SPECIFYING SERVICES USING THE SERVICE ORIENTED ARCHITECTURE MODELING LANGUAGE (SOAML)

A Baseline for Specification of Cloud-based Services

Brian Elvesæter, Arne-Jørgen Berre
SINTEF ICT, P. O. Box 124 Blindern, N-0314 Oslo, Norway

Andrey Sadovykh
SOFTEAM, 21 avenue Victor Hugo, 75016 Paris, France

Keywords: Service specification, Service oriented architecture, Service modelling, SoaML.

Abstract: The Service oriented architecture Modeling Language (SoaML) is a new specification from the Object Management Group (OMG) that provides support for modelling services. The SoaML specification defines three different approaches to specifying services; simple interfaces, service interfaces and service contracts. In this paper we provide an overview of the SoaML language constructs and discuss the three different ways to specify services. Furthermore, we provide practical modelling guidelines for how the different SoaML service specification approaches can be aligned and used as a baseline for specifying cloud-based services.

1 INTRODUCTION

The Service oriented architecture Modeling Language (SoaML) specification (OMG, 2009) defines a UML profile and a metamodel for the design of services within a service-oriented architecture. The goals of SoaML are to support the activities of service modelling and design and to fit into an overall model-driven development approach, supporting SOA from both a business and an IT perspective.

The SoaML specification defines three different approaches to specifying services; simple interfaces, service interfaces and service contracts. The different approaches prescribe using different parts of UML, and understanding how these relate is not obvious from reading the specification. Due to this we have seen some confusion amongst software engineering practitioners trying to apply SoaML.

In this paper we provide an overview of the SoaML language and discuss the different ways to specifying services. Furthermore, we provide practical modelling guidelines for how the different SoaML service specification approaches can be aligned. The guidelines are based on our experience from developing SoaML modelling tools and methods, and proof-of-concept implementations in industrial case studies (Stollberg et al., 2010).

The paper is structured as follows: In Section 2 we give an overview of the SoaML language. Section 3 presents an illustrative example and describes and discusses the different approaches to service specification. In Section 4 we provide some practical guidelines for how to align these different SoaML specification approaches. In Section 5 we discuss methodology issues and related work. Finally, Section 6 concludes this paper.

2 OVERVIEW OF THE SOAML LANGUAGE CONSTRUCTS

The SoaML specification defines a UML profile and a metamodel that extends UML to support the range of modelling requirements for SOA, including the specification of systems of services, the specification of individual service interfaces, and the specification of service implementations. The SoaML metamodel extends the UML metamodel to support an explicit service modelling in distributed environments. This extension aims to support different service modelling scenarios such as single service

Elvesæter B., Berre A. and Sadovykh A..
SPECIFYING SERVICES USING THE SERVICE ORIENTED ARCHITECTURE MODELING LANGUAGE (SOAML) - A baseline for Specification of Cloud-based Services.
DOI: 10.5220/0003393202760285
In *Proceedings of the 1st International Conference on Cloud Computing and Services Science* (CLOSER-2011), pages 276-285
ISBN: 978-989-8425-52-2

description, service-oriented architecture modelling, or service contract definition. This is done in such a way as to support the automatic generation of derived artefacts following the approach of Model Driven Architecture (MDA) (OMG, 2003).

UML is a general-purpose modelling language for visualising, specifying, constructing and documenting artefacts of software-intensive systems. A UML profile customizes UML for a specific domain or purpose by using extension mechanisms such as stereotypes and metaclasses. Figure 1 shows the main stereotypes defined in the UML profile for SoaML, e.g. the stereotype «ServiceInterface» extends the UML metaclass *Class*.

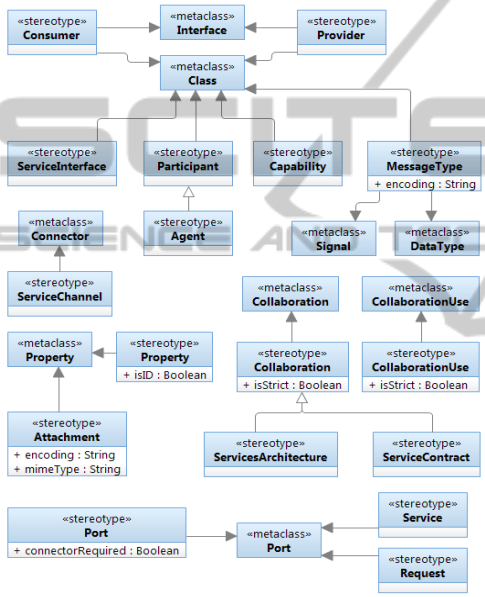

Figure 1: Main UML extensions defined as stereotypes in the UML Profile for SoaML (no relationships shown).

SoaML extends UML in six main areas: Participants, service interfaces, service contracts, services architectures, service data and capabilities.

Participants are used to define the service providers and consumers in a system. A participant may play the role of service provider, consumer or both. When a participant acts as a provider it contains *service ports*, and when a participant acts as a consumer it contains *request ports*.

