PARIS NORD UNIVERSITY - SORBONNE PARIS CITÉ
Laboratoire d'Informatique de Paris 13, CNRS UMR 7030,
École Doctorale Galilée

PHD THESIS

to obtain the title of

PhD of Science of Université Paris 13
Specialty: Informatique

Khanh Dung TRAN

Rigorous and formal approaches for modelling and designing service systems

Thesis Director
Prof. Christine CHOPPY

Defended on September 20, 2013

JURY

Reviewers:	
Prof. Nicole Levy	- CNAM, Paris, France
Prof. Pascal Poizat	- Université Paris Ouest Nanterre la Défense, France
Examinators:	
Prof. Christophe Fouquere	- LIPN, Université Paris Nord, France
Adviser:	
Prof. Gianna Reggio	- DIBRIS, Università di Genova, Italy
Director:	
Prof. Christine Choppy	- LIPN, Université Paris Nord, France

Acknowledgements

Above all, I would like to express my deepest gratitude to my thesis director Christine CHOPPY, who never accepts me less than my best efforts. I would like to thank for her guidance in last three years, not only in my studying but also in my daily life. I owe a great debt of gratitude for her help and support.

I would like to express my deep and sincere thanks to my adviser Gianna REGGIO, who has not only served as my adviser but also has encouraged and challenged me through my studies. Thanks so much for her support, patience, as well as professional knowledge and ideas.

I wish to express my warm thanks to reviewers Nicole LEVY, Pascal POIZAT, and Chair of the jury Christophe FOUQUERE for their time spent on reading my thesis and remarks made.

I would like to thank to the whole administrative board at LIPN for their support, especially Laure PETRUCCI - Head of LIPN, Christophe FOUQUERE - Head of LIPN in my first year, and Brigitte GUEVENEUX - Secretary of LIPN, also Michael FORTIER and Mamadou SOW for their technical support.

My sincere thanks are due to all my friends in Paris 13, who share with me PhD life and help me a lot: Phuong, Loc, Trung, Nghia, Quang, Nguyen, Binh, Le, Trang, Tung-Dung, Nga-Hoan, Phong, Hoan-Ly, Khue, Cuong, Bao, Cuc, and due to all members of LIPN, for their warm and friendly atmosphere.

Last, but by no means least, I want to thank my parents Tam and Du for being my inspiration and motivation to become a PhD student in France. Thanks to my brothers: Minh and Duy, my nephews and nieces: Hien, Ngoc, Mai, and Phuong Nghi for being by my side despite the geographic distance between us. And I could not express how much I am grateful to my husband Duc, for his love, his measureless support, as well as for his belief in me.

Abstract

RIGOROUS AND FORMAL APPROACHES FOR MODELLING AND DESIGNING SERVICE SYSTEMS

Abstract: The emergence of Service Oriented Architecture (SOA) allows application functionalities to be provided and consumed as sets of services, and enables a community, an organization or a system of systems to work together more cohesively using services without getting overly coupled. SOA has been associated with a variety of approaches and technologies and became a solution for building service oriented systems.

In this thesis, we focus on proposing precise (*exactly or sharply defined or stated*) methods for modelling and designing service oriented systems with the UML, as well as with formal specification notations to have both advantages of practical visual notations.

For implementing SOA successfully, our first effort is to define a mechanism for defining and modelling a *business* and its business processes, then we have developed a formal method for modelling SOA services and service systems not only visually but also formally and precisely.

In terms of designing service systems (i.e., how is a service system defined for realizing the business processes), we proposed a method which follows the Model Driven Architecture approach and results in transforming models in designing a high-quality service systems for particular enterprises. Our method is based on model transformation, in which some transformation patterns should be defined in order to transform a business model into a design model of a service system.

In result, we have developed two precise methods for modelling service systems (one based on a standard widespread not formal notation (UML), and the other based on a formal notation (Casl4Soa)); a precise method for modelling a *business* and its processes; a precise method for placing a system on a *business model*; a set of transformation patterns to transform a business model with placement to a *design model* of a service system (with assuming that some services already available).

Besides, we introduce a mechanism to verify the realization of a business on a designed service system.

Key words: SOA, Service-oriented system, Business process, Design pattern, MDA, UML, SoaML, Casl-Mdl, Casl4Soa

RESUME

APPROCHES RIGOUREUSES ET FORMELLES POUR LA MODELLISATION ET LA SPECIFICATION DES SYSTEMES ORIENTES SERVICE

Résumé: L'émergence de l'architecture orienté services (SOA) permet que les fonctionnalités d'une application soient fournies et consommées comme des ensembles de services, et qu'une organisation ou un système de systèmes travaille de manière cohérente. SOA est associée à une variété d'approches et de technologies et est une solution pour construire des systèmes de services.

Nous proposons une nouvelle méthode pour la modélisation et la conception de systèmes orientés services avec UML, ainsi qu'avec les notations de spécification formelle et visuelles pratiques.

Pour l'implémentation, notre premier effort consiste à définir un mécanisme pour la définition et la modélisation d'un système et de ses processus, puis nous développons une méthode formelle pour modéliser les services SOA et les systèmes de manière non seulement visuelle mais aussi formelle.

Pour la conception, nous présentons une méthode qui respecte l'architecture dirigée par les modèles et aboutit à transformer les modèles pour concevoir un système de services de haute qualité. Notre méthode est basée sur le modèle de transformation, dans laquelle des modèles de transformation sont définis pour transformer un modèle d'activité en un modèle de conception d'un système de services.

En conclusion, nous avons proposé: deux méthodes pour modéliser les systèmes de services (l'une est basée sur une notation non formelle (UML), l'autre sur une notation formelle (CASL4SOA)); une méthode pour modéliser d'un système et de ses processus; une méthode pour placer un système dans un modèle d'activité; un ensemble de modèles pour transformer un modèle de système en un modèle de spécification d'un système de services.

Mots-clés: SOA, Systèmes orientés service, Processus de systèmes, Patron de conception, MDA, UML, SoaML, CASL-MDL, CASL4SOA

Contents

Chapter 1

Introduction

Contents

1.1 Statement of problem

The emergence of Service Oriented Architecture (SOA, c.f., Sect. 2.1) has enabled business functionalities to be invoked over a remote network, and thus requires specific methods to develop service-oriented systems.

Up to now, SOA has been associated with a variety of approaches and technologies and became a solution for building systems that are easily modified. SOA allows application functionalities to be provided and consumed as sets of services, and enables a community, an organization or a system of systems to work together more cohesively using services without getting overly coupled.

The traditional development methods, related techniques, and notations have been found inadequate to support the development of service oriented systems, so this has motivated our work on development methods for SOA based system. Moreover, developing service systems is a difficult task, and methods can help SOA developers avoid common errors.

For the modelling of services for the design of service systems (i.e., how are services defined), several proposals using the UML notation [25] and formal semi-visual notations have been developed. However, we found that several currently available modelling notations have a typical problematic aspect, i.e., the lack of a formal semantics if not of a well-defined syntax. We recall here a statement of Object Management Group[1] (OMG) that *"Good SOA services cannot be achieved by simply exposing legacy applications and data directly. Rather, they need to be transformed to support enterprise semantics"*.

It is needed to have a precise modelling notation with a formal semantics playing a role in specification, analysis, and even checking the quality of models. At a precise level, the specification of services and service systems shall be effective in supporting their evolution to maintain service systems for future change. This problem urges us to develop a formal method for modelling a service system not only visually but also formally and precisely.

In terms of designing service systems (i.e., how is a service system defined for realizing the business processes), the experience of solving the wide business and architectural issues still stands at an early stage. Moreover, there is an important aspect in the implementation of SOA, that is: we cannot link our business processes to our service models without following the MDA modeling standards, i.e., Model Driven Architecture[2]. MDA standards offer the capability to design a complete SOA

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

[2] MDA - "The Architecture Of Choice For A Changing World", http://www.omg.org/mda/index.htm

solution through models, and minimize the effort invested in specific technologies or protocols. Those aspects motivate the need for a method which follows the MDA approach and then results in transforming models in designing a high-quality service systems for particular enterprises.

Hence, the relevant work of this thesis is devoted to develop modelling and designing methods for SOA services and service systems.

The starting point is an informal description of a *business*, then the developers have to build a design model of the service system before developing the applications for this service system. We use the concept of *business* to denote the activities performed within an enterprise (i.e., organization, formally or informally established, that performs specific activities for some specific goals).

For implementing SOA successfully, first of all, we should have in hand a precise model of the business and its processes to be supported. Thus, our first aim is to define a mechanism for defining and modelling a business and its business processes. In fact, there are a lot of business analysts studying the way companies work and defining business processes with simple flow charts, such as, BPMN 2.0 [24]. We are also comfortable with visualizing business processes in a flow-chart format, but moreover, they should be modelled in a precise way that does not to lead the readers to the confusion.

For building modelling methods, we have used both the UML and a CASL4SOA, a CASL-MDL [9] based notation. The UML offers a very large set of constructs to build the diagrams, such as activity diagrams, class diagrams, and use case diagrams, etc. and we can take advantage of its numerous features and supporting tools, and also define our own methods. For the formal description of semantics of services, we choose CASL-MDL that is a visual formal notation derived by CASL-LTL. CASL-LTL is a textual formal specification language based on a first-order many-sorted branching time temporal logic. It is an extension for dynamic system of CASL that resulted from a unification effort of the algebraic specification community.

We have followed the MDA to propose a method for building the service system to support a business. Thus, this method should be based on model transformation, in which some transformation patterns should be defined in order to transform a business model into a design model of a service system (illustrated visually in Fig. 1.1).

As a part of the development of a service system to support a business, it is necessary to have mechanism for placing a service system being built to support a business on its model. This work helps us in deciding which part of that business will be automatized by the service system. The placement work should be a starting phase in our design method.

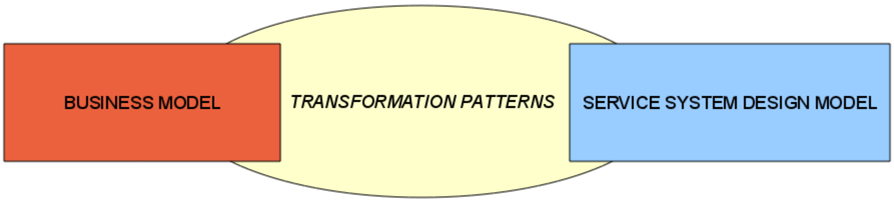


Figure 1.1: Transformation patterns filling the gap between the Business Model of a business and the Design Model of a service system supporting this business

In the early stage of this thesis, we chose SoaML [27] as a based-modelling language to develop our modelling method and this method seems different from our proposed modelling method based on Casl4Soa [8]. However, in our working progress, we investigate that we are able to use UML2.0 to define our own modelling method, and we can unify our proposed modelling method based on Casl4Soa with this one in a common view of a SOA service system. Thus, we finally built a unique conceptual model of service system for both the two proposed modelling methods, one using UML 2.0, instead of SoaML, and one using Casl4Soa.

1.2 Contributions

We focus on proposing a precise[3] method for modelling and designing service oriented systems with the UML, as well as with formal specification notations to have both advantages of practical visual notations.

Fig. 1.2 gives an overview of our proposed approach to service system development. The first point is building the model of the business, thus we have built a precise UML-based modelling method supporting for this work. This first stage results in a Business Model. In the next step, we deal with deciding which part of the business will be automatized by the service system through the Business Model. This work should be done before building the design model of a service system supporting this business. Hence, we also define a precise method for placing a system over a business, i.e., over a Business Model. The Business Model with Placement marked is the input of the next stage, i.e., transforming the Business Model to a Design Model of a Service System. For this stage, we propose a set of patterns to drive the transformation of this Business Model with Placement until obtaining a Design model of a Service system. Then the designed Service system should be validated with respect to the selected business part defined in the placement stage.

[3] Precise: exactly or sharply defined or stated (Merrian-Webster Dictionary)

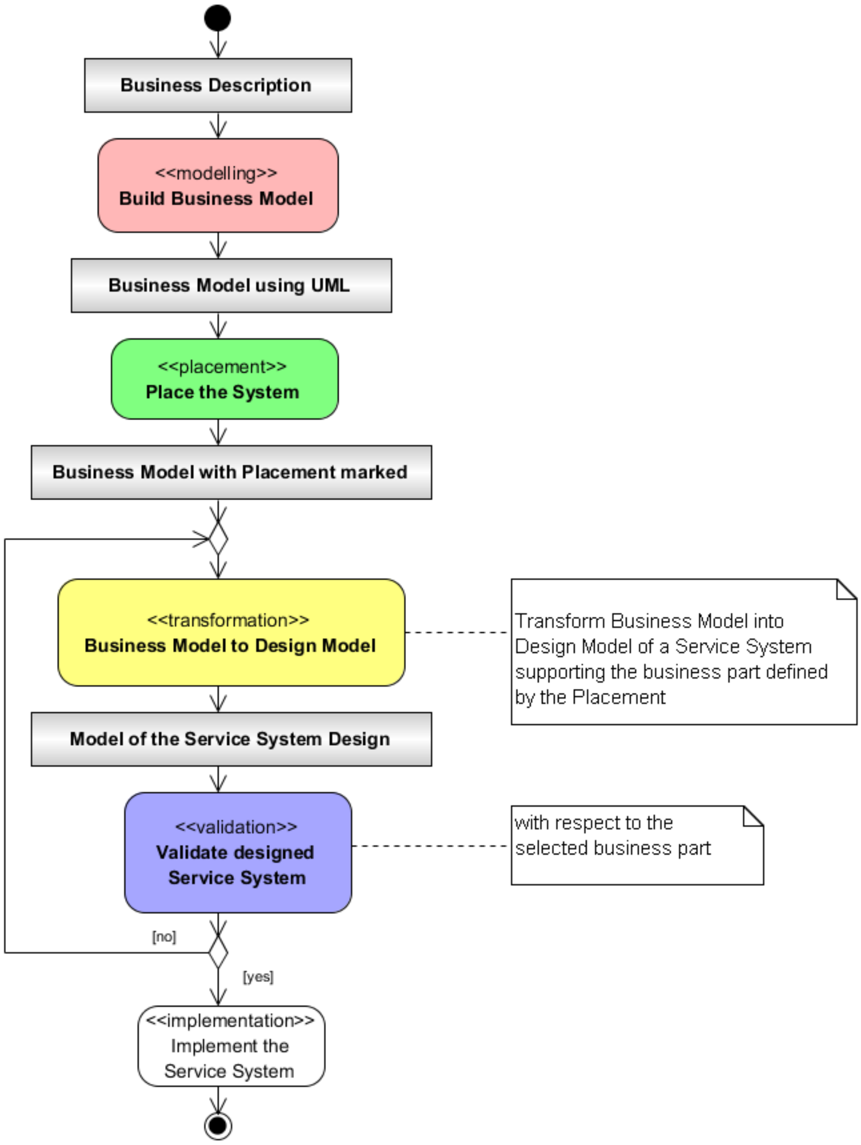

Business Description
<<modelling>>
Build Business Model
Business Model using UML
<<placement>>
Place the System
Business Model with Placement marked
<<transformation>>
Business Model to Design Model
Transform Business Model into Design Model of a Service System supporting the business part defined by the Placement
Model of the Service System Design
<<validation>>
Validate designed Service System
with respect to the selected business part
[no]
[yes]
<<implementation>>
Implement the Service System

Figure 1.2: Proposed approach to service system development

The last stage is the implementation of the service system, in which the models may be transformed into execution languages, description languages, or other technique fragments by a transformer in a Model Driven Environment built for SOA. However, we do not consider this stage at our work, it will be subject of future work.

In every proposed method in this thesis, for presenting the models of a business and a service system, we use a UML profile, defined by a set of stereotypes, and we explicitly define the form of the models by means of a metamodel and a set of well-formedness rules.

In total, the following precise methods for modelling and a set of transformation patterns for designing a service system are proposed, they are:

1. Two precise methods for modelling service systems, one based on a standard widespread not formal notation (UML), and the other based on a formal notation (CASL4SOA). The applications of these methods are illustrated by two case studies.
2. A precise method for modelling a business and its processes
3. A precise method for placing a system on a business model
4. A set of transformation patterns to transform a business model with placement to a design model of a service system (with assuming that some services already available).

Those precise methods includes a set of *well-formedness constraints* on the models investigated to guide the development of good quality models and avoid frequent mistakes. Detailed guidelines for applying those methods are defined to help the developers

Besides that, we introduce a mechanism to verify the realization of a business on a designed service system.

1.3 Thesis layout

The thesis is organized in 11 chapters, and their contents are summarized as following:

Chapter 1 (this chapter) introduces the motivations of doing this thesis, and its structure.

Chapter 2 presents related works and recent works of other colleges and IT organizations in our subjects of interest, i.e, SOA, modelling and designing service systems.

Chapter 3 provides the description of two case studies, i.e., the Dealer Network case study that was adopted from OMG Adopted Specification of SoaML [27], and the services provided by the set of software tools composing a Office-like suite. We will illustrate the use of our proposed methods in this thesis by the application of them to such systems.

Chapter 4 presents our view on services and service systems, in which we introduce our conceptual model of a service-oriented system, that is the basis of the work in the following chapters.

Chapter 5 gives a general introduction to business and business processes, and describes the precise business modelling method that we propose.

Chapter 6 introduces a method to model service systems using a profile of the UML, called PreciseSOA. PreciseSOA method has been inspired by SoaML [27].

Chapter 7 presents our extension of Casl-Mdl models (that offers a visual syntax to a subset of the Casl-Ltl [45] formal textual specifications) to develop Casl4Soa as a formal visual notation used to model service systems formally and effectively.

Chapter 8 presents a method for designing a service system being built to support a business that follows the MDA. The method provides also a set of transformation patterns to help the developer works.

Chapter 9 discusses difference and the relationship between the two notations, i.e., the UML profile PreciseSOA and the formal visual notation Casl4Soa, essentially the level of precision (and thus the expressiveness) both at the level of constructs and at the semantics level, and the evaluation of the application of those approaches on the case studies.

Chapter 10 concludes and presents our future work.

All the diagrams included in the thesis have been created by using the community edition of the tool Visual Paradigm (available on [44]).

Chapter 2

State of Art

Contents

Let us present here the related works and the recent works of other colleges and IT organizations in our subjects of interest, i.e, SOA, modelling and designing service systems. First of all, we collect the definitions about SOA from various organizations, and present our own view of SOA and its concepts. Then we take a closer look on how the extended UML and formal specification languages used for modelling visually and formally the concepts of a SOA system in existing works. After that, we introduce some methods for designing service systems that their proposed techniques and design phases give us the inspiration to develop our method.

2.1 Understanding SOA (Service Oriented Architecture)

Several IT organizations (OMG, W3C, IBM, ...) have been writing about SOA [14, 28, 42] with different meaning according to their own definitions.

The World Wide Web Consortium (W3C) refers to SOA as "*A set of components which can be invoked, and whose interface descriptions can be published and discovered*" [30].

The Object Management Group[1] (OMG) considers that SOA is "*An architectural paradigm for defining people, organizations and systems provide and use services to achieve results*" [27].

IBM defines SOA as the following: "*SOA is a business-driven IT approach that supports integrating a business as linked, repeatable business tasks, or service*" [33].

Anyway there will not be an exhaustive definition because of the specific audiences that they address.

So what is SOA finally? Is it an approach, a paradigm, a strategy or a framework?

For us, we thoroughly follow the understanding of IBM [13] about SOA that: "SOA presents an *approach* for building distributed systems that deliver application functionality as services to either end-user applications or other services". SOA approach strongly reinforces well-established, general software architecture principles, such as information hiding, modularization, and separation of concerns.

Let us take a first look at a SOA by Fig. 2.1 where the elements might be observed in it are shown [13]. The SOA is divided into two halves, with the left half addressing the functional aspects and the right half addressing the quality of service aspects. These elements are briefly described as follows:

- *Transport* is the mechanism used to move service requests from the service consumer to the service provider, and service responses from the service provider

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

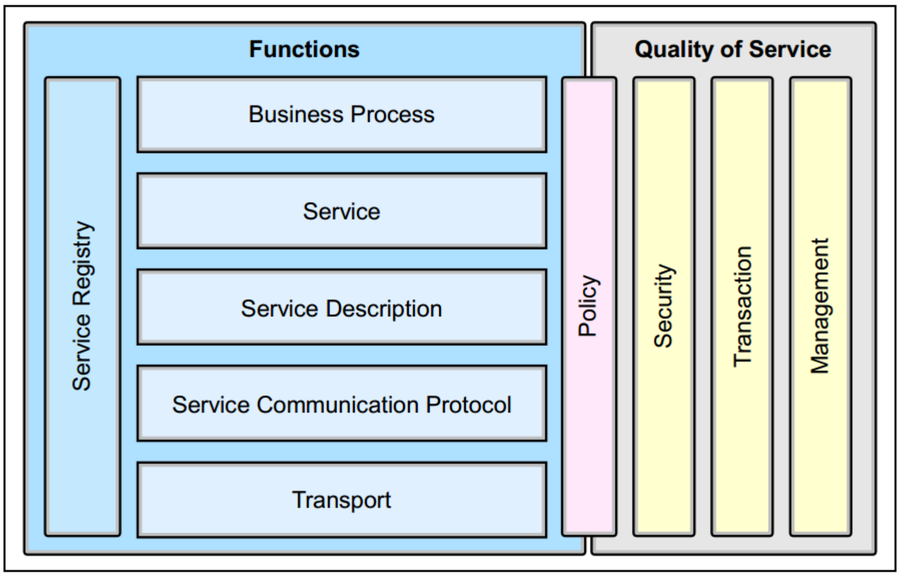


Figure 2.1: Elements of a service-oriented architecture

to the service consumer.

- *Service Communication Protocol* is an agreed mechanism that the service provider and the service consumer use to communicate what is being requested and what is being returned.

- *Service Description* is an agreed schema for describing what the service is, how it should be invoked, and what data is required to invoke the service successfully.

- *Service* describes an actual service that is made available for use.

- *Business Process* is a collection of services, invoked in a particular sequence with a particular set of rules, to meet a business requirement.

- *Service Registry* is a repository of service and data descriptions which may be used by service providers to publish their services, and service consumers to discover or find available services. The service registry may provide other functions to services that require a centralized repository.

- *Policy* is a set of conditions or rules under which a service provider makes the service available to consumers. There are aspects of policy which are functional, and aspects which relate to quality of service; therefore the policy function is in both functional and quality of service areas.

- *Security* is the set of rules that might be applied to the identification, authorization, and access control of service consumers invoking services.
- *Transaction* is the set of attributes that might be applied to a group of services to deliver a consistent result. For example, if a group of three services are to be used to complete a business function, all must complete or none must complete.
- *Management* is the set of attributes that might be applied to managing the services provided or consumed.

Service in the context of SOA

The important concept in SOA is "***service***", i.e., IT services that are distinguished from general business services. There are many definitions for the concept of "service" in context of IT, such as:

- A component capable of performing a task that is described using WSDL (Web Service Definition Language) [50].
- A vehicle by which a consumer's requirement is satisfied according to a negotiated contract [50].
- A logical representation of a repeatable business activity that has a specified outcome [29].
- A service is defined as the delivery of value to another party, enabled by one or more capabilities [27].
- A module that can be invoked, that is assigned to a specific function and that offers a well defined interface [37], etc.

The definition of service that we consider is given in [27]: "*A service is value delivered to another through a well-defined interface and available to a community. A service results in work provided to one by another*". This leads us to realize two key roles in SOA: the *service provider* who publishes a service description and provides the implementation for the service and the *service consumer* who can use the service directly or can find the service description in a service registry and invoke the service.

We have an agreement on all aspects used to characterize a service in [16] as following:

- *Defined*: services are defined in terms of what they do (e.g., the process(es) they perform), the interfaces used to communicate with the services (e.g., how to invoke the service to perform the process), the data passed to and returned from the services, and how the service is managed;

- *Implemented*: the service has to be implemented in software in such a way that: it can respond to requests to perform its function, it can perform the requested process, and it may return or disseminate results;

- *Deployed*: the service must be made available for use by others;

- *Managed*: the deployed service implementation is under the control of some management authority to insure that the service is available and operates as defined, and to provide the necessary underlying IT infrastructure to manage the operation of the service;

- *Reusable*: by providing defined, discrete functionality, independent of how it is used, a service can be used (or reused) at any point in the overall business process where the functionality is needed;

- *Communicating:* a service is accessed by sending it a request, e.g., from a IT system or from another service. Results are communicated to those that use the service;

- *Abstracting*: services (through their operations) define only what process the service provides and how to communicate with it. How the service is implemented to provide the defined functionality is not defined;

- *Composable*: a service may be combined with other services to implement the overall business process.

Also an agreement on the additional implementation and operation specific characteristics of a service, they are:

- *Granularity*: the complexity of the business process provided by the service may range in scale. The service may define a small process (or many small processes) where each request performs a simple operation and many requests are needed to complete a business process i.e. fine grained or the service may provide a substantial process that maps directly to a business-level step i.e. coarse grained;

- *Coupling*: while services need to interact to solve business problems, a service may be defined to be independent of any other service and can function without the knowledge of how other services work, or it may have explicit knowledge about not only what other services are available, but how they perform their operations. Services that are not dependent on other services are loosely coupled, while those that require other services are (more) tightly coupled;

- *Autonomy*: what a service requires to perform its operations is local knowledge only to the service. Autonomy is closely related to both coupling and abstraction;

- *State*: to fulfil a request, the operational service may need to know about historic use and invocation of the service i.e. to access data about the "state" of the business operation maintained by the service itself. The service may require no information about state i.e. stateless, or it may require knowledge of prior operations, i.e., stateful;

- *Discoverability*: service definitions may be made available so that existing services can be found, enabling reuse and composition. Discoverability is independent of the service itself, but part of the overall environment in which the service is defined and managed.

Finally, whatever definitions proposed for a service, eventually they must be fully described such that a service consumer can search, bind and invoke them. And all kinds of services must be designed to be reused in different contexts of applications.

Characteristics of SOA system

To create a system called SOA-based, one needs techniques for the analysis, design and realization of services, such that this system has the following aspects:

- *Loosely coupled*: client of a service is essentially independent of the service. The way a client (which can be another service) communicates with the service does not depend on the implementation of the service.

- *Reuse*: beyond what is described in the service contract, services hide logic from the outside world, the logic is divided into services with the intention of promoting reuse.

- *Composition*: enables to share modules between applications and inter-application interchanges.

- *Uncoupling*: in order to reduce the coupling between modules, the coupling towards the platform and infrastructure, the coupling between the client of a service and a specific implementation of this service.

- *Permanence*: supporting current and future technologies.
- *Flexibility*: since every application lives, has a precise life cycle, can be enriched with new modules and has to answer new business needs.
- *Openness and interoperability*: in order to share modules between platforms and environments.
- *Distribution*: the ability of a service consumer to invoke a service regardless of its actual location in the network so that modules can be remotely accessed and so that they can be centralized.

According to [50] in Microsoft Developer Network (MSDN) library [40], there are three important architectural perspectives for SOA as following:

- The *Application Architecture*, where the consumers are focused on, is the business facing solution which consumes services from one or more providers and integrates them into the business processes.
- The *Service Architecture* provides a bridge between the implementations and the consuming applications, creating a logical view of sets of services which are available for use, invoked by a common interface and management architecture.
- The *Component Architecture*, where the provider is focused on, describes the various environments supporting the implemented applications, the business objects and their implementations.

2.1.1 Views of SOA

OMG's view of SOA

"SOA is an architectural approach that seeks to align business processes with the services protocols and the underlying software components and legacy applications that implement them."

This is an official definition of SOA that the OMG states. The OMG highly recommends to understand clearly the model of the business processes to develop a SOA supporting them successfully. We should understand how to model business processes, services, and components and how to tie all the models together in a consistent manner.

In one specification[2], the OMG adopted a Fig.2.2 from an article on BPM and SOA [48] to annotate an organization of SOA environment.

[2] http://www.omg.org/attachments/pdf/OMG-and-the-SOA.pdf

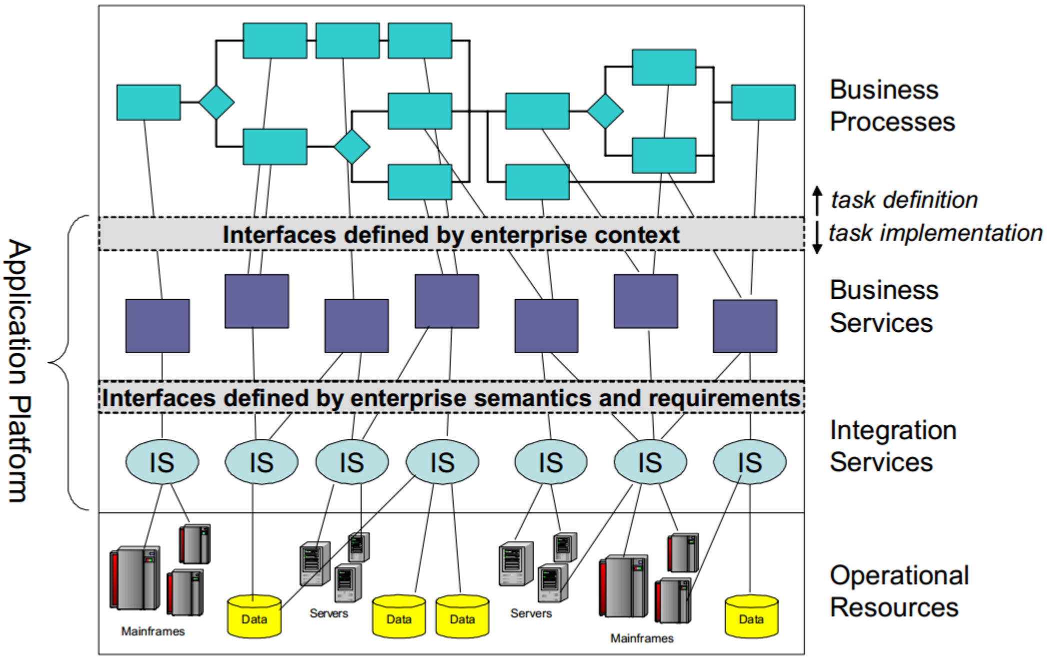


Figure 2.2: An organization of SOA environment

The SOA environment is divided into four general layer as following:

- Business Process: describes business processes made up of a sequence of business activities,
- Business Services: defines business services capable of automating specific business process activities,
- Components: defines software components and orchestrations that allow the business services to link to and call enterprise-level shared resources as needed ,
- Operational Resources: illustrates applications, packages and databases that might be called upon by the various components.

