UML Extensions for Service-Oriented Systems *

Howard Foster[1], László Gönczy[2], Nora Koch[3,4], Philip Mayer[3], Carlo Montangero[5], and Dániel Varró[2]

[1] Imperial College London, UK
[2] Budapest University of Technology and Economics, Hungary
[3] Ludwig-Maximilians-Universität München, Germany
[4] Cirquent GmbH, Germany
[5] Universitá di Pisa, Italy
howard.foster@imperial.ac.uk, {nora.koch,philip.mayer}@pst.ifi.lmu.de, monta@di.unipi.it, {varro,gonczy}@mit.bme.hu

Abstract. A trend in software engineering is towards model-driven development. Models are used to document requirements, design results, and analysis in early phases of the development process. However, the aim of modeling is very often more ambitious as models are used for automatic generation in so-called model-driven engineering approaches. The relevance of models leads to the need of both, high-level domain specific modeling languages (DSML), and metamodels which are the basis for the definition of model transformations and code generation.
For the service-oriented computing domain we developed within the Sensoria project a DSML for building and transforming SOA models. This DSML is defined as a family of UML profiles, which complement the SoaML profile for the specification of SOAs structure. Our family of profiles focus on orchestration of services, service-level agreements, non-functional properties of services, implementation of service modes and service deployment.

1 Introduction

A range of domain-specific languages and standards are available for engineering service-oriented architectures (SOAs) such as Web Services Description Language (WSDL), Web Services Business Process Execution Language (WS-BPEL), Web Services Choreography Description Language (WS-CDL), WS-Policy and WS-Security. These languages deal with the various aspects of SOA systems, such as service descriptions, orchestrations, policies and non-functional properties of services at a specification level. However, more systematic and model-driven approaches are needed for the development of service-oriented software. Models of SOAs are required for providing a complete – whenever possible a graphical – picture of the systems represented at a high level of abstraction. Achieving the properties of service-oriented systems mentioned above requires then model elements that ease the understanding of the individual artefacts of a system, and their integration.

* This work has been partially sponsored by the project Sensoria, IST-2 005-016004.

Within the Sensoria project, we have created ways of modeling these different aspects with the help of the Unified Modeling Language (UML)[24]. The UML is accepted as lingua franca in the development of software systems. It is the most mature language used for modeling. However, plain UML is not expressive enough for the specification of structural and behavioral aspects of services. Service modeling introduces a new set of key distinguishing concepts, for example partner services, message passing among requester and provider of services, compensation of long-running transactions, modes, and policies associated to services. Without specific support for those concepts in the modeling language, diagrams quickly get overloaded with technical constructs, degrading their readability.

Several attempts have been made to add service functionality to the UML. Most notably, SoaML [25] is an upcoming standard UML profile of the OMG for specification of service-oriented architectures, which does only cover structural aspects. Our own contribution to the field of UML service modeling complements SoaML, and consists in introducing more service-specific model elements mainly for the behavioral aspects of services-oriented software. In a first step, metamodels are defined as a conservative extension of the UML metamodel, i.e. they do not imply any adjustment in the UML metamodel. In a second step, UML profiles are created for these metamodels using the UML extension mechanisms provided by mapping stereotypes to the metaclasses. The result is the Sensoria family of UML profiles for the development of SOAs.

The use of the UML for modeling has many advantages when compared to the use of proprietary modeling techniques. These advantages are (1) to be able to use existing CASE tool support, which is provided by commercial and open source tools; (2) to avoid the definition from scratch of a new modeling language, which would require an own project to detail their syntax, semantics and provide user-friendly tool support. These metamodels and the corresponding UML profiles constitute the basis for model transformations and code generation defining a model-driven development process. In particular, the MDD4SOA (Model-Driven Development for SOA) transformers – also developed within the scope of the Sensoria project – are model transformations implemented as Eclipse plug-ins. They automatically transform service orchestrations specified with our UML4SOA profile to executable code, such as BPEL/WSDL, Java and Jolie.

In the following sections, we will discuss the individual UML extensions which form our Sensoria family of profiles for SOA development and the SoaML profile (section 3), which are jointly used to model the different aspects of service-oriented software. The Sensoria family of profiles comprise UML4SOA, a profile for service orchestration (section 4), for non-functional properties of services (section 5), business policies (section 6), for implementation modes of SOAs (section 7), and service deployment (section 8). These UML profiles can be used separately or in combination, depending on the software requirements and the decisions of the service engineer. The running example belongs to the case study from the automotive domain, which is detailed in section 2. Finally, in section 9 we present some related work and conclude in section 10.

2 Case Study

The SENSORIA family of profiles that are presented in the following sections are illustrated by models of the *On Road Assistance* scenario of the automotive case study [15,29]. In this scenario, the diagnostic system reports a failure in the car engine, for example, the vehicle's oil lamp reports a low oil level. This triggers the in-vehicle diagnostic system to perform an analysis of the sensor values. The diagnostic system reports e.g. a problem with the pressure in one cylinder head, and therefore the driver will not be able to reach the planned destination. The diagnostic system sends a message for starting the assistance system, which orchestrates a set of services.

Based on availability and the driver's preferences, the service discovery system identifies and selects the appropriate services in the area: repair shops (garage) and rental car stations. The selection of services takes into account personalized policies and preferences of the driver to find these "best services". We assume that the owner of the car has to deposit a security payment before being able to order services. In order to keep the scenario simple, we limit the involved services, but they could be easily extended e.g. to identify as well a towing service, providing the GPS data of the stranded vehicle in case the vehicle is no longer drivable. In such a case, the driver makes an appointment with the towing service, and the vehicle will be towed to the shop.

The *On Road Assistance* scenario is complemented with the *Emergency* scenario [15] that is needed when the damaged car blocks the route and a convoy behaviour is required from other cars. It is used to illustrate the reconfiguration issues of a service-oriented system. In case of an emergency, the vehicles that are driven in a default mode are reconfigured to be driven in a convoy mode guided by the Highway Emergency System. The master vehicle is then followed by the other vehicles of the convoy. In the *Emergency* scenario the car navigation system is able to react to events which cause the switching between the modes specified in the different architecture configurations.

3 Modeling Structural Aspects of SOAs

The basic structure of a software system is the ground layer on which other specifications are based – this holds true not only for traditional architectures, but also for the SOA-based systems we have considered in SENSORIA. Although the UML does include mechanisms for modeling structural aspects of software, the specific requirements of SOA systems – for example, the central concept of a service and the separation of requested and provided services – cannot be expressed in a concise way, as services and service providers are not first level citizens of the UML.

We therefore need an extension of the UML to be able to express these ideas. In SENSORIA, we have chosen to use the existing profile SoaML, which is currently in a beta 2 phase and on its way to becoming an OMG standard. We feel that we can adequately express our ideas of structural aspects of services in SoaML, and have therefore sought to integrate our own specific profiles presented in later sections with SoaML.

In this section, we introduce some of the basic concepts specified in SoaML which we need for modeling our case study and as a basis for defining our profiles. For the complete description, please refer to the SoaML specification [25].

Structural service modeling employs the basic UML mechanisms for modeling composite structures, enhanced with stereotypes from the SoaML profile – ≪participant≫, ≪servicePoint≫, ≪requestPoint≫, ≪serviceInterface≫ and ≪messageType≫ (listed in Table 1). The basic unit for implementing service functionality is a service participant, modeled as a class with the stereotype ≪participant≫. A participant may provide or request services through ports, which are stereotyped with ≪requestPoint≫ or ≪servicePoint≫, respectively. Each port has a type, which is a ≪serviceInterface≫ implementing or using operations as defined in a standard UML interface definition.