Service interfaces are used to describe the operations provided and required to complete the functionality of a service. A service interface can be used as the protocol for a service port or a request port.

Service contracts are used to describe interaction patterns between service entities. A service contract is used to model an agreement between two or more parties. Each service role in a service contract has an interface that usually represents a *provider* or a *consumer*.

Services architectures are used to define how a set of participants works together for some purpose by providing and using services. The services are expressed as service contracts in a services architecture.

Service data are used to describe service messages and message attachments. The *message type* is used to specify the information exchanged between service consumers and providers. An *attachment* is a part of a message that is attached to rather than contained in the message.

Capabilities represent an abstraction of the ability to affect change. Capabilities identify or specify a cohesive set of functions or resources that a service provided by one or more participants might offer.

3 APPROACHES TO SPECIFYING SERVICES USING SOAML

SoaML supports different approaches to SOA. This has resulted in the definition of different but overlapping language constructs in the UML profile. The specification distinguishes between three different approaches to specifying a service:

- The **simple interface** based approach uses a UML interface to specify a one-way service interaction.
- The **service contract** based approach extends a UML collaboration to specify a binary or n-ary service interaction.
- The **service interface** based approach extends a UML class to specify a binary or n-ary service interaction.

Both the service contract and service interface based approaches entail the specification of simple interfaces, typically one for each of the roles participating in the service interaction. Thus a service contract or a service interface can be seen as an extension of the simple interface based approach.

The following subsections introduce an illustrative example and describe and discuss the three approaches in light of the example in more details.

3.1 Illustrative Example

In this paper we have adopted the Dealer Network Architecture example from the SoaML specification. The example in the specification is somewhat

difficult to read because of the different approaches to service specifications. On closer inspection of the example in the SoaML specification, we found out that there were in fact two different variants of the example; one favouring the service contract based approach and the other favouring the service interface based approach. These two variants are slightly different and have been modelled using two different modelling tools, thus the reader is faced with a somewhat unclear and inconsistent running example throughout the specification. A full example favouring the service contract based approached is documented in (Casanave, 2009) and a full example favouring the service interface based approach is documented in (Amsden, 2010). Both of these authors were heavily involved in the SoaML specification process. Furthermore, we also found out that some UML tools do not properly support UML collaboration, resulting in poor support for the SoaML service contract based approach.

In this paper we merged and unified the two different variants of the example used in the specification and modelled the example using the Modelio modelling tool (www.modeliosoft.com) that equally supports both the service contract and service interface based approaches of SoaML. The example was also extended to show a full integration and benefits of multi-party and compound service contracts in specifying services architectures. Figure 2 shows the services architecture for the unified example.

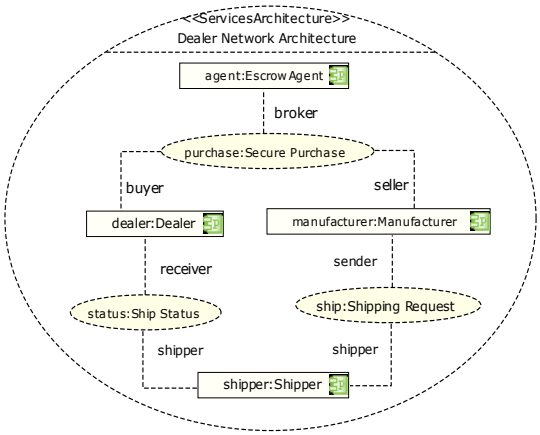

Figure 2: Services architecture for the Dealer Network Architecture illustrative example with four participants and the roles they play in the three service contract specifications.

The Dealer Network Architecture consists of four participants (*dealer*, *manufacturer*, *agent* and *shipper*) interacting and fulfilling their roles defined in the three service contracts: *Secure Purchase* (specifying the roles *buyer*, *seller* and *broker*), Shipping Request (specifying the roles *sender* and *shipper*) and *Shipping Status* (specifying the roles *receiver* and *shipper*). In the services architecture the participants are bounded to the roles defined in the service contracts through the collaboration uses *purchase* (instance of *Secure Purchase*), *ship* (instance of *Shipping Request*) and *status* (instance of the *Ship Status*). Details of these service contracts will be described in the subsequent sections.

3.2 Simple Interface based Approach

The **simple interface based approach** focuses attention on a one-way interaction provided by a participant on a port represented as a UML interface. The participant receives operations on this port and may provide results to the caller. This approach can be used with "anonymous" callers and the participant makes no assumptions about the caller or the choreography of the service.

In the Dealer Network example there are three services identified (as service contracts). Some of these may in fact be simple one-way interactions and could thus be modelled using the simple interface based approach. Let us consider the *Ship Status* service as a simple one-way interaction. Figure 3 shows the specification of this service, consisting of the provider interface *ShippingStatus* (modelled as a «Provider» UML interface) and the two message types *ShipmentStatusRequest* and *ShipmentStatus* (modelled as «MessageType» UML classes). The message types represent the types of the input parameter and the return type of the operation *queryShippingStatus* defined in the provider interface.