At the top level, Business Process, OMG has adopted the Business Process Modelling Notation (BPMN) [24] that is working on Business Process Definition Metamodel (BPDM). At the Business Service level, the OMG has defined the Unified Modelling Language (UML) version 2.0 [25]. At the Component and Operational levels, the OMG has developed a variety of different standards, e.g., BPDM has all the information needed to generate Business Process Execution Language (BPEL) [34] code.

The OMG declares that good SOA services should be transformed to support enterprise semantics and should not be achieved by simply exposing legacy applications and data directly. The OMG has the responsibility in creating the common semantic modelling system made up of specific modelling standards and a common approach to cross-model communication - the Model Driven Architecture (MDA) [22].

2.1.1.1 IBM technicians' view of SOA

The elements of SOA:

The authors in [36] adapted this set of *elements* for SOA from Three Architectural Perspectives given in [50].

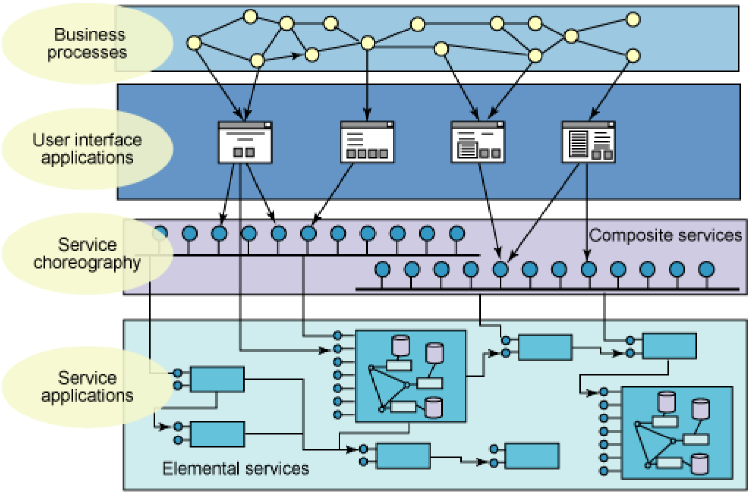


Figure 2.3: The elements of SOA

Fig. 2.3 shows the elements of SOA from the perspective of applications. Business processes are supported by user interface applications and service applications. A step in a business process is either manual or supported by a user interface application. User interface applications implement a lot of micro work flow, and they also consume services that implement business functionality.

In the service choreography layer, composite services are defined by means of a choreography language, such as BPEL. The choreography of composite services defines their flow and composition from elemental services. The choreography layer should be supported by a choreography tool that allows graphical specification.

The elemental services, used by the service choreography layer and also by user interface applications, are implemented by service applications. In turn, the service implementations might call other services, often from other service applications.

The layers of SOA:

Ali Arsanjani, Chief Architect, SOA and Web services center of Excellent, IBM build an architectural template for a SOA in [2].

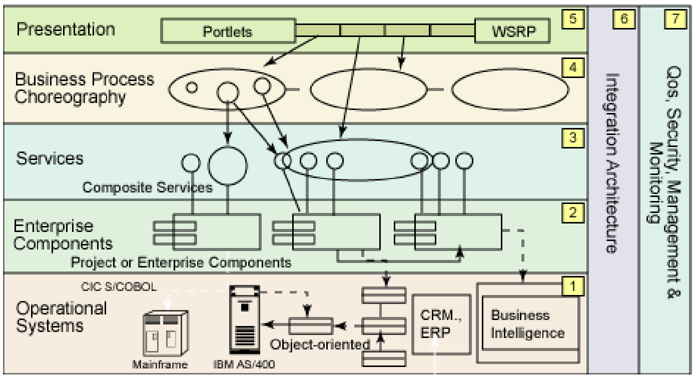


Figure 2.4: The layers of SOA

Fig.2.4 depicts a representation of this type of architecture. For each of these layers, we must make design and architectural decisions.

The operational systems layer consists of existing custom built applications, and called legacy systems.

The enterprise components layer consists of enterprise components that are responsible for realizing functionality and maintaining the *Quality of Service* (QoS) of the exposed services. These special components are a managed, governed set of enterprise assets that are funded at the enterprise or the business unit level.

The services layer consists of the services that the business chooses to fund and expose. They can be discovered or be statically bound and then invoked, or possibly, choreographed into a composite service. This service exposure layer also provides for the mechanism to take enterprise scale components, business unit specific components, and in some cases, project-specific components, and externalizes a subset of their interfaces in the form of service descriptions.

The business process composition or choreography layer defines the compositions and choreography of services exposed in layer 3. Services are bundled into a flow through orchestration or choreography, and thus act together as a single application. These applications support specific use cases and business processes.

The access or presentation layer is usually out of scope for discussions around a SOA, but the author depict it here because there is an increasing convergence of standards and other technologies, that seek to leverage Web services at the application interface or presentation level. We can think of it as a future layer that you need to take into account for future solutions.

The integration enables the integration of services through the introduction of a reliable set of capabilities, such as intelligent routing, protocol mediation, and other transformation mechanisms. Web Services Description Language (WSDL) specifies a binding, which implies a location where the service is provided.

The QoS provides the capabilities required to monitor, manage, and maintain QoS such as security, performance, and availability.

2.1.1.2 Our view of SOA

In our view, SOA is an IT architecture for building business applications as a set of components that are considered as service providers and consumers. SOA is one of the architecture framework used to define and build IT systems as a compositions of services. SOA provides a set of specific characteristics identifying how the services are defined, and how they interact. The most common enablement technology used to implement SOA is Web Service. Web Services is a realization of the SOA model for building an IT systems using a particular technology platform.

SOA is an architecture that is able to respond to future changes. Up to now, SOA has been associated with a variety of approaches and technologies to become a solution to build a system for change. The policies and frameworks of SOA provide the ability to enable application functionality to be provided and consumed as sets of services, and the ability to enable a community, organization or system of systems to work together more cohesively using services without getting overly coupled.

A service oriented system (*service system*, in short) consists of a set of *participants* providing and consuming services. A participant may provide or consume several services, whereas a service establishes a connection exactly between two participants: the provider and the consumer. A participant may be also structured in terms of services to be able to be fulfill the contracts corresponding to the provided and consumed services.

A participant may use internal services (i.e., local service) or external services (or both) to provide a service in a services architecture. Thus, there are two kinds of participant: one is *monolithic participant* that may only use external services and does not consist of any subparticipant, and the other is *structured participant* that may have subparticipants and use internal services. The structured participants may be further described by an internal services architecture. A participant services architecture (called as participant architecture for short) used to specify the architecture for a participant would consist of a number of other participants interacting through service contracts. The participant architecture illustrates how realizing participants and external collaborators work together.

We consider a service system as a special kind of participant that does not provide or consume any service, but it includes inner participants consuming and providing local services. For an image, assuming that a *port*, i.e., a UML notation, is used to model an interaction point of a participant, in which a participant consumes or provides a service, then a participant may have any number of ports corresponding to a number of its providing and consuming services, hence a service system is a participant having no *port*. It is a our typical conceptual point of our view for a SOA system, it governs all our issues in this thesis.

In our opinion, a service is a functionality unit in a service oriented system, used to support a/some business process(es) of a business (i.e., organization performs specific activities for some specific goals). A service is a set of operations that are encapsulated together by means of the capabilities of a participant.

A service is provided by a participant acting as the provider and used by a participant acting as the consumer. Those participants provide and consume several services to fulfill a purpose. The specification of a service includes: the roles each participant plays in the service such as *provider* or *consumer*; the *interfaces* provided and used by each participant; the messages exchanged between the participants while enacting the service; the choreography of the interactions between the participants.

Differently from various opinions, we consider that a service is an atomic component of a service system. In other words, there is no conception of *composite service* (i.e., service contains service) in our work. Also, a service must provide a specific capability and can be used in combination to support the needs of the business.

2.1.2 SOA vs Web Service and Object Oriented Model

Web Services are the set of protocols by which services can be published, discovered and used in a technology neutral, standard form [50]. Web Services are not a mandatory component of a SOA but Web Services propose one technical solution to implement SOA. SOA is much wider, as SOA services can be realized as Web services however all Web services are not equal to SOA services. Web services infrastructures allow for changing the service provider at runtime without affecting the consumers.

In Object Oriented Model (OOP), there are dependencies between the presentation layer and the business objects. Here is a three tier typical architecture with an object model according to [37]:

The client code must interact with the object model of the business layer, which increases the coupling and requires an important amount of calls between those

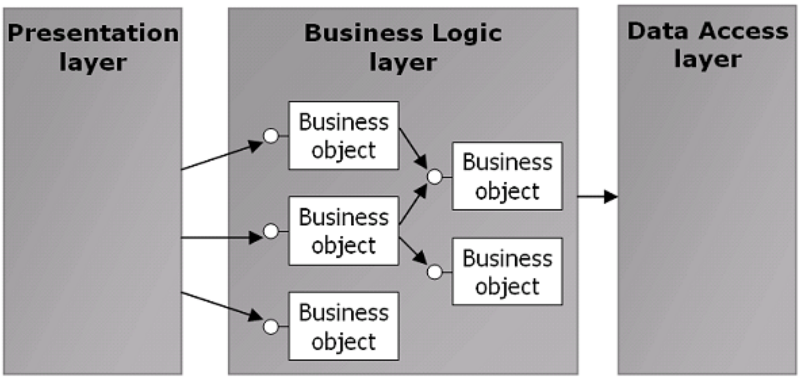


Figure 2.5: Three Tier Typical Architecture

layers. Such an amount of methods invocations between layers is a problem when business objects are located on a remote machine. The amount of business objects that the presentation layer has to manipulate reduces the independence between the layers and makes it difficult to learn how to use the business layer.

In SOA, a new abstraction called "Service" layer is introduced. Here is a service oriented architecture that would rely on the business objects:

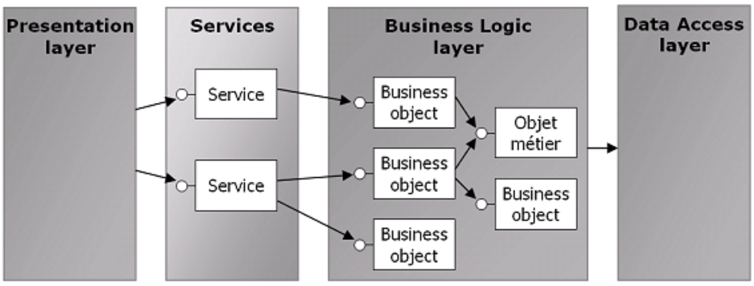


Figure 2.6: A Service Oriented Architecture Perspective

The Presentation layer uses the services to access the business objects. Business objects are located in class libraries that the services load into memory, since the service layer and the business layer are now located in the same process. Services give an abstraction of the object model and only expose a reduced set of features, which reduces the interchanges between layers.

2.2 Extended UML for SOA

The UML (Unified Modelling Language) [25] is a well-known language for modelling software system with support ranging from requirement modelling to structural overviews of a system, and down to behavioural specification of individual component [38]. However, the UML has been built within object-oriented environment, thus there is also a fact that, UML needs to be extended with specific elements which are geared towards expressing the new concepts of SOA systems. The UML-based modelling approach is by now widely adopted by both industry and research in modelling and designing SOA systems.

OMG is an international and open membership organization which has responsibility in setting the standard and profile for a wide range of technologies. And the UML is the core modelling standard notation of OMG. They has built UML version 2.0 [25, 26] that suits well for representing SOA concepts such as *services*, *interfaces*, *participants*, *protocols*, *choreography*, and *communications*. OMG also develop a standard, called SoaML [3] (Service oriented architecture Modeling Language), as a UML profile for modelling services and service oriented architecture (specification adopted in May 2012 [27]) for SOA systems.

Taking the advantage of UML 2.0, , a service-oriented profile, namely Uml4Soa, is devised in [35] as an extension of the UML 2.0. The Uml4Soa profile has been developed within the Sensoria project[4]. Uml4Soa profile defines a domain-specific language for service-oriented systems. Uml4Soa includes a set of service-specific model elements to ease the modelling activity of SOA. Uml4Soa diagrams enjoy formal analysis support through the Sensoria Development Environment (SDE) [39] and integrated tools. Uml4Soa complements the SoaML profile that focuses on the structural aspects of SOA systems and can be used in combination with other UML profiles, such as the MARTE profile[5], which has been used for performance analysis within the scope of the Sensoria project.

In the following, let us take a closer look on how the extended UML used for modelling concepts of a SOA system in those UML profiles.

SoaML provides the capability to model a service oriented architecture at the enterprise and system levels. SoaML models can be used to implement SOA solutions on popular SOA technology standards such as Web Services and Business Process Execution suites. Many UML tools have been extended to provide specific services modelling capabilities based on SoaML. With MDA tools, the services and service components defined in SoaML can be part of "source code" for service implementations.

In SoaML, a service is defined by a *contract* (defined by a UML collaboration use), a *interface* (defined by UML class) and a *choreography* (defined by UML sequence diagram). A service contract specifies an agreement between the involved

[3] Version 1.0.1 - http://www.omg.org/spec/SoaML/1.0.1/PDF/

[4] http://www.sensoria-ist.eu/

[5] http://www.omgmarte.org/

participants on how the service is provided and consumed. A service interface define the roles of the participants through the interfaces, the messages data of the service, and an associated choreography illustrating the order of messages within this service.

For each service, there will be one providing participant type and one consuming participant type. A participant is of type *service provider* if it offers a service and is of type *service consumer* if it uses a service. The UML mechanism of the Ports are used to indicate the points of interaction through which participants interacts with each others to enact services. There are two kinds of port that a participant may have, one is stereotyped by ≪Service≫ known as a service port where a service is provided by this participant, one is stereotyped by ≪Request≫ where a participant makes a request for service from other participant. A port typed by a service interface.

In SoaML, a *service architecture* presents how a system of participants provides and uses services to achieve business value. It consists of a set of services and a set of participants that work together by providing and consuming services for business goals. It is is defined by a UML collaboration with stereotype ≪Service Architecture≫. A service is provided and consumed by two participants, this associated relationship is represented by a collaboration use of the service contract. There may be several collaboration uses of the different service contracts in a service architecture, and each of them involves a different set of roles and connectors. Inside a service architecture, a participant is displayed as a UML part (a solid rectangle) that contains the role of participant and the type of this role. The two participants, who play the providing and consuming roles in a service, will be defined in the service contract of this service. The participant model is characterized by the set of modelled participants.

Now we move to Uml4Soa. The Uml4Soa, current version 3.0, the UML profile is defined for specifying behavioural aspects of service-oriented architectures on a high level of abstraction. In particular, it focuses on service orchestrations, i.e. compositions of services, by means of an orchestration workflow. An orchestration, in turn, is another service to be used externally, or in other orchestrations. UML 2.0 activity diagrams are selected as the base for modelling such workflows, and UML 2.0 state machines for modelling their externally visible behaviour with regard to a certain partner. The author extends both notations by SOA specific stereotypes, thereby enabling developers to model SOA orchestrations in a high-level fashion. The extension is minimal, i.e. existing UML 2.0 elements are used wherever possible, only extending the UML 2.0 where is required additional semantics, or if it adds to the overall clarity of the diagrams.

UML4SOA provides a notation that allows for intuitive and expressive specifications of service-oriented system, where some of elements provide shortcuts for service patterns. Based on the SENSORIA Ontology [19], a set of MOF (Meta Object Facility) metamodels of service-oriented concepts is built to specify SOA. They present UML meta-classes which are directly related to the SOA concepts. They are metamodel presenting model elements for structural and behavioural aspects, and for business goals, policies and non-functional properties of SOA systems. For instance, Fig.2.7 shows the metamodel including model elements to support the specification of structural aspects of services.

In those metamodels, they represent existing classes from the UML metamodel with a yellow background, while the new UML4SOA classes have a white background.

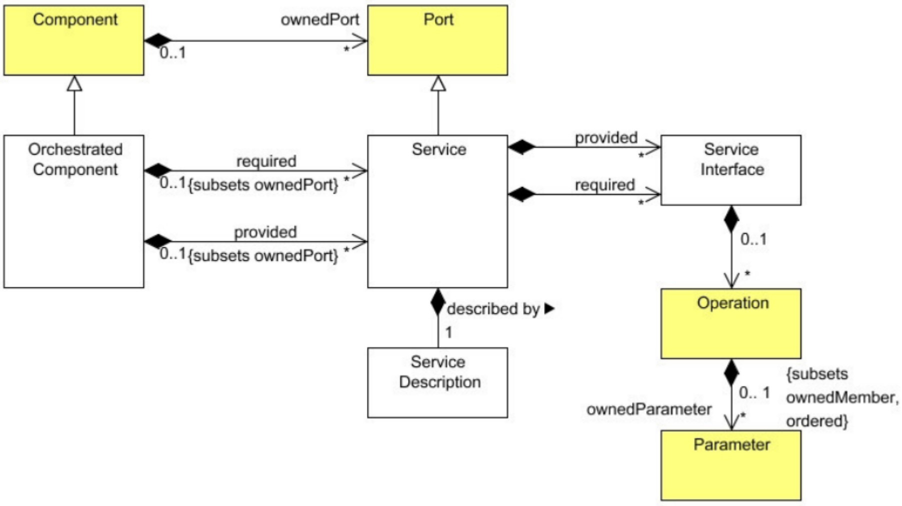


Figure 2.7: UML4SOA metamodel: package Structure

For each element of a metamodel, a stereotype is defined establishing an extension relationship to an appropriate UML meta-class. Besides, a set of rules which must be followed to create UML4SOA models are defined.

Several relationships in the model are defined, e.g., "*an orchestrated component may contain several services implemented as ports*", "*each service may contain a required and a provided interface*", etc.

For specifying behavioural aspects of SOA systems, another UML4SOA metamodel (see Fig.2.8), provides model elements supporting this work. The metamodel is presented in two parts: service orchestration along with compensation activities; and the service protocol and requirements for SOA. Service orchestration is the process of combining existing services together to form a new service to be used like any other service. To allow modelling of such compositions in UML, specific service-aware elements are added to be used in activity diagrams.

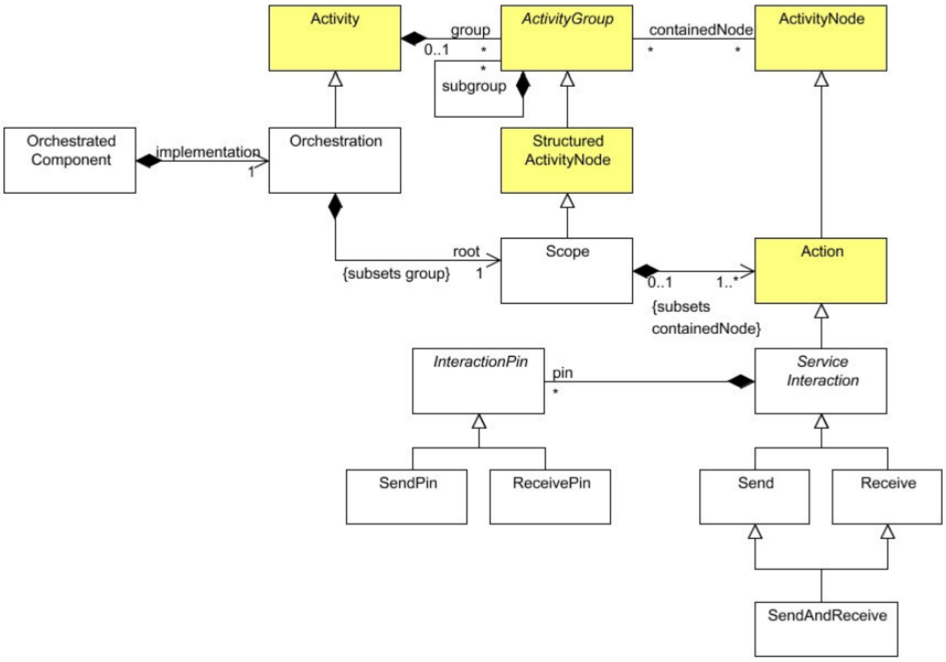


Figure 2.8: Uml4Soa metamodel: package Behaviour - Orchestration

At service level, the behaviour of the *port* providing services to or requiring services from other parties is specified using an UML *state machine*. The requirements are mainly given by the transformations to be defined to follow a model-driven approach.

The UML extension for modelling non-functional parameters of services is also described. The semantic basis for this extension is defined in the Sensoria Ontology while the style of the UML metamodel and the refinement of the concepts is inspired by the *UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms*. Uml4Soa is a UML implementation of the Sensoria Ontology, View Non-functional properties of services, refined to meet the needs of service development. The style and elements of the UML metamodel is guided by the *OMG specification UML Profile for Modelling Quality of Service and Fault Tolerance Characteristics and Mechanisms*.

In this extension, they focused on the QoS parameters that refers to any kind of runtime attribute which can be measured. Using a *SLA-driven* approach, the profile is built to support contracts between provider and requester parties. They consider that the profile must be specified by means of stereotypes and their relationships to the classes of the UML metamodel in order to be able to use the elements of the Uml4Soa metamodel in UML 2.0 CASE tools. They define a stereotype for all non-abstract classes of the metamodel defined.

For modelling the structural specification with UML extension, they present stereotype OrchestratedComponent indicating that the component is the result of the orchestration of service, stereotypes for specifying services, i.e. Service, ServiceInterface, and ServiceDescription, the stereotypes for distinguishing different types of communication paths, i.e. Permanent, Temporary, and the more service specific OnTheFly.

With *Business Policies*, the main point to be addressed remains the representation of the tasks, such as their behaviour in relation to policies. UML provides two Actions, to model the invocation of a behaviour in an activity model: CallOperationAction and CallBehaviorAction.

They need also a framework to complete the profile and obtain a modelling environment. The StPowla profile (described in later section) is used to deal with tasks, as elements of the workflows.

Besides introducing two general attributes of the StPowla task type (such as automation and actor) the framework accounts for their standard behaviour. The task types used in the workflow will inherit this behaviour, but they can redefine its signature, to take care of the arguments and results of the specific requested service.

They consider that the UML profile for service-oriented systems is enriched with constructs for modelling non-functional properties of services: stereotypes for the requester and provider specification and for the characteristics and dimensions of these characteristics.

An overview of the architecture, which is represented as a UML 2.0 *Deployment Diagram* is depicted. The model shows the distribution of the components within the different physical devices of the vehicle. The two UML4SOA stereotypes, ≪temporary≫ and ≪on the fly≫, visualize *temporary* and *on-the-fly* communication paths. The ≪permanent≫ is not visualized in order to avoid an overloaded diagram. The *Orchestrator* is an architectural element that is in charge to achieve a goal by means of composition of services.

To keep the diagram understandable, they do not cover all possible aspects, the aim was to show how different aspects such as the performance, the secure communication, the availability... can be incorporated using SENSORIA profile. Some example attributes are also listed to illustrate the detailed definition of *NFDimensions*, however, the list is not intended to be exhaustive.

To their knowledge, there is no much work on the use of UML in relation with policies. Like they do, UML is used to provide the context for the use of policies, rather than to express them. They think that they will be able to provide comments and suggestions for the UPMS approach, which is performed within the scope of an

OMG standard process. They really have the experience using the UML4SOA profile in modelling the architectural and behavioural aspects of the On Road Assistance scenarios of the automotive case study and the experience done so far with the StPowla profile and framework. At last, they suggest to do some more work in two directions: more extensive modelling of case studies to straighten the details, and experimentation with UML design tools that fully support the profiling mechanism.

There are also model transformation tools available for converting UML4SOA diagrams to BPEL/WSDL, JAVA, and JOLIE (Java Orchestration Language Interpreter Engine) [41]. This issue is described isolation in the SENSORIA Model Driven Approach (SDA) [51].

2.3 Formal modelling for SOA

One of our major purposes is to develop a method for modelling services and service systems not only visually but also formally. In this section, let us introduce some interested works of other collages in this field, modelling and specifying services formally.

The two authors in the group developing UML4SOA [19], Nora Koch and Philip Mayer, have coordinated with other collages in equipping UML4SOA with formal analysis called Architectural Design Rewriting (ADR) in [5]. ADR enables formal analysis of the UML specification.

There is a "inadequate-formal" approach (said by us) for modelling services using a natural policy language, APPEL [47], is introduced in StPowla [21]. They develop a workflow based approach to business process modelling that integrates a simple graphical notation and APPEL, to provide the necessary adaptation to the varied expectations of the various business stakeholders, and the SOA, and to assemble and orchestrate available services in the business process. StPowla combines policies and work flows that adds to each of the concepts being used on their own, since it permits to capture the essential requirements of a business process by *workflow* notation, and at the same time to express the inevitable requirements variability by policies in a descriptive way, at a similar level of abstraction. In StPowla, a task has a type and attributes. These are used to refer to the state of the execution of the workflow, and characterise properties of individual tasks or of the whole workflow. Besides, a task may have input and output values, it is carried out by at least one service, and that the choice of the services may be specified by policies associated to the task. The policies can express their choices by inspecting and/or using both the attributes and the input values. Let us show an example for a task with its policy made of the modelling language given in StPowla:

```
appliesTo makeCoffee when taskEntry([])
do req(glassOfWater, [], [Cost = 0])
andthen
req(main, [], [] )
```

A typical formal approach that we consider is developed in [7]. The authors has extended the *REsource Model for Embedded Systems* (REMES [49]) to formally describe the services. They focus on describing the behaviours of a service and checking the service correctness. The service behaviour should be described formally in such a way that their impact on QoS attributes can be exposed. They enriched the REMES with some specific information about service, i.e., service type, service capacity, time-to-serve, service status, and pre/postconditions on inputs, and output of the services.

The pre/postconditions are built in the Guarded Command Language (GCL) (defined in Dijstra's guarded command language [11]).

These conditions are exploited to describe the service behaviours and employed to check service correctness. For us, we understand that the service correctness here is the correctness of service compositions.

In REMES, a service is modelled by an *atomic mode* (do not contain submode(s)) or *composite mode* (contain submode(s)), *called* REMES *service*. The extended attributes of a service are exposed at the interface of mode, here they are:

- service type: specifies the type of given service (e.g., web service, database service, network service, etc.);

- service capacity: specifies the maximum number of messages handled per time units;

- time-to-serve: specifies the worst-case time needed to respond and serve a given request;

- service status: specifies the current status of service (i.e., *passive*, *idle*, and *active*);

- service precondition: is a predicate (Pre) conditioning the start of a REMES service execution;

- and service postcondition: is a predicate (Post) must hold at the end of a REMES service execution.

The formal specification of a service they proposed is a Hoare triple, {p}Service{q}, where Service is described in the GCL, p is the mode‘s precondition, and q is the postcondition.

Being aware that the services need to synchronize their behaviours, they propose a special kind of REMES models as a synchronization mechanism, i.e., AND mode and OR mode. If two services need to synchronize some actions, the respective edges will be decorated with *channel variables*, e.g., *out x* and *in x.* It means that the respective edges are taken simultaneously in both service, one writes variables that the other is reading. The question that we consider here is: How is the situation if two service need to synchronize more than two actions in a time point?

The authors extended REMES to model a service, and they define the Hierarchical Language for Dynamic Service Composition (HDCL) to check the correctness of service compositions. The HDCL is used in addressing the dynamic aspects of the services, it allows creating new services, adding, and deleting services from lists.

They introduce a *Hoare-triple* in the GCL to describe the REMES composite mode service, then the correctness will be checked by using the strongest post condition semantics. This is a formal way that they investigate for ensuring the correctness of the composition.

A set of REMES interface operations is defined by pre/postcondition specification, they are: Create service, Delete service, Create service list, Delete service list, Add service to a list, Remove service from the list, Replace service in the list, and Insert service at specific position. Let us introduce here two typical examples of them:

- Create service: *create service_ name*

  [pre]: service_ name = NULL

  create : *Type x N x x "passive" x* $(\sum \rightarrow$ *bool) x* $(\sum \rightarrow$ *bool)* $\rightarrow$ *service_ name*

  post : service_ name $\neq$ *NULL*

- Add service to a list: *add service_ name, s_ list*

  [pre] : service_ name $\notin$ *s_ list*

  add: *s_ list* $\rightarrow$ *s_ list*

  post : service_ name $\in$ *s_ list*

where $\sum$ is the set of service states, i.e., the current collection of variable values, they are service type *Type*, service capacity *N*, time-to-serve *N*, service status *"passive"*, the precondition $\sum \rightarrow$ *bool*, and the postcondition $\sum \rightarrow$ *bool.*

Then, a hierarchical language supporting dynamic REMES service composition (HDCL) is defined to provide means for connecting REMES services. This language facilitates modelling of nested sequential, parallel or synchronized services, it allows an theoretically infinite degree of nesting.

The Sensoria Reference Modelling Language (SRML) [18] is a service modeling framework that relies on UML state machines to model service behavior, which could help to spread its use among researchers. The benefit of the approach comes with the mechanism that supports the formal analysis formal analysis of functional and timing properties via model-checking; however, the analysis of extra-functional properties, other than timing, is not addressed.

The SRML language provides a number of semantic modelling primitives for service-oriented systems that are independent of the languages and platforms in which services are programmed and executed. In SRML, the orchestration of services is expressed in terms of a number of internal and external parties that are connected to each other through interaction protocols and jointly execute a business process. The configuration of this business process may change at run time as the discovery of required services is triggered. An example for service representation in SRML is shown in Fig.2.9.