Table 1. SoaML metaclasses and stereotypes (excerpt)

Metaclass	**Stereotype**	**UML Metaclass**	**Description**
Participant	≪participant≫	Class	Represents some (possibly concrete) entity or component that provides and/or consumes services
ServicePoint	≪servicePoint≫	Port	Is the offer of a service by one participant to others using well defined terms, conditions and interfaces. It defines the connection point through which a participant provides a service to clients
RequestPoint	≪requestPoint≫	Port	Models the use of a service by a participant and defines the connection point through which a participant makes requests and uses or consumes services
ServiceInterface	≪serviceInterface≫	Class	Is the type of a ≪servicePoint≫ or ≪requestPoint≫ specifying provided and required operations.
MessageType	≪messageType≫	DataType, Class	Is the specification of information exchanged between service requesters and providers

As an example for using these stereotypes, we present the structural diagram for the scenario introduced in the previous section (see Fig. 1). As can be seen from the figure, the central orchestration of the case study – i.e., the component which coordinates the actions of all the services – is modeled as a ≪participant≫. The *OnRoadAssistant* participant has seven ports, six of which are ≪RequestPoint≫s, indicating that a certain service is requested. The last port is a ≪ServicePoint≫, indicating that a certain service is provided.

As mentioned above, each ≪RequestPoint≫ and ≪ServicePoint≫ is typed with a ≪ServiceInterface≫ which defines, though interface realizations and usage assocations, the operations required or provided at the given port. In our case, the orchestration provides, through the ≪ServiceInterface≫ *ClientInterface*, the operation *startAssistant* to clients. In the other direction, it requires e.g. the operation *selectBestGarage* from another service, which is indicated through the ≪ServiceInterface≫ *SelectBestInterface* which is the type of the ≪RequestPoint≫ *selectBestGarage*.

With the basic structure of service-based systems and our case study specified using SoaML, we can move on to define profiles for additional aspects of SOA systems.

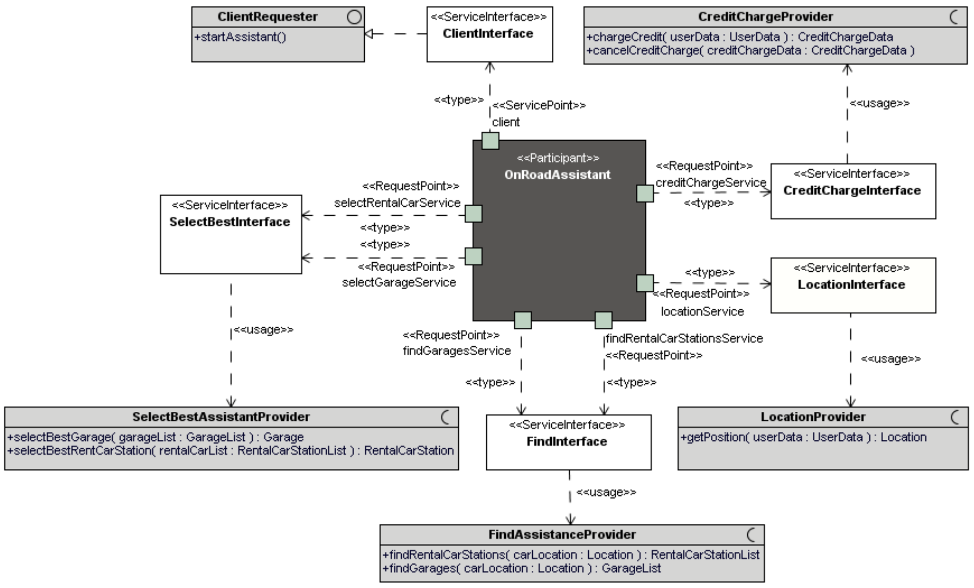


Fig. 1. SoaML structural diagram of the On Road Assistance scenario

4 Service Orchestrations

A key aspect of service-orientation is the ability to compose existing services, i.e. creating a description of the interaction of several services, which has come to be known as an *orchestration*. An orchestration is a behavioral specification of a service component, or ≪Participant≫ in SoaML. As with structural aspects, the UML does contain mechanisms for specifying behavior – for example, as activity or sequence diagrams – but does not contain specific support for constructs used in service orchestrations such as message passing, compensation, event handling, and the combination of these.

To enable developers to model service orchestration behavior in an easy and concise way, we have created UML4SOA, a profile for UML which defines a high-level domain specific modeling language (DSML) for behavioral service specifications. UML4SOA was first introduced in [19] and described in more detail in [20]. It has been used as the central language for the specification of the SENSORIA case studies and enjoys the support of several formalisms and formal tools.

4.1 Metamodel

An excerpt of the UML4SOA metamodel is shown in Fig. 2, which includes the main concepts of our DSML and the relationships among these concepts. For example, we introduce elements such as *ServiceSendAction* for modeling the asynchronous invocation of a service, i.e. without waiting for a reply from the external partner. Another specific concept of the service-oriented domain is the compensation of long-running transactions. Therefore we define model elements such as *CompensationAction* and *CompensationEdge*. For each non-abstract class of the metamodel we defined a stereotype with the objective of producing semantically enriched and increased readable models of service-oriented systems, e.g. a stereotype ≪sendAction≫ for the *ServiceSendAction* metaclass, and a stereotype ≪compensate≫ for the *CompensateAction* metaclass. Table 2 provides an overview of the elements of the metamodel, the stereotypes that are defined for these metamodel elements (they comprise the profile UML4SOA), the UML metaclasses they extend, and a brief description. For further details on UML4SOA, including the full metamodel, the reader is referred to [18].

UML4SOA proposes the use of UML activity diagrams for modeling service behavior, in particular for modeling orchestrations which coordinate other services. We assume that business modelers are most familiar with this kind of notation to show dynamic behavior of business workflows. An UML4SOA ≪ServiceActivity≫, as noted above, can be directly attached as the behavior of a ≪Participant≫.

4.2 Example

As an example for modeling a service-oriented system in UML4SOA, we show the implementation of the *On Road Assistance* scenario defined in Fig. 1.

The process *On Road Assistance* is modeled as a UML4SOA orchestration (see Fig. 3) . It illustrates how the assistance process interacts with its client and its partners through ports. It starts with a receipt (≪receive≫) of the call *startAssistant* through the *client* port, receiving the request to start the assistance. Note that the initial call to *startAssistant* starts the complete activity – a convention we chose to make the workflow more explicit. Furthermore, note that the port is given in the ≪lnk≫ pin, while the operation is denoted in the main body of the action.

Table 2. UML4SOA metaclasses and stereotypes

Metaclass	**Stereotype**	**UML Metaclass**	**Description**
ServiceActivity Node	≪serviceActivity≫	Activity, StructuredActivityNode	Represents a special activity for service behavior or a grouping element for service-related actions
ServiceSendAction	≪send≫	CallOperationAction	Is an action that invokes an operation of a target service asynchronously, i.e. without waiting for a reply. The argument values are data to be transmitted as parameters of the operation call. There is no return value
ServiceReceiveAction	≪receive≫	AcceptCallAction	Is an accept call action representing the receipt of an operation call from an external partner. No answer is given to the external partner
ServiceSend&Receive	≪send&receive≫	CallOperationAction	Is a shorthand for a sequential order of send and receive actions
ServiceReplyAction	≪reply≫	ReplyAction	Is an action that accepts a return value and a value containing return information produced by a previous ServiceReceiveAction action
CompensationEdge	≪compensation≫	ActivityEdge	Is an edge which connects an orchestration element to be compensated with the one specifying a compensation. It is used to associate compensation handlers to activities and scopes
EventEdge	≪event≫	ActivityEdge	Is an edge connecting event handlers with an orchestration element during which the event may occur. The event handler attached must contain a receive or a timed event at the beginning.
CompensateAction	≪compensate≫	Action	Triggers the execution of the compensation defined for a certain named service activity (can only be inserted in compensation or exception handlers)
CompensateAllAction	≪compensateAll≫	Action	Triggers compensation of all nested service activities from the service activity attached to the current compensation or exception handler. The nested service activities are compensated in reverse order of completion.
LinkPin	≪lnk≫	InputPin	Holds a reference to the partner service by indicating the corresponding service point or request point involved in the interaction
SendPin	≪snd≫	InputPin	Is used in send actions to denote the data to be sent to an external service
ReceivePin	≪rcv≫	OutputPin	Is used in receive actions to denote the data to be received from an external service