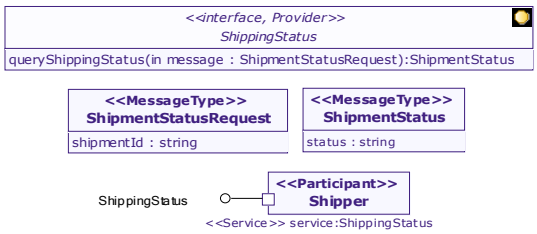

Figure 3: Specification of the *Ship Status* service using the simple interface based approach, consisting of a simple provider interface, its operations and message types, and the corresponding port on a participant.

According to the services architecture *Dealer Network Architecture* the participant *Shipper* must realize the *shipper* role, which means to provide the provider interface through a service port. Thus, to complete the specification, we add the service port *service* (modelled as a «Service» UML port) to the participant *Shipper* (modelled as a «Participant»

UML class or a «Participant» UML component (a subclass of UML class)). The service port is typed with the provider interface *ShippingStatus* and exposes its provided interface.

The simple interface based approach can be seen as a degenerate case of both the service contract and service interface based approaches, only entailing one provider interface. Figure 4 shows how the simple interface can be specified as a service contract *Ship Status* (modelled as a «ServiceContract» UML collaboration). Note that a contract involves at least two roles, e.g. consumer and provider. In this case we use the role names *receiver* and *shipper* instead of the generic consumer and provider role names. Since we model a one-way interaction only the provider side, i.e. the *shipper* role, has a type, namely the provider interface *ShippingStatus*. In this case we have only introduced the UML collaboration, all other elements from the simple interface based approach remain unaffected.

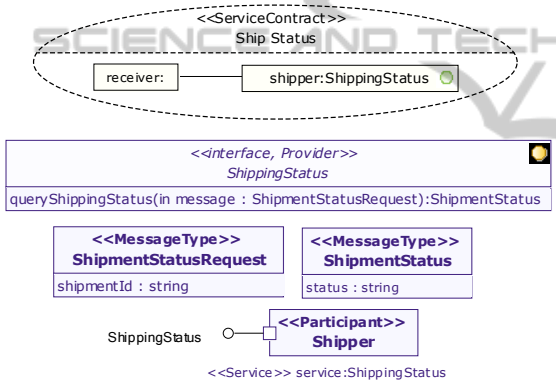

Figure 4: Specification of the *Ship Status* service using the service contract based approach.

Figure 5 shows how the simple interface can be modelled as a service interface. Here the service contract has been replaced by the service interface *ShipStatusService* (modelled as a «ServiceInterface» UML class). The service interface contains a part *shipper* that denotes the role which has the provider interface type *ShippingStatus*. Furthermore, the service interface class also realizes the *ShippingStatus* interface. Note also, that in the case of using the service interface based approach, the corresponding port on the participant *Shipper* has been typed by the service interface *ShipStatusService* instead of the provider interface *ShippingStatus*.

Here we could also have added the *receiver* role to the service interface in order to completely resemble the service contract, but this is not really needed in this case since we do not define any consumer interface type. A service contract however needs to have a generic consumer role to be complete, even though we do not have a consumer interface, to ensure that the roles of the two interacting participants can be linked using role binding in a services architecture.

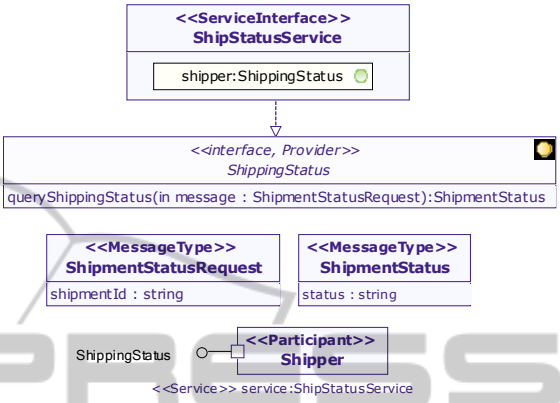

Figure 5: Specification of the *Ship Status* service (here named *ShipStatusService*) using the service interface based approach.

From the examples above we see that modelling simple interfaces as either a service contract or a service interface requires additional modelling effort. Thus, the simple interface based approach is recommended for modelling one-way service interactions, since the addition of a service contract or a service interface is unnecessary. However, if you want to illustrate the particular use of a one-way interaction service in a services architecture, you will need to add a service contract modelled as a UML collaboration. As explained above, this fortunately does not affect any of the other elements already specified using the simple interface based approach.

3.3 Service Contract based Approach

A **service contract based approach** defines service specifications that define the roles each participant plays in the service (such as provider and consumer) and the interfaces they implement to play that role in that service. These interfaces are then the types of ports on the participant, which obligates the participant to be able to play that role in that service contract.