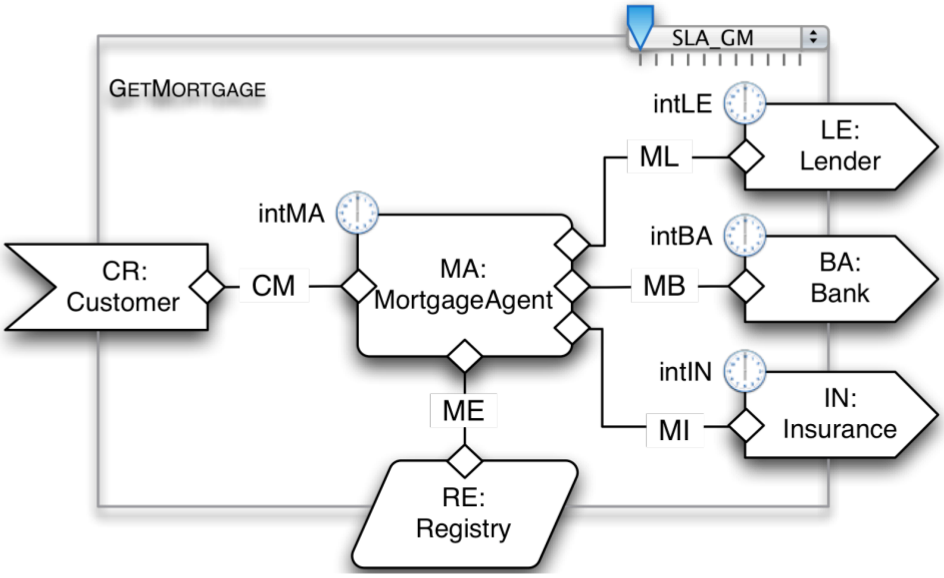


Figure 2.9: A SRML service representation

A formal computation and coordination model offers a layer of abstraction for capturing, orchestrating and analyzing properties of the conversational protocols that characterize service-oriented interactions. The properties of required and provided services are specified in temporal logic and can be analyzed over orchestrations defined in terms of state transition systems using the model checker UMC[6] that works over UML state machines. Time-related properties of services can be

[6] http://fmt.isti.cnr.it/umc/V4.1/umc.html

analyzed using the Markovian process algebra PEPA (developed at the University of Edinburgh).

An algebraic operational semantics supports the run-time discovery, selection and binding mechanisms of the language and offers a business reflective model of dynamic (re)configuration. SLA (Service Level Agreements) constraints and the associated ranking and selection mechanisms are formalized over the c-semiring approach to constraint optimization. The extensions of use-case and message-sequence diagrams provide support for a number of methodological aspects of engineering business services and activities.

2.4 Methods for developing service-oriented systems

In recent eras, the methods for analysis and design information systems have evolved through various generations, including structured programing, process orientation, object orientation methods. The emergence of SOA that enables the capabilities of businesses to be invoked over a distributed network, has given unprecedented key concepts, such as services, components and flows. The traditional development methods, related techniques, and notations have been found inadequate to support the development of service-oriented systems, so this has motivated work on development methods for SOA based systems.

Some interesting techniques are proposed in a method named *Service-Oriented Modeling and Architecture* (SOMA) developed in IBM for designing and building SOA-based solutions [1]. The structure of the method enables one to effectively analysis, design, implement, and deploy SOA projects as part of a fractal model of software development. The SOMA method incorporates the key aspects of overall SOA solution design and delivery.

In SOMA, a prioritization of the service model is conducted and based on a service-dependency diagram and takes into account the risk factors involved in the IT aspects of the architecture. A subset of services is prioritized for the next implementation release in which technical feasibility is planned for, measured, and exercised within a proof-of-concept prototype.

SOMA is composed of capability patterns representing techniques applied in the method. Many of these capability patterns are executed in all phases in SOMA with different degrees of elaboration and precision.

The SOMA method includes the seven major phases shown in Fig. 2.10

Those phases in SOMA are not linear, they are fractal phases applied in a risk-

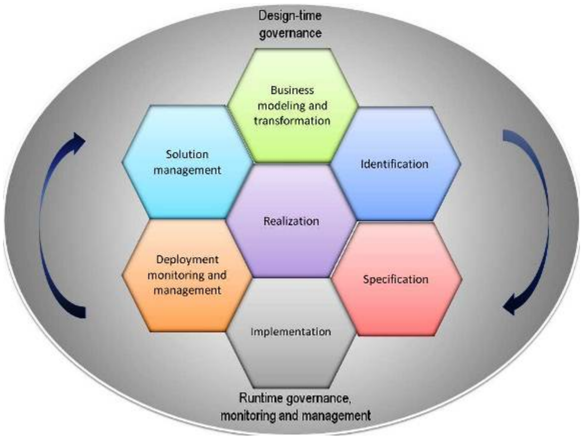

Design-time governance
Business modeling and transformation
Solution management
Identification
Realization
Deployment monitoring and management
Specification
Implementation
Runtime governance, monitoring and management

Figure 2.10: SOMA phases - a fractal model of software development

driven, iterative, and incremental approach using a nuance peculiar to the SOA life cycle. The life cycle of an SOA project uses a fractal approach that is a combination of two key principles. The first principle of fractal software development is the application of method tasks to self-similar scope: that is, tasks are done in a similar way in larger or smaller boundary scopes (whether enterprise-wide, line-of-business, or single project initiatives). The second principle of fractal software development is successive iteration. The concepts of iterative and incremental development life cycles have existed for a long time. They focus on prioritization and mitigation of risk factors in order to ensure the product quality of the solution.

For all phases in SOMA, a set of techniques is executed. Those techniques are harvested from the experiences working with hundreds of projects of the authors.

The first phase of SOMA is *Business modelling and transformation*. In this phase, the business is modelled, stimulated and optimized by defining business architecture and business models. This phase is not strictly required but highly recommended because a focal area for transformation identified may drive the using the set of the SOMA phases.

We see that the most interesting techniques are proposed in the second and the third phase of SOMA, they are *Identification* and *Specification*.

The Identification phase focuses on identifying candidate services and creating a service portfolio of services that support the business processes and business goals. There are three main complementary service identification techniques: *Goal service modelling (GSM), Domain decomposition and Existing assets analysis.*

GSM maintains the alignment of services with business goals. A generalized statement of business goals will be decomposed into subgoals that must be met in order for the high level goals to be met. To provide an objective basis for evaluating the goals, the Key performance indicators (KPIs) is identified. GSM combines the top-down and bottom-up approaches.

The *Domain decomposition* phase focuses on top-down analysis of business domains and business process modeling to identify not only services but also the components and flows. Both static view and dynamic view of the business are analyzed. A static view is harvested by partitioning the business domain into coarse-grained functional areas. The key elements are business functions, and business entities. A dynamic view is reflected by decomposing the business processes. Thus, the business processes are identified and modelled, and a process hierarchy is developed for business domain. Those processes, subprocesses and process activities will be considered as candidate services.

The *Existing asset analysis* uses a bottom-up approach to take a look at the existing assets (i.e, applications) in order to identify good candidates for service exposure and to leverage existing resources. The focus is on existing assets that play a role in business processes and functions. A coarse-grained mapping of business processes to the portfolio of existing application will be performed.

In the Identification phase, *Service refactoring and rationalization* is also an important technique. This activity consists of three parts: refactoring of services, the Service Litmus Tests (SLTs), and rationalization. The services are refactored in such a way that lower-level services (in the service hierarchy) are grouped together under a higher-level service, then the SLTs are applied to the set of candidate services to yield a set of exposed services. Rationalization consists of a review of the service model with business stakeholders to verify the continued relevance of the services.

We may consider the *Specification* phase in SOMA as a design phase. In this phase, the information models are specified and further analysis of existing assets are performed. The conceptual data model is elaborated into a logical data model to populate the attributes to be implemented, and the service message models which include input, output and error messages are designed. The *Service specification* is the core technique of this phase, it focuses on elaborating the detailed design of the services. Service operations that are invoked to execute a business function will be designated and identified in the service hierarchy. The result of this task is a *service context diagram*, this diagram depicts the ecosystem of a group of related service consumers and service providers.

Besides the service specification task, there are two tasks, i.e Subsystem analysis and Component specification which should be executed. The major output of the Subsystem analysis is a Subsystem dependency diagram that shows the dependencies of all subsystems and their interfaces. The output of the Component specification is the *enterprise component pattern* to represent the inner structure of components.

For the three remaining phases, i.e Realization, Implementation and Deployment, no specific techniques are proposed, however the authors recommend some key elements and guidance to achieve the tasks.

"SOA Principles of Service Design" by Thomas Erl [15] is an encyclopedia of service design principles needed to build SOA solutions. In the early phase of describing services, it is useful to use a common template or a form to collect similar meta-data from all the services. This book represents a service profile. Such a profile can be created early on during the analysis phase, and it is updated later when the service goes through various changes. The author give out the detail explaining what the profile might contain.

The author suggests standardizing the following vocabularies, also offering a good set of terms to start with:

- Service-Oriented Computing Terms
- Service Classification Terms
- Design Principle and Characteristic Types, Categories, Labels
- Design Principle Application Levels
- Service Profile Keywords

A list of organizational roles, with clear specifications for each one, creates a better picture of what everybody is supposed to do and how they relate to each other. The author presents a set of roles associated with service-oriented design principles, such as: Service Analyst, Service Architect, Service Custodian, Schema Custodian, etc,.

The author gives a description of each role mentioning the principles associated with it. For example, the Service Analyst role is associated with *Service Reusability*, *Service Autonomy*, *Service Discoverability.*

We pay attention to two main parts, i.e., fundamentals, and design principles. The fundamentals focus on describing what Service Oriented, Service-oriented Computing are, and give us an understanding about Design Principles in these fields. The second part provides typical design principles in SOA, they are: Service Contracts, Service Coupling, Service Abstraction, Service Reusability, Service Autonomy, Service Stateless, Service Dicoverability, Service Composibility. We recommend that we should be aware of those principles when we choose SOA to develop service-oriented applications. Because of space reason, we refer you to this book for throughout descriptions of those principles .

We are also interested in design patterns from the book “SOA Design Patterns” of Thomas Erl [17]. Thomas Erl provides service architects with a broad palette of reusable service patterns that describe service capabilities that can cut across many SOA applications. Service architects taking advantage of these patterns will save a great deal of time describing and assembling services to deliver the real world effects they need to meet their organizations specific business objectives. These patterns enable easily communicated, reusable, and effective solutions, allowing us to more rapidly design and build out the large, complicated and interoperable enterprise SOA into which our IT environments are evolving.

In particular, chapter 10 presents three *Inventory Governance Patterns*, they are: *Canonical Expression*, *Meta-data Centralization*, and *Canonical Versioning.* When first designing a service inventory, there are steps that can be taken to ensure that the eventual effort and impact of having to govern the inventory is reduced. The set of patterns supply some fundamental design-time solutions specifically with the inventory's post-implementation evolution in mind. *Canonical Expression* refines the service contract in support of increased discovering ability, which goes hand-in-hand with *Meta-data Centralization*, a pattern that essentially establishes a service registry for the discovery of service contracts. These patterns are further complemented by *Canonical Versioning*, which requires the use of a consistent, inventory-wide versioning strategy.

We pay our attention to chapter 16 which presents *Service Governance Patterns*, comprising a number of 8 patterns: *Compatible Change* and *Version Identification* deal with service versioning (focused on the versioning of service contracts); *Termination Notification* addresses the final phase of a service, it is retirement; *Service Refactoring* explains how to deal with changing service contracts; *Service Decomposition*; *Proxy Capability*, and *Decomposed Capability* include techniques needed to express coarse-grained services through multiple fine-grained ones; *Distributed Capability* helps increasing service scalability through processing deferral. The most fundamental pattern in this chapter is *Service Refactoring*, which leverages a loosely (and ideally decoupled) contract to allow the underlying logic and implementation to be upgraded and improved. The trio of *Service Decomposition*, *Decomposed Capability*, and *Proxy Capability* establish techniques that allow coarser-grained services to be physically partitioned into multiple fine-grained services that can help further improve composition performance. Distributed Capability provides a specialized, refactoring-related design solution to help increase service scalability via internally distributed processing deferral.

In term of implementation SOA systems, a development life cycle methodology for web services is proposed in *Service-Oriented Design and Development Methodology* by Prof. M.P. Papazoglou and Prof. W.V.D. Heuvel [43]. Their methodology is partly based on other related development models, such as Rational Unified Process (RUP) [10], Component-based Development [32], and Business Process Modelling [31]. Fig.2.11 depicts their methodology concentrating on the levels of the web services development life cycle hierarchy. It is based on an iterative and incremental process that includes several main phases concentrating on business processes and services, i.e., Planning, Analysis, Service Design (including: Service Design Concerns, Specifying Services, Specifying Business Process), Service Construction, Service Test, Service Provisioning, Service Development, Service Execution, and Service Monitoring.

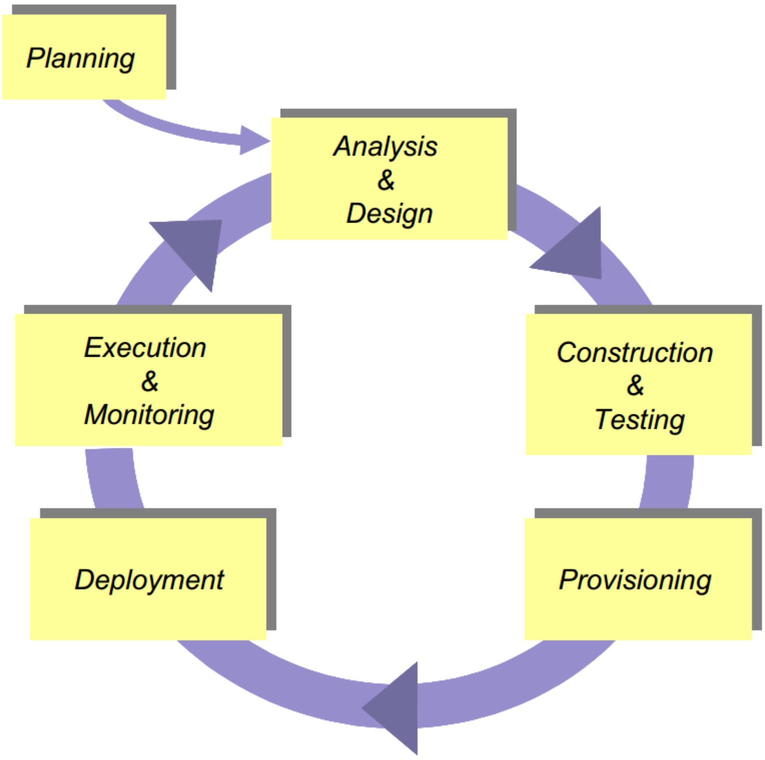


Figure 2.11: Phases of the service-oriented design and development methodology

We recommend two phases of interest, they are: Analysis phase, and Service Design phase. We evaluate that Service Design is the fundamental phase of this methodology.

During the Analysis phase, the requirements of a new system are investigated. This includes reviewing business goals and objectives that drive the development of business processes. This phase also examines the existing services portfolio at the size of service provider to understand which policies and processes are already in place and which need to be introduced and implemented. The Analysis phase comprises four main activities: *process identification, process scoping, business gap analysis,* and *process realization.* In *process identification*, the authors suggest to apply the design principles of coupling and cohesion for identifying the functionalities that should be included in a business process and the functionalities that is best incorporated into other business processes. In *process scoping*, they define that the *scope of a business process* is an aggregation of aspects that include where the process starts and ends, the typical users of the process, the inputs and outputs that the users expect, the external entities and providers that the process is expected to interface with, and the different types of events that start an instance of the process. *Gap analysis* starts with comparing candidate service functionalities with available software service implementations that may be assembled within the enclosures of a newly conceived business process. There are various options for *process realization analysis* that emphasizes the separation of specification from implementation that allows Web services to be realized in different ways, e.g., top-down or meet-in-the-middle development. The *process realization* results in a business architecture represented by business processes and the set of normalized business functions extracted from the analysis of these processes.

The Service Design phase is based on a twin-track development approach that provides two production lines: one to produce services (possibly out of pre-existing components), and another to compose services out of reusable service constellations. This calls for a business process model that forces developers to determine how services combine and interact jointly to produce higher level services. The Service Design Concerns include *managing service granularity*, *designing for service reuse*, and *designing for service composability*[7].

The Service Specification comprises a set of three specification elements, they are: Structural specification that defines the service types, messages, port types and operations, Behavioural specification that entails the effects and side effects of service operations and the semantics of input and output messages, and Policy specification that denotes policy assertions and constraints on the service. During Service Design phase, service interfaces that were identified during the analysis phase are specified based on service coupling and cohesion criteria as well as on the basis of the Service Design Concerns.

In Specifying Business Process, the business processes are described in the abstract. This step comprises four separate tasks, one deriving the process structure, one linking it to business roles, which reflect responsibilities of the trading partners, and one specifying non-functional characteristics of business processes. The

[7] Composability is a system design principle that deals with the inter-relationships of components - Wikipedia

first task is to one choosing the type of service composition. The choice is between *orchestration* versus *choreography*. If a choice for orchestration is made three tasks follow to orchestrate a process. These are defined using the web services BPEL [34].

CHAPTER 3

CASE STUDIES

Contents

We will illustrate the use of proposed methods in this thesis by applying to some case studies. The created models of those case studies are also used to compare and evaluate the methods.

3.1 Dealer Network

The Dealer Network case study that was taken from OMG Adopted Specification of SoaML [27]. The Dealer Network is a business community including three primary parties: the dealers, the manufacturers and the shippers. They are independent parties but they want to work together. Moreover, they have their own business processes and do not want to change their existing systems. A service oriented architecture is required to be designed to enable this business environment.

3.1.1 Business Dealer Network

In the business Dealer Network, a manufacturer sells some products, a dealer buys some products from a manufacturer, and a shipper delivers some packages to a dealer. The goals of this business are to keep those entities working together conveniently and to satisfy the needs of them as soon as possible. For the dealers, they can buy things with best price and receive the products fast. For the manufacturer, they can sell the products to the buyers at the wished prices and fast. For the shipper, they can have shipping orders as many as possible and fast.

There are two relevant processes in this business.

Process Buying goods

We define a process, named Buying goods, that will be performed by the entities in business Dealer Network as following:

A dealer wants to buy a product from a manufacturer. He may ask for the price of the product. He will send an order to the manufacturer to buy the product. When the manufacturer receives the order, he will check the product availability.

If the product is available, the order will be accepted, otherwise, the order will be rejected.

The dealer will pay for the order. Once the order is paid, the manufacturer will send a confirmation to the buyer, and will ask a shipper to deliver the product to the dealer.

If the order is not paid, the manufacturer will cancel the order.

When the shipper receives a shipping order from the manufacturer, he will inform the manufacturer of the date for picking up the shipment. Once the shipment is delivered to the dealer, the shipper will send a confirmation to the manufacturer.

Variants: The manufacturer may send shipping order to some shippers. The shippers will inform of the price and expected delivery date. The manufacturer may choose one of shippers and send him a request for delivering the product.

Process Handling reputation information

The goals of the process Handling reputation information are to define and handle the reputation information of some parties in Dealer Network in order to minimize the risk in the transactions of the business. The various parties may express their satisfaction on the services and the products provided by other parties with whom they interact.

For instance, a dealer may express his satisfaction on a product, on a manufacturer, and on the quality of the service of a shipper.

Based on the reputation of the shippers, the manufacturers may choose a good one to deliver their products to the dealers. Based on the reputation of the manufacturers and the products, the dealers may buy good products and meet less risk.

The reputation of the manufacturers and the shippers is managed by an Authority. This Authority can receive the evaluation of those parties from the dealers, then make available the information to the community.

3.1.2 DealerNetwork Service system

Dealer Network service system has to be developed to automate the business Dealer Network described above.

There are three services to be built: Place Order, Get Ship Status and Request Shipping.

The dealers use service Place Order to make an order for the product that they want to buy from the manufacturers. The service supplies the dealers with the quote of the product and issues a confirmation when the order is accepted. If the order is confirmed, the dealer will receive further information about the order, such as the waybill number to track the shipment information.

The manufacturers use service Request Shipping to have the products relative to an order delivered to the dealer by a shipper. The delivery schedule is produced by the shipper. The shipper informs the manufacturers of the package picking date and the delivery date. Then,the manufacturer will receive the delivery confirmation when the dealer receives the products.

The dealers use the Get Ship Status service to get the information about the status of the shipment for the products they ordered (identified by the waybill number that he received from the manufacturer).

3.2 Office Service system

Office System is a Microsoft Office-like system. There are various components in Microsoft Office system, but we consider only some of them. Our case study includes a set of components supporting office works such as printing, checking spelling and grammar of a given language and publishing a web page. We name *application* a (not better specified) component using them. There are the following services to build: Print, Check French, Check English, Check Italian and Publish on Web.

Service Printing provides the application means to require the printing of a document, this service can communicate the result of the request such as no paper, document is not in A4 format or document printed. If the paper is not in A4, then the service will receive a request to reduce the page or to cancel the printing from the user.

Three services allow to check the spelling and the grammar of a text written in specific language: English, French or Italian. A service of this kind will refuse to do the check, if the text is not in the language it is able to handle. The result of the checking will be the list of the found errors.

Service Publish on Web allows to publish web pages on the Internet. The web page must be in HTML format to be published. The service will refuse to publish a page not written in correct HTML format. Them if the provided URL is correct and the corresponding Web server is available, then the service will publish the page, otherwise it will informs of the problems the service user.

We chose the services provided by the set of software tools composing a Office-like suite because they are simple but complete enough to show many of the modeling concepts and extensions defined in PreciseSOA (see Sect. 6), also in Casl4Soa (see Sect. 7).

Chapter 4

A View on Services and Service Systems

In this chapter we briefly present our view on services and service systems, that will be the basis of the work in the following chapters.

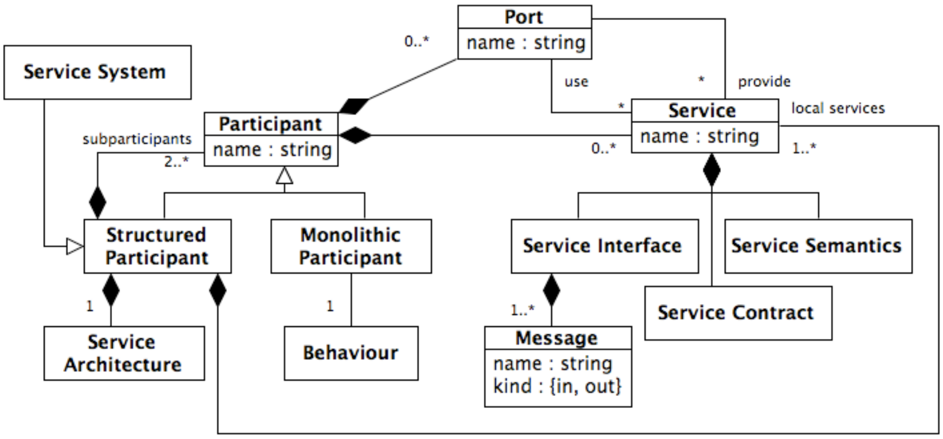


Figure 4.1: Service Systems Conceptual Model

Fig. 4.1 presents our conceptual view of services and service systems by means of a UML model.

We classify the participants into two kinds, structured participants and monolithic participant. A *structured participant* may include some inner participants (*subparticipants*), whereas this is not possible for a monolithic participant, but it can be still structured for example in terms of components.

A *participant* is characterized by a number of *ports* (where a port represents an interaction point where a participant consumes or provides a service), a number of services, those that it provides or uses at some of its ports. A structured participant is characterized also by the set of its subparticipants. A monolithic participant should be associated with a description of its behaviour. A *service system* is then a structured participant neither offering nor requiring services (and thus without any port), so a service system is a special case of a participant.

The *service architecture* presents how the subparticipants of a participant P provides and uses their services to allow P to provide its services, using obviously the services required by P.

A service is characterized by *service contract*, a *service interface* and a *service semantics.*

The service interface provides the static information needed to interact with the service. A service interface is conceptually seen as a set of in and out messages, were each message is characterized by a name and a list of typed parameters; the in messages are used to require the service functionalities to the service provider and the out messages to answer to such requests.

The service contract focuses on the protocol between the provider and the consumer of the service. The service contract specifies conceptually which are the allowed interaction between who requires the service and who provides it, and it may be represented by a labelled transition tree where the transitions are labelled by messages. All the complete paths (i.e., going from the root till a final node) on such tree should start with a transition labelled by a in message. Reaching a final node on such tree means that the service has terminated its activities started as a reaction to the reception of a request from a consumer.

The semantics tries to define which are the functionalities offered by the service. We assume that a service is able to act over a portion of the real world, that we name the *realm* of the service, and that it may modify such realm as the result to having received an in message form a service user, and that its answers to who uses the service (out messages) depend on the current status of the realm. Thus, the semantics of a service consists of a description of the realm, and of a refinement of the labelled transition tree representing the contract where the transition are equipped with conditions on the realm, and with the description of possible modifications of the realm itself.

Given a service interface SI, we denote by $\tilde{\text{SI}}$ the conjugate of SI. The messages of $\tilde{\text{SI}}$ are those of SI but after having changed the input kind in output kind and vice versa on each message.

A participant is a service provider if it offers a service and it is a service consumer if it uses a service. A participant may provide and consume any number of services. It means that the same participant may be "provider" of some services and a "consumer" of others.

A port must be typed by a service interface.

A *service architecture* defines which are the possible roles for subparticipants and how the pairs of ports of such (roles for) participants are connected (obviously only if typed by a pair of conjugate service interfaces) to offer and use services, defining also which are the handled services.

The conceptual schema presented in Fig. 4.1 could be the starting point to produce an informal model of a service system, that we name *conceptual level model.* It will consists of the lists of participants, ports and services, and of informal presentation of the services constituents (interface, contract and semantics) by means of list of messages and labelled transition trees, of the service architectures (by a graphs whose nodes are labelled by participant roles and whose arcs are labelled by service names), and of the behaviour of the monolithic participants.

This kind of informal model will be the starting point to prepare a model of the service system using the UML or the algebraic specification language CASL4SOA.

Fig. 4.2 and 4.3 present by means of two UML activities the guidelines for helping the modeller to produce a conceptual level model of a service system.

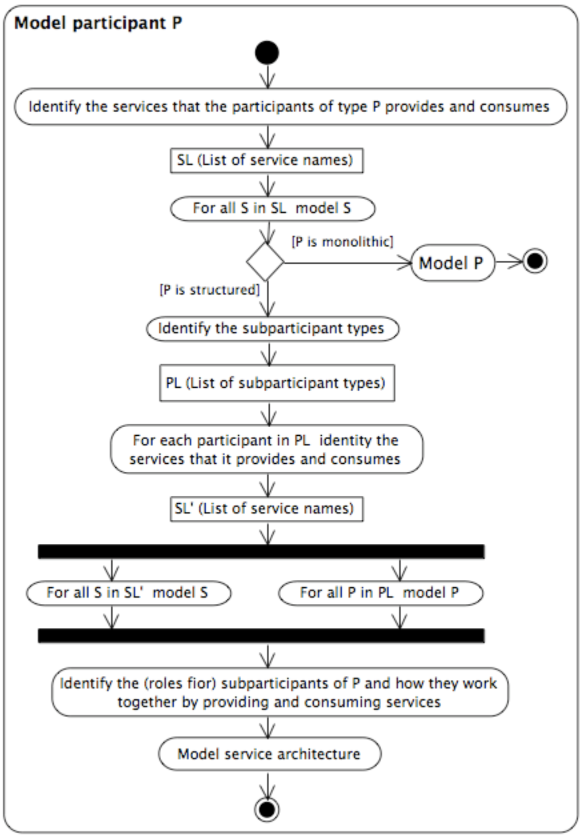

Model participant P
Identify the services that the participants of type P provides and consumes
SL (List of service names)
For all S in SL model S
[P is monolithic]
Model P
[P is structured]
Identify the subparticipant types
PL (List of subparticipant types)
For each participant in PL identity the services that it provides and consumes
SL' (List of service names)
For all S in SL' model S
For all P in PL model P
Identify the (roles fior) subparticipants of P and how they work together by providing and consuming services
Model service architecture

Figure 4.2: How to produce a conceptual level model of a participant

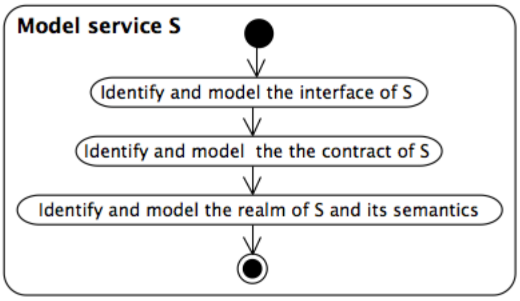

Model service S
Identify and model the interface of S
Identify and model the the contract of S
Identify and model the realm of S and its semantics

Figure 4.3: How to produce a conceptual level model of a service

CHAPTER 5

BUSINESS MODELLING

Contents

This fifth chapter aims to give a general introduction to business and business process, and give the description of the precise business modelling method that we propose. In later chapters, we will introduce our approaches to the modelling and the designing of a service system being built to support a business and its processes. As in other cases in the thesis, for presenting the models of a business, we use a UML profile, defined by a set of stereotypes, and we explicitly define the form of the models by means of a metamodel and a set of well-formedness rules. The content of this chapter is based on [3].

5.1 Business and business process

There are several definitions of business and business process. Each definition is useful in a specific context, none is agreed in the software industry. We give here some definitions of the meaning of the business and business process.

Definitions given by Wikipedia[1]*:*

"A **business** (also known as enterprise or firm) is an organization involved in the trade of goods, services, or both to consumers. Businesses are predominant in capitalist economies, where most of them are privately owned and administered to earn profit to increase the wealth of their owners."

"A **business process** is a collection of related, structured activities or tasks that produce a specific service or product (serve a *particular goal*) for a particular customer or customers. It often can be visualized with a flowchart as a sequence of activities with interleaving decision points or with a process matrix as a sequence of activities with relevance rules based on the data in the process."

Definitions given in Business Dictionary[2]*:*

A **business** is "an organization or economic system where goods and services are exchanged for one another or for money. Every business requires some form of investment and enough customers to whom its output can be sold on a consistent basis in order to make a profit. Businesses can be privately owned, not-for-profit or state-owned."