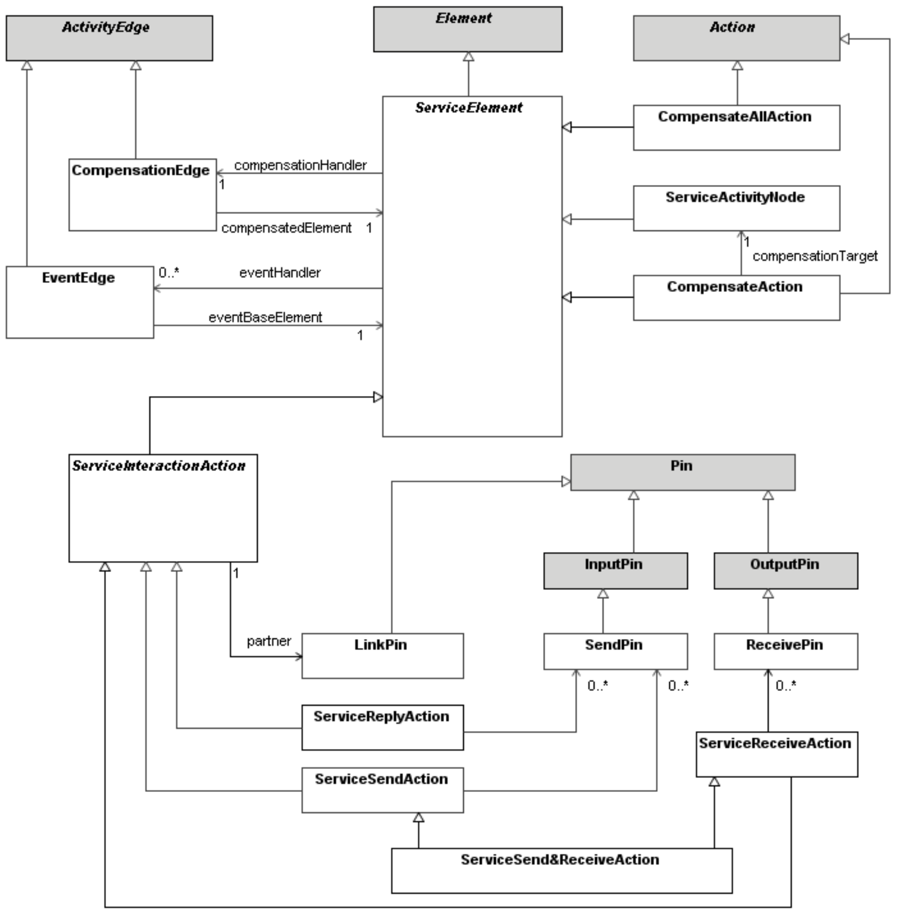


Fig. 2. Excerpt of the UML4SOA metamodel (includes some highlighted UML metaclasses)

Once the initial request has been received, the process goes on to interact with partner services. The process first charges the credit card of the user to ensure that payment is available for later actions. This is done with the help of a ≪send&receive≫ action, invoking the operation ≪chargeCredit≫ on the service attached to the ≪RequestPoint≫ *creditChargeService*. The ≪send&receive≫ action also uses a ≪snd≫ pin for denoting the information to be sent (the variable *userData*, in this case) and the variable in which the return information will be stored (*creditChargeData*, defined in the ≪receive≫ pin). A similar call is placed to retrieve the position of car using the *locationService* port.

Once this initial setup phase has completed, the process enters the *findAssistance* service activity. Here, it simultaneously interacts with two external services available through the *findGaragesService* and *findRentalCarStationsSer-*

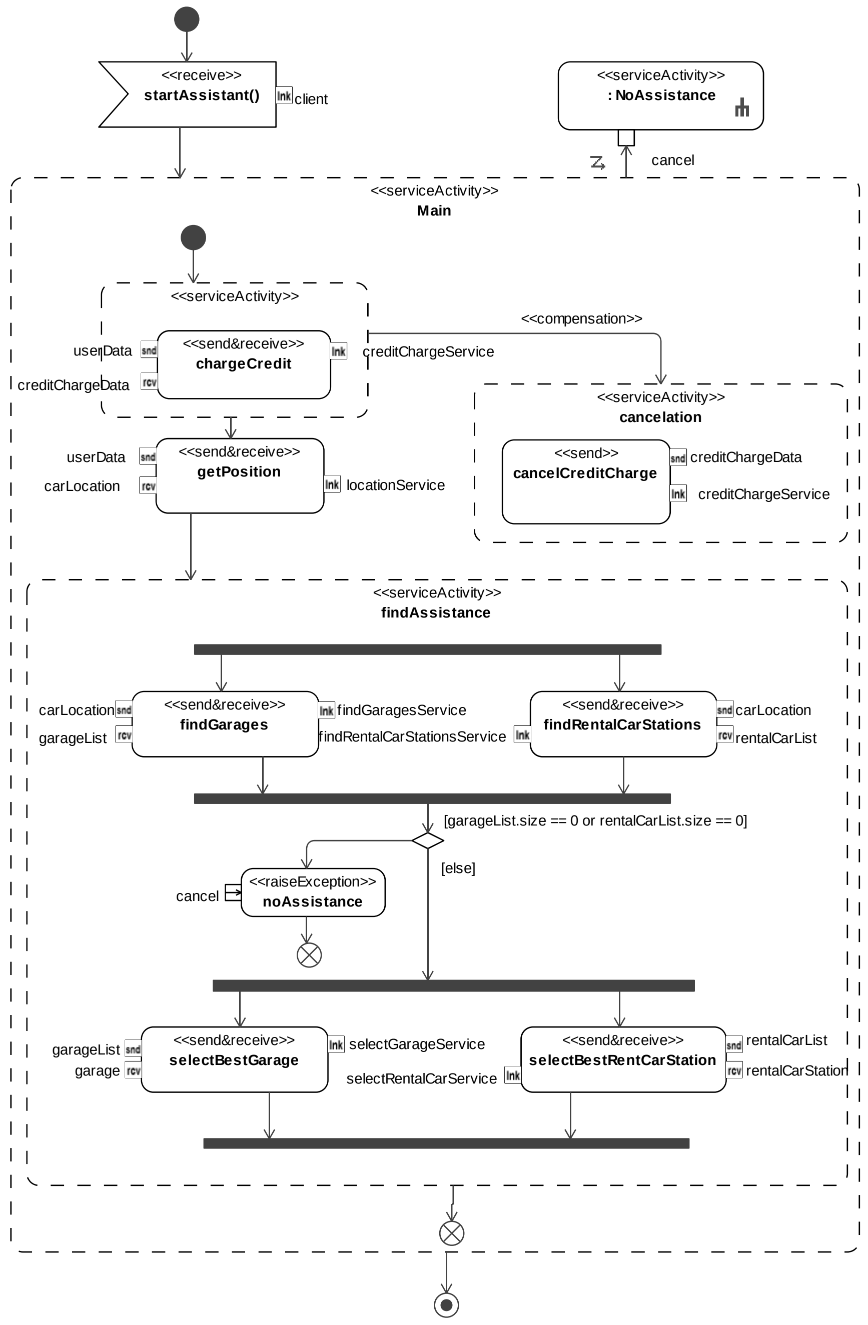


Fig. 3. UML4SOA activity diagram showing the OnRoadAssistance participant

vice ports. If the process finds both a garage and a rental car station, it continues to retrieve the nearest one. If it is not able to find at least one garage and one rental car station, an exception is thrown.

Note that there is a standard UML exception handler attached to the service activity. Inside the exception handler, the process invokes a ≪compensateAll≫ action. The meaning of this action is to undo previously and successfully completed work. In this case, the process refers of course to the credit card charge. To be able to undo this operation, a compensation handler is attached to that action, which consists of an action canceling the charge with another service call to the service identified by the *CreditChargeInterface* port.

As can be seen from Fig. 3, the behavioral specification of this process is concise and very readable. The specification also directly uses elements defined in the structural diagram in Fig. 1, thus exploiting the information defined there, not repeating it unnecessarily.

4.3 Model-Driven Development Support

UML4SOA specifications can be used for more than just modeling to understand the semantics of a system. With the MDD4SOA (Model-Driven Development for SOA) transformers, UML4SOA orchestrations can be automatically transformed to executable code in BPEL/WSDL, Java, and Jolie by using model transformations.

Also, UML4SOA models enjoy formal methods support – the SENSORIA project includes tools and methods for checking qualitative and quantitative properties of orchestrations, as well as checking protocol compliance of an orchestration. UML4SOA was used to model different scenarios of the SENSORIA case studies. We refer the interested reader to Chapter 7-1 of this book for more information.