The service contract based approach extends a UML collaboration to model the structural part of the service interaction. The approach can be used to specify services in which there is a contractual obligation, i.e. an agreement, between two or more parties. This is the case where you have an

interaction pattern that involves an exchange of messages which specify (simple) interfaces on both sides.

Let us now consider the *Secure Purchase* service contract from the *Dealer Network* and demonstrate the use of the service contract based approach to define a binary service contract, a multi-party service contract and a compound service contract. First assume that the *Secure Purchase* service contract can be modelled as two separate service contracts, one specifying the order interaction and the other specifying the purchase interaction.

Figure 6 shows the specification of the *Place Order* service contract, with the two roles *consumer* and *provider* and their respective *OrderPlacer* and *OrderTaker* consumer and provider interface types. The service contract states that there is a dependency between these two interfaces and this must also be modelled explicitly using UML dependencies. The participants interacting in this service contract fulfil their roles by realizing the corresponding interfaces and expose them through ports. From the role bindings in the services architecture we deduce that the Dealer has a request port *Request* typed by the *OrderPlacer* interface, and the *Manufacturer* has a service port *Service* typed by the *OrderTaker* interface. Notice that we have not fully specified the operation signatures and message types for the interfaces. We will come back to this in Section 4.1 of this paper.

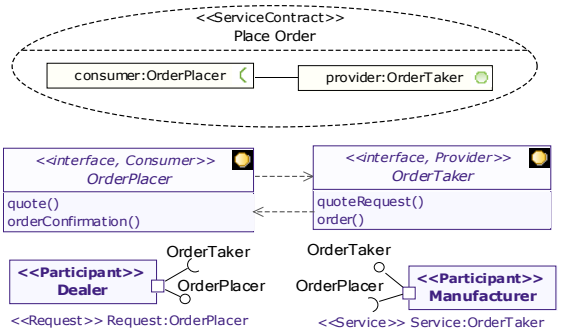

Figure 6: Specification of the *Place Order* service, consisting of two roles, their respective consumer and provider interface type, and the corresponding ports on the participants.

In the example above the service contract acts as a packaging of the two interfaces, ensuring that both interfaces are a part of **one** service specification and not specified as **two** independent service specifications as two separate simple interfaces. In addition to this structural specification it is recommended to specify the behaviour of the service contract, i.e. the service choreography or service protocol. In fact, one can argue that the specification of the service choreography is essential in order to understand how to design the interfaces to support the exchange of messages. SoaML is agnostic with regards to behavioural modelling and basically states that any UML behaviour, e.g. interaction models, activity models or state machines, can be used.

Figure 7 shows the specification of the service choreography using a UML interaction. Here we see that we specify a conversation, i.e. message exchange, between the two participants, and this requires an interface to be implemented at both sides. In this case the simple interface based approach falls short as it is not able to capture this as one single service specification.

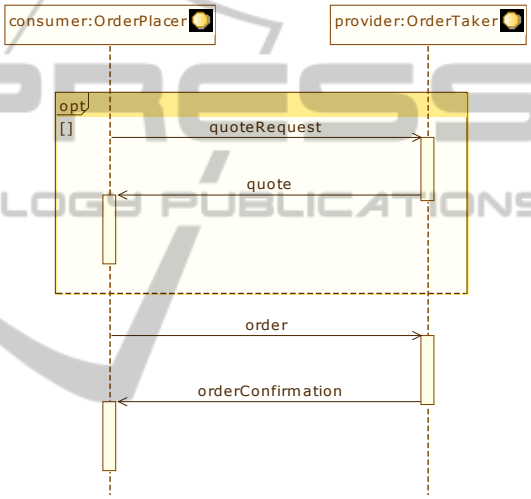

Figure 7: Specification of the *Place Order* service choreography (behaviour).

The service contract based approach is useful when specifying interactions between two or more roles that implies the establishment of some agreement e.g. through message exchanges. A service contract also serves as a reusable specification element that can be re-used at design time to connect different participants. Furthermore, the approach also supports modelling of multiparty service contracts involving three or more participants and compound service contracts where existing service contracts can be used to define more granular service contracts.

Let us first look at a multiparty service contract. Our example uses an Escrow purchase, where the interaction between a buyer and a seller is mediated through an Escrow broker. Figure 8 shows the specification of the *Escrow Purchase* service contract, with the three roles *buyer*, *seller* and *broker* and their respective *Purchaser*, *Seller* and *EscrowAgent* consumer and provider interface types.

The dependencies between the interfaces are explicitly modelled using UML dependencies and the participants have ports corresponding to the role bindings in the services architecture. As can be seen the ports on the *Dealer, Manufacturer* and *EscrowAgent* participants each provides one and requires two interfaces in order to comply with the dependencies between the interfaces.