A **business process** is "a series of logically related activities or tasks (such as planning, production, or sales) performed together to produce a defined set of results."

[1] http://en.wikipedia.org/wiki/Business

[2] http://www.businessdictionary.com/

In a specification by OMG, i.e., Business Motivation Model version 1.1 ([23], section 7.3.9, page 15), they define what are business processes:

Business processes realize courses of action. Courses of action are undertaken to ensure that the enterprise makes progress towards one or more of its *goals*. They provide processing steps, sequences (including cycles branches and synchronization), structure (decomposition and reuse), interactions, and connections to events that trigger the processes.

We use the concept of *"business process"* to denote the activities performed within an *"business"* for some specific goals. The term *business* is used in a general meaning, it is dedicated to *companies*, *organizations*, etc. We consider an enterprise as whatever organization, formally or informally established, that performs specific activities for specific goals (not only monetary profit goal). Here are some samples of enterprise: companies, banks, universities, government agencies, and non profit associations.

In our opinion, a business process has a well defined goal and consists of a set of activities joined to realize that goal. The goal is the reason the enterprise does this work, it should be defined in terms of satisfying the business needs. A business process will typically produce one or more outputs of value to the business, either for internal use or to satisfy external requirements. An output may be a physical object, a transformation of raw resources into a new arrangement or an overall business result such as completing a customer order. An output of one business process may feed into another process, either as a requested item or a trigger to initiate new activities.

In a business there is a number of entities which perform the activities or are handled by the activities, that we will call *business entities*. The business entities may be human beings, existing softwares and hardwares, and physical/logical elements. For example, in a bank, the business entities are the clerks, the bank accounts, and the ATM machines, etc. The activities that the business entities perform are considered as *business tasks*.

Our first aim is to define a mechanism for defining and modelling a business and its business processes. In fact, there are a lot of business analysts studying the way companies work and defining business processes with simple flow charts, such as, BPMN 2.0 [24]. We are also comfortable with visualizing business processes in a flow-chart format, but moreover, they should be modelled in a precise way that does not lead the readers to the confusion.

Our purpose is to transfer a given description of the business to a model by means of the UML notation. Using the UML, we can take advantage of its numerous features and supporting tools, and also define our own methods for modelling. The UML offers a very large set of diagrams, e.g., activity diagrams, class diagrams, and use case diagrams, and a large set of constructs to build such diagrams. UML is a very large notation, the subset used in this theses has been reported in Appendix A.1, together some additional well-formedeness rules on models that help to avoid common errors and to produce “precise” models.

5.2 A precise method for business modelling

The main ingredients of a business are the entities taking part in the business itself (business entities) and the processes that they perform to reach some goals of the business (business processes).

We classify the business entities into three types:

- *business workers* that are human entities performing activities in some business process (e.g., a clerk, a buyer, an agency),
- *business objects* that are logic and physical things being handled in some business processes (e.g., an account, an invoice, packages, goods),
- and *systems* that are existing software or hardware systems involved in some business process (e.g., ATM machine, Paypal, DBMSs).

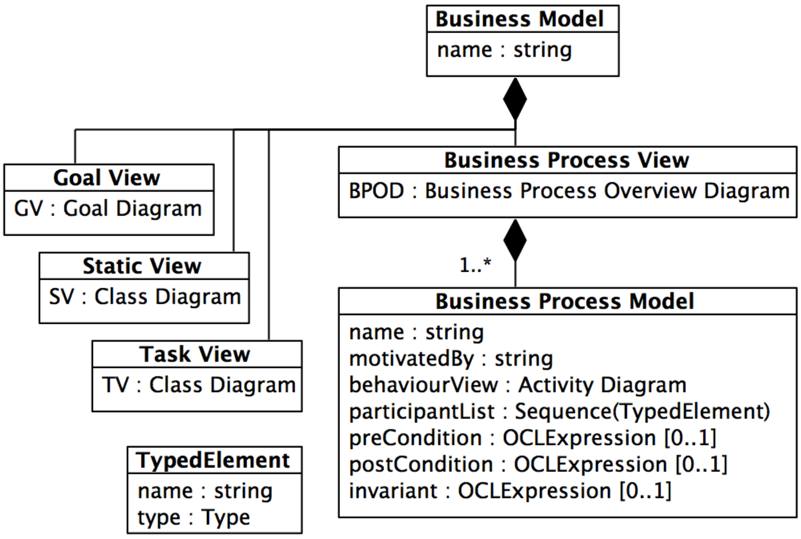


Figure 5.1: Business Model Metamodel

The metamodel in Fig. 5.1 shows the overall structure of a business model, whereas the well-formed constraints are given in Table 5.1. A business model (denoted by BMOD) is organized in four views, where each view shows a relevant aspect of the business. They are: the Goal View, the Static View, the Task View, and the Business Process View.

A business is motivated by the need to reach some goals, and all its business processess must be motivated by those goals. The goals of the business are described in the Goal View of the BMOD. The Goal view will lead the business activities and the determination of the business processes. The Static View describes the types of all the entities that participate in the business and all the data they handle. The Task View defines all the basic tasks of the business. The Business Process View presents all business processes that define how the business is made by means of the Business Process Overview Diagram. The Business Process View includes a number of Business Process Models. Each Business Process Model presents a process of the business. It is motivated by some goals, is represented by one Behaviour View, and includes the list of the entities taking part in it; there may be some conditions of kind invariant, pre and post about the entities of the processes.

Goal View

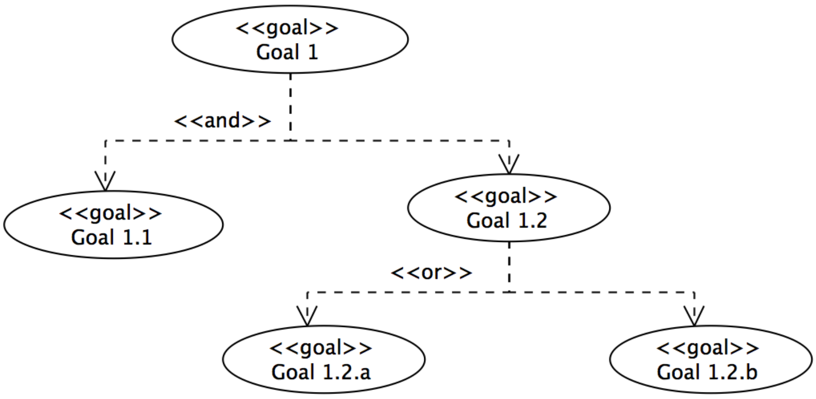


Figure 5.2: A generic Goal View

The Goal View is a UML use case diagram (described in Appendix A.1.5), in which, the use cases present the goals of the business and are stereotyped by ≪goal≫ (see an example in Fig. 5.2). Each goal has a name that describes the full meaning of this goal.

The goals of a business may be described at different levels, i.e., from high to low of the generalization level. There may be many goals for a business at the top level, and the others are subgoals needed to be satisfied to reach the main goals. A subgoal may belong to a number of other (sub)goals. The dependence is the only relationship among the goals, and it must be stereotyped by ≪and≫ or ≪or≫. While a goal G depends on a number of goals G_i $(i = 1, \ldots, n)$, if it is necessary to get all the subgoals G_i $(i = 1, \ldots, n)$ for getting G, the dependencies between the goal G and its subgoals G_i are stereotyped by ≪and≫; if it is sufficient to get at least one among G_i $(i = 1, \ldots, n)$ for getting G, the dependencies between the goal G and its subgoals G_i are stereotyped by ≪or≫.

Static View

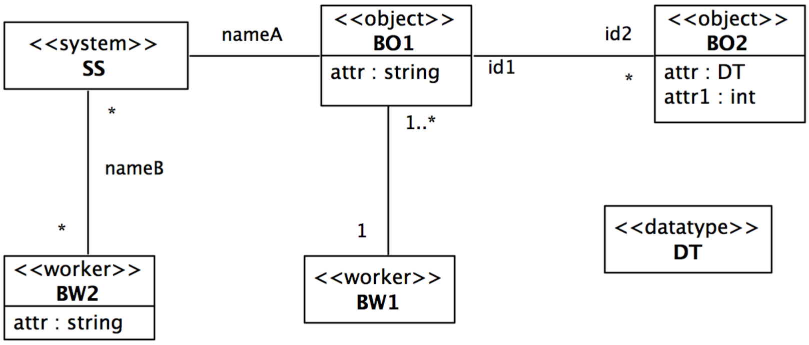


Figure 5.3: A generic Static View

The Static View is a UML class diagram that defines the types (i.e., the classes) of all the business entities belonging to the business and their relationships (see an example in Fig. 5.3). The business entities classes are further classified by the stereotypes ≪worker≫ (human entities performing the autonomous actions of the business), ≪system≫ (existing software/hardware systems taking part in the business), and ≪object≫ (things over which the business actions are performed).

The data used by the business processes (called *process data*) are defined by datatypes in the Static View. The datatypes used to define the entity classes are also included in the Static View.

So the Static view includes the classes typing all business entities, and all the needed datatypes. The behaviour and the details of the classes present in the Static View may be defined using the various features of the UML, such as constraints, methods and state machines.

Task View

The Task View is a UML class diagram (described in Appendix A.1.4) including various classes modelling the basic tasks of the business processes and any other classes needed to describe them. An example of a Task View is shown in Fig. 5.4.

A type of basic tasks in the Task View is represented by a class stereotyped by ≪task≫, called *task class*, (e.g., Task1 in Fig 5.4).

A task class must be connected by means of associations with the classes typing the business entities taking part in such task. An association between a task class and another class may be named, e.g., name1 in Fig. 5.4, and it may have the association end (presented in Appendix A.1.4.1) typed by the entity class named, e.g., id2 in Fig. 5.4. The association names or the association end names will allow to refer to the entities taking part in the task; for example given T an instance of the task class Task2 in Fig. 5.4, T.id will denote the entity of class BW2 taking part in T.

If a task generates a new business entity, the association between the task class and the class of the created entity must be stereotyped by ≪out≫.

A task class may have any number of attributes typed by datatypes; they represent the data handled by the task. If a task generates a value, then the attribute representing it must be stereotyped by ≪out≫ (e.g., Task2 in Fig. 5.4 will generate a new string).

A task class may be characterized by means of pre/post/invariant constraints concerning the entities taking part in such tasks and handled the data. Each of these constraints will be expressed using the OCL (presented in Appendix A.2). Moreover, the behaviour of the task may be detailed by means of an activity diagram.

Business Process Overview Diagram

The Business Process Overview Diagram, presenting an overall view of all processes of the business, is a UML use case diagram, in which a use case stereotyped by ≪process≫ represents a business process, an actor represents a role for business entities of a given type (e.g., Y:ClassY in Fig. 5.5). The use cases have the names of the business processes (e.g., BP1 in Fig. 5.5). The actors have the names of the roles of the entities taking part in the processes, Name:ClassName, where Name is the name of the role, and ClassName is the name the class typing that role. A role for human entities is represented by a stick man (e.g., X:ClassX in Fig. 5.5), a role for

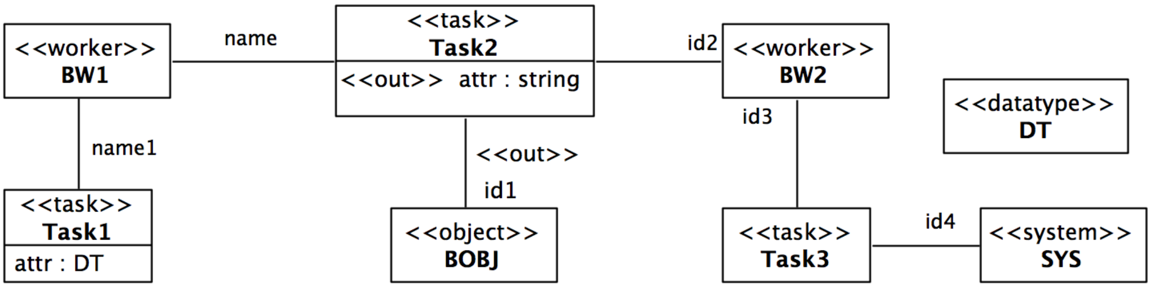


Figure 5.4: A generic Task View

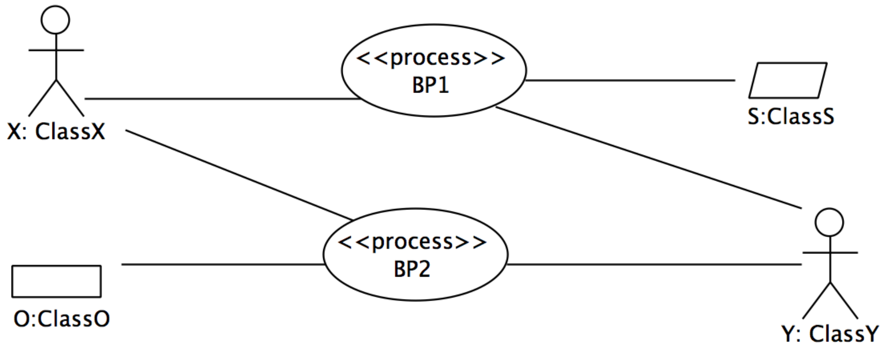


Figure 5.5: A generic Business Process Overview Diagram

systems is represented by a parallelogram (e.g., S:ClassS in Fig. 5.5), and a role for business objects is represented by a rectangle (e.g., O:ClassO in Fig. 5.5).

Note: For the readability of the Business Process Overview Diagram, the the roles typed by classes stereotyped by ≪object≫ may be hidden.

Business Process Model

A Business Process Model has the name of a business process. A business process must be motivated by at least one of the goals of the business, therefore the names of the goals that motivate that business process are used to fill attribute the motivatedBy in the metamodel (see Fig. 5.1). There is a specific number of entities taking part in a business process, and they are typed by the classes defined in the Static View stereotyped either by ≪worker≫ or ≪system≫, or ≪object≫. There may be some *process data* in a process, and they will be typed by datatypes defined in the Static View or by primitive types.

The behaviour of those entities in the process (and thus the behavioural aspects of the whole process) are presented by means of an activity diagram (described in Appendix A.1.8) in the Behaviour View of a business process model. The list of the typed roles for the entities taking part in the modelled process should be written explicitly and attached to the activity diagram by means of a UML note. If there are some process data in that process, they will be also listed in that note.

An action node in the behaviour view corresponds to a basic task, and it is labelled by an instance of this task class.

We will use the following notation to represent instances of task classes.
Let TC be a task class that has n roles for business entities:

R_1:CE_1, …, R_n:CE_n (CE_1, …, CE_n are classes defined in the Static View),
and m attributes typed by datatypes:

P_1:DT_1, …, P_n:DT_m (DT_1, …, DT_m are either defined in the Static View or are

primitive types),
then

$\mathsf{TC}<\mathsf{R}_1 \leftarrow \mathsf{E}_1, \ldots \mathsf{R}_n \leftarrow \mathsf{E}_n, \mathsf{P}_1 \leftarrow \mathsf{V}_1, \ldots, \mathsf{P}_m \leftarrow \mathsf{V}_m>$

is the instance of TC where the various roles are played by $\mathsf{E}_1, \ldots, \mathsf{E}_n$ and the used data are $\mathsf{V}_1, \ldots, \mathsf{V}_m$. Whenever there are neither two roles typed by the same class nor two parameters with the same type, it is possible to drop the role names and the parameters name and the attached arrows, and write within the angular parenthesisonly the entities and the values instantiating the task.

The instances of a task class will be used to label the action nodes of the behaviour views of this business processes.

To improve the readability of the business model:

- The Static View can be shown together with the Task View (i.e., the elements of the two views are shown together in a unique class diagram);
- For each process, we can add some diagrams that are fragments of the Static View and the Task View to show only classes relative to this process, where some classes, some relationships, and some details of classes, e.g attributes, are hidden.

Goal View	– A dependency relationship between the goals must be stereotyped by either ≪and≫ or ≪or≫ – All use cases must be stereotyped by ≪goal≫ – A goal cannot be decomposed both using ≪and≫ and ≪or≫ – There is no cycle built using the dependency relationships
Static View	– All classes should be stereotyped either by ≪system≫, ≪worker≫, or ≪object≫ – If a class is stereotyped by ≪X≫, and it is specialized by another class, the latter should be stereotyped by ≪X≫
Task View	– All classes should be stereotyped either by ≪system≫, ≪worker≫, ≪object≫, or ≪task≫ – A task class must have at least one association with one of the business entity classes – Only the attributes of a task class, and the entities associated with the task class by means of the associations may appear in the pre/post/invariant constraints of that task class

	– Neither an attribute stereotyped by ≪out≫ nor an entity role reachable by navigation with an association stereotyped by ≪out≫ may appear in a precondition expression – There is no association between two task classes in a Task View
Business Process Overview Diagram	– A process must have a relationship with at least an entity – There is no relationship between entities – All types of business entities involved in a process must be linked to this process.
Business Process Model	– The motivatedBy attribute must not empty and must be made of goal names – The list of entities taking part in a process must be presented in the Behaviour View by a note – The actions of the Behaviour View must be labelled with instances of task classes – The conditions of decision nodes must be built of process data

Table 5.1: Business model: Well-formedness constraints

- The name of a goal should have the form of a short sentence, and should start with a verb (such as "allow", "avoid", "prevent", . . .)

- The names of the roles of the entities of a process are written in capital letters

Table 5.2: Naming convention for business models

5.3 How to model a business

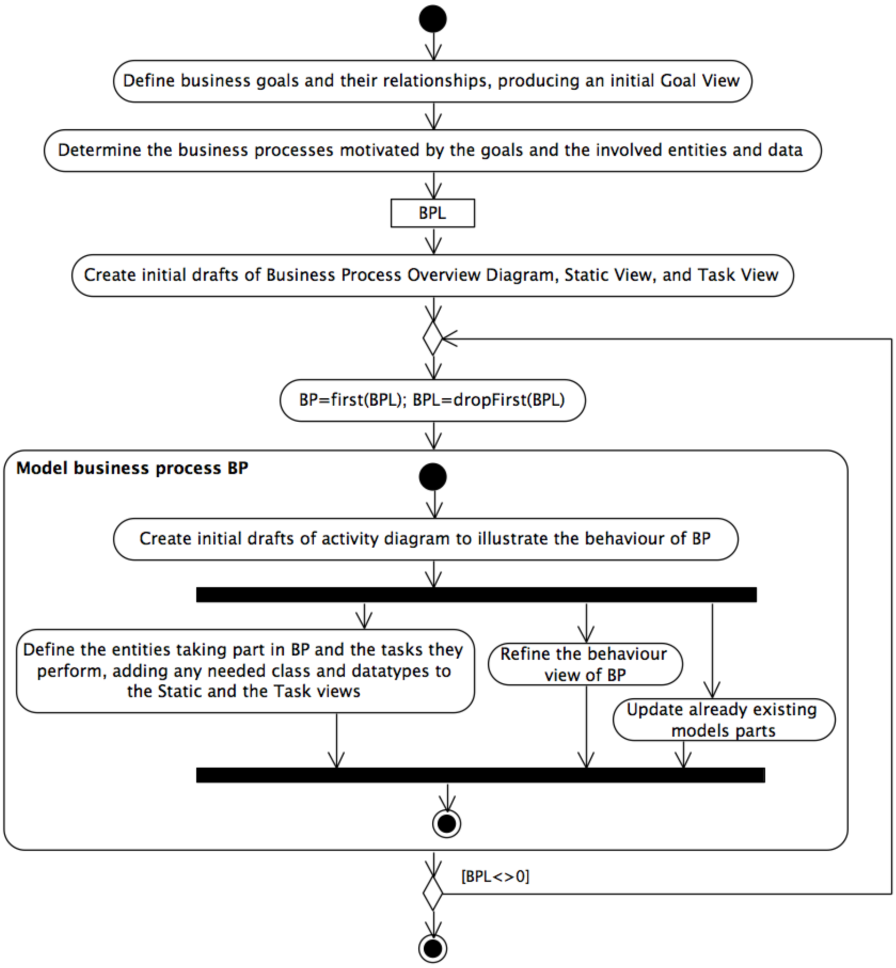

Define business goals and their relationships, producing an initial Goal View
Determine the business processes motivated by the goals and the involved entities and data
BPL
Create initial drafts of Business Process Overview Diagram, Static View, and Task View
BP=first(BPL); BPL=dropFirst(BPL)
Model business process BP
Create initial drafts of activity diagram to illustrate the behaviour of BP
Define the entities taking part in BP and the tasks they perform, adding any needed class and datatypes to the Static and the Task views
Refine the behaviour view of BP
Update already existing models parts
[BPL<>0]

Figure 5.6: How to produce a business model

We present a set of steps to model a business by the activity diagram in Fig. 5.6 The modelling of a business is not a sequential progress, there are some interleaved steps, and parallel steps. The main point of the progress is the refining the flow of tasks in a business process. The method leads us start from definition of business goals and the identification of business processes motivated by those goals. As a result, all the views of the business will be described by means of diagrams following the indication of Sect. 5.2.

First of all, we have to define the goals of the business and the relationships between them the goals to produce the Goal View. Next, we will define the business processes that are motivated by those goals. We will not build the Business Process Overview, the Static View and the Task View of the business directly, and in fact, it is not possible. We will start at some initial draft of those views, and of the activity diagrams presenting the behaviour of the business processes. Completing the activity diagrams corresponding to the behaviour view of the business process models, the other parts of the business model will be updated and finally completely defined.

Steps in creating initial drafts of activity diagram to illustrate the behaviour of a process

- Describe the process informally in natural language;

- Sketch an informal activity diagram to illustrate the flow of basic actions in this process (the action nodes are labelled by natural language);

- Define the entities taking part in this process. Create an initial list of entities;

- Define the logic entities that are handled by the process. Create an initial list of business objects;

- Define the tasks that the entities perform to reach the specific goals. Create an initial list of tasks;

Parallel steps in updating existing parts of the business model

- Refine the Behaviour View of each business process model. Define the constraints of the entities taking part in the tasks in the business processes by OCL expressions.

- Add new entities, new tasks, and any needed class and datatype to Static View and Task View.

- Add new business processes.

5.4 Modelling the Dealer Business

We illustrate the application of the business modelling method by modelling the Dealer Business.

Dealer Business model: Goal View

Fig. 5.7 shows the Goal View that describes the goals of the Dealer Business (cf. Sect. 3). There is a main goal, and it is decomposed in three subgoals by the ≪and≫ relationship.

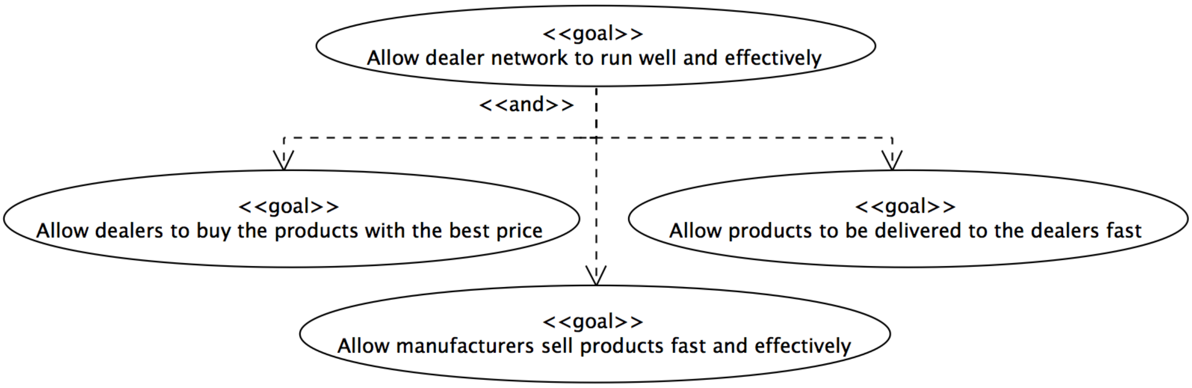

<<goal>>
Allow dealer network to run well and effectively
<<and>>
<<goal>>
Allow dealers to buy the products with the best price
<<goal>>
Allow products to be delivered to the dealers fast
<<goal>>
Allow manufacturers sell products fast and effectively

Figure 5.7: Dealer Business Model: Goal View

Dealer Business model: Business Process Overview Diagram

Fig. 5.8 shows the Business Process Overview Diagram of Dealer Business. There are two processes, one is Handling reputation information and the other is Buying goods (described informally in Sect. 3). Notice that for the moment we do not model the business process Handling reputation information. The model of business process Buying goods is presented in next few paragraphs.

There are four types of entities taking part in process Buying goods: Payment, Dealer, Manufacturer, and Shipper, where Payment is an existing application providing authorization to payments by credit cards, and the others are human entities.

Dealer Business model: Static View

The class diagram in Fig. 5.9 is the Static View of the model of Dealer Business. The human business workers, i.e., Dealer, Manufacturer and Shipper, are represented by classes stereotyped by ≪worker≫, and the existing payment applications are typed by the class Payment with stereotype ≪system≫. Those four types of entities perform the basic tasks in the processes over three types of business objects, precisely Order, ShippingRequest, and Invoice, which are represented by classes with stereotype ≪object≫. The relationships among them are modelled by the associations between the corresponding classes.

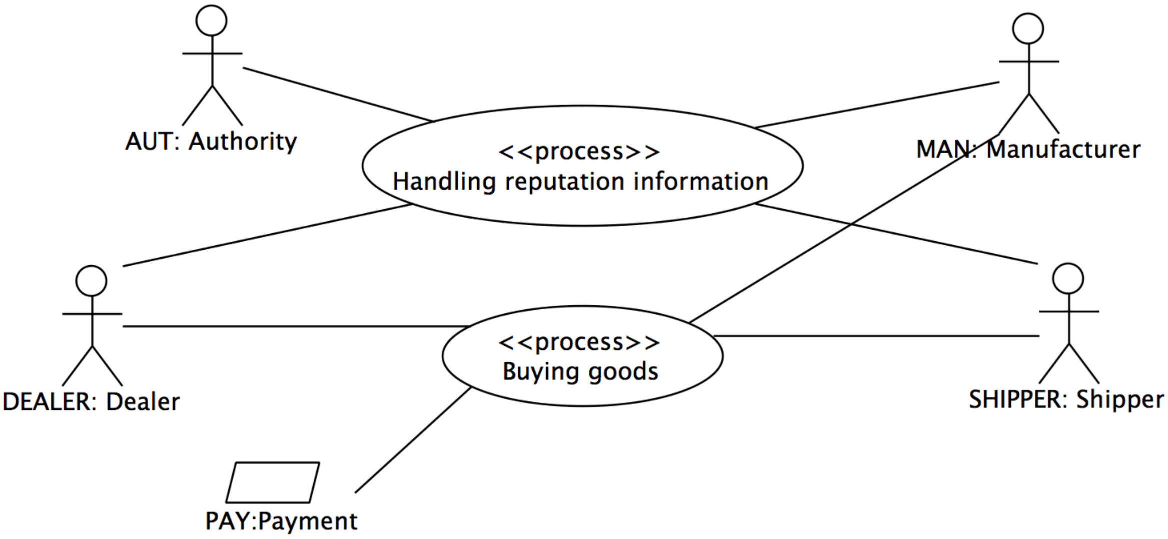


Figure 5.8: Dealer Business Model: Business Process Overview Diagram

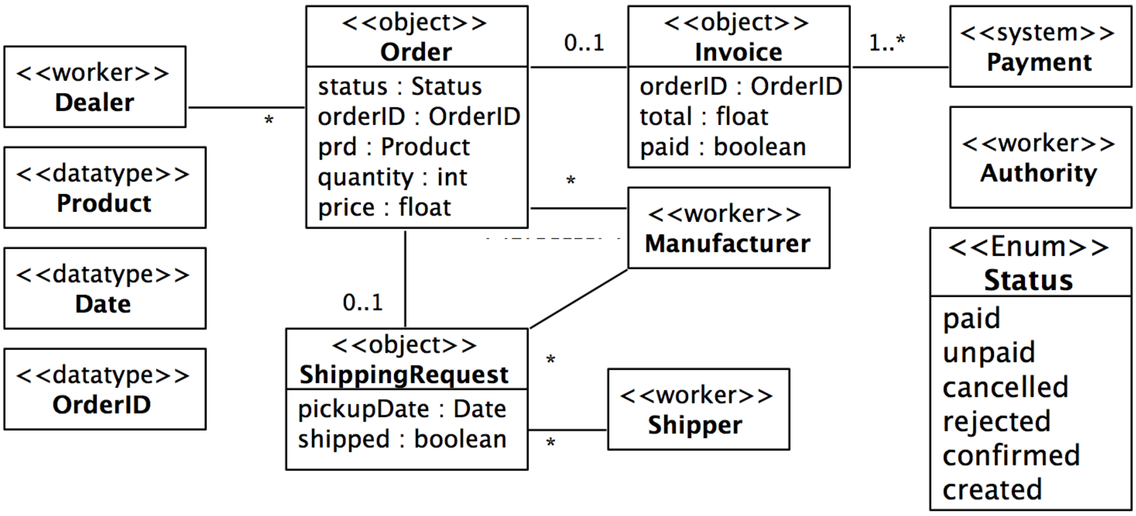
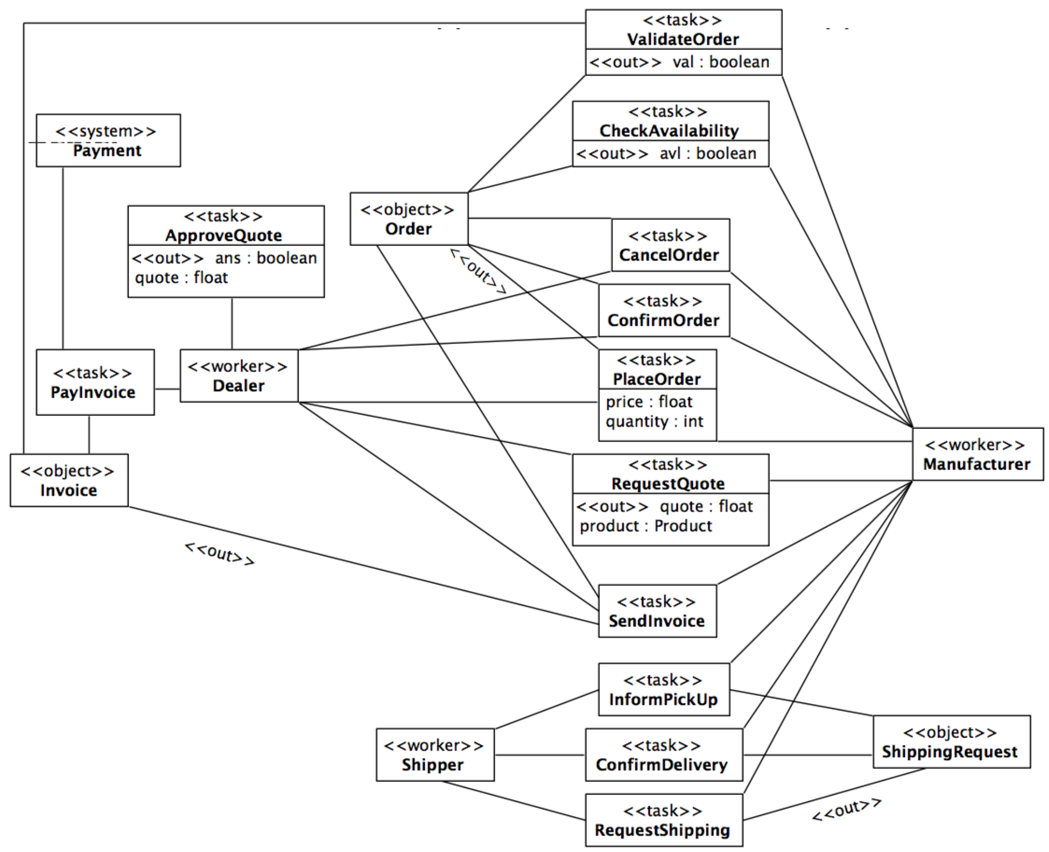


Figure 5.9: Dealer Network Business Model: Static View

Dealer Business model: Task View

All the basic tasks in the Buying goods process are modelled by means of classes with stereotype ≪task≫, they are presented together with the related business workers and business objects in Fig. 5.10, where the relationships between business workers and business objects are omitted. This class diagram is the Task View of the model of Dealer. The constraints on the tasks in the Task View are described by the OCL expressions. Some informal descriptions of the basic tasks are added, see Fig. 5.11.