5 Non-Functional Properties of Services

Non-functional extensions of UML4SOA aim to provide the modeling of arbitrary "quality of service" properties defined for a particular given client-server pair. Since in real service configurations, service properties can vary for different classes of clients, we follow a contract-based approach, where non-functional properties of services are defined between two *participant* components, namely, the *service provider* and the *service requester*. These contracts are modeled by ≪nfContracts≫. Different non-functional aspects (performance, security, etc.) are modeled in corresponding ≪nfCharacteristics≫ which group different properties in ≪nfDimensions≫ (where a ≪runTimeValue≫ is associated to each dimension). The reason for creating separate classes for these instead of storing in properties is to correlate real SLAs where most parameters are typically bound to a range of allowed values. Moreover, concepts like average values, deviation, etc. need to be modeled in a uniform way.

During a negotiation process, participants create an agreed contract of the provider and requester. Finally, properties of services need to be monitored at runtime (modeled as ≪monitor≫) either by the participating parties or by involving a separate entity.

5.1 Metamodel

A metamodel of UML4SOA-NFP is shown in Fig. 4. The profile was motivated by the UML 2.0 Profile for QoS & Fault Tolerance [23]. However, we followed a more simple way of defining a general framework for QoS, which then can be "instantiated" by defining concrete aspects such as performance, security, etc. Table 3 shows the usage of the stereotypes.

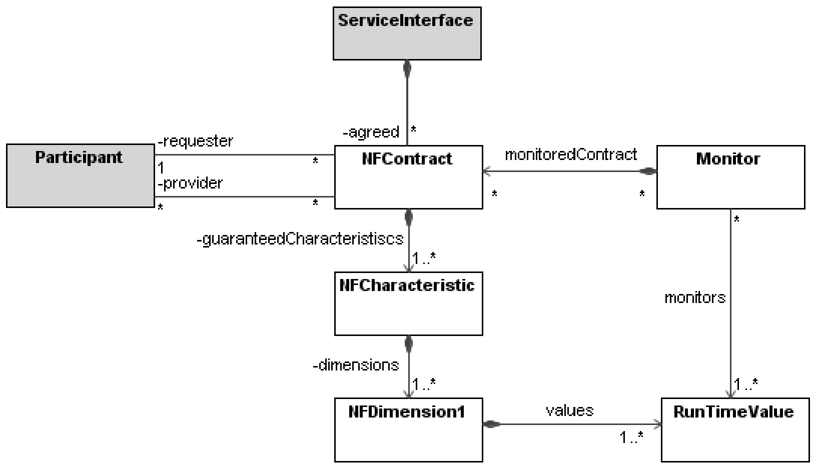


Fig. 4. Metamodel of non-functional extensions (includes some highlighted SoaML metaclasses

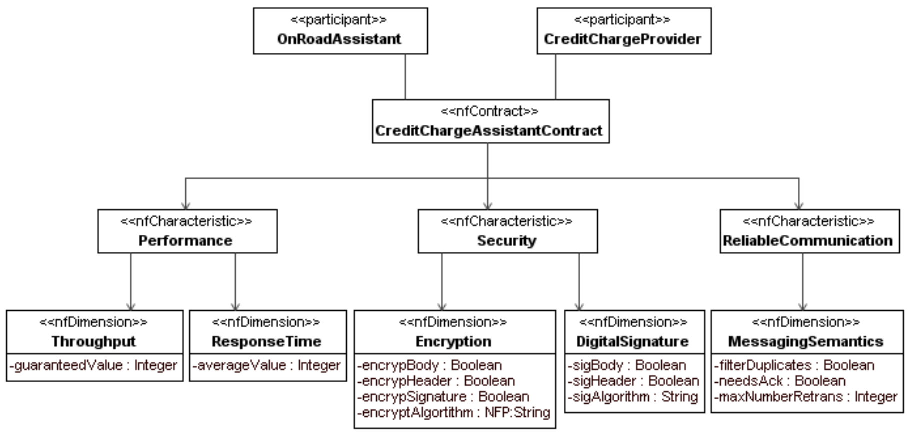
5.2 Examples

In the *On Road Assistance* scenario, several QoS requirements can be formed on service connections. To illustrate the use of UML4SOA-NFP, we modeled (part of) a contract between *OnRoadAsssistant* (the orchestrator component) and *CreditChargeProvider*. First we show a brief textual specification of non-functional requirements:

– All communications between these services must be secure, e.g. message content must be encrypted and digitally signed.
– All messages from the orchestrator component to the credit card manager must be acknowledged when received.

– As all succesful scenarios pass this step, the throughput of the service must be high enough (1000 requests per hour) with a reasonable response time.

Fig. 5 shows an excerpt of a concrete contract. Note that the class diagram corresponds to a template which is filled (instantiated as object diagram). This will include concrete values for encrypting methods, response time, etc.



<<participant>> OnRoadAssistant
<<participant>> CreditChargeProvider
<<nfContract>> CreditChargeAssistantContract
<<nfCharacteristic>> Performance
<<nfCharacteristic>> Security
<<nfCharacteristic>> ReliableCommunication
<<nfDimension>> Throughput
-guaranteedValue : Integer
<<nfDimension>> ResponseTime
-averageValue : Integer
<<nfDimension>> Encryption
-encrypBody : Boolean
-encrypHeader : Boolean
-encrypSignature : Boolean
-encryptAlgortithm : NFP:String
<<nfDimension>> DigitalSignature
-sigBody : Boolean
-sigHeader : Boolean
-sigAlgorithm : String
<<nfDimension>> MessagingSemantics
-filterDuplicates : Boolean
-needsAck : Boolean
-maxNumberRetrans : Integer

Fig. 5. Non-functional paramaters in (part of) the On Road Assistance scenario

5.3 Model-Driven Development Support

As this profile may describe arbitrary types of requirements (logging, security, performance, etc.), the development support for different aspects obviously vary for different development phases (early design/analysis/deplyoment/operation).

Table 3. UML4SOA-NFP metaclasses and stereotypes

Metaclass	**Stereotype**	**UML Metaclass**	**Description**
NFContract	≪nfContract≫	Class	Represents a non-functional contract between a service provider and a service requester
NFCharacteristic	≪nfCharacteristic≫	Class	Represents a non-functional aspect such as performance, security, reliable messaging, etc.
NFDimension	≪nfDimension≫	Class	Groups non-functional properties within a non-functional aspect (characteristics)
RunTimeValue	≪runTimeValue≫	Attribute	An actual non-functional property
Monitor	≪monitor≫	Class	A run-time service to monitor a contract (not used in the paper)

UML4SOA-NFP has support for middleware-level performability analysis as described in [7] and [9]. This enables the early estimation of a trade-off between reliability and performance. Evolving model transformations are developed to support the automated code generation of middleware configuration with QoS constraints. Details of this technology are described in [6] (modeling), [8] (performability analysis) and [9] (transformations for deployment). These transformations are based on the VIATRA framework, described in details in Chapter 6-2 of this book.

Also a transformation-based technique is currently under development, which will help to create simple transformations on UML4SOA-NFP models extended with additional information on their intended usage (e.g. security analysis) and/or target platform (e.g. Apache stack). These models and transformations give a flexible tool to support the quickly changing WS-* platforms in every phase of service engineering. This transformation set will include validation steps to check both modeling errors and domain specific requirements.

6 Business Policies Support

This part of the UML4SOA profile deals with the connection of services and business policies, in the context of StPowla [10]. The goal of StPowla is to define the business process so that the business stakeholder can easily adapt it to the current state of affairs, by controlling the resources used by the basic tasks in the workflows. To this purpose, the stakeholders issue *policy* definitions, which constrain the resource usage as a function of the state of the workflow when a task is needed.

Here we show how to define business workflows in terms of *taskSpecifications*, that is, interfaces of ServicePoints as from SoaML, enriched with information on the ranges of variability in the use of resources (*service level dimensions*).

We note that the profile we present here does not cover policies explicitly. This is why it is called Business Policies *Support* profile. Indeed, policies are better expressed as tables than as UML models. In another chapter of this book (Chapter 1-3) it is shown how to integrate workflows and policies in a SOA, to support flexible workflow enactment.