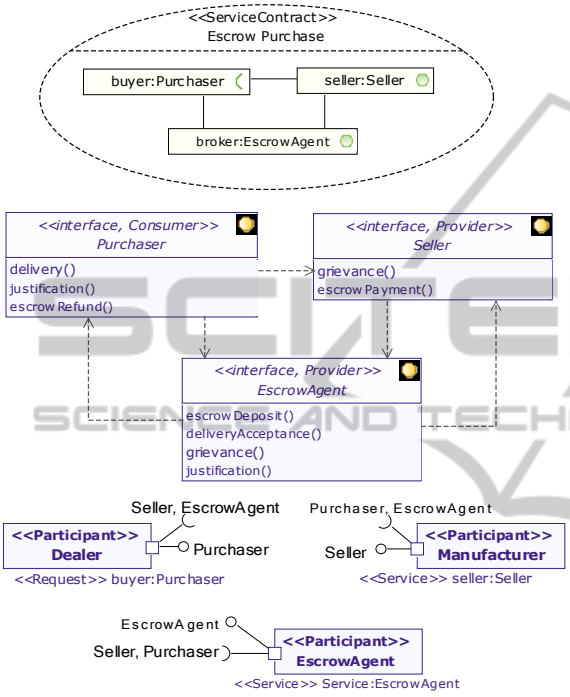

Figure 8: Specification of the *Escrow Purchase* service, consisting of three roles, the respective consumer interface and the two provider interface types, and the corresponding ports on the participants.

Figure 9 shows the specification of the service choreography using a UML interaction. Note that this is a multiparty service contract because the *buyer* also interacts with the *seller* directly through the *delivery* message. With the exception of the delivery, all other interactions are mediated through an Escrow broker. The service interaction starts with a deposit made by the *buyer* to *broker*. At a later time a delivery is made and either accepted or grievance is sent to the *broker* who forwards it to the *seller*, which may file a justification in order to clarify whether to accept or refund the payment. This process repeats until the broker concludes the transaction and either makes the *escrowPayment* to the *seller* or *escrowRefund* to the *buyer*.

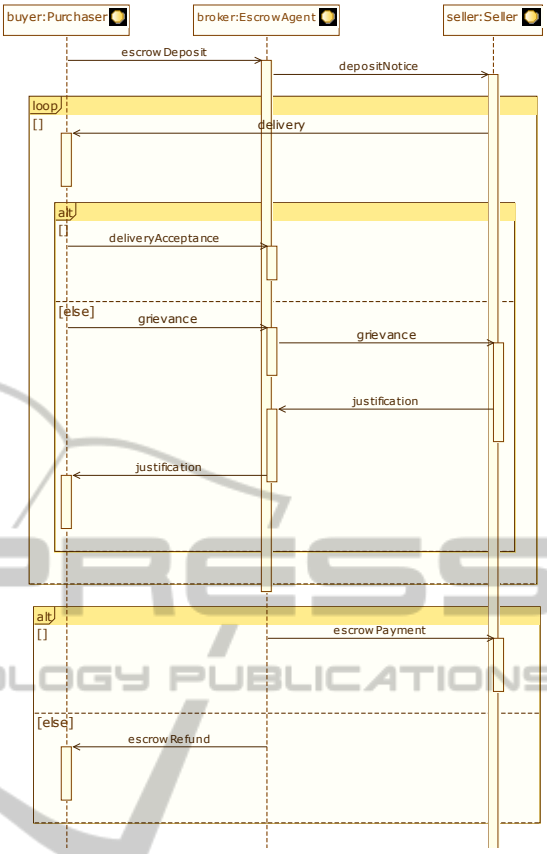

Figure 9: Specification of the Escrow Purchase service choreography (behaviour).

Let us now look at a compound service contract. Note that a compound service contract should not be confused with a service that is implemented by calling other services. A compound service contract defines a more granular service contract based on other service contracts. Figure 10 shows the specification of the *Secure Purchase* compound service contract, which combines the *Place Order* and *Escrow Purchase* service contracts. In the case of compound service contracts, the SoaML specification prescribes that the types of the roles should be modelled as classes instead of interfaces. Moreover, according to the SoaML specification they can in fact be of type service interfaces which are explained in Section 3.4. In this example we use UML classes stereotyped as either «Consumer» or «Provider». Note that the *Buyer* and *Seller* have two ports, each corresponding to the role played in the *Place Order* and *Escrow Purchase* services. When a compound service is used it looks no different than any other service in a services architecture, thus hiding the detail of the more granular service in the high-level architecture yet providing traceability through all levels.

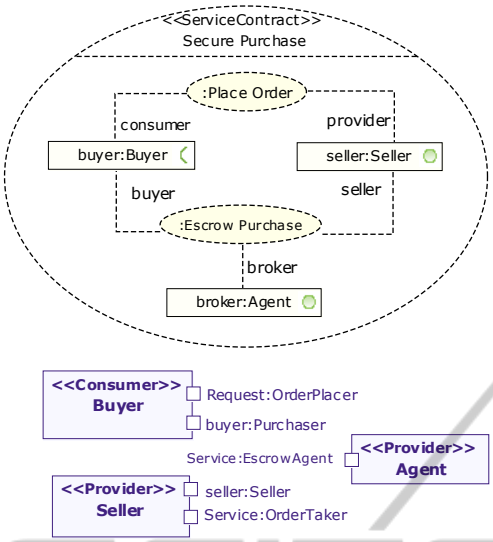

Figure 10: Specification of the *Secure Purchase* compound service contract which combines the two-party *Place Order* and the multi-party *Escrow Purchase* service contracts.