Dealer Business model: Buying goods process model



ApproveQuote: Dealer approves the price of the product. The task returns a boolean value
CancelOrder: Manufacturer cancels an order and inform the Dealer
CheckAvailability: Manufacturer checks if the ordered products are available. The task returns a boolean value
ConfirmDelivery: Shipper informs Manufacturer that the ordered products have been delivered
ConfirmOrder: Manufacturer confirms an order to the Dealer
InformPickUp: Shipper informs Manufacturer of the date to pick up the ordered products
PayInvoice: Dealer pays pays an invoice using the Payment system
PlaceOrder: Dealer places an order to Manufacturer. The task returns an order
RequestShipping: Manufacturer sends a request to Shipper for asking a delivery. The task returns a shipping request
RequestQuote: Dealer asks to Manufacturer the price of a product . The task returns a price
SendInvoice: Manufacturer creates and sends an invoice of an order to Dealer ValidateOrder: Manufacturer validates an order checking whether it has been paid within 24 hours after the invoice was sent

```
context RequestQuote post:  quote>0
context ApproveQuote pre:  quote>0
context PlaceOrder pre:  quantity>0 and quote>0
post:
       order.status = created and order.price = price and
       order.prd = product and order.quantity = quantity
context SendInvoice post:
       invoice.orderID = order.orderID and
       invoice.total = order.quantity* order.price and invoice.paid = false
context PayInvoice pre:  invoice.order.status = unpaid
context ValidateOrder post:  order.status = invoice.paid
context ConfirmOrder
       pre:  order.status = paid
       post:  order.status = confirmed
context CancelOrder post:  order.status = cancelled
context RequestShipping
       pre:  order.status = confirmed
       post:  shippingRequest.orderID = order.orderID
context ConfirmDelivery pre:  shippingRequest.delivered = true
```

Figure 5.10: Buying goods Process Model: Task View

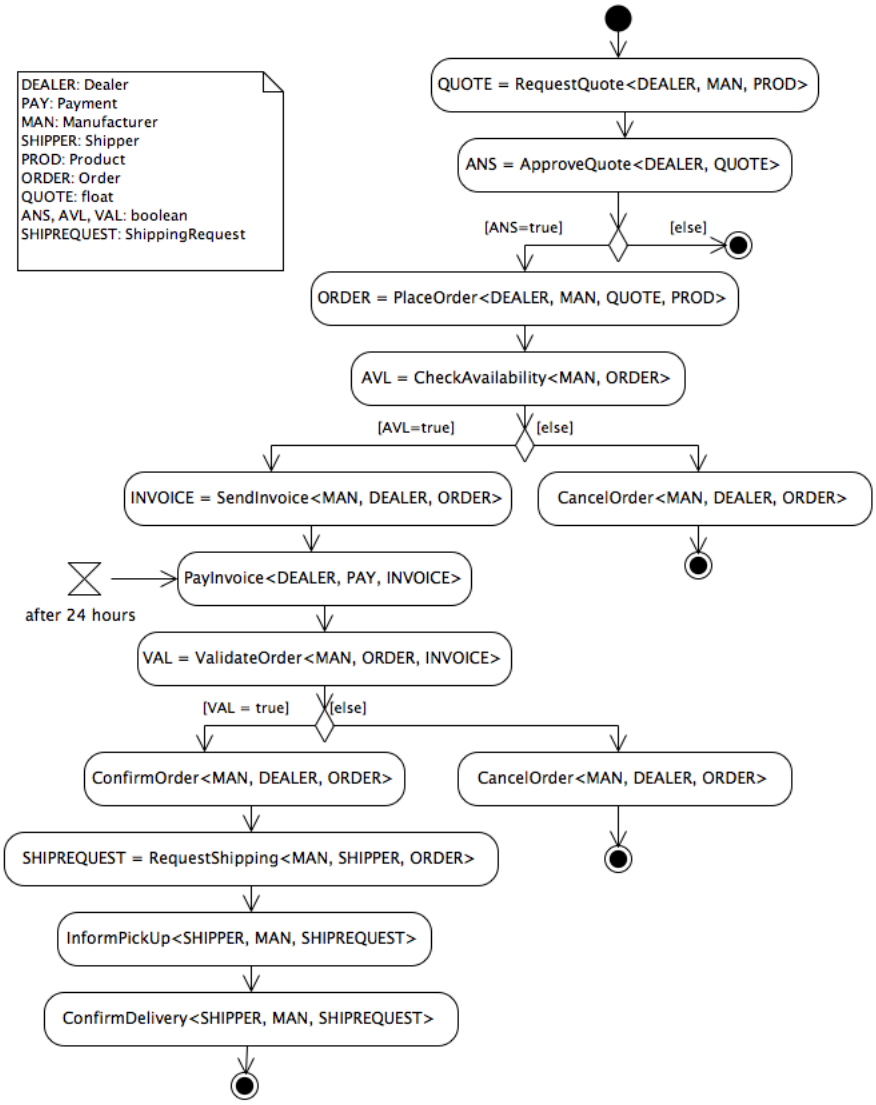

DEALER: Dealer
PAY: Payment
MAN: Manufacturer
SHIPPER: Shipper
PROD: Product
ORDER: Order
QUOTE: float
ANS, AVL, VAL: boolean
SHIPREQUEST: ShippingRequest
QUOTE = RequestQuote<DEALER, MAN, PROD>
ANS = ApproveQuote<DEALER, QUOTE>
[ANS=true]
[else]
ORDER = PlaceOrder<DEALER, MAN, QUOTE, PROD>
AVL = CheckAvailability<MAN, ORDER>
[AVL=true]
[else]
INVOICE = SendInvoice<MAN, DEALER, ORDER>
CancelOrder<MAN, DEALER, ORDER>
PayInvoice<DEALER, PAY, INVOICE>
after 24 hours
VAL = ValidateOrder<MAN, ORDER, INVOICE>
[VAL = true]
[else]
ConfirmOrder<MAN, DEALER, ORDER>
CancelOrder<MAN, DEALER, ORDER>
SHIPREQUEST = RequestShipping<MAN, SHIPPER, ORDER>
InformPickUp<SHIPPER, MAN, SHIPREQUEST>
ConfirmDelivery<SHIPPER, MAN, SHIPREQUEST>

Figure 5.11: Buying goods Process Model: Behaviour View

The activity diagram in Fig. 5.11 gives a description of the behaviour of the business process Buying goods. There are four roles for entities, i.e., DEALER, MAN, SHIPPER, and PAY of corresponding types: Dealer, Manufacturer, Shipper, and Payment. Those entities are reported by the note attached in the activity diagram. The tasks they perform are represented by action nodes, and the set of control flows between actions node is the flow of tasks in this process. The variables (e.g., the answer ANS) are declared as containers for data values. Since there are no tasks with two roles/two parameters with the same type, we used the short form for writing the task instantiations.

Conclusion: At the beginning of this chapter, we gave the definitions of a business and a business process in our opinion. For modelling them, we have proposed a method, in which a UML profile is defined and equipped with a set of well-formedness rules. These well-formedness rules keeps the users at applying the method precisely in order to produce precise models. We also build the guideline for the application, and visualized them by an activity diagram illustrating the steps required to be done. Finally, the application of the method is illustrated by a typical case study.

CHAPTER 6

PreciseSoa: A PRECISE METHOD FOR MODELLING SERVICE SYSTEMS USING UML

Contents

We propose a method to model service systems using the UML, called PreciseSoa. PreciseSoa has been inspired by SoaML (Service oriented architecture Modeling Language, c.f., Sect. 2.2). SoaML is the standard OMG profile to architect and model SOA solutions, adopted in May 2012 [27].

6.1 PreciseSoa

We present the structure of the PreciseSoa service system models by means of a metamodel shown in Fig. 6.1, whereas the well-formed constraints are given in Table 6.1.

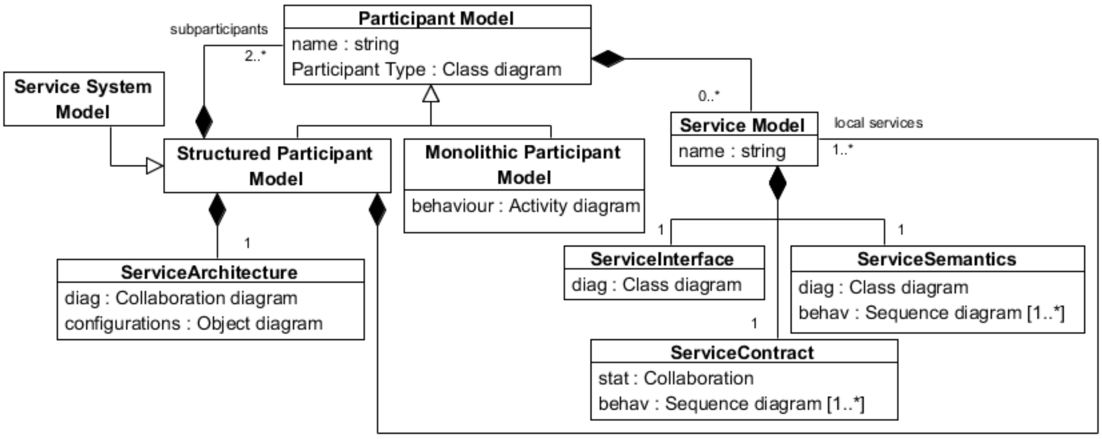


Figure 6.1: Service System model: metamodel

PreciseSoa provides two different kind of models for structured and monolithic participants; both offer the possibility to model the services required and provided by a participant, whereas for the structured participants it is possible to define the subparticipants and how they are organized into a service architecture, and for the monolithic participants instead it is possible to define their behaviour.

Recall that a *service system* is a structured participant neither offering neither consuming services (and thus without ports). Thus, we consider that a *service system model* is a special case of a participant model. A *participant model* includes a number of *service models*, one for each of the services that the participant consumes and provides. A *structured participant* model includes also some participant models (one for each kind of its subparticpants) and a *service architecture*. The behaviour of a *monolithic participant* may be depicted by an UML activity diagram.

The *service architecture* presents how the subparticipants of a particpant P interact among them using the services they provide and use, in the meantime they allow P to provide its services and obviously they use the services P uses.

A *service model* describe a service; the *service interface* define the messages together with the associate message data needed to use the service, distinguished in in and out, the *service contract* illustrates the allowed sequences of messages that can be received sent by the service itself, and the *semantics* the effect of the in messages on the service realm and how the realm itself influence the out messages.

In the case of a structured participant the models of its subparticipants are also part of its model.

Service Model	***Service Interface*** –A service interface Diagram consists of three interfaces (a service interface without operation, stereotyped by ≪Service Interface≫, the other two interfaces can have the operations) and any number of data types (possibly none). –A service interface uses and realizes two interfaces (one is the type of the role of service consumer and another is the type of the role of service provider). –All operations of the service interfaces should have “data type argument”. If a data type is not in kind of the primitive type, it must be defined. –The identify attributes in MessageType should have specific data types. ***Service Contract*** –The collaboration part of a service contract should be - named as the service itself, - have exactly two parts connected by a UML connector, one stereotyped by ≪use≫ and the other by ≪provide≫, and such parts must be typed by the interfaces realized or used by the service interface. –A sequence diagram representing the behaviour part of a contract should have exactly two lifelines corresponding to the two parts in the collaboration. –All messages in the behaviour sequence diagrams should be built by operations of the used and provided interfaces, and all these interface operations should appear at least in one of the behaviour sequence diagrams.

	–All interface operations appearing as the messages of the sequence diagrams should be shown in full forms with their parameters. ***Service Semantics***
Participant Model	–All services of a participant must have different names. –All sub participant models must have different names. – The interfaces realized and used by a participant must be attached at its ports. –A port must be typed by a service interface.
Service Architecture	–All services and participants should appear in the corresponding parts of the model. –Each service has to conform to a service Model. –Each service is bound to two participants by bindings of the corresponding collaboration use. –The bindings between parts and collaboration uses must be labelled by the nouns of the roles. –A structured participant has at least one port to interact with other participants in a service architecture. –The ports that are connected must be typed by the same service interfaces. ***Architecture Configuration*** -Each collaboration use is an instance of a service in the service architecture. -Each instance is an instance of a participant in the service architecture. -A collaboration use is bound to exactly two participant instances.

Table 6.1: PreciseSoa model: Well-formedness constraints

– The names of the roles of the participants should be written in lower case.
– The names of the roles of the participants should be ending with “er”.
– The names of the services and interfaces should be written in the case where each new word begins with a capital letter.

– The names of the operations of an interface should be written in mixed case starting with lower case. When there are more than two words in the name, use underscores to separate them.

– The parameters of an operation should be written in upper case.

– The names of data types must be nouns and written in the mixed case where each new word begins with a capital letter.

– The attributes of the data types should be written in mixed case starting with lower case.

Table 6.2: Naming convention for PreciseSoa models

6.1.1 Service Model

A *service model* consists of a name, a service interface, a service contract, and a service semantics. As stated in Chapter 4 a service interface present the in/ out messages that the service exchanges, a service contract specifies an agreement between who offers and who consume the service on how it is provided and consumed, and the semantics says what are the effects of the received messages on the realm of the service and what will be returned.

A *service interface* is defined by a class stereotyped ≪Service Interface≫ and named as the service itself. It should realize and use respectively two UML interfaces, defining the in and the out messages by means of operations. The operations of the interfaces correspond to the messages exchanged between the service and the participants using and providing it. The operations may have (in) parameters that must be typed by message types, but cannot have a return type. A *message type* is a UML datatype stereotyped by ≪Message Type≫. The attributes of a message type must be typed either by primitive type, or datatypes, or other message types. The definition of the needed message types should be given together with the the two interfaces, thus in Fig. 6.1 a service interface consists of a class diagram, that will include a class stereotyped by ≪Service Interface≫, the two interfaces and all needed message types.

Fig. 6.2 shows a generic service interface. The class stereotyped by ≪Service Interface≫ should realize the provided interface (represented by the UML realization symbol: the dashed arrow with closed head) and use the required interface (represented by the UML dependency: the dashed arrow with open head).

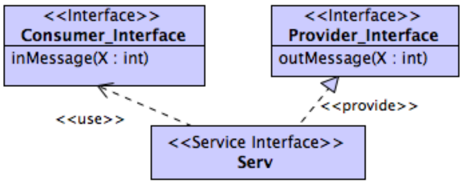


Figure 6.2: A generic service interface

A *conjugate* service interface is suggested as a mechanism to connect the consuming participant and the providing participant. Each service interface has one conjugate service interface that is named by the name of the corresponding service interface starting with " $\sim$ "; and it is defined transforming the in messages into out messages, and similarly the out messages into in messages, i.e., the realized interfaces becomes the used one and vice versa.

A *service contract* consists of a UML collaboration stereotyped by ≪Service Contract≫ and named as the service, and by a behaviour represented by a set of UML sequence diagrams. The collaboration has exactly two parts corresponding to the roles the service provider and the consumer, and the sequence diagrams have exactly two lifelines (one for the service provider roles and one for the service user role).

Fig. 6.3 shows a generic service contract. The dashed oval is the icon of the collaboration, whereas the inside boxes represent the collaboration parts and are used to model the roles of who provides and of who consumes the service (the stereotypes ≪use≫ and ≪provide≫ allow to distinguish the two roles). The parts are typed by interfaces. The sequence diagrams present all possible stories of the provider using the service showing which messages and in which order the provider and the consumer exchange in each story.

Thus, in Fig. 6.3, Prov1 is the role for provider and the Provider_Interface is the interface that it implements to play that role, whereas Cons1 is the role for consumer and Consumer_Interface is the interface that it implements to play that role in Serv. The two parts are connected by a UML connector, to emphasize that they will communicate. Serv is quite simple, and so after receiving an integer number will return the same number increased by 3, thus a unique sequence diagram is enough to model its contract.

The *service semantics* should be defined by modelling how the received messages will result in modifications of the service realm and how the current status of the

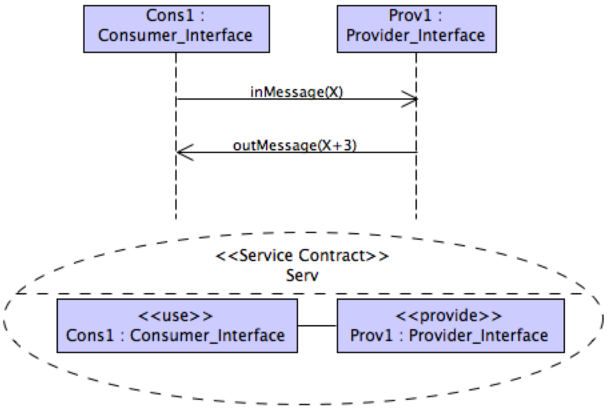

Cons1 : Consumer_Interface
Prov1 : Provider_Interface
inMessage(X)
outMessage(X+3)
<<Service Contract>>
Serv
<<use>>
Cons1 : Consumer_Interface
<<provide>>
Prov1 : Provider_Interface

Figure 6.3: A generic service Contract

realm influences the messages sent out by the service provider. In PreciseSoa we model the service semantics by introducing a class Service_Realm realizing the provided interface, its attributes will define the current status of the service realm. Then the sequence diagrams, i.e., the behaviour part of the service contract, should be refined by adding action specifications to modify the realm status and further guards to influence the choice of which messages to send out and which values they are carrying.

6.1.2 Participant Model

In PreciseSoa the participants of a service system may of different kinds and each kind is described by a specific participant model.

A *participant model* introduce a class stereotyped ≪Participant≫, that will be used to type the specific participants (instances) of that kind, and that we will name participant class.

A participant is a service provider if it offers a service and is a service consumer if it uses a service. A participant may provide and consume any number of services. It means that the same participant may be "provider" of some services and a "consumer" of other. The UML mechanism of the ports is used to indicate the points of interaction through which participants interactwith each others to enact services, and the needed ports are added to the participant class. There are two kinds of port that a participant class may have, one is stereotyped by ≪Service≫ known as a service port where a service is provided by participant of this type, one is stereotyped by ≪Request≫ where a participant makes a request for service from other participants. A port is then typed by a service interface. A service port has the type of the relevant service interface and the request port has the type of the conjugate service interface.

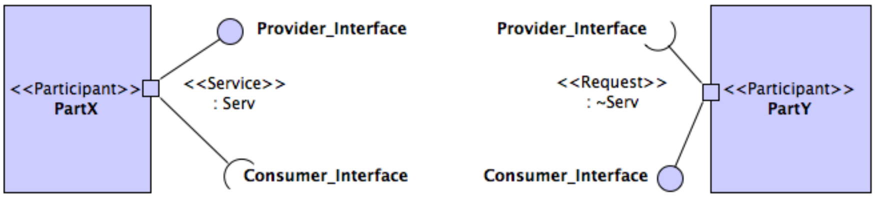


Figure 6.4: A generic participant view

All the models of the subparticipants of a structured participant are collect in a *participant view.*

Fig. 6.4 presents a generic participant view including two participant classes, The instances of the participant class PartX are the provider of Serv and thus the class has a ≪Service≫ port, typed by the conjugate service interface of Serv, denoted by ∼Serv. Participant class PartY types the consumers of this service and has a ≪Request≫ port. The port of PartX provides the Provider_Interface interface and requires the Consumer_Interface interface of Serv, and vice versa for PartY. These ports are the points for engaging two participants PartX and PartY to enact Serv.

6.1.3 Service Architecture

A *service architecture* is defined by a UML collaboration with stereotype ≪Service Architecture≫, as shown in Fig. 6.5, and by a set of *architecure configurations*. A service architecture consists of a set of services and a set of (roles for) participants that work together by providing and consuming services for some business goals.

A service is provided and consumed by two participants, and we use the wording "the service is used" or "service usage" to indicate it, this associated relationship is represented by a collaboration use of the collaboration part of the service contract (such as S1:Serv). There may be several usages of the same or of different services in a service architecture, and each of them involves a possible different set of roles (and of the related connectors). The name of a service in a collaboration use is structured of S: Serv, where S is optional (as in our example in Fig. 6.10) and can be a short name for service Serv, or be the same as Serv (Serv: Serv for example) as in SoaML examples (c.f, [27]), and Serv is the name of the service, suggesting its purpose. This structure abides the name of a UML collaboration element.

A participant role in the service architecture is displayed as a UML part (a solid rectangle) that contains the role name and the participant class typing the role, i.e., X : PartX, where X is the role name and PartX is the type of this role. The two participants, who play the providing and consuming roles in a service, will be defined in the contract of this service. The roles that the two participants play in a service usage, i.e., who is provider and who is the consumer, are represented by the labels on the dashed line connecting the parts and the collaboration use.

Fig. 6.5 shows a generic service architecture for a service system, i.e., a structured participant without any port, and thus unable to interact with the outside by means of service calls.

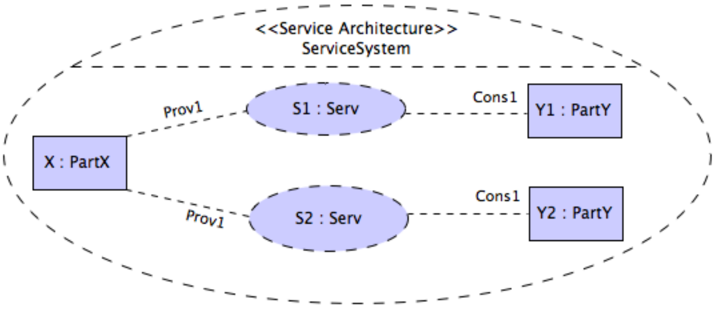


Figure 6.5: A generic service architecture for a service system

In case of a structured participant P offering and using services in the service architectures there will be also special parts, denoted by dashed boxes, representing the roles of the those using or providing services to P.

The service architecture of a generic structured participant is illustrated in Fig. 6.6. In this service architecture, the roles of other participants are demonstrated by the roles with dashed outlines (i.e., X : Part X), whereas the roles played by subparticipants within structured participant are normal roles.

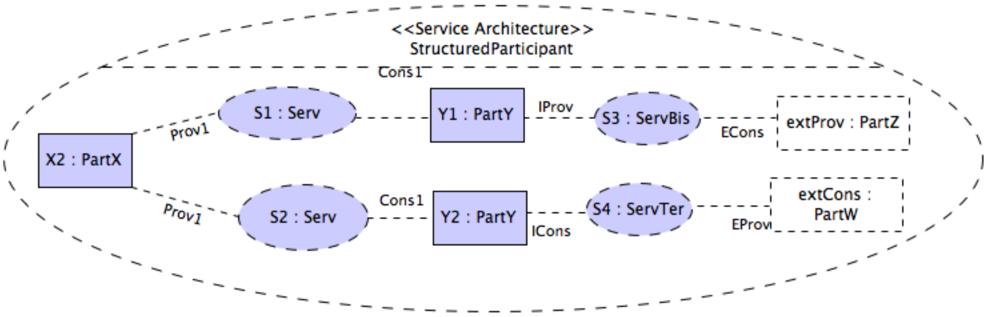


Figure 6.6: A generic service architecture for a structured participant

An *architecture configuration* shows a snapshot of the architecture at a specific point in time. A configuration is presented by a UML object diagram. It includes a set of participant instances and the services that they offer and use at that particular time. Each service is is represented by a use of the collaboration part of the definition of its contract. The links among them are bindings of the collaboration.

A structured participant class, say PC, should be a UML structured class having as subclasses the participant classes typing its subparticipants; all these classes will have their respective ports. There will connectors between the ports of the subparticipant classes and between the ports of the subparticipant classes and the ports of PC. The class in Fig. 6.7 is a generic structured participant class.

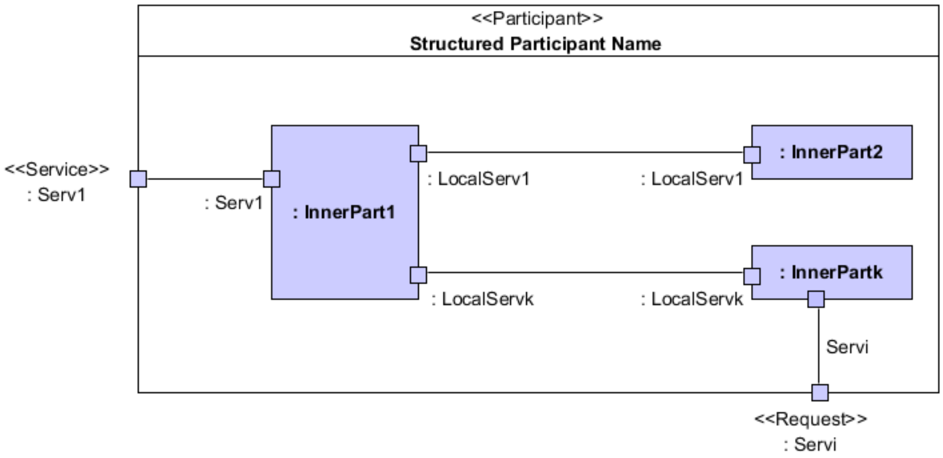


Figure 6.7: A generic structured participant

Monolithic participant Behaviour

A UML activity diagram used to model the behaviour of a monolithic participant consist of one initial state, sending actions, accepting actions and one final state. The sending actions are the operations of required interfaces of the services that it consumes, the accepting actions are the operations of provided interfaces of the services it provides.

6.1.4 How to produce a PreciseSoa model

Following the indications of Chapter 4 (see Fig. 4.2, 4.3), we present a set of steps to produce a PreciseSoa model of a service system being built by the activity diagram in Fig. 6.8. The method leads us to start from participants identification to the service architecture modelling. The list of participants should be indentified first, then the service that each participant consumes or offers will be defined, in result, a list of service names is identified, called SL. The contracts and interfaces of those services will be modelled by notations presented in Sect. 6.1.1.

For modelling a service S, let take a look at the actions of the activity named **Model S** in Fig. 6.9).