6.1 Metamodel

The metamodel for business policies support consists of a series of related elements, relationships, and a number of constraints. Not all the concepts are new, since we exploit the *NFDimension* concept from the NF-UML4SOA profile, (cfr. Section 5). We deal first with the elements devoted to the basic tasks in (Fig. 6):

- *ServiceInterface* specifies the interface of the service point a Task connects to at enaction time. Constraint: just one operation specified.
- *Requires* is used to link a TaskSpecification to its ServiceInterface.

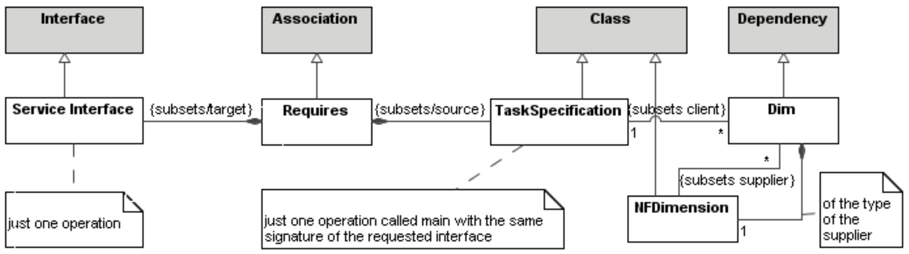


Fig. 6. Metamodel for business policies support: Task specification (includes some highlighted UML classes)

- *TaskSpecification* specifies a Task, identifying (via Requires) the ServiceInterface. It also specifies (via Dim) the non-fuctional dimensions that characterize the service to invoke.
- *TaskSpecification* owns an operation called main, with the same parameters and return type of the required service. Indeed, main triggers the search and invocation of a suitable service, and returns the computed result. The search identifies a service implementation that satisfies the current policies.
- *Dim* allows specifying the relevant service level dimensions in a TaskSpecification, by linking to ≪nfDimension≫ from NF-UML4SOA. It also defines a default value, which is used to select the service provider, when no policy with specific requirements for the target dimension is in place.

The next concepts are depicted in Fig. 7:

- *WfSpecification* defines a workflow, specifying its attributes and internal behavior. The formers can be used to express conditions in the policies. The behavior is specified by the owned WfActivity.
- *Workflow* is an activity action that calls the specified behavior, i.e., a lower level workflow.
- *Task* is an activity action that calls the specified main operation.
- *WfActivity* defines the behavior of a workflow. Constraint: an owned action is either a Workflow or a Task.

All the concepts above are rendered as stereotypes in the UML profile shown in Table 4. The defaultValues are rendered as tagged values of the ≪dim≫ dependency: the tag is defaultValue, and the type is given by the target dimension.

6.2 Examples

To show how StPowla supports flexibility in the *On Road Assistance* scenario, we consider, within the general OnRoadAssistance workflow, a single task, namely the one that selects the best garage, and a policy that allows the driver to choose directly the repairing services which he knows and trusts, in his own town:

Table 4. Business policies support metaclasses and stereotypes

Metaclass	**Stereotype**	**UML Metaclass**	**Description**
ServiceInterface	≪serviceInterface≫	Interface	Specifies the interface of the service point a Task connects to at enaction time
Requires	≪Requires≫	Association	Associates a TaskSpecification to the signature of the services that implement it
TaskSpecification	≪taskSpecification≫	Class	Specifies a Task, functionally via Requires, and non-functionally via Dim. The latter identifies the QoS dimensions that characterize the service to invoke. It owns a main operation, with the same parameters and return type of the required service, whose behavior is to trigger the search and invocation of a suitable service (i.e., one whose QoS characteristics satisfy the current policies), and to return the computed result
Dim	≪Dim≫	Dependency	Allows specifying the relevant service level dimensions in a TaskSpecification, by linking to ≪nfDimension≫ from UML4SOA-NFP
WfSpecification	≪WfSpecification≫	Class	Defines a workflow, specifying its attributes and internal behavior. The latter is specified by the owned WfActivity
WfActivity	≪WfActivity≫	Activity	Defines the behavior of a workflow. Constraint: an owned action is either a Workflow or a Task
Workflow	≪Workflow≫	CallBehavior-Action	Calls the specified behavior, namely, a lower level Workflow
Task	≪Task≫	CallOperation-Action	Calls the specified main operation

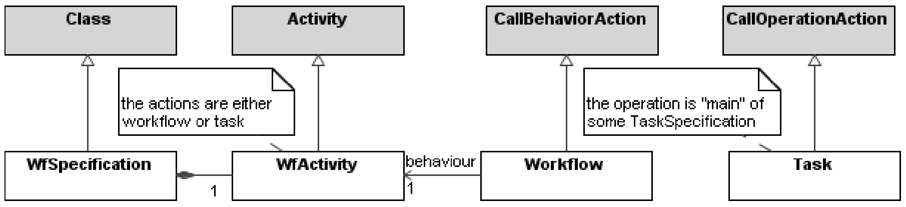


Fig. 7. Metamodel for business policies support: Workflow specification (includes some highlighted UML classes)

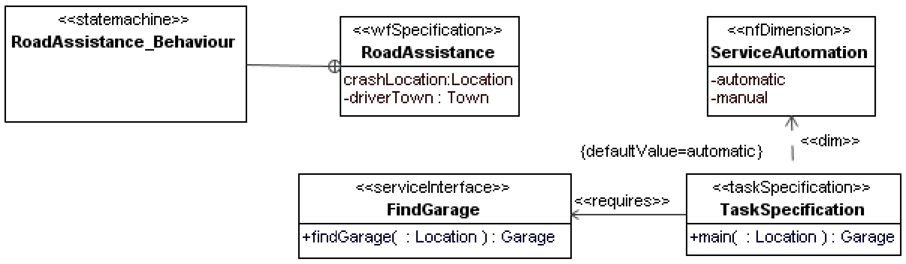


Fig. 8. Fragments of the On Road Assistance model

If the car fault happens in the driver's town, then let him select the services to be used. Otherwise choose the services automatically.

Fig. 8 shows an excerpt from the model of the scenario just outlined, exemplifying the use of the concepts both at the workflow and at the task level. To formalize the policies, the modeler needs to define, for the workflow, the attributes that specify the driver's home town and the car crash location, as detected by the embedded car GPS. Indeed, they are needed to express the conditions in the policy. So, the ≪wfSpecification≫ RoadAssistance, at the top-centre of the figure, lists the two attributes crashLocation and driverTown. The relevant part of the related ≪wfActivity≫ is shown to the left: the actions appear as shown: here, the task invokes the main operation of FindGarage, whose ≪taskSpecification≫ appears to the right (bottom). The name of the node is of little importance, being useful only to distinguish two nodes in the same workflow, when they use the same ≪taskSpecification≫.

Moreover, ≪taskSpecification≫ FindGarage requires the findGarage service (at its left), and declares the main operation accordingly. The modeler here has to introduce a suitable ≪nfDimension≫ to express the choice between the service that searches for a garage nearby the crash location, and the service that interacts with the driver to contact his own choice. This is AutomationLevel (top-right). The default value is fixed as automatic, via the tagged value for ≪dim≫.

7 Service Modes for Adaptive Service Brokering

In this section we describe a part of the Sensoria family of profiles that addresses service adaptation and reconfiguration based upon operational states of the service system being described. The Service Modes profile complements the UML4SOA profile for orchestration by providing an abstraction of service system adaptation through architecture, behavior and constraints. Service Modes are an extension of Software Architecture Modes.

Software Architecture Modes are an abstraction of a specific set of services that must interact for the completion of a specific subsystem task, i.e., a mode will determine the structural constraints that rule a (sub)system configuration at runtime [13]. Therefore, passing from one mode to another and interactions among different modes formalize the evolution constraints that a system must satisfy: the properties that reconfiguration must satisfy to obtain a valid transition between two modes which determine the structural constraints imposed to the corresponding architectural instances. A Service Mode represents a Software Architecture Mode scenario of a service system. It combines a service architecture with behavior and policy specifications for service components within the service system and is intended to be evolved as new requirements are desired from the system. In this section we detail the specification of service modes by way of a Service Modes profile in the UML notation.