3.4 Service Interface based Approach

The **service interface based approach** is quite similar to the service contract based approach in that it also focuses on binary and n-ary service interactions, requiring us to specify a set of related (simple) interfaces as **one** service specification. Whereas the service contract based approach prescribes using UML collaboration, the service interface based approach focuses on UML components and allows the interconnection between these components through ports. In order to connect components through ports, the ports must specify both required and provided interfaces.

The service interface based approach introduces the concept of a **service interface** and a **conjugate service interface** to type the ports on the provider and consumer side respectively. Let us show this through a typical bidirectional service interaction, where a "callback" from the provider to the consumer is specified as part of a conversation between the participants. Figure 11 shows the specification of the *Place Order* service previously modelled as a service contract using the service interface based approach instead. We name the service interface *PlaceOrderInterface*, which specifies the roles *consumer* and *provider* with their corresponding interface types *OrderPlacer* and *OrderTaker* respectively as before. The service interface realizes the provider interface *OrderTaker* and uses the consumer interface *OrderPlacer*. A conjugate service interface denoted with the ~ (tilde) prefix is used to type the port on the participant playing the consumer role. The conjugate service interface reverses the realization and uses associations, i.e. it realizes the consumer interface *OrderPlacer* and uses the provider interface *OrderTaker*.

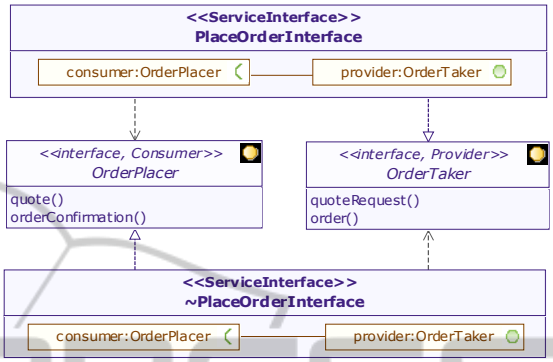

Figure 11: Specification of the *Place Order* service using the service interface based approach.

Figure 12 shows how the participants fulfilling the consumer and provider roles can be connected through their ports. The *Dealer* participant has a request port typed by the conjugate service interface *~PlaceOrderInterface*, whereas the *Manufacturer* participant has a service port typed by the service interface *PlaceOrderInterface*. Both ports are compatible with regards to the required and provided interfaces, which thus can be connected. This allows us to specify composite structures in UML where we can connect the ports through service channels, e.g. as illustrated with the *Dealer Network Architecture* in Figure 12. Graphically, the interconnection between these ports can now be simplified by only showing the connector and mask out any required and provided interfaces which only tend to clutter the diagram.

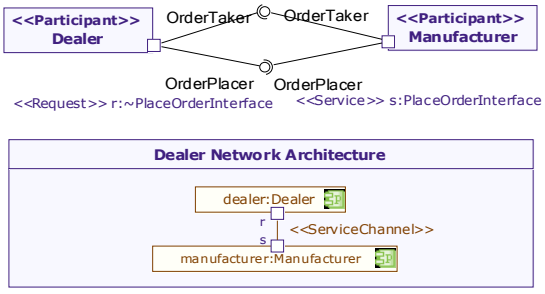

Figure 12: Consumer and provider participants connected through their request and service ports.

The service interface based approach adds further details in order to align with UML component modelling, which allows components to be

composed and connected through ports. The addition of the conjugate service interface type arguably adds extra complexity, but it ensures syntactical correctness in the model when connecting the ports. Furthermore, the **conjugate service interface** type could be automatically created in a SoaML modelling tool based on the modelled **service interface** type.

4 ALIGNING THE SERVICE SPECIFICATION APPROACHES

Although the three different service specification approaches of SoaML are different they are still somehow intertwined with respect to the fact that simple interfaces are structural parts in both the service contract and service interface based approaches. The simple interface can be viewed as degenerate cases of both these approaches as explained in Section 3.2 of this paper.

Basically, we see two main approaches, the service contract based approach that extends UML collaboration and the service interface based approach that extends UML class but focused on components and composite structures connected through ports. Reading the SoaML specification it is unclear how to use and combine these approaches. Furthermore, combining them may also lead to duplicate modelling effort, in particular with respect to the specification of the behavioural parts, i.e. the service choreographies.

Our experience with SoaML modelling in different project settings suggests that there are two main ways of aligning the service contract and service interface based approaches, namely through refinements or through views. We have developed a methodology that provides SoaML modelling guidelines (Elvesæter et al., 2011) that includes the two approaches, but favours the use of refinements when starting from a top-down modelling approach. Which approach to choose though, depends on the complexity, size and technical focus of the modelling scope at hand, and they may in fact also be combined.