Model service S

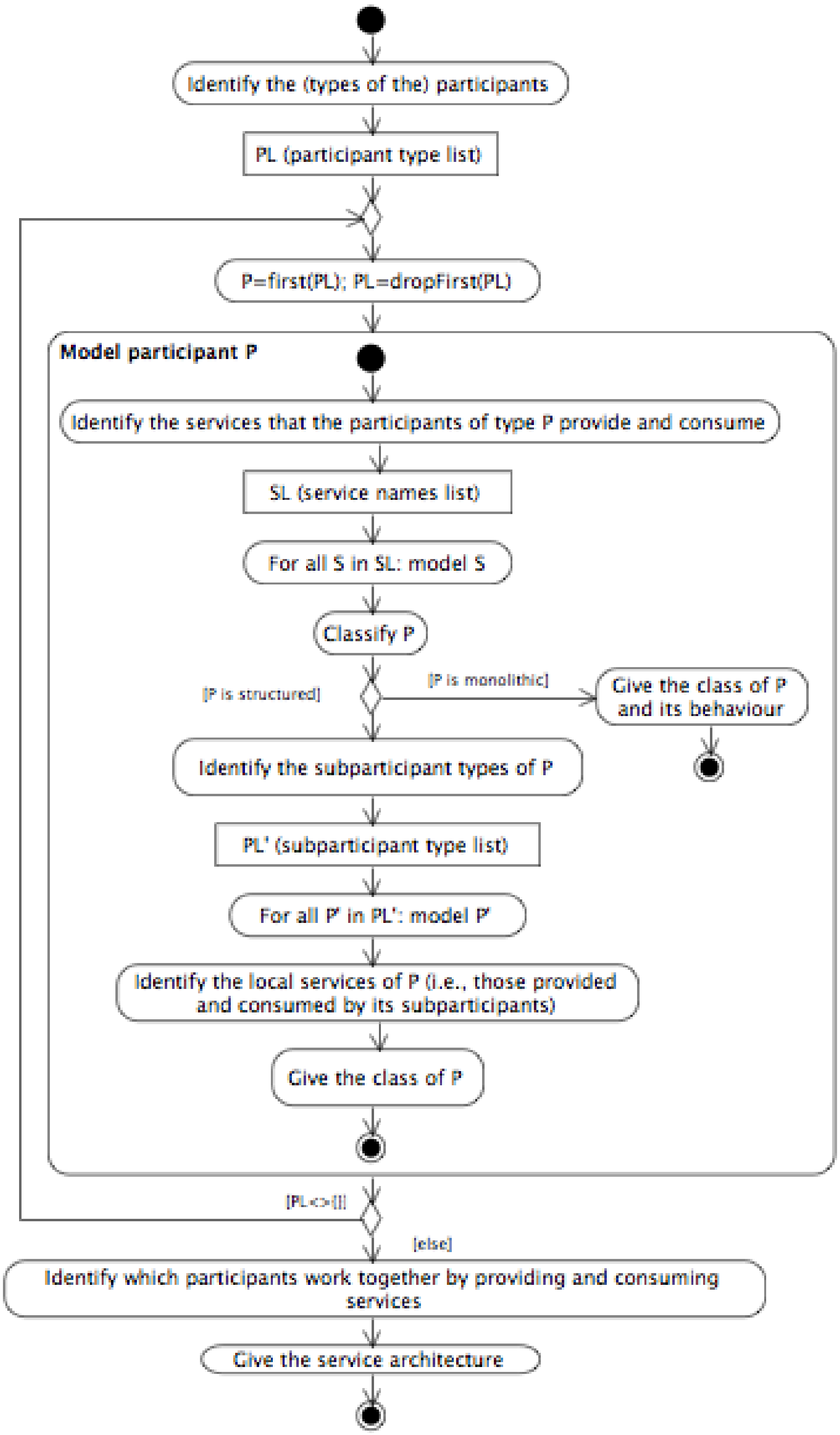

Identify the (types of the) participants
PL (participant type list)
P=first(PL); PL=dropFirst(PL)
Model participant P
Identify the services that the participants of type P provide and consume
SL (service names list)
For all S in SL: model S
Classify P
[P is structured]
[P is monolithic]
Give the class of P and its behaviour
Identify the subparticipant types of P
PL' (subparticipant type list)
For all P' in PL': model P'
Identify the local services of P (i.e., those provided and consumed by its subparticipants)
Give the class of P
[PL<>[]]
[else]
Identify which participants work together by providing and consuming services
Give the service architecture

Figure 6.8: How to develop a PreciseSoa model

Model the interface of S

- Identify the messages that consumer and provider exchanges when enacting S, and the relative message data.

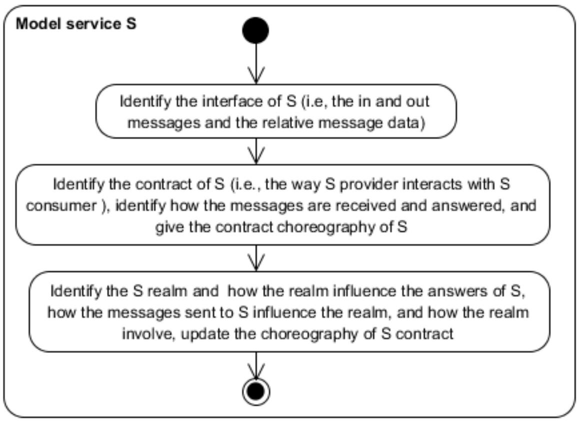

Model service S
Identify the interface of S (i.e, the in and out messages and the relative message data)
Identify the contract of S (i.e., the way S provider interacts with S consumer), identify how the messages are received and answered, and give the contract choreography of S
Identify the S realm and how the realm influence the answers of S, how the messages sent to S influence the realm, and how the realm involve, update the choreography of S contract

Figure 6.9: How to develop a service model

- Use classes with stereotype ≪Interface≫ to model: 1. required interface: includes the operations, i.e. *out* messages, that S consumer will receive, and 2. provided interface includes the operations, i.e., *in* messages, that S provider will receive.
- Use an empty class with stereotype ≪Service Interface≫ to model S interface. Connect the service interface to required interface by a dependence with stereotype ≪use≫ and to provided interface by a realization with stereotype ≪provide≫.
- Define all the message types by classes with stereotype ≪Message Type≫.

Model the contract of S

- Identify the roles of provider and consumer in S.
- Use collaboration with stereotype ≪Service Contract≫ to model the contract of S. Name the collaboration by the name of S itself.
- Type the role of provider by an interface type (called provided interface, this is the interface that the provider will require on a port to provide S). Model the provider role by a part stereotyped by ≪provide≫.
- Type the role of consumer by an interface type (called required interface, this is the interface that the consumer will implement on a port to consume S). Model the consumer role by a part stereotyped by ≪use≫.
- Use a sequence diagram having two life lines representing two roles of S provider and S consumer to illustrate the order of the messages exchanged between them using the messages between two life lines. Specify the

conditions for the occur of the messages to define the guard of combinators if any.

Model the semantics of S

- Identify the S realm, how the realm influence the answers of S, how the messages sent to S influence the realm, and how the realm involves.
- Refine the sequence diagrams, i.e., the part of S contract, by adding action specifications and further guards to modify the realms status and to influence the choice of which messages to send out and to define the values carried.

For modelling a participant P, let us describe the actions of the activity named **Model P** in the activity diagram (see Fig. 6.8) as following:

Model participant P

If P is structured participant, then:

- Use UML structured class with stereotype ≪Participant≫, named P, to model participant P.
- Identify all subparticipants of P, model each of them by a subclass without stereotype and named by the name of this subparticipant, put them inside ≪Participant≫ structured class P.
- Identify the ports of each subparticipant, in which it consumes and provides local services, typed them by the interfaces of those local services.
- Connect each pair of ports of two subparticipants by a connector for representing their cooperation in the enacting a local service.
- For a service that P provides/consumes, define a port of ≪Participant≫ structured class P, connect it to the port of the corresponding subparticipant which provide/consume this service, stereotype this port by ≪Service≫/≪Request≫ and type it by the interface of this service.

If P is monolithic participant, then:

- Use UML class with stereotype ≪Participant≫, named P, to model participant P.

- Identify the ports of P, in which it consumes and provides services, typed them by the interfaces of those services, stereotype ports by ≪Service≫ if P provides services, and by ≪Request≫ if P consumes them.
- For service S that P provides, use a *lollipop* of the corresponding ≪Service≫ port (i.e, port typed by S interface) for representing the provided interface of S and a *cup* for representing the required interface of S.
- For service S that P consumes, use a *lollipop* of the corresponding ≪Request≫ port (i.e, port typed by S interface) for representing the required interface of S and a *cup* for representing the provided interface of S.
- Use a UML activity diagram to illustrate the behaviour of P if necessary.

After we have in hand a set of modelled participant types and a set modelled services, define how participant types and services are combined to consume and provide services, then model the service architecture of the service system.

Model the service architecture

- Use collaboration with stereotype ≪Service Architecture≫ to model the service architecture of the service system.
- Identify all participant types, i.e, consumers and providers of services in the service system.
- Model each participant type by a part, put inside ≪Service Architecture≫ collaboration.
- Identify all services that the participants consume and provides.
- Model each service by a collaboration use, put inside ≪Service Architecture≫ collaboration.
- Bind each service, i.e., a collaboration use, with exact two participants, i.e., two parts representing one consuming and another providing it, the connectors are labelled by the roles of those two participants played in this service (seller for example).

6.2 Modelling Dealer Networking System following PreciseSoa

In this section we present the model of the service system Dealer Networking System following PreciseSoa.

6.2.1 Service Architecture and Configurations

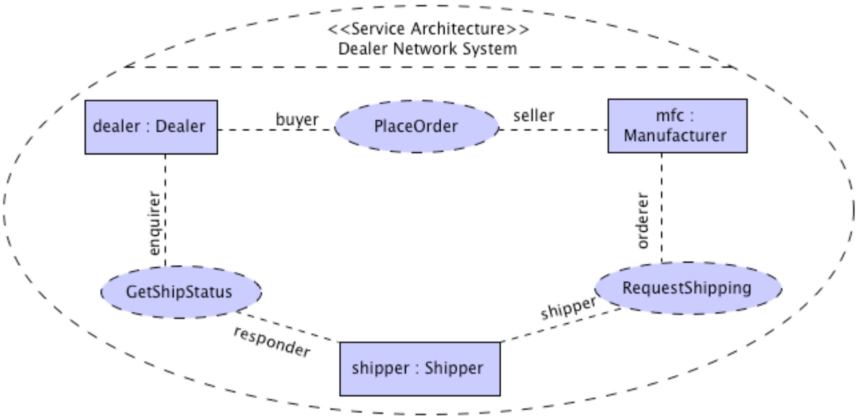


Figure 6.10: Dealer Networking System Service Architecture

The service architecture of the Dealer Networking System is shown in Fig. 6.10 and in 6.11 (where an architecture configuration is shown). The Dealer Networking System architecture depicts a community of participants providing and consuming services for realizing the aims of the Dealer Network. There are three roles for the participants in this architecture: dealer, shipper and mfc typed respectively by the participant classes Dealer, Shipper and Manufacturer, they are involved in three services: “Place Order”, “Get Ship Status” and “Request Shipping”. dealer plays the role buyer (i.e., consumer) and mfc plays the role seller (i.e., provider) in service Place Order. Instead, dealer plays role of the enquirer in service Get Ship Status whose provider is the shipper. mfc plays the role of provider in service Place Order, but in service Request Shipping, it plays a role as a consumer - orderer precisely. To take part in the services, each participant class will have some service ports, as shown in Fig. 6.19, corresponding to the service they participate in.

Fig. 6.10 illustrates the possible roles for participants in the high level view of how they work together in the Dealer Networking System. The three services and the participants appearing in this diagram will be described by service and participant models in the following sections. Fig. 6.11 shows instead a possible configuration of this architecture, where several participants of several types play various roles using and providing services; notice how at the same time several participants may use some service, offered by the same or by different other participants.

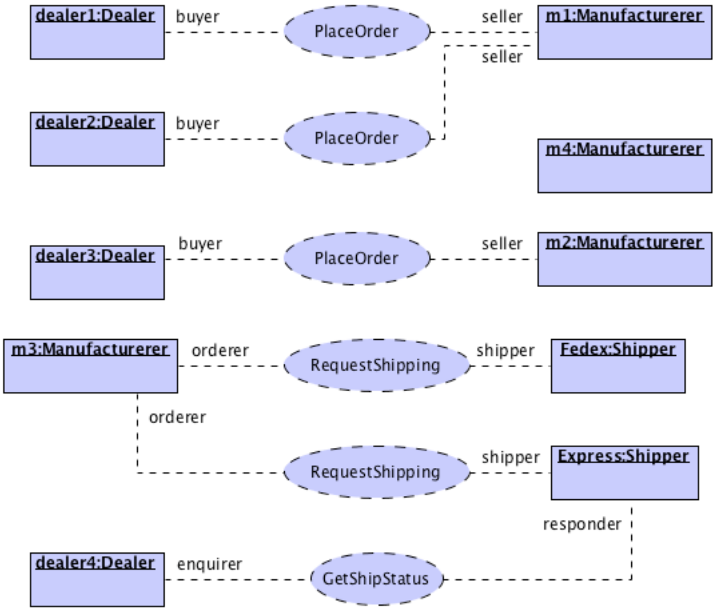


Figure 6.11: A Dealer Networking System Architecture Configuration

6.2.2 Service Models

6.2.2.1 Service Place Order

Fig. 6.12 shows the interface of the service Place Order. It realizes and uses respectively the interfaces: OrderPlacer and OrderTaker, which define the operations that provider and consumer implement to play their own roles. The type of the provider role: OrderTaker is the interface including the operations whose calls the provider seller will receive when enacting the service. The type of the consumer role: OrderPlacer is the interface including the operations whose calls the buyer will receive (correspondingly sent by the provider). We define two possible cases for an order: confirmed or cancelled that may be the values of attribute status typed by the enumeration type ConfirmationType. Moreover, if the order is accepted, the buyer will receive further information about the order represented by other attributes of the datatype OrderStatus, they are the providerID of the order, the deliveryDate of the shipment, and the wBN of the shipment. The identification of the buyer is defined by the attribute customerID in the definition of datatypes QuoteRequest and Order.

Fig. 6.13 presents the Place Order contract. The collaboration states that the role for the participant using the service is named buyer and is typed by the interface

OrderPlacer, whereas the role for the participant offering the service is named seller and is typed by interface OrderTaker. Those parts are bound to fulfill service Place Order following its contract. The sequence diagrams describe how to use use the service, i.e., which messages to send and which may be the possible answers. The buyer may either send a quote request or place an order, and the two sequence diagrams model these two cases.

Fig. 6.14 finally shows the Place Order semantics. The realm of the service is characterized by the amount of product in stock, so the PlaceOrder_Realm class has an attribute modelling the stock amount stock: int. The modified sequence diagram for the case of placing an order shows that the order is confirmed only whenever the ordered quantity is in stock, otherwise is cancelled, and after an order has been accepted the stock is reduced by the ordered amount.

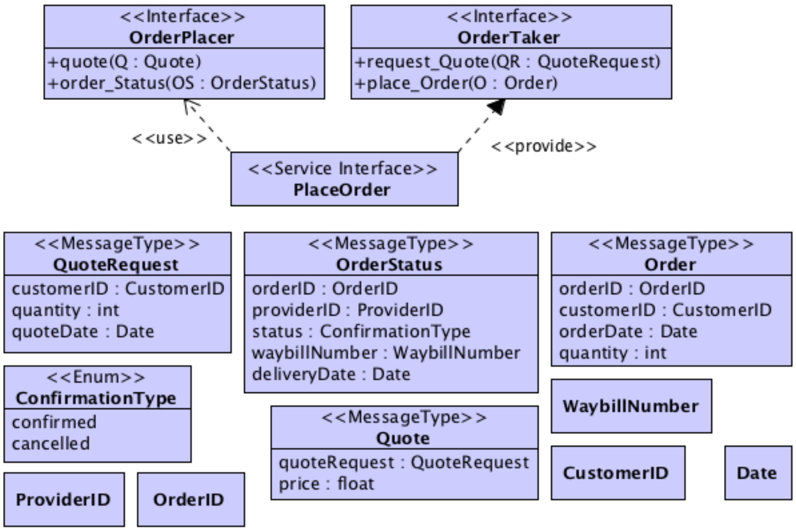


Figure 6.12: Place Order service Interface

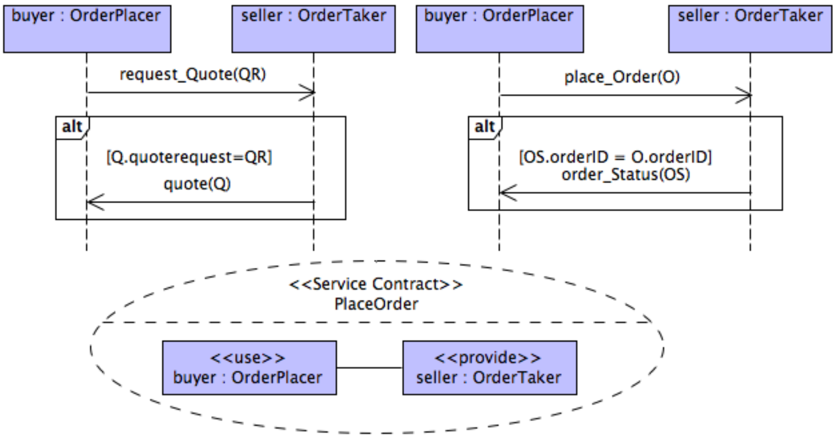


Figure 6.13: Place Order service contract

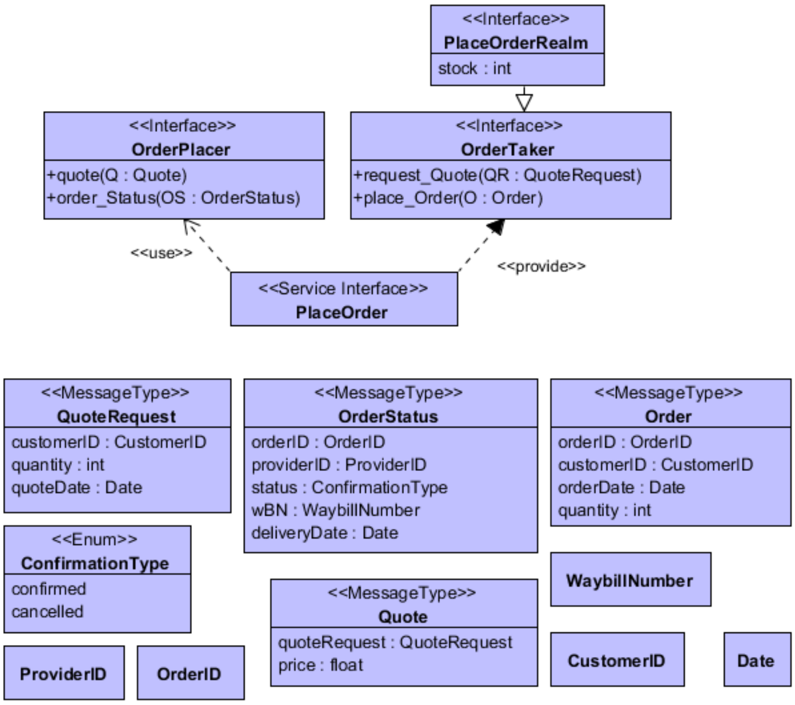


Figure 6.14: Place Order semantics

6.2.2.2 Service Request Shipping

Service Request Shipping provides the capability to send a shipping request to a shipper in order to deliver goods to a customer for a filled order. Fig. 6.15 shows the interface of the Request Shipping service. The provided interfaced contains a unique operation (request_Shipping(Request)), i.e., the service has a unique in message, that will be used to request a shipping (contain the information of the order, i.e., this is waybill number, and the sender and receiver's address); whereas the required interface contain two operations, i.e., package_PickUp(PickUpInfo) and confirm_Delivery(Confirmation), corresponding to two out messages to communicate the info on the pick up (pickup date and an estimated delivery date) and to confirm the delivery (containing the delivery date). Some obvious conditions relate the estimated delivery date, the expected delivery date, the pick up date and the request date can be seen in several message types of Request Shipping interface.

Fig. 6.16 shows the Request Shipping contract. In the sequence diagram to ensure that the package pickup message is sent to corresponding shipping request, we require that the parameters R and PUInfo in satisfy the condition R.wBN = PUInfo.wBN, similarly for R and conf.

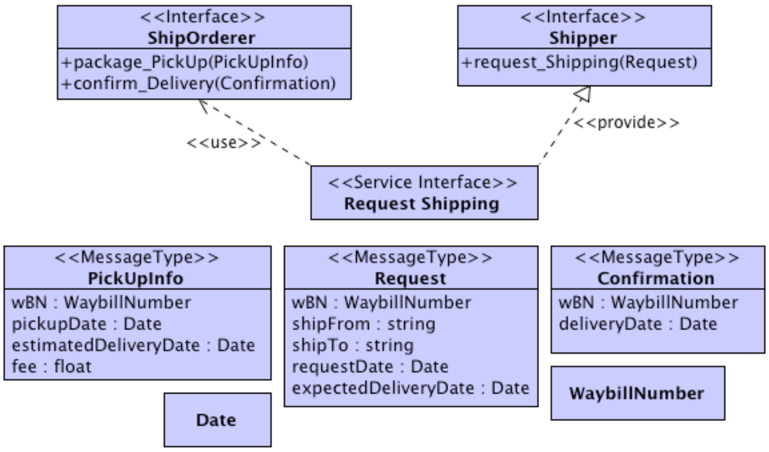


Figure 6.15: Request Shipping service Interface

We do not give the semantics of the service Request Shipping since to know exactly why the shipper propose a date for the pick up or another one is of no interest for the service user.

6.2.2.3 Service Get Ship Status

The interface and the contract of the service Get Ship Status are shown in Fig. 6.17 and 6.18; similarly to the the service Request Shipping we do not give its semantics.

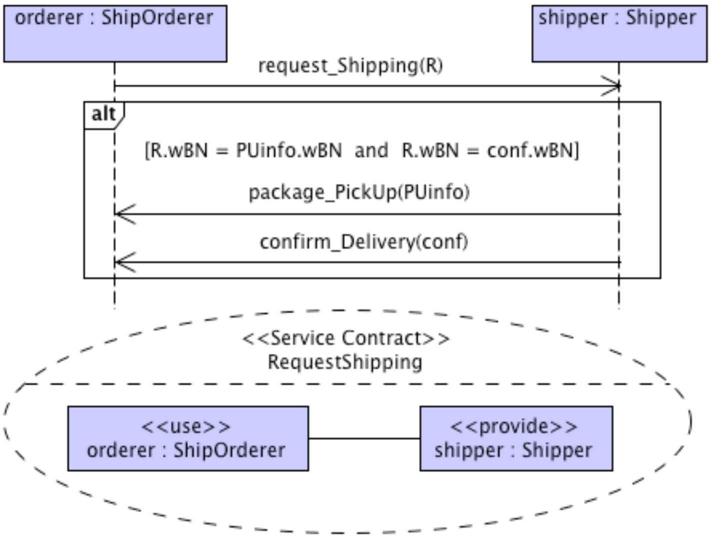


Figure 6.16: Request Shipping service contract

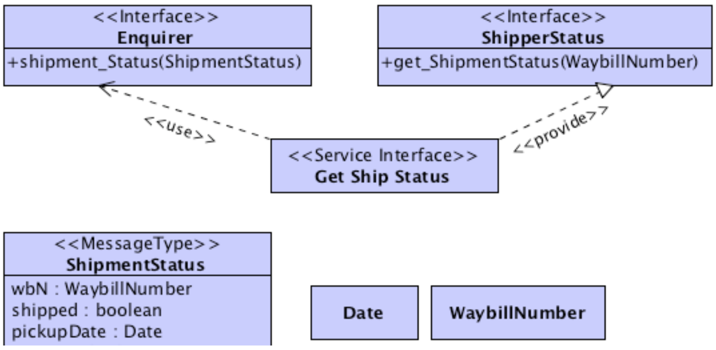


Figure 6.17: Get Ship Status service Interface

The collaboration in 6.18 binds the two parts representing the consumer and provider of service Get Ship Status. The consumer plays a role as a enquirer that is typed by interface Enquirer (defined in Get Ship Status interface in Fig. 6.17). Enquirer includes an operation to receive the status of the shipment, i.e., shipment_Status(ShipmentStatus). In OMG specification of SoaML [27], this consumer is not required to provide any operations, hence service Get Ship Status is a simple service that has a *simple service interface*. But in our precise approach, the interface of the consumer can be an “empty class” or it has one operation as we built (e.g., Enquirer in Fig. 6.17). The empty compartment means that there are no operations, therefore "empty class" is an interface without operations in the reality.

The type of the provider role responder is ShipperStatus interface that contains one operation get_ShipmentStatus(W:WaybillNumber) called by the consumer to enact this service. If we could not find any suitable names for the roles, we may let them have the same names as the interface themselves (shipperStatus : ShipperStatus for example). The purpose of the service is resulting shipment status for the enquirer. The status of the shipment will be contained in the message type ShipmentStatus (see Fig. 6.17). The order of messages that service Get Ship Status receives and sends out is illustrated by the sequence diagram built in Fig. 6.18.

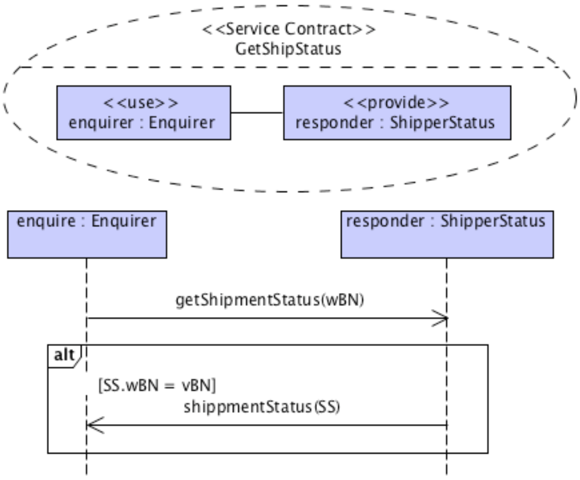


Figure 6.18: Get Ship Status service contract

6.2.3 Participant View

All the kinds of participants that provide and consume services in the Dealer Networking System are presented by participant view show in Fig. 6.19 by just giving their participant classes (here we do not further model these participants, since we not even define if they are structured or monolithic). There are three kinds of participant: Dealer, Manufacturer, and Shipper. Each participant (type) has service ports and request ports for the services they provide and consume that are stereotyped by ≪Service≫ and ≪Request≫. Manufacturer participant is a service provider for service Place Order, so it has a ≪Service≫ port typed by service interface Place Order to offer the service through this port. Manufacturer itself is a service consumer of service Request Shipping, then it has a ≪Request≫ port typed by conjugate ∼ Request Shipping service interface to consume the service.

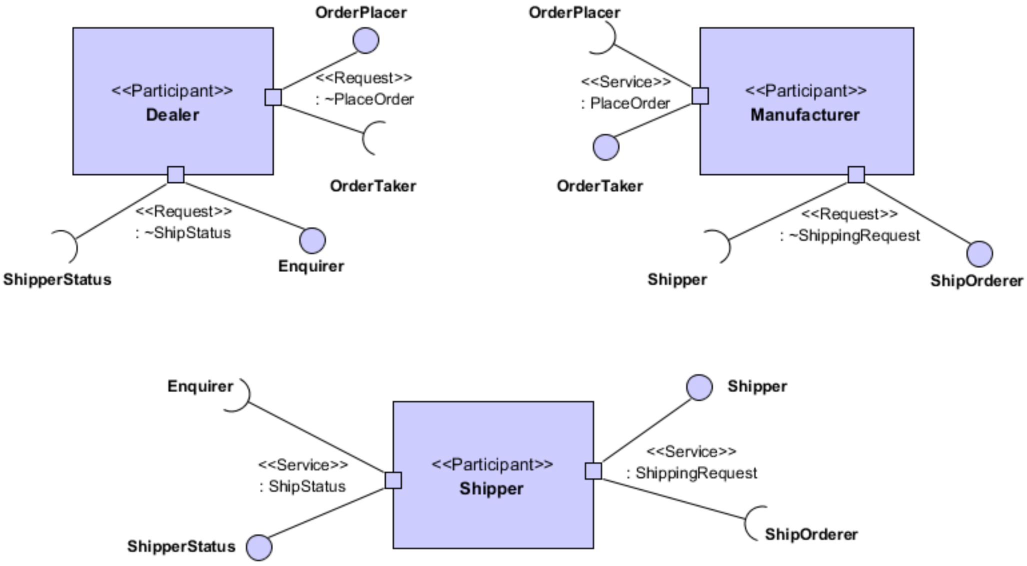


Figure 6.19: Dealer Networking System participant view

6.3 Office System Model Model following PreciseSoa

In this section we model the Office System described in Chapter 3 following the PreciseSoa method introduced in Sect. 6.

6.3.1 Service Architecture and Configurations

The service architecture shown in Fig. 6.20 presents the set of services and the participants that provide and consume those services for Office System. There are six roles for participants: *Office Component*, *Printing Center*, *Language Centers* for three languages (italian, French, and English), and *Web Publisher*. In this architecture, *Office Component* plays the consumer role to the services: Print, Check French, Check English, Check Italian, and Publish on Web.

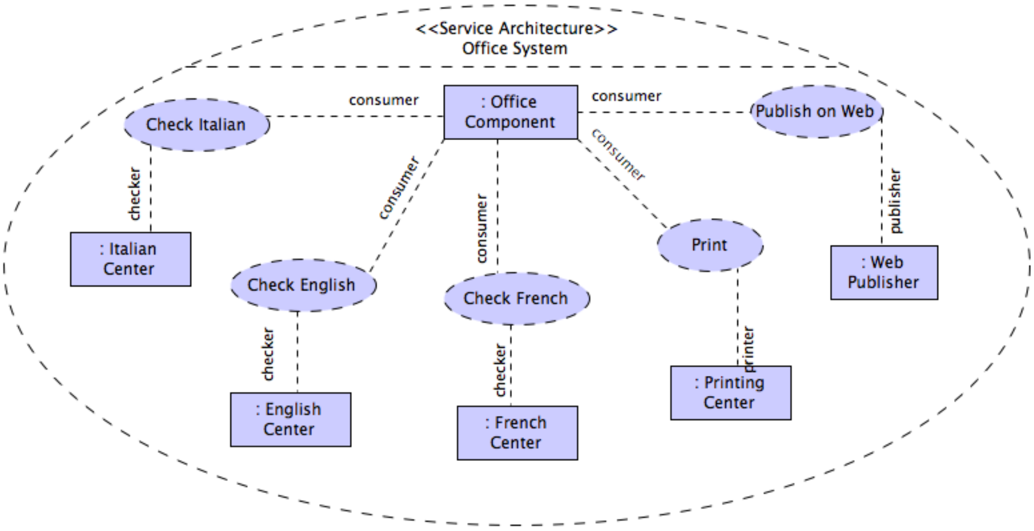


Figure 6.20: Office System architecture

6.3.2 Service Models

6.3.2.1 Service Print

The purpose of the Print service is to allow to print documents.