7.1 Metamodel

A metamodel for service modes (illustrated in Fig. 9) extends and constrains a number of UML core elements. As an overview, a *ModeModel* defines a package which contains a number of service architecture scenarios (as *Mode* packages) and components and also contains a *ModeModelActivity* to define how to switch between different service scenarios. Each scenario is defined in a *Mode* package which is a container for a *ModeCollaboration* and describes the role that each component plays within the scenario (e.g. a service requester and/or a provider). Each *ModeCollaboration* holds a *ModeActivity* which describes the process in which the mode orchestration is fulfilled. Each *ModeCollaboration* also refines the components of the *Mode* for additional service adaptation requirements (such as the constraints for service brokering). A *ModeConstraint* specifies a constraint on adaptation of ModeCollaborations. These constraints can also specify Quality-Of-Service (QoS) attributes for service components, and we reuse the *QoSRequired* and *QoSProvided* stereotypes related to the QoS Profile (as discussed in section 5.1). We now elaborate on service mode architecture, behaviour and adaptation relationships through examples.

7.2 Examples

A **Service Modes Architecture** consists of specifying the service components, their requirements and capabilities and interface specifications. A high-level architecture configuration is given in UML to represent the component specifications and their relationships. Each component will offer services to its clients,

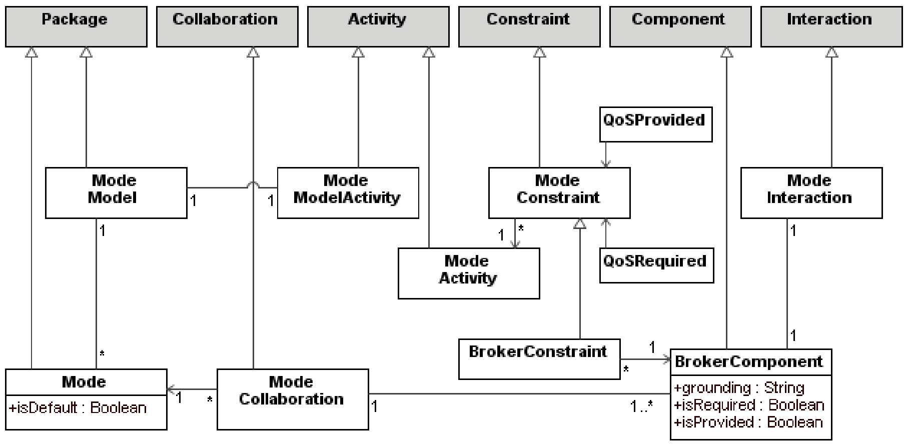


Fig. 9. Metamodel for service modes and service brokering specification (includes some highlighted UML classes)

each such service is a component contract. A component specification defines a contract between clients requiring services, and implementers providing services. The contract is made up of two parts. The static part, or usage contract, specifies what clients must know in order to use provided services. The usage contract is defined by interfaces provided by a component, and required interfaces that specify what the component needs in order to function. The interfaces contain the available operations, their input and output parameters, exceptions they might raise, preconditions that must be met before a client can invoke the operation, and post conditions that clients can expect after invocation. These operations represent features and obligations that constitute a coherent offered or required service. At this level, the components are defined and connected in a static way, or in other words, the view of the component architecture represents a complete description disregarding the necessary state of collaboration for a given goal. Even if the designer wishes to restrict the component diagram to only those components which do collaborate, the necessary behavior and constraints are not explicit to be able to determine how, in a given situation, the components should interact. An example composite structure diagram for a service modes architecture is illustrated in Fig. 10 for the Emergency scenario of the Automotive Case Study (discussed in section 2). Note that the architecture represents both local services (via a localDiscovery component) and remote services (remoteDiscovery via a Vehicle Services Gateway).

Service Mode Behavior specification is a local coordinated process of service interactions and events for mode changes. The behavior is similar to that of service orchestrations, for which orchestrations languages such as WS-BPEL are widely adopted. Service mode behavior may be formed as described in section 4.

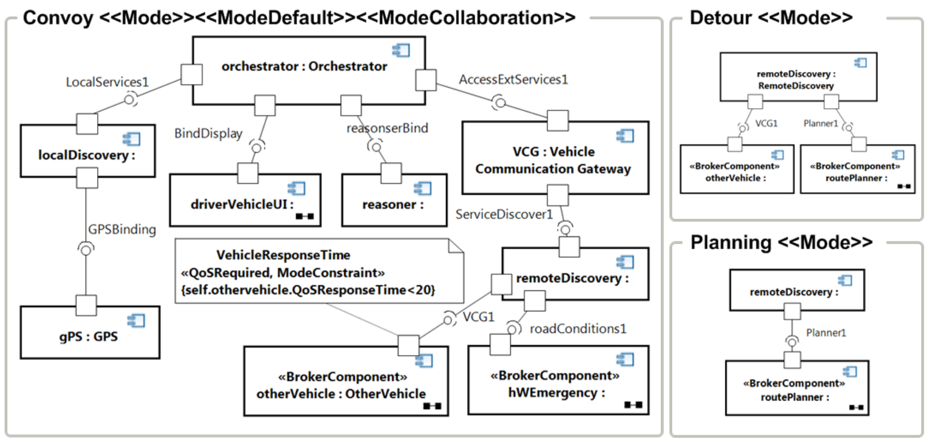

Convoy <<Mode>><<ModeDefault>><<ModeCollaboration>>
orchestrator : Orchestrator
LocalServices1
AccessExtServices1
localDiscovery :
BindDisplay
reasonserBind
VCG : Vehicle Communication Gateway
driverVehicleUI :
reasoner :
GPSBinding
ServiceDiscover1
VehicleResponseTime
«QoSRequired, ModeConstraint»
{self.othervehicle.QoSResponseTime<20}
remoteDiscovery :
gPS : GPS
VCG1
roadConditions1
«BrokerComponent» otherVehicle : OtherVehicle
«BrokerComponent» hWEmergency :
Detour <<Mode>>
remoteDiscovery : RemoteDiscovery
VCG1
Planner1
«BrokerComponent» otherVehicle :
«BrokerComponent» routePlanner :
Planning <<Mode>>
remoteDiscovery :
Planner1
«BrokerComponent» routePlanner :

Fig. 10. Emergency service brokering architecture with Modes profile

At design time however, the activities for mode orchestration consist of two concepts. Firstly, orchestrating the *default* composition of services required and provided in the specified mode architecture. Secondly, the orchestration should also be able to react to events which cause mode changes, or in other words cater for the switching between the modes specified in the different architecture configurations. To specify mode changes, the engineer adds event handlers (and follow on activities) to react to certain events which cause a mode change. An example Service Mode Behavior is illustrated in Fig. 11. Note the events that lead to mode changes, for example receiving notification of an accident from an highway emergency service leads to a mode switch to a detour mode configuration.

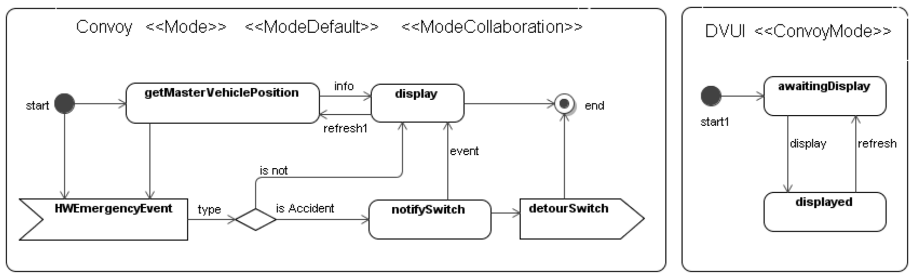


Fig. 11. Convoy service mode behavior specified in an activity diagram

Service Dynamism and Adaptation focuses on constraining changes to architecture and services, identifying both functional and non-functional variants on the specification. Using the Service Modes Profile we identify ModeCollaborations (composite structure diagrams) with *ModeConstraints* (UML constraints) which are categorised further by a constraint stereotype. Additionally, architectural constraints may be specified in the Object Constraint Language (OCL) or another constraint based language. The constraint language adopted becomes an implementation-dependent aspect of analysing models in UML. The ModeConstraint is itself extended to support a specific kind of adaptation, that for service brokering. A *BrokerComponent* defines a service component which is included in service brokering specifications and can be used to identify the role of the brokered component (either requested or provided), and holds a specification for the service profile. Additionally, one or more (*BrokerConstraints*) can be associated with a BrokerComponent, to identify the QoS either requested or provided by the service. An example constraint applied to a BrokerComponent is also illustrated in Fig. 10, in this case for the requirement that a *QoSResponseTime* should be offered less than 20ms by the other vehicle service.