4.1 Aligning Service Contracts and Service Interfaces as Refinements

UML collaborations are often regarded as more appropriate for modelling high-level architectures rather than detailed design. Thus, one way of combining the service contracts and service interfaces is to regard service interfaces as refinements of service contracts. We have found this approach appropriate to describe business-level architectures using service contracts and system-level architectures using service interfaces. Figure 13 and Figure 14 illustrate this approach.

In Figure 13 we have the specification of "high-level" or "business-level" consumer and provider interfaces for the *Place Order* service contract as explained earlier in this paper. In Figure 14 we refine these interfaces and add more details with respect to IT service implementation. Notice that this approach can be supported by a one-to-one mapping from a service contract and its interfaces to a service interface and its interfaces. In the service interface and its corresponding interfaces you may refine the models towards more IT-level system specifications.

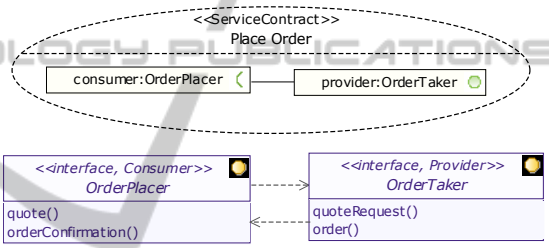

Figure 13: Service contracts models the business interfaces.

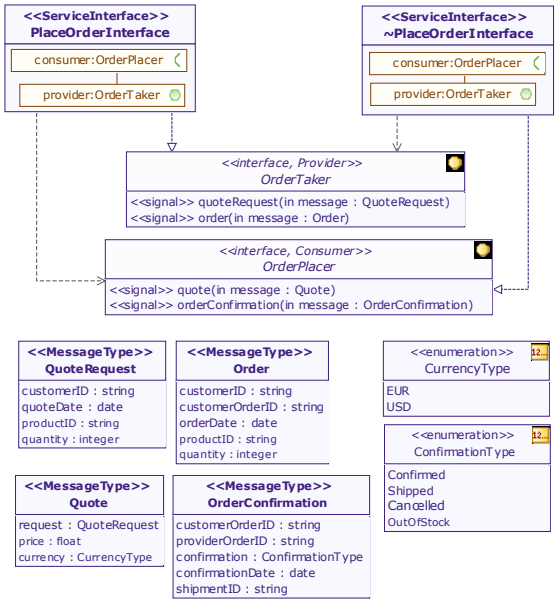

Figure 14: Service interface refines the service contract.

4.2 Aligning Service Contracts and Service Interfaces as Views

Another way of aligning the service contract and service interface based approaches is to consider these as two different views on the same model, both existing on the same abstraction level. Figure 15 illustrates this approach. Here the consumer and provider interfaces types defined are the exact same model elements used to type the roles in both the service contract and the service interface. Thus, the service contract and the service interface can be seen as two different notations for the exact same thing and one can argue that this duplicates some modelling effort and should be avoided. However, as the SoaML specification prescribes modelling services architectures using service contracts, you are required to specify service contracts if you also want to specify a services architecture. Or to put it in another way, it is only necessary to model the service contracts views that are needed for the services architectures that you have specified.

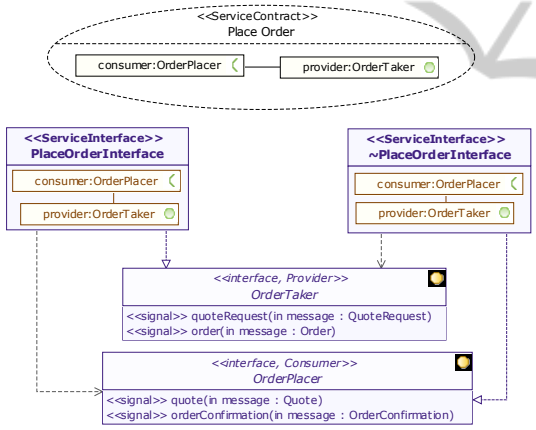

Figure 15: Service interfaces and service contracts are two different views.

Note that, in this alignment approach it is recommended to only specify behaviour in either the service contract or the service interface, thus avoiding some duplicate modelling effort for the behaviour part. In the approach using refinements, the service choreography (behaviour) of a service interface would be a behavioural refinement of the service choreography of a service contract.

5 METHODOLOGY DISCUSSION AND RELATED WORK

Cloud computing and SOA are recognized game-changing technologies for a cost-efficient and reliable service delivery. Software as a Service (SaaS) paradigm becomes more and more popular enabling flexible license payment schemas and moving the infrastructure management costs from consumers to service providers. However, building a SaaS system from scratch may require a huge investment in time and efforts. Moreover, the organizations' legacy systems are difficult to reuse due to platform, documentation and architecture obsolescence.

The Model Driven Architecture (MDA) (OMG, 2003) and related efforts around domain-specific languages have gained much popularity. These technologies put the model in the centre of the software engineering process. The software products are built with subsequent model refinements and transformations from business models (process, rules, motivation), down to component architectures (e.g. SOA), detailed platform specific design and finally implementation.

