open and UML4SOA/Strict have been introduced. The former focuses on maximum expressiveness and integration with existing UML constructs, while the latter adds a set of constraints for ensuring unambiguous models ready for code generation and analysis.

Finally, Chapter 3 has shown a practical example of how to model with UML4SOA in the form of diagrams for the eUniversity case study of Sensoria. More examples can be found in [EGK+10].

Bibliography

[EGK+10] Jannis Elgner, Stefania Gnesi, Nora Koch, , and Philip Mayer. *Specification and Implementation of Demonstrators for the Case Studies*, chapter 7.1. Springer Verlag, 2010.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2005.

[HÖ7] Matthias Hölzl. D8.4a: Distributed E-University Management and E-Learning System: Requirements modelling and analysis of selected scenarios. Deliverable for the eu project sensoria, reporting period october 2006 - september 2007, SENSORIA Project 016004, 2007.

[OMG09] OMG (Object Management Group). Service oriented architecture Modeling Language(SoaML), Beta 2. Technical report, OMG (Object Management Group), 2009.

[OMG10a] OMG (Object Management Group). Unified Modeling Language: Infrastructure, version 2.3. Technical report, OMG (Object Management Group), 2010.

[OMG10b] OMG (Object Management Group). Unified Modeling Language Superstructure. Specification, OMG (Object Management Group), 5 2010. `http://www.omg.org/spec/UML/2.3/Superstructure/`.