As a summary of the semantics for the Service Modes profile, we list each profile metaclass, stereotype and UML metaclass in Table 5. Service Mode models built using the specification described in this section can be analysed for safety and correctness using the approach described in [3] and used for generating runtime service broker requirements and capability specifications as described in [5].

Table 5. Service Modes metaclasses and stereotypes

Metaclass	**Stereotype**	**UML Metaclass**	**Description**
ModeModel	≪ModeModel≫	Package	A Model containing Mode packages
ModeModelActivity	≪ModeModelActivity≫	Activity	The process flow for a ModeModel (policy)
ModeCollaboration	≪ModeCollaboration≫	Collaboration	Contains composite structure and interactions
ModeActivity	≪ModeActivity≫	Activity	The process flow for a Mode (orchestration)
ModeConstraint	≪ModeConstraint≫	Constraint	Constraints on mode service or activity action
ModeInteraction	≪ModeInteraction≫	Interaction	Interaction protocol between Mode components
BrokerComponent	≪BrokerComponent≫	Component	Service component to be brokered within a Mode
BrokerConstraint	≪BrokerConstraint≫	ModeConstraint	A constraint on a BrokerComponent

8 Service Deployment

In this section we describe a part of the Sensoria family of profiles that addresses describing service composition deployment and more specifically, how service orchestrations are configured with appropriate infrastructure nodes and resources. A Service Deployment profile complements the UML4SOA profile for orchestration by providing an abstraction of service composition deployment through infrastructure nodes such as web server and servlets.

Service compositions, implemented as web services using BPEL or other execution languages, are executed by a specialist container, sometimes called a service composition engine or run-time environment. These containers use various system resources depending on the activities specified in the composition. BPEL engines will, for example upon receiving a SOAP message to start a BPEL process, instantiate this process and execute it in a separate thread concurrently with other ongoing BPEL processes. Again BPEL engines typically have configurable database connections and thread pools and they would delay the start of a BPEL process until they can assign a thread from a pool. Both Web service and BPEL containers typically map these threads efficiently to a set of operating system threads. The amount of operating system threads however, is finite due to the finite amount of memory required to handle the stack segment of the thread. Administrators must therefore carefully configure the thread pools to avoid exhaustion of the operating system resources. A Service Deployment model of the architecture can describe the characteristics of the host server and orchestration, and can be used to analyze such configurations for safety and correctness.

8.1 Metamodel

The Service Deployment metamodel (illustrated in Fig. 12) focuses on modeling the deployment architecture nodes (Servlet, WebServer) and deployment artifacts (ServiceOrchestration and Resource). One or more service orchestrations

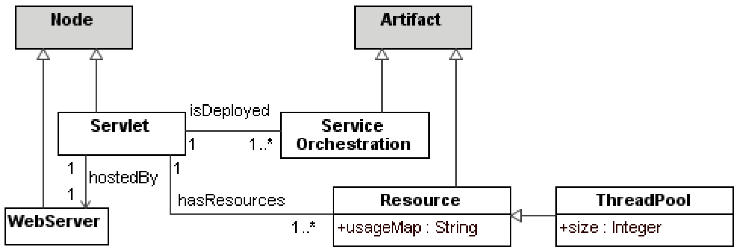


Fig. 12. Metamodel for service composition deployment and resources (includes some highlighted UML classes)

(of type ServiceOrchestration) are modeled as artifacts which are deployed on to servlet nodes. A service orchestration can only be deployed to one servlet instance. Servlets are hosted on web server nodes (a web server is a web container which manages the creation and deletion of servlet instances). A servlet also has pre-defined resource allocations, which are modeled as one or more objects of type Resource artifact. Resource is a general object for any finite system allocation object, however in this example we also illustrate a ThreadPool type of Resource.

Semantics The metamodel for service deployment consists of a series of related elements, relationships and a number of constraints. For each element we list each profile metaclass, stereotype and UML metaclass in Table 6.

Table 6. Service Deployment metaclasses and stereotypes

Metaclass	**Stereotype**	**UML Metaclass**	**Description**
ServiceOrchestration	≪ServiceOrchestration≫	Artifact	A reference to a service orchestration process
Servlet	≪Servlet≫	Node	An execution container for orchestration processes
WebServer	≪WebServer≫	Node	A host for servlet containers
Resource	≪Resource≫	Artifact	A type of resource used by a Servlet or Webserver
ThreadPool	≪ThreadPool≫	Resource	A resource collection of threads

8.2 Examples

A deployment model using the Service Deployment profile is illustrated in Fig. 13. Two service orchestrations (RoadAssitance and RoutePlanning) are deployed to a single servlet artifact. The servlet artifact manages a collection of threads in a ThreadPool. The servlet is also hosted by a WebServer. The example can be used to model check that the collaborating service orchestrations, along with the management of thread acquisition and release, is safe and correct.

For a more complete example of using the Service Deployment profile, along with detailed analysis of the model, the reader is invited to refer to [4].

9 Related Work

Several other attempts exist to define UML extensions for service-oriented systems. Most, however, do not cover aspects such structural, behavioral and non-functional aspects of SOAs. For example the UML2 profile for software services [14,17] provides an extension for the specification of services addressing

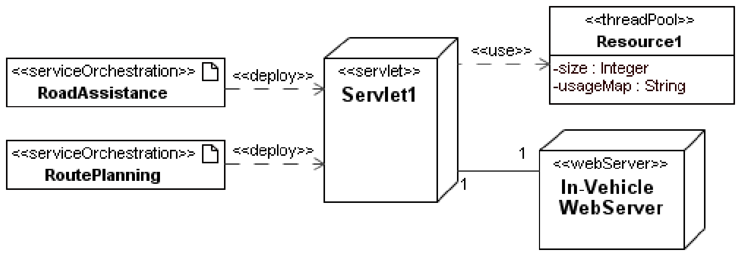


Fig. 13. Example: Deployment model for two service orchestrations and one servlet

only their structural aspects. The UML extension for service-oriented architectures described by Baresi et al. [1] focuses mainly on modeling SOAs by refining business-oriented architectures. The extension is also limited to stereotypes for the structural specification of services. Other modeling approaches require very detailed UML diagrams from designers trying to force service-oriented languages (like BPEL) on top of UML in order to facilitate automated transformation from UML to BPEL [11]. The approach lacks an appropriate UML profile preventing building models at a high level of abstraction; thus producing overloaded diagrams. Some other extensions, conversely to UML4SOA, do not cover vital parts of service orchestrations such as compensation handling, e.g. the UML profile described in [2]. Our UML4SOA approach tries to fill this gap providing a UML profile for service orchestrations.

The OMG also started an effort to standardize a UML profile and metamodel for services (SoaML) [25]. The current beta version focus on structural aspects of services, such as service components, service specifications, service interfaces and contracts for services. We see our family of UML profiles as a complementary set to the profile SoaML.

With respect to business policies, we have already mentioned that several of the stereotypes introduced here bear some relationships to SoaML ones. For instance, a ≪wfSpecification≫ is a ≪capability≫, which can ≪use≫ only (the capabilities offered by) other ≪workflow≫'s and ≪task≫'s. Similarly, a ≪taskSpecification≫ is also a ≪capability≫ , whose ≪contract≫'s can only span the space defined by the ≪NFDimension≫'s indentified via the ≪dim≫ dependencies. In either cases, the ≪serviceInterface≫ is a simple SoaML ≪serviceInterface≫, i.e. a plain UML interface. Finally, the ≪partecipant≫'s that implement these capabilities can be actually invoked only if they fulfill the current contract, as idenfified by the policies in place. Therefore, from the business policies perspective, the Business Policies profile could be seen as a specialization of SoaML, to address the concerns of a large share of the stakeholders, explicitely.

A few words are needed in relation to another widely known standard specification, namely Web Services Policy [28]. In fact, this is a machine-readable language to represent the capabilities and requirements, the *policies*, of a Web service. As such, the standard addresses low level issues, related to the automation of service selection, and will help in the implementation of StPowla .