The SoaML modelling language, as presented in this paper, was designed to support the MDA and as such provides a baseline modelling language for the specification of any services within a service-oriented environment, which included cloud-based services. The SoaML language itself can be said to be quite small, as it only specifies around twenty main extensions to UML. These extensions provide the key language constructs for specifying the structure of services. As explained earlier, SoaML do not specify which kind of behavioural notation to use. The goal of SoaML was not to be a fully-fletched modelling language supporting all aspects of service-oriented architectures, but rather to be a small core that can be extended and integrated with other modelling languages, e.g. BPMN for behavioural modelling. As such, the SoaML language should be regarded by software engineers and researchers looking into cloud-based services in as a baseline or starting point for which to define other language extensions required in cloud computing.

We see evidence that SoaML is being supported by UML tool vendors and incorporated as part of their service-oriented methodologies. In particular, IBM has incorporated SoaML in their Service-Oriented Modeling and Architecture (SOMA) methodology which is supported by their Rational Software Architect (RSA) modelling tool (Amsden, 2010). Other vendors that provide SoaML support are ModelDriven.org, NoMagic, SOFTEAM and Sparx Systems (see www.soaml.org). In addition, SoaML-based tools and methods for model-driven

engineering of service-oriented landscapes were developed in the European research project SHAPE (Stollberg et al., 2010).

The current tools and methodologies using SoaML focus mainly on supporting the MDA approach, which emphasises models as the essential artefacts. Similarly, the Architecture-Driven Modernization (ADM) (see adm.omg.org) proposes to start with knowledge discovery to recover models and to re-build the new system in a forward MDA process. The ADM initiative may be another starting point in order to support migration of legacy systems into the cloud. The European research project REMICS (www.remics.eu), in which the authors participate, aims to develop a complete process including methods and tools for creating SoaML models from the legacy artefacts and re-building cloud-based systems by applying SOA and cloud patterns.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have presented an overview of the SoaML modelling language, and discussed the different SoaML approaches to specifying services. Our discussion has tried to clarify the differences and similarities between the different SoaML approaches in order to make SoaML more comprehensible to software engineering practitioners. Furthermore, we have presented a set of practical modelling guidelines for how to align the different approaches to specifying services using SoaML. These guidelines can be adopted by methodologists that want to include SoaML as part of their own service engineering method. Finally, we argue that SoaML could be used a baseline for specification of cloud-based services. SoaML is a language that can be extended with new modelling constructs and integrated with other modelling languages, to support a richer model-driven approach to specifying cloud-based services.

The results presented in this paper are based on experience from research and development of SoaML tools and methods that have been evaluated in proof-of-concept implementations in industrial case studies. One aspect of our guidelines that requires further work is better advice for behavioural modelling. SoaML is quite open with regards to behavioural modelling, and we are currently investigating the use of BPMN 2.0 (OMG, 2011) as an extension to SoaML for this purpose in the European research project NEFFICS (www.neffics.eu). Furthermore, we are researching how SoaML can be applied in model-based migration approaches where legacy systems are modernized and migrated to new service-oriented and cloud-based platforms. In fact, our future work in the research project REMICS (www.remics.eu) involves the specification of the SoaML4Cloud language, which will extend SoaML to support cloud-based services.

ACKNOWLEDGEMENTS

The origin of this research was co-funded by the European Union in the frame of the SHAPE project (FP7-ICT-216408) (www.shape-project.eu). The research is being continued in the frame of the NEFFICS project (FP7-+ICT-258076) (www.neffics.eu), the REMICS project (FP7-ICT-257793) (www.remics.eu) and the SINTEF project SiSaS (sisas.modelbased.net).

REFERENCES

Amsden, J., 2010, *Modeling with SoaML, the Service-Oriented Architecture Modeling Language*, Technical article series, IBM.

Casanave, C., 2009, *Enterprise Service Oriented Architecture Using the OMG SoaML Standard*, White Paper, Model Driven Solutions, Inc.

Elvesæter, B., Carrez, C., Mohagheghi, P., Berre, A.-J., Johnsen, S. G., Solberg, A., 2011, *Model-driven Service Engineering with SoaML*, in *Service Engineering - European Research Results*, pp. 25-54, Springer.

OMG, 2003, *MDA Guide Version 1.0.1*, Document omg/03-06-01, Object Management Group (OMG).

OMG, 2009, *Service oriented architecture Modeling Language (SoaML), FTF Beta 2*, Document ptc/2009-12-09, Object Management Group (OMG).

OMG, 2011, *Business Process Model and Notation (BPMN), Version 2.0*, Document formal/2011-01-03, Object Management Group (OMG).

Stollberg, M., Benguria, B., Berre, A.-J., Cerri, D., Elvesæter, B., Fischer, K., Hahn, C., Jacobi, S., Landre, E., Panfilenko, D., Sadovykh, A., 2010, *SHAPE Whitepaper*, SHAPE Collaborative Project.