Fig. 6.22 shows the interface of Print. Fig. 6.23 depicts the printing service contract where the two parts OfficeComponent and Printer are the consumer and the provider respectively. In the part stereotyped by ≪provide≫, the role of provider Printer is named as printer (as labeled on the connector as shown in Fig. 6.20) is bound to its type that is Print interface. This Print interface contains three operations to

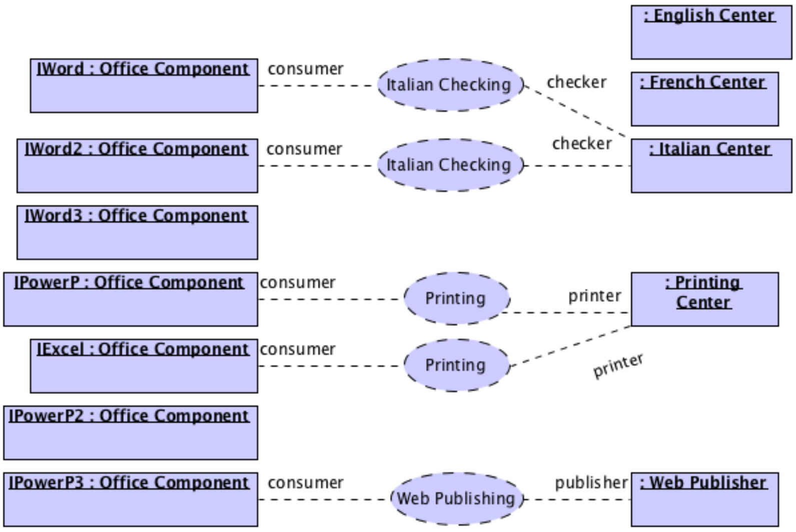


Figure 6.21: As Office System Architecture Configuration

enact the service, as shown in Fig. 6.22. The role of participant OfficeComponent is named as consumer for all services, and in this service, the type of consumer role is Writer, which is an interface that contains three operations corresponding to the three cases: printed if the document printed successfully, notA4 if the page is not in A4 format and noPaper if the printer is out of paper.

The service interface is used to type the ports of these two participants shown in Fig. 6.29. The messages exchanging between parts to offer and use that service is represented by the sequence diagram in Fig. 6.22.

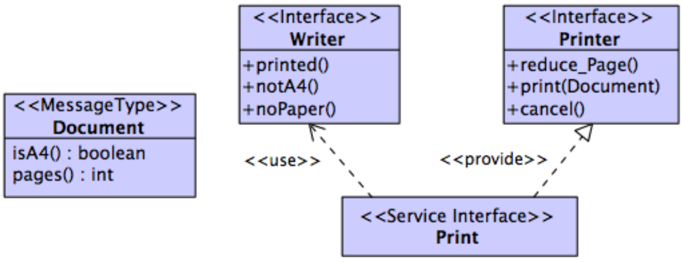


Figure 6.22: Print service interface

6.3.2.2 Check Italian service

There are three services that supply capabilities to check the spelling and the grammar of a text written in one of three specific languages: English, French and Italian (as described in Chapter 3). Here we give the model of the service Check Italian considering the Italian language, the models of Check French and Check English are similar.

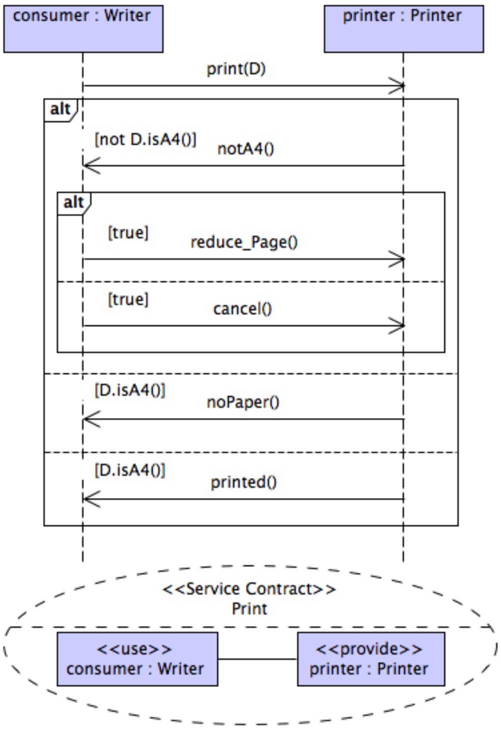


Figure 6.23: Print service contract

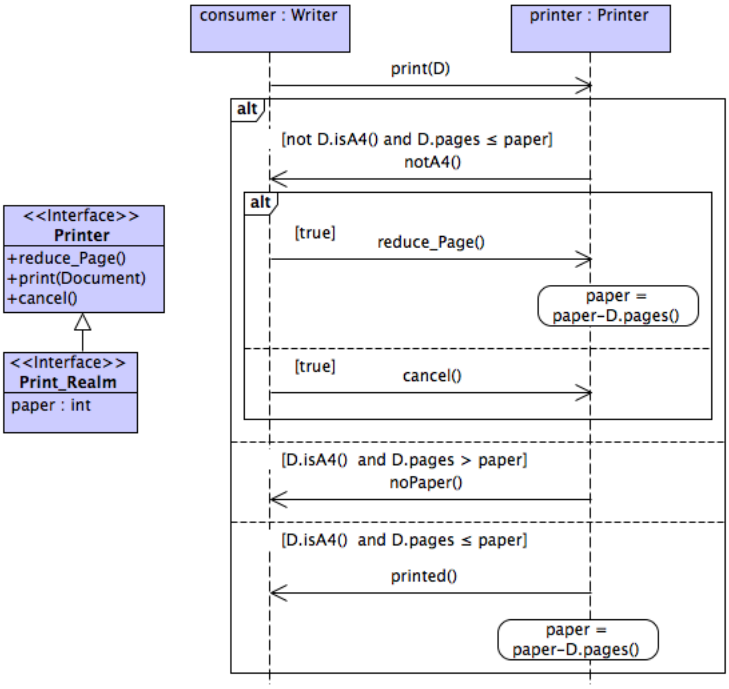


Figure 6.24: Print service semantics

The interface of Check Italian is given in Fig. 6.25. The provided interface Checker comprises two operations, i.e., check_Spelling(Text) and check_Grammar(Text), they are two types of in message that this service can receive, one for spelling checking requirement, and the other for grammar checking requirement. The required interface Editor comprises three possible out messages that service may respond to those in messages, i.e., spelling_Errors(SpellErrors) if the service found any spelling error in Text, grammar_Errors(GrammErrors) if the service found any gramma error in Text, and wrong_Language() if Text is not written in three predefined languages.

There is a contract that if service Check Italian receive message check_Spelling(Text), it may return only one of two messages: spelling_Errors(SpellErrors) or wrong_Language(), or only one message wrong_Language(), it cannot send out message grammar_Errors(GrammErrors). Meanwhile, if service Check Italian receive message grammar_Errors(GrammErrors), it may return one of two or both of two messages: spelling_Errors(SpellErrors) and grammar_Errors(GrammErrors), or only one message wrong_Language(). With only model of Check Italian interface, we cannot see this aspect, it will be illustrated in contract of this service.

The contract of service Check Italian, see Fig. 6.26, represents two roles, in which checker is the role of provider Italian Language Center, and consumer is the role of consumer OfficeComponent. The type of provider role is Checker interface that the provider will require on a port to provide this service. Whereas, the type of consumer role is Editor interface containing the operations that he may receive when enacting service. The messages stereotyped by ≪Message Type≫ are sent from the consumer to the provider in order to request a language checking service. In case, the consumer only requires to check the spelling, the provider will require Checker interface to return a list of spelling errors if any, or a message to note that the text is not in the set of languages, i.e., the operations spelling_Errors(SE: Spelling Errors) and wrong_Language(). Otherwise, if the consumer requires to check the grammar, the provider may return a list of spelling errors if any, and/or a list of grammar errors (if there is), or a message wrong language, i.e., the operations grammar_Errors(GE: Grammar Errors) and wrong_Language(). We model two sequence diagrams for this service contract behaviour to illustrates those two possible results, see Fig. 6.26.

We do not give the semantics view, since this service does not depends on a realm: we assume that the correctness of the spelling and of a language do not depend on any changeable aspects of the real world, and moreover this service does not modify anything.

6.3.2.3 Service Publish on Web

The consumers use the Publish on Web service to publish web pages on the Internet. There are two preconditions for the service: the URL is correct and the

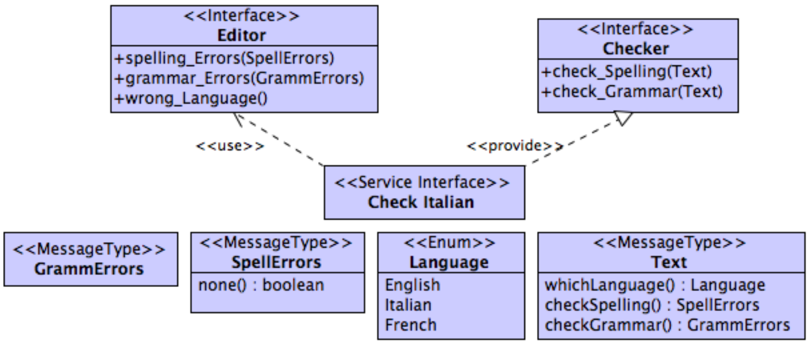


Figure 6.25: Check Italian service interface

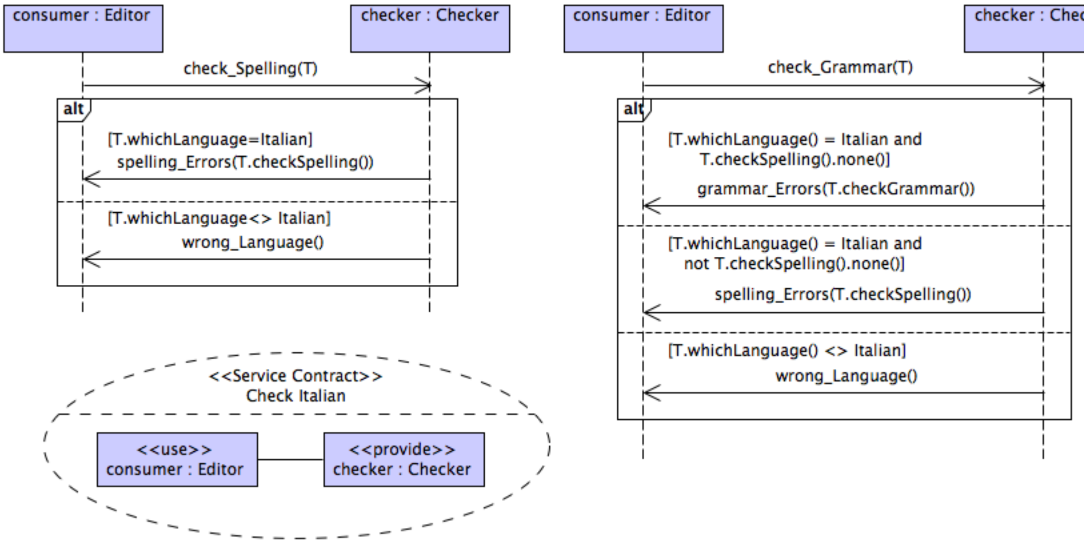

consumer : Editor
checker : Checker
check_Spelling(T)
alt
[T.whichLanguage=Italian]
spelling_Errors(T.checkSpelling())
[T.whichLanguage<> Italian]
wrong_Language()
consumer : Editor
checker : Chec
check_Grammar(T)
alt
[T.whichLanguage() = Italian and
T.checkSpelling().none()]
grammar_Errors(T.checkGrammar())
[T.whichLanguage() = Italian and
not T.checkSpelling().none()]
spelling_Errors(T.checkSpelling())
[T.whichLanguage() <> Italian]
wrong_Language()
<<Service Contract>>
Check Italian
<<use>>
consumer : Editor
<<provide>>
checker : Checker

Figure 6.26: Check Italian service contract

corresponding server is available.

Fig. 6.27 shows the interface of service Publish on Web. The role for the participant using the service is named consumer and the role for the participant offering the service is named publisher, as shown in Fig. 6.20. In the collaboration in Fig. 6.28, they appear again in two parts of the service contract and are bound to their types, i.e Web Editor interface and Publisher interface respectively, as shown in Fig. 6.27. The operations that are sent from the publisher to the editor can be: wrongURL() if the URL is wrong, notHTML() if the page is not written in HTML, serverNotAvailable() if the server requested is not available at this moment, and published() if the pages are published such that the editor will have his page published as a web page on the Internet. Fig. 6.28 illustrates the behaviour for this service contract.

Two variant cases should be concerned to this service. The first case is that the page is not written in HTML, the service will not only send an alert but also convert it to HTML format. The second case is that the URL is not correct, then the consumer will be required to supply another one with any number of times. We consider those cases as internal interactions of the service, and we wish to propose a mechanism to model them, particularly in CASL4SOA profile (cf. Chapter 7).

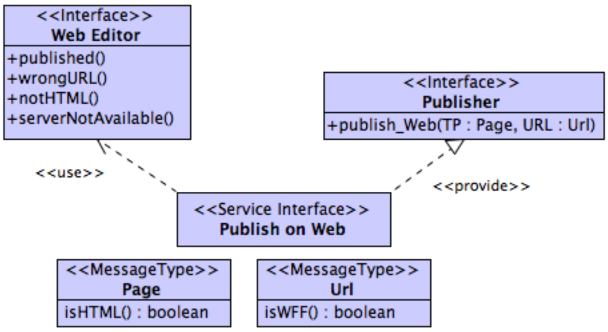


Figure 6.27: Publish on Web service interface

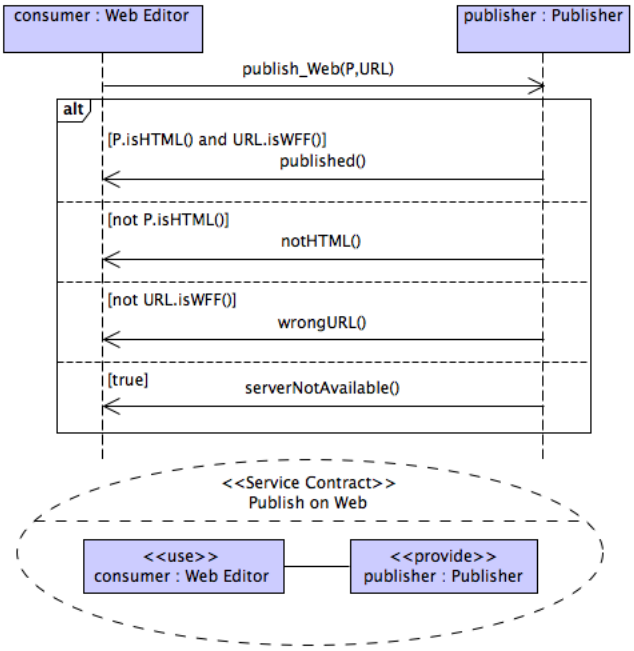


Figure 6.28: Publish on Web service contract

6.3.3 Participant View

There are six kinds of participants that provide and consume services in Office System. They are represented by the classes stereotyped by ≪Participant≫ in the participant view in Fig. 6.29 and named: Italian Center, English Center, French Center, Printing Center, Office Component, and Web Publisher. Those participants realize their interfaces by service ports and request ports that are stereotyped by ≪Service≫ and ≪Request≫. The participant typed Office Component is the unique consumer and does not play any provider role in the system, since all its ports are stereotyped by ≪Request≫ and typed by the conjugate service interfaces of the providers for corresponding services. For the participants that are of kind of providers, their ports are stereotyped by ≪Service≫ and typed by the service interface of the service itself. The participant typed Printing Center is provider for Print service, its port is stereotyped by ≪Service≫ and typed by the service interface Print Service. Those participants realize their interfaces to offer the service through their service ports.

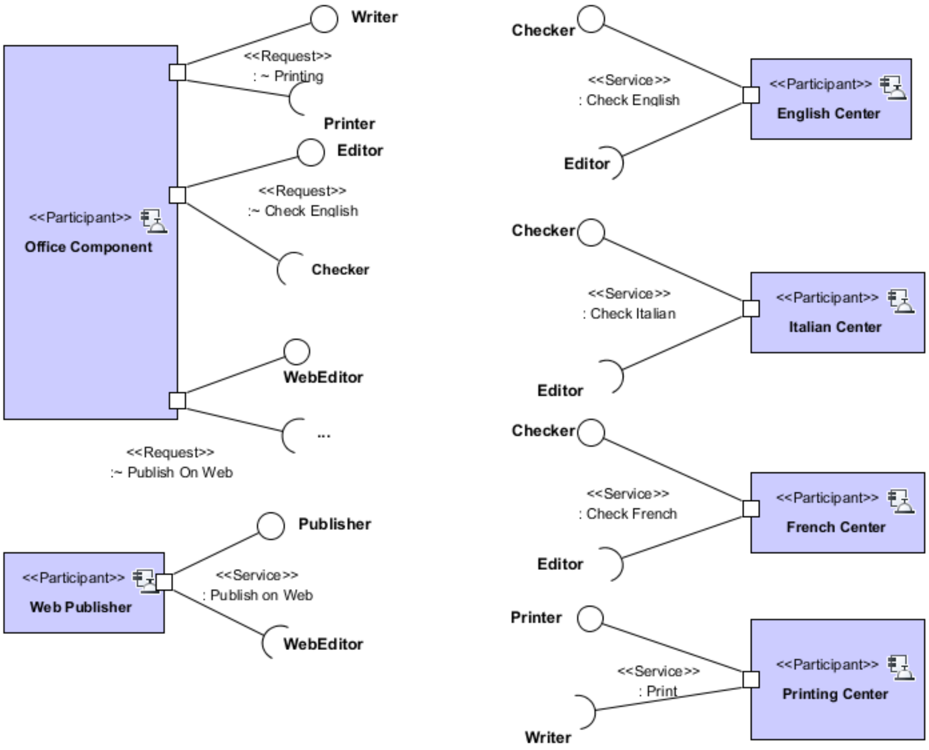


Figure 6.29: Office System Participant View

Conclusion: In this chapter, we have presented our proposed PreciseSoa profile, i.e., a UML profile, and a method designed over SOA paradigm presented in Chapter 4 to model service systems using this profile. We described the structure of the PreciseSoa service system models and investigated the well-formed constraints on these models. A set of steps to produce a PreciseSoa model of a service system is presented, moreover the use of this modelling method is illustrated by the application on Dealer Networking System and Office System case studies (presented in Chapter 3).

Chapter 7

Design Model of Service system in Casl4Soa

«««< .mine

Contents

In this chapter, we present our extension of Casl-Mdl models that offers a visual syntax to a subset of the Casl-Ltl [45] formal textual specifications to develop Casl4Soa as a formal visual notation used to model service systems not only visually but also formally and effectively.

7.1 Overview of Casl4Soa

Based on Casl-Ltl (an extension for dynamic systems of the algebraic specification language Casl, see [45]), Casl-Mdl [9] has been developed as a non objected-oriented visual formal notation. Casl-Mdl offers a visual syntax to a subset of the Casl-Ltl formal textual specifications, precisely each Casl-Mdl model can be translated into a Casl-Ltl specification.

In Casl-Mdl we have a type diagram introducing the datatypes and the dynamic types, which are types of dynamic systems, that allow to model datatypes and either simple or structured dynamic systems.

A dynamic system is seen as a labelled transition system, where the labels are "elementary interactions" corresponding to the interactions of the system with its context. Then, the behaviour of data and dynamic types is defined either following a property-oriented style using logical formulas (of a first-order many sorted branching time with edge-formulas logic) or constructively defining the operation behaviour by conditional rules and the system behaviour by interaction machines.

At the visual level the constructs of Casl-Mdl have been defined reusing the visual constructs of the UML, thus we can use the UML editing tools to produce the Casl-Mdl models.

Casl4Soa [8] (Common Algebraic Specification Language for Service Oriented Architecture) has been developed as a profile of Casl-Mdl with the aim to provide an effective notation to model SOA systems. The profiling mechanism used for defining Casl4Soa is similar to the profiling mechanism of the UML, and it was inspired by it. Thus we use stereotyped Casl-Mdl constructs (Appendix A.3) to define the new Casl4Soa constructs. Moreover, each Casl4Soa model corresponds to a Casl-Mdl model, that in turn corresponds to a Casl-Ltl specification (see Fig. 7.1), which has a well-defined formal semantics, thus also Casl4Soa has a well-defined formal semantics.

Casl4Soa has been designed over SOA paradigm presented in Chapter 4, and here we will use all the terminology defined in such chapter.

There are two kinds of Casl4Soa service models, constructive and property-oriented. In a constructive model, the behavioural aspects of the services and of

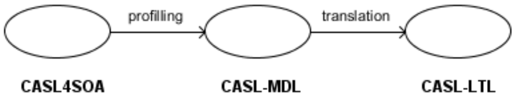


Figure 7.1: Relationships among Casl4Soa, Casl-Mdl, and Casl-Ltl

the participants are expressed by means of the interaction machines (see Appendix A.3.4), whereas in a property oriented model, such aspects are expressed by means of first-order temporal logic formula (see Appendix A.3.5).

In Casl4Soa, *dynamic system* denotes any kind of dynamic entities, i.e., entities with dynamic behaviours without making further distinctions, and are formally considered as labelled transition systems, that we briefly summarize below.

A *labelled transition system* is a triple $(\mathit{State}, \mathit{Label}, \rightarrow)$, where $\rightarrow \subseteq \mathit{State} \times \mathit{Label} \times \mathit{State}$ is the *transition relation*.

A dynamic system is thus modelled by a transition tree determined by a labelled transition system and an initial state $s_0 \in \mathit{State}$. A dynamic type corresponds to the state of a labelled transition system, thus its values correspond to dynamic systems.

The labels of the transitions of a dynamic system are named *interactions* and are descriptions of the information flowing in or out the system during the transitions, thus they truly correspond to interactions of the system with the external world. The states of simple systems are characterized by a set of typed attributes (precisely the states of the associated labelled transition system).

We use dynamic types to model services and participants.

7.2 Casl4Soa Constructive Service System Model

The form of the Casl4Soa constructive participant models is shown in Fig. 7.2 by means of a metamodel, whereas Table 7.1 shows the constraints defining the well-formed models.

A service system, as said in Chapter 4, is a particular structured participant neither offering nor using services, and thus without any service point. Thus a Casl4Soa *constructive service system model* is a special case of a participant model for a participant without any port, thus neither offering nor using services.

A participant model consists of the definition of a participant type by means of a type diagram including the definition of a dynamic type, and of the models of the provided and used services. The dynamic type defining the participant may be either simple (for the case of the monolithic participants) or structured (for the

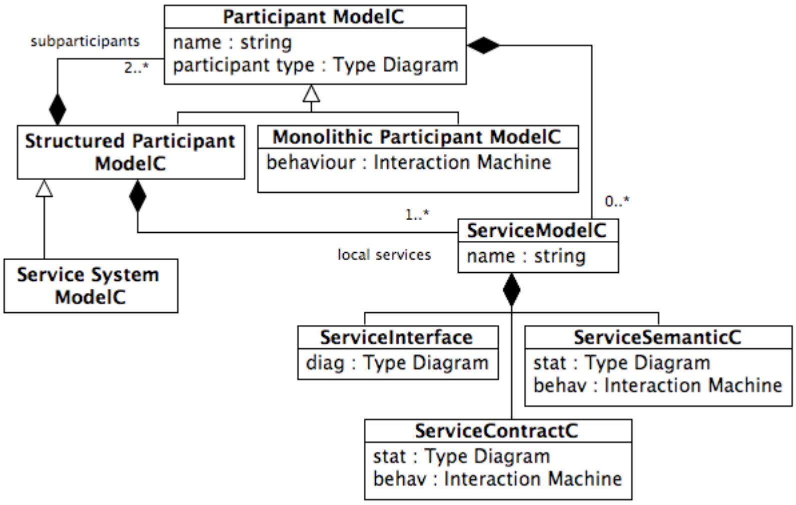


Figure 7.2: Casl4Soa constructive model: metamodel

structured participants), in this case the structured dynamic type definition will allow to represent its service architecture. The model of a structured participant will include also the models of a set of *participant types*, its subparticipants, and the models of the local services, i.e., the service used by its subparticipants to interact among them.

A service system, as said in Chapter 4, is a particular participant neither offering nor using services, and thus without any service point. Thus a Casl4Soa *constructive service system model* is a special case of a participant model for a participant without any port, thus neither offering nor using services.

The Casl-Mdl dynamic type modelling a participant will be stereotyped by ≪Participant≫, whereas in the case of service system the stereotype ≪ServiceSystem≫ will be used.

Participant Model	– All services have different names. – All subparticipants have different names. – A subparticipant must have at least a port to offer or use at least one service. – Connectors between two ports of two different participants must be labelled by the name of a service that those participants offer and use. – Many different connectors may leave or enter the same port of a participant but all of them must be labelled with the same service. – The services used to label the connectors must be already presented in the model.
Service Model	– All the parameters of the interactions of a service interface must be typed by datatypes. – A service interface must have at least one input elementary interaction. – If a service is named SN, then the simple system appearing in the contract should be named SN_Contract and the one appearing in the semantic should be named by SN_Semantics. – There is only one initial state in the interaction machine representing service behaviour.

Table 7.1: CASL4SOA constructive model: Well-formedness constraints

Naming Convention

–In the names of the services and interfaces, each word should begin with a capital letter (e.g., Place Order, OrderTaker).

–In the names of the interactions, word should be written in mixed case starting with lower case. When there are more than two words in the name, use underscores to separate them (e.g., ?_place_Order).

–The parameters of an interaction should be in upper case (e.g., QR).

–Names of datatypes must be nouns, and each word of the names should begin with a capital letter (e.g., OrderStatus).

–The attributes of datatypes should be written in mixed case starting with lower case (e.g., orderDate).

Table 7.2: Naming convention for Casl4Soa models

7.2.1 Service Model

The structure of the service model is shown as a part of Casl4Soa Constructive Model in Fig. 7.2 (together with Participant Model which is described in Sec.7.2.2), and the associated well-formedness constraints are reported in Table 7.1. A Casl4Soa *constructive service model* consists of the *service name*, a *service interface*, a *contract* and a *semantics.* As stated in Chapter 4, a service interface provides the static information needed to interact with the service, the service contract focuses on the protocol between the provider and the consumer of the service, and the semantics allows to understand the functionalities provided by the service to its users.

A *service interface* is an interface for a dynamic system, visually represented by a box with the stereotype indication ≪Service Interface≫ (see a generic service interface shown in Fig. 7.3), it is named as the service itself and it defines the elementary interactions needed to use the service, distinguished in input and output interactions by a naming convention ?_yyyy (input) and !_xxxx (output). The input elementary interactions model the requests sent to the service, whereas the output ones model the answers that the service sent out to the user. They are characterized by a name and a possible empty list of parameters.

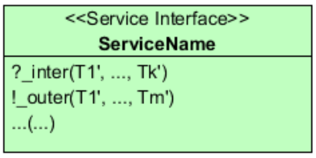


Figure 7.3: A generic service interface

A *service contract* in constructive style is represented by a simple dynamic system stereotyped by ≪simpleSystem≫ (see a generic simple system shown in Fig. 7.4) and an associated *interaction machine* (see Fig.7.5), in which the behaviour of the service is modelled as seen at the service point where it is provided.

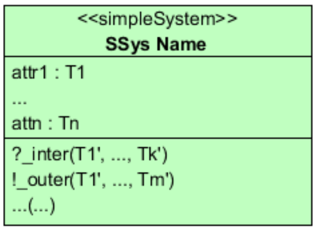


Figure 7.4: A generic simple dynamic system

The simple system should extend the one used for modelling the interface but without adding any new interaction, thus it will exactly the same elementary interactions as the service interface, whereas it may have some new attributes, which are needed to abstractly model the relationships between the in and out messages. The interaction machine modelling a service contract should follow a specific pattern to mimic the informal conceptual description of a service contract proposed in Chapter 4 to illustrate the fact that: the service may receive initially possible requests, then it will answer in many different ways (even going in a final state), after that the service is ready to receive other requests, then answer them. A visual generic schematic example of an interaction machine is given in Fig.7.5, and for further description, see Appendix A.3.4).

The realm of a service is the part of the real world affected and known by the service itself. The *semantics* of a service describes the effects of the requests received by the service itself (in messages) on its realm, and how the realm status determines the answers sent out by the service itself (out messages). The semantics of a service in constructive style is given by a simple dynamic system extending the one used for the service contract (again without adding new interactions), and the behaviour of this system is modelled as usual by means of an interaction machine.

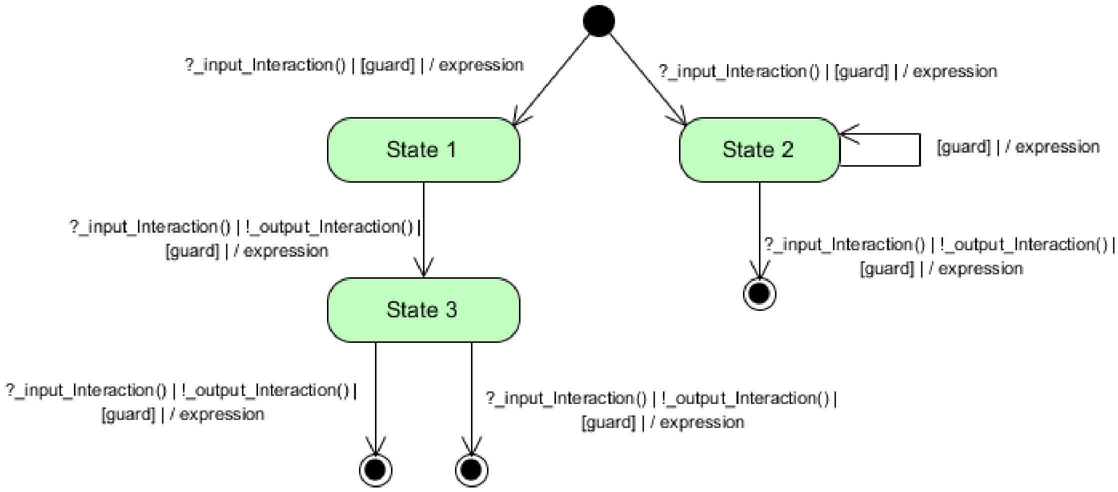

?_input_Interaction() | [guard] | / expression
?_input_Interaction() | [guard] | / expression
State 1
State 2
[guard] | / expression
?_input_Interaction() | !_output_Interaction() | [guard] | / expression
?_input_Interaction() | !_output_Interaction() | [guard] | / expression
State 3
?_input_Interaction() | !_output_Interaction() | [guard] | / expression
?_input_Interaction() | !_output_Interaction() | [guard] | / expression

Figure 7.5: A generic schematic example of an interaction machine

7.2.2 Participant model

A participant type is expressed as a CASL-MDL dynamic type (see Appendix A.3.3) stereotyped by ≪Participant≫ (see generic participant shown in Fig. 7.6), the ports of which are characterized by service interfaces to indicate that a participant of that type provides or consumes a service. The ports are used to structure the elementary interactions of a dynamic system and to define the cooperation inside the structured dynamic systems.

If a participant is offering a service S, the port where S is offered is typed by the interface of S, whereas if it is using a service S' throughout a port, then such port is typed by the conjugate of the interface of S'. Recall that the conjugate of an interface I is denoted by ∼I, and the elementary interactions of ∼I are those of I where the input and output types are swapped.

The participant type and all the needed datatypes are collected in a type diagram.

In case of a monolithic participant, a generic model is given in Fig.7.6, its behavior should be expressed by means of an interaction machine (a generic interaction machine is given in Fig.7.5).

In case of a structured participant, the corresponding CASL-MDL dynamic type will be a structured dynamic type (see Fig.7.7), and its definition will allow to express the service architecture of the participant itself.

As described in Chapter 4 a service system is a special case of structured participant without any port, and we will use the stereotype ≪ServiceSystem≫ (see Fig.7.8)

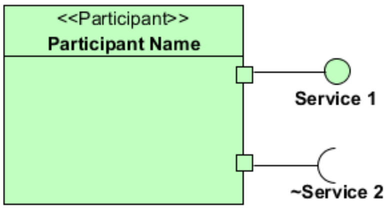


Figure 7.6: A generic participant

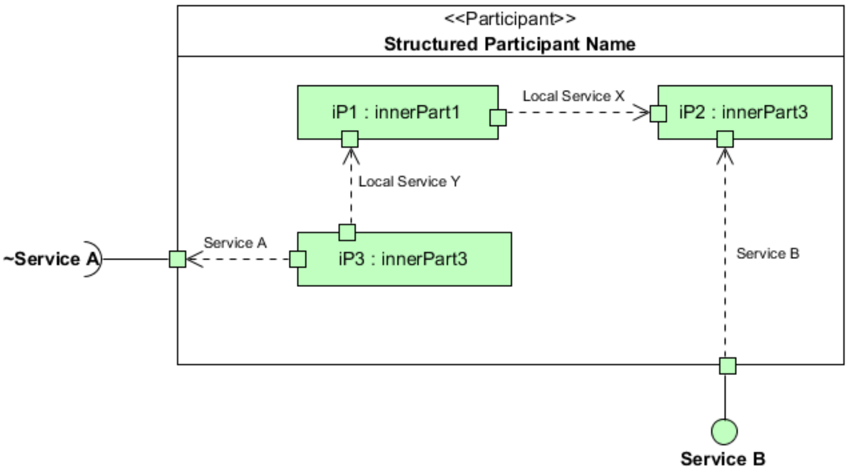


Figure 7.7: A generic structured participant

instead of ≪Participant≫ for denoting the corresponding Casl-Mdl structured dynamic type.

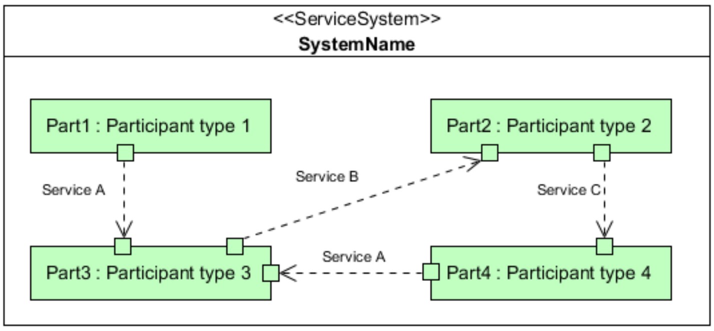

<<ServiceSystem>>
SystemName
Part1 : Participant type 1
Part2 : Participant type 2
Service A
Service B
Service C
Part3 : Participant type 3
Service A
Part4 : Participant type 4

Figure 7.8: A generic service system

7.3 Casl4Soa property oriented model

The structure of the Casl4Soa property oriented models is shown in Fig. 7.9. The form of these models is similar to the one of the constructive models shown in Fig. 7.2, but now the behaviour of the dynamic systems is modelled by means of a set of logical formulas, instead of an interaction machine. The constraints defining the well-formed models are the same as the constructive models, shown in Table 7.1.

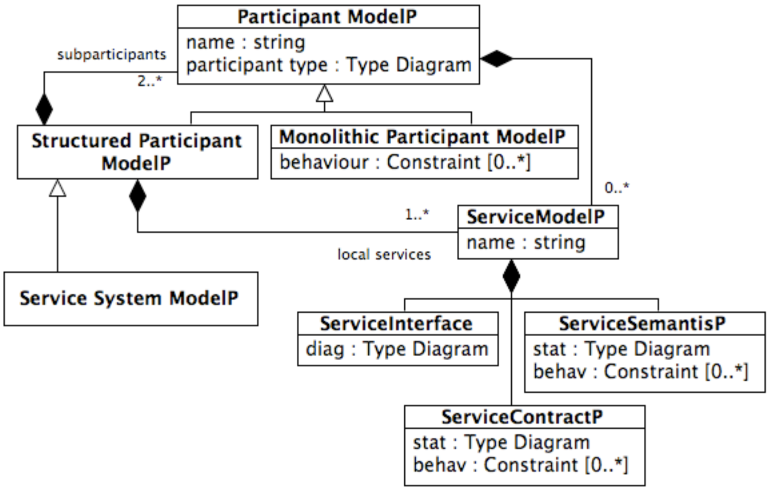


Figure 7.9: CASL4SOA property oriented model: metamodel

A *service contract* in property-oriented style is represented by a dynamic type whose behaviour is specified by a set of constraints on the type itself, i.e., by a set of temporal formulas described in Appendix A.3.5. Similarly the semantics of a service in property-oriented style is given by a simple dynamic system extending the one used for the service contract, and the behaviour of this system is modelled again by means of a set of constraints.

A generic example of a formula is presented as following:

[in_any_case] [sometimes | always] [path_form] ⇒ *[eventually | always | next][path_form]*

The formulas comprise first-order logic combinators, together with temporal combinators (for a path formula) to address whether a property is satisfied in states of a path from a given state. The form of formulas should conform to a grammar structure defined in Appendix A.3.5.

7.4 How to develop a CASL4SOA model

Following the indications of Chapter 4 (see Fig. 4.1 and 4.2), we give first a conceptual model of the service system of interest, and then model all the parts using CASL4SOA; all the steps are summarized in Fig. 7.10.

Recall that a service system is a particular structured participant neither offering nor using services, we assume that the service system being built is the first con-

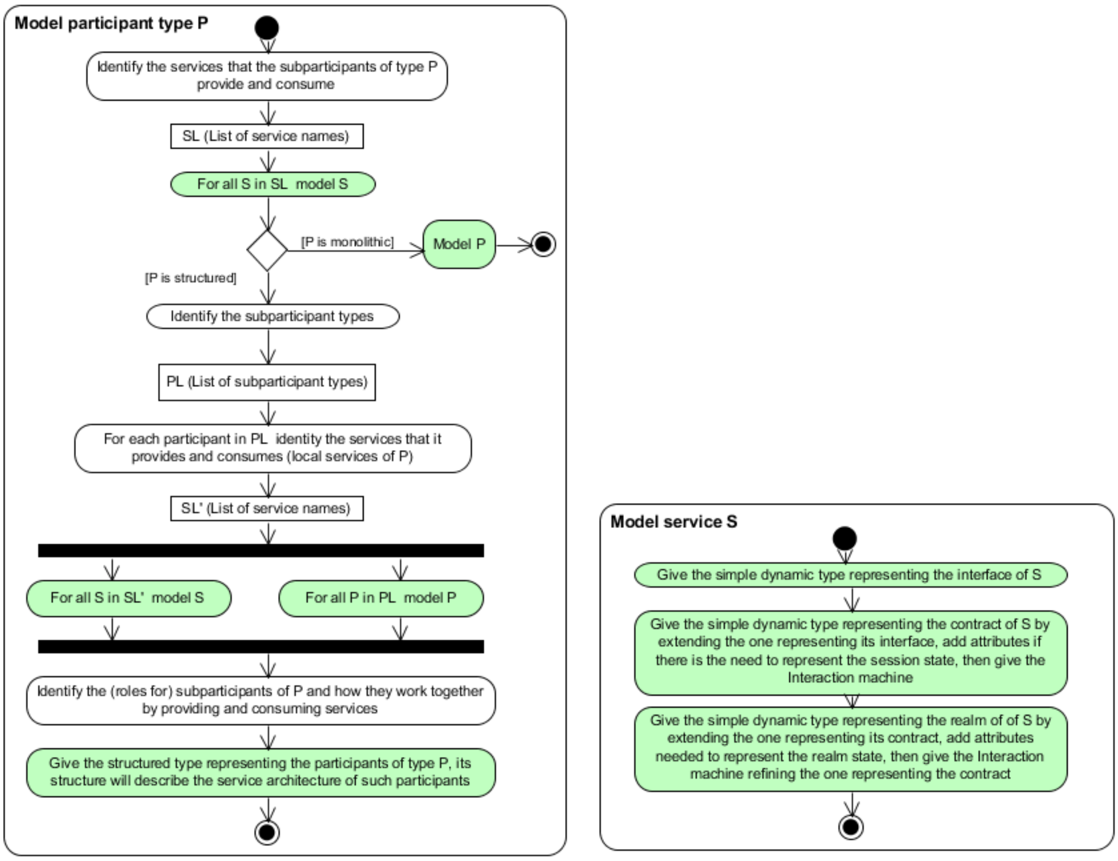

Model participant type P
Identify the services that the subparticipants of type P provide and consume
SL (List of service names)
For all S in SL model S
[P is monolithic]
Model P
[P is structured]
Identify the subparticipant types
PL (List of subparticipant types)
For each participant in PL identity the services that it provides and consumes (local services of P)
SL' (List of service names)
For all S in SL' model S
For all P in PL model P
Identify the (roles for) subparticipants of P and how they work together by providing and consuming services
Give the structured type representing the participants of type P, its structure will describe the service architecture of such participants
Model service S
Give the simple dynamic type representing the interface of S
Give the simple dynamic type representing the contract of S by extending the one representing its interface, add attributes if there is the need to represent the session state, then give the Interaction machine
Give the simple dynamic type representing the realm of of S by extending the one representing its contract, add attributes needed to represent the realm state, then give the Interaction machine refining the one representing the contract

Figure 7.10: How to develop a CASL4SOA model

sidered structured participant of type P (see activity named **Model participant type P** in Fig. 7.10), then it may include a number of monolithic participants and structured participants, and a number of services.

Identifying the services that the subparticipants of P provide and consume is the first steps.

To model a service S in CASL4SOA, all the steps are shown in the actions of the activity named **Model service S** in Fig. 7.10. Those actions are described in details as following:

Give the simple dynamic type representing the interface of S

What to do

Define set of input and output interactions for a simple dynamic system representing interface of S. The interactions are either of kind input or output. Define datatypes needed to type the elements of interface.

What is supposed to be clear before building interface of S

- The specific requests that service S will receive from the users.

- The specific responses that service S may send to the users for corresponding requests.

The composition of S interface

- Name of S interface
- The elementary interactions of kind *input* and *output*.

Building steps

1. Use Casl-Mdl simple dynamic type with stereotype ≪serviceInterface≫ to define S interface.
2. Name S interface by the name of S itself (suggesting the purpose of the service)
3. Build input interactions based on the requests to service S, give them names (starting with convention ?_) and define the necessary parameters.
4. Build output interactions based on the requests to service S, give them names (starting with convention !_) and define the necessary parameters.
5. Use construct Datatype to define the elements of S interface if they are not of primitive type.

Note:

- The name of a input interaction should contain and start with a verb.
- The input interactions should be listed before the corresponding output interactions.

Give the simple dynamic type representing the contract of S

What to do

Extend the simple dynamic type that represents the interface of S, add attributes if needed to represent the session state.

What is supposed to be clear before building the contract of S

- The interface of S.
- The attributes needed to model the activities of S providers and to express the realm of the service.

The composition of S contract in terms of a simple dynamic type

- Name of S contract (named with extension _Contract).

- Some attributes determining the internal states of the dynamic system representing S contract, and the elementary interactions as defined in S interface.

Building steps

1. Use CASL-MDL simple dynamic type with stereotype ≪simpleSystem≫ to define S contract.

2. Name the simple dynamic system by S name with extension _Contract, and insert the elementary interactions that are predefined in the interface of S.

3. Insert attributes representing the session state and type them by datatypes/primitive types.

Give the interaction machine of S contract

What to do

Build interaction machine of S contract.

What is supposed to be clear before building an interaction machine of S contract

- The possible states of the simple system when S receives a specific request and when S issues the responses.

- The possible activities in the interactions of S.

- The conditions for the occurrence of each interaction, i.e the guards of the arc.

- The parameters exchange between the interactions of S.

The pattern of an interaction machine of S contract

- An initial state.

- The notes representing the possible interaction states of simple dynamic system representing S.

- The arcs representing the possible transitions of simple dynamic system representing S, labelled by an interaction occurrence, a guard and an effect in the form of *interact-occur [guard]*, where *interaction-occur* may be an input interaction and an output interaction that are defined in S interface.

- A number of final states according to a number of possible ends of S behaviors.

Building steps

1. Start the graph with an initial state

2. Build the arc for the first input interaction.
3. Define and name the next states of the simple dynamic system representing S when S receives the requests or processes the responses.
4. Define the attributes of the data of the input interaction and their possible values to build the boolean expressions for the guards of the following transitions.
5. Corresponding to the specific guards, build the next arcs for the following transitions.
6. Mark the points where the behaviours end in final states.

Note:

- An interaction machine may have any number of final states.
- Whenever there is an arc with a condition there should be also the arc with opposite condition.
- If no condition is satisfied for an elementary interaction, the matching interaction will never be matched.

Give the simple dynamic type representing the realm of S
Model interaction machine of a semantics

What is supposed to be clear before building an interaction machine in a semantics

- The service contract
- The domain of the value of the simple dynamic system attributes.
- The effect of the activities of the simple dynamic system.

The pattern of an interaction machine in a semantics

- An initial state.
- The notes represent the possible interaction states of the system.
- The arcs represent the possible transitions of the system, labelled by an interaction occurrence, a guard and an effect in the form of *interact-occur [guard] / effect*, where *interaction-occur* may be a input interaction and a output interaction that are defined in the provided service interface, the *[guard]* is the

boolean expression built over the simple dynamic system attributes, and the *effect* is the action over those attributes.

- Final states (also none).

Building steps an interaction machine in a semantics
On the basis of the provided interaction machine of the service contract,

1. Create the arcs to define the initial value for the simple dynamic system attributes if any.

2. Supplement the possible value of the attributes of the simple dynamic system to create the *guard* for the transitions.

3. Define the effect of the interactions over the simple dynamic system attributes to create the *effect* for the transitions if any.

Note:

- The main difference between the service contract and the semantics is the presence of the values of the Simple system attributes in the *effect* of the transitions.

Model participant P
A-Model monolithic participant P
What to do
Define the monolithic participant P and identify the services that P provides and consumes
Building steps

- Use CASL-MDL simple dynamic type with stereotype ≪Participant≫ to define P, name this simple system with the name of P.

- Create a port with a lollipop for each service that P provides, name this port by the service name.

- Create a port with a cup for each service that P consumes, name this port by the service name starting with conjunction ∼.

B-Model structured participant P
What to do
Identify all subparticipants of P and the local services that they provides and consumes.
Building steps

- Use a structured system (i.e., CASL-MDL structured dynamic type) with stereotype ≪Participant≫ to define P, name this structured system by the name of P.
- Use a subsystem to define each subparticipant. Insert them into the structured system P.
- For each subparticipant, create a port for a service that it offers or uses.
- Connect the pair of participants involved in a service by a connector labelled by the name the service.
- Create a port for participant P for a service that it provides (a port having a lollipop) or consumes (a port having a cup). Connect this port to the corresponding port of subparticipant which provides or consumes this service.

Model structured participant P as service architecture of the service system

A service architecture is represented by a CASL-MDL structured system of a number of subsystems.

What is supposed to be clear before building a service architecture:

- All the participant types of the service system.
- All the services of the system.

The composition of the service architecture

A structured system stereotyped ≪serviceArchitecture≫ includes:

- The participant types are represented by subsystems, on which the ports are the points they offer or use the service.
- The connectors between the participants.

Building steps for a service architecture:

- Use a structured system (i.e., CASL-MDL structured dynamic type) with stereotype ≪Service Architecture≫ to define P, name this structured system by the name of service system.
- Use a subsystem to define each subparticipant. Insert them into the structured system P.
- For each subparticipant, create a port for a service that it offers or uses.

- Connect the pair of participants involved in a service by a connector labelled by the name the service.

Note:

- The connector leaves and ends at the ports of the participants.
- The interactions occurring among the participants are in terms of services.

Model service S in property oriented style

A-Build the contract of service S in property oriented style

The contract of S in property oriented style characterized by formulas expresses all possible occurrences of the interactions in the particular conditions.

What is supposed to be clear before building the contract of service S in property oriented style

- The protocols to be followed by the providers and the consumers of the service, what they guarantee to each other and what they expect in terms of dynamic behaviour.
- What will happen when the service receives a specific request.
- What conditions match each interaction and how they match with each other, such as what conditions for a request to be accepted, or what conditions for a response be implemented.
- The attributes of the data in a request sent to the service by the consumers, what is true or not.
- The relationships between the interactions (the order, the dependency and the priority)

The composition of S contract in property oriented style

- A set of formulas expresses all possible cases of the occurrence of the interactions defined in the provided service interface according to the specific matching conditions.

Building steps

1. Define the context of the transition of the Simple system to select the quantifications on paths and quantification on states if any.

2. Define the possible values of the parameters and the attributes of the data of the specific input interaction, combine them with this input interaction to build the **premise** of the formulas.

3. Based on the relationships between the conditions and the interactions, between the input interactions and the output interactions, use the logic combinators and given quantifications to build the **conclusion** of the formula, such that each formula may expresses all possible interactions of the system corresponding to each specific premise part.

B-Model the semantics of service S in property oriented style

The semantics of S in property oriented style not only expresses all possible occurrences of the interactions in the particular conditions but also the possible internal transitions associating with the effect of the transitions if any.

What is supposed to be clear before building the semantics of S in property oriented style

- What contributes the answer of the service.
- What is examined, controlled, affected by the service itself.
- The provided attributes of the simple system that can determine its internal states, the internal actions over those attributes.
- The effect of the request sent to the service itself.
- The effect of the response to the service itself.

The content of S semantics in property oriented style

- A set of formulas expresses all possible cases of the occurrence of the interactions defined in the provided service interface according to the specific matching conditions; express the internal states of the simple system and the internal actions over the simple system attributes associating with such interactions.

Building steps

1. On the basic of the premise parts of the provided formulas in the service contract in property oriented style, combine them with the possible properties of the attributes of the simple system to create premise parts for the formulas.

2. Corresponding to the premise parts, create the the conclusion parts of the formula for the occurrences of the interactions and supplement the effect of the transitions if any.

3. Create necessary formulas to define the conditions for the simple system attributes to guarantee the realization of the transitions.

Note: The effect of the transition is usually attached with the corresponding interaction by logic combinator AND.

7.5 Dealer Network Model in Casl4Soa

In this section, we model the Dealer Network using Casl4Soa, first following the constructive style and later following the property-oriented one. The description of Dealer Network case study has been given in Sect. 3.1.2.

The Dealer Network is a service system, and thus is a special structured participant without ports for offering and consuming services. The Dealer Network Casl4Soa model, in both styles, is thus a special case of participant model consisting of:

- the definition of a dynamic type stereotyped by ≪ServiceSystem≫ corresponding to the Dealer Network and named DealerNetwork, by means of a type diagram including such type; DealerNetwork is a Casl-Mdl structured dynamic type (thus the definition of this type will also express the service architecture of the Dealer Network);
- the models of the local services, that are Place Order, Get Ship Status, and Request Shipping;
- the models of (the types of) its subparticipants, that are Dealer, Manufacturer, and Shipper.

7.5.1 Dealer Network Model in Casl4Soa: constructive style

7.5.1.1 DealerNetwork type

DealerNetwork, the dynamic type for the service system corresponding to the Dealer Network, is shown in Fig. 7.11 by a box (the dynamic type icon of Casl-Mdl) stereotyped by ≪ServiceSystem≫ to express that it is a service system. Recall that in Casl4Soa, we denote the participants of a service system (as well as the subparticipants of a generic structured participant) by subsystems in the structured system stereotyped by ≪ServiceSystem≫ (stereotyped by ≪Participant≫). Each subsystem (depicted by a box) represents a role for (sub)participants of a specific type (the type name is depicted in the box after the colon). Moreover, the fact that

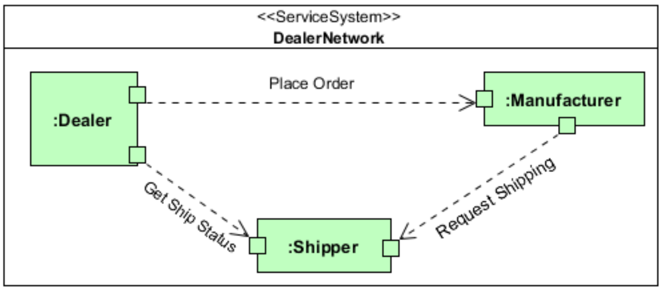


Figure 7.11: DealerNetwork type

a (sub)participant interact with another (sub)participant by means of a service is shown by a dashed arrow going from the port of who uses the service towards the port of who provides it. Thus Fig. 7.11 shows also the service architecture of Dealer Network in Casl4Soa. We can see that there are several participants of three different types Dealer, Manufacturer, and Shipper. Each participant has a number of ports to indicate that it provides or consumes various services. For example, the participants typed by Shipper have two ports for the two services that they provide, they are Get Ship Status and Request Shipping. The dashed arrows entering in the Shipper ports are labelled by the names of the services that it provides.

7.5.1.2 Local Service Models

Service Place Order

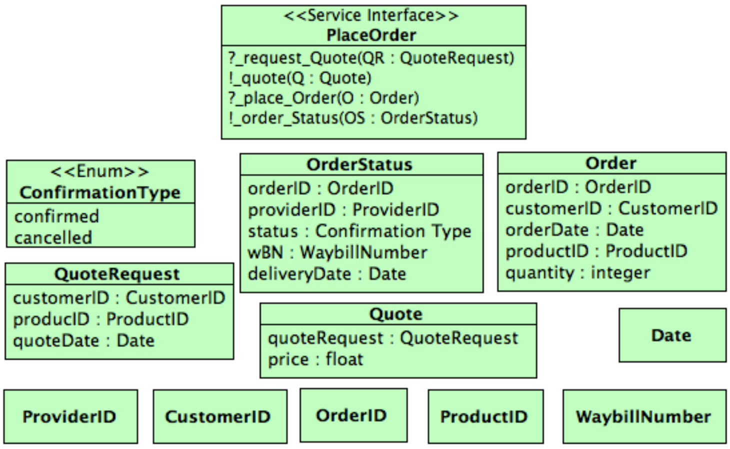


Figure 7.12: Place Order Service: interface

Service Interface The interface of service Place Order, shown in Fig. 7.12, is a type diagram that contains a dynamic type named Place Order with stereotype

≪serviceInterface≫ modelling the the service interface itself, and the definitions of the datatypes needed to type the parameters of its interactions.

The interface of service Place Order consists of four interactions: ?_request_Quote and ?_place_Order of kind input, and !_quote and !_order_Status of kind output. The service can receive the quote request from the buyers by interaction ?_request_Quote with a parameter typed by the datatype QuoteRequest, then the service may respond with the quote contained in a parameter typed by datatype Quote of the output interaction !_quote. When the service receives the request to place an order from the buyer by the interaction ?_place_Order, it will respond with a confirmation by the interaction !_order_Status to communicate the status of the order. An order may be confirmed or cancelled, and the attribute status of the datatype OrderStatus typed by the enumeration type ConfirmationType contain this information. Moreover, if the order is confirmed, the buyer will receives further information about the order contained in the other attributes of OrderStatus, they are the providerID of the order, the delivery date of the shipment, and the wBN (waybill number) of the shipment. The identification of the buyer is defined by the attribute customerID in the definition of datatype QuoteRequest and Order.

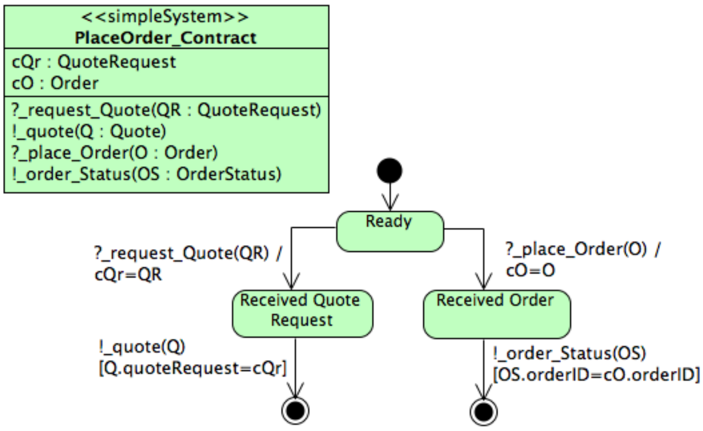


Figure 7.13: Place Order service: contract (constructive style)

Service Contract The contract of service Place Order is represented by the simple dynamic system PlaceOrder_Contract and the interaction machine shown in Fig. 7.13; this dynamic system has the same interactions of the one modelling the service interface, and all of them will appear on transitions between of this interaction machine. The interaction machine expresses that the service may receive two requests: the quote request and the order placement from the consumers, through the two transitions leaving the state Ready labelled respectively by ?_request_Quote(qr:QuoteRequest) and ?_place_Order(O). The attribute cQr of the simple dynamic system allows to store the value of the parameter quote request QR; similarly, the attribute cO stores the order O. Those values are used in the transition guards to determine the cases when the interactions may happen. For instance, to guarantee that the quote of each quote request will correspond to such request, the guard for the output interaction !_quote(Q) should be [Q.QuoteRequest=cQr].

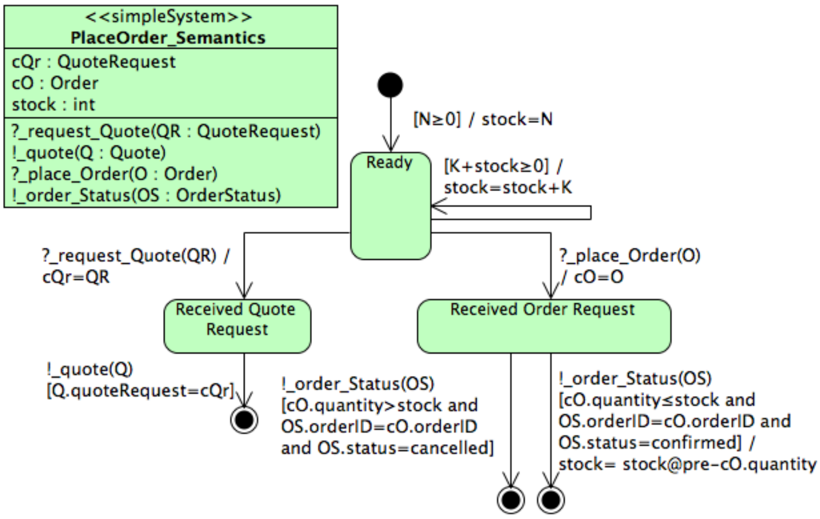

<<simpleSystem>>
PlaceOrder_Semantics
cQr : QuoteRequest
cO : Order
stock : int
?_request_Quote(QR : QuoteRequest)
!_quote(Q : Quote)
?_place_Order(O : Order)
!_order_Status(OS : OrderStatus)
[N≥0] / stock=N
Ready
[K+stock≥0] / stock=stock+K
?_request_Quote(QR) / cQr=QR
?_place_Order(O) / cO=O
Received Quote Request
Received Order Request
!_quote(Q) [Q.quoteRequest=cQr]
!_order_Status(OS) [cO.quantity>stock and OS.orderID=cO.orderID and OS.status=cancelled]
!_order_Status(OS) [cO.quantity≤stock and OS.orderID=cO.orderID and OS.status=confirmed] / stock= stock@pre-cO.quantity

Figure 7.14: Place Order service: semantics (constructive style)

Semantic View The semantics of service Place Order is modelled by the simple dynamic system PlaceOrder_Semantics and the interaction machine shown in Fig. 7.14. In this example, the interaction machine of the semantics has a similar shape to the one of the service contract (see Fig. 7.13), however it takes also into account the information about the realm. In this case, the quantity of product in stock is this information, thus we introduce attribute stock in system PlaceOrder_Semantics. The interaction machine in Fig. 7.14 models that, if the ordered quantity is greater than the product quantity in stock, the order will be cancelled; otherwise the order will be confirmed, and the product quantity in stock will be reduced by the ordered quantity.

Service Request Shipping

Fig. 7.15 shows the interface of the service Request Shipping. The service provides a means to require a shipping request by ?_request_Shipping(R:Request) and it can respond the two possible results, one for package packing and another for confirmation of delivery, in !_package_PickUp(PP:PackagePickUp)

and !_delivery_Confirmation(DC:Confirmation).

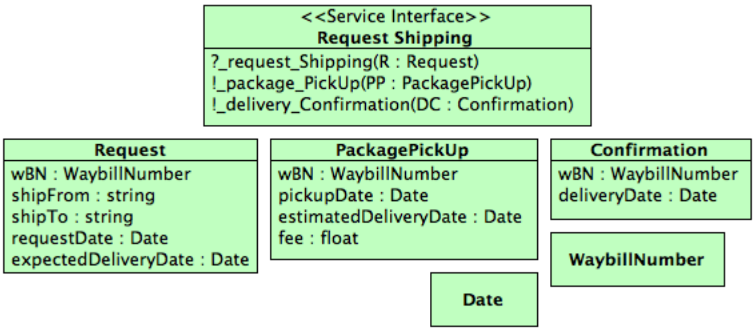


Figure 7.15: Request Shipping service: interface