As for non-functional properties, the UML Profile for QoS and Fault Tolerance [23] and UML Profile for Schedulability and Time [12] were considered during the development of UML4SOA-NFP. As our profile is general purpose (i.e., not bound to any specific aspect like security or performance), it can be extended to describe typical patterns for SLAs which is an ongoing work.

What is generally missing from the existing profile approaches is the ability to identify the requirements and capabilities of services and then to elaborate on the dynamic changes anticipated for adaptation or self-management. For the design of service compositions the dynamic composition of services has largely focused on planning techniques, such as in [26,21], generally with the specification of a guiding policy with some goals of service state. Runtime service brokering also plays an important role in being able to adapt component configurations [22] between requesters and providers yet there is little detail on providing analysis of requirements for brokering. *Software Architecture Modes* were perhaps first introduced in [13], in which they identify a mode as an abstraction of a specific set of services that must interact for the completion of a specific subsystem task. Hirsch's introduction to modes included architectural configuration but did not elaborate on component behavioral change as part of mode adaptation. Consequently, the concept of mode architectures has been extended with behavioral adaptation in [16], focusing on modes as behavioral specifications relating to architecture specification albeit indirectly. We provide a UML profile for service modes.

10 Conclusions

As service-oriented computing continues to gain support in the area of enterprise software development, approaches for handling SOA artefacts and their integration on a high level of abstraction while keeping a semantic link to their implementation become imperative. In this paper, we have focused on a UML-based domain specific modeling language for the specification of service-oriented software. Such a modeling language is the basis for the definition and use of model transformers to generate code in executable target SOA languages like BPEL and WSDL, in a model-driven development process.

Our main contribution are a set of UML profiles for modeling of services that comprise modeling of service orchestration, business policies and non-functional properties of services, service modes for adaptive service brokering and service deployment. Each profile provides a small set of model elements that allow the service engineer to produce diagrams which visualize services and their functionality in a simple fashion.

These are profiles for separate purposes, which share some basic concepts (e.g. service, participant, etc.). It is the service engineer who decides which profiles to use as they cover different steps of the development lifecycle, e.g. QoS parameters bound to an *SLA* could be transformed to the input of *Modes* while they can be also used in *St-Powla*. The policy support profile depends on UML4SOA-NFP, insofar as it imports *NFDimension* to characterize the QoS of the services subjected to the policies.

Further details on the profiles and tools discussed in this paper are available on the Sensoria project website [27].

References

1. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-Based Modeling and Refinement of Service-Oriented Architectures. *Journal of Software and Systems Modeling (SOSYM)*, 5(2):187–200, 2005.
2. V. Ermagan and I. Krüger. A UML2 Profile for Service Modeling. In *International Conference on Model Driven Engineering Languages and Systems*, volume LNCS 4735 of *IEEE*, pages 360–374. Springer-Verlag, 2007.
3. H. Foster. Architecture and Behaviour Analysis for Engineering Service Modes. In *Proceedings of the 2nd Workshop on Principles of Engineering Service Oriented Systems (PESOS09)*, Vancouver, Canada, 2009.
4. H. Foster, W. Emmerich, J. Kramer, J. Magee, D. Rosenblum, and S. Uchitel. Model Checking Service Compositions under Resource Constraints. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the foundations of Software Engineering*, pages 225–234, New York, NY, USA, 2007. ACM.
5. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Leveraging Modes and UML2 for Service Brokering Specifications. In *Proceedings of the 4th Model-Driven Web Engineering Workshop (MDWE 2008)*, Toulouse, France, 2008.
6. S. Gilmore, L. Gönczy, N. Koch, P. Mayer, and D. Varró. Non-Functional Properties in the Model-Driven Development of Service-Oriented Systems. *Journal of Software and Systems Modeling*, 2010. Accepted for publication.
7. L. Gönczy, Z. Déri, and D. Varró. Model Driven Performability Analysis of Service Configurations with Reliable Messaging. In *Proc. of Model Driven Web Engineering Workshop (MDWE 2008)*, 2008.
8. L. Gönczy, Z. Déri, and D. Varró. Model Transformations for Performability Analysis of Service Configurations. pages 153–166, Berlin, Heidelberg, 2009. Springer-Verlag.
9. L. Gönczy and D. Varró. *Developing Effective Service Oriented Architectures: Concepts and Applications in Service Level Agreements, Quality of Service and Reliability*, chapter Engineering Service Oriented Applications with Reliability and Security Requirements. IGI Global, 2010. To be published.
10. S. Gorton, C. Montangero, S. Reiff-Marganiec, and L. Semini. StPowla: SOA, Policies and Workflows. In E. D. Nitto and M. Ripeanu, editors, *ICSOC'07 Workshops Revised Selected Papers*, volume 4907 of *Lecture Notes in Computer Science*, pages 351–362. Springer, 2009.
11. R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Style-Based Modeling and Refinement of Service-Oriented Architectures. In *Eighth IEEE International Enterprise Distributed Object Computing Conference (EDOC'04)*, IEEE, pages 47–57. IEEE, 2004.
12. O. M. Group. *UML Profile for Schedulability, Performance and Time Specification*, 2005. http://www.omg.org/technology/documents/formal/schedulability.htm.

13. D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for Software Architectures. In *Proceedings of EWSA 2006, 3rd European Workshop on Software Architecture*, Lecture Notes in Computer Science. Springer Verlag, 2006.
14. S. Johnston. UML 2.0 Profile for Software Services, available at http://www-128.ibm.com/developerworks/rational/library/05/419soa. *Request For Proposal - AD/02-01/07*, 2005.
15. N. Koch and D. Berndl. Requirements Modelling and Analysis of Selected Scenarios: Automotive CASE Study. Technical Report D8.2a, SENSORIA Deliverable, 2007.
16. J. Kofroň, F. Plášil, and O. Šerý. Modes in Component Behavior Specification via EBP and their application in Product Lines. *Information and Software Technology*, 51(1):31–41, 2009.
17. R. J. Machado, J. M. Fernandes, P. Monteiro, and H. Rodrigues. Transformation of UML Models for Service-Oriented Software Architectures. In *In Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, Washington, DC, USA*, pages 173–182, 2005.
18. P. Mayer, N. Koch, and A. Schroeder. The UML4SOA Profile. Technical report, Ludwig-Maximilians-Universität München, July 2009.
19. P. Mayer, A. Schroeder, and N. Koch. A Model-Driven Approach to Service Orchestration. In *Proceedings of the 2008 IEEE International Conference on Services Computing (SCC 2008)*, volume 2, pages 533–536, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
20. P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *The 12th IEEE International EDOC Conference (EDOC 2008)*, pages 203–212, Munich, Germany, 2008. IEEE Computer Society.
21. B. Medjahed, A. Bouguettaya, and A. Elmagarmid. Composing Web Services on the Semantic Web. *VLDB Journal*, pages 333–351, 2003.
22. A. Mukhija, A. Dingwall-Smith, and D. S. Rosenblum. QoS-Aware Service Composition in Dino. In *ECOWS '07: Proceedings of the Fifth European Conference on Web Services*, pages 3–12, Halle, Germany, 2007. IEEE Computer Society.
23. OMG. *UML for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, v1.1* , 2008. `http://www.omg.org/spec/QFTP/1.1/`.
24. OMG. Unified Modeling Language: Superstructure, version 2.2. Technical Report formal/2009-02-02, Object Management Group, 2009.
25. OMG. Service oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services (UPMS), revised submission. Specification, Object Management Group, 2010. `http://www.omg.org/spec/SoaML/1.0/Beta2/`, Last visited: 22.07.2010.
26. M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.
27. SENSORIA. Software Engineering for Service-Oriented Overlay Computers. http://www.sensoria-ist.eu/, last visited 15.03.2010.
28. W3C Working Group. Web Services Policy 1.5 - Primer. `http://www.w3.org/TR/ws-policy-primer/`. Last visit 22.10.2009.
29. R. Xie and N. Koch. Automotive CASE Study: Demonstrator (Tutorial). Technical report, Cirquent GmbH, 2009. http://www.sensoria-ist.eu/, last visited 15.03.2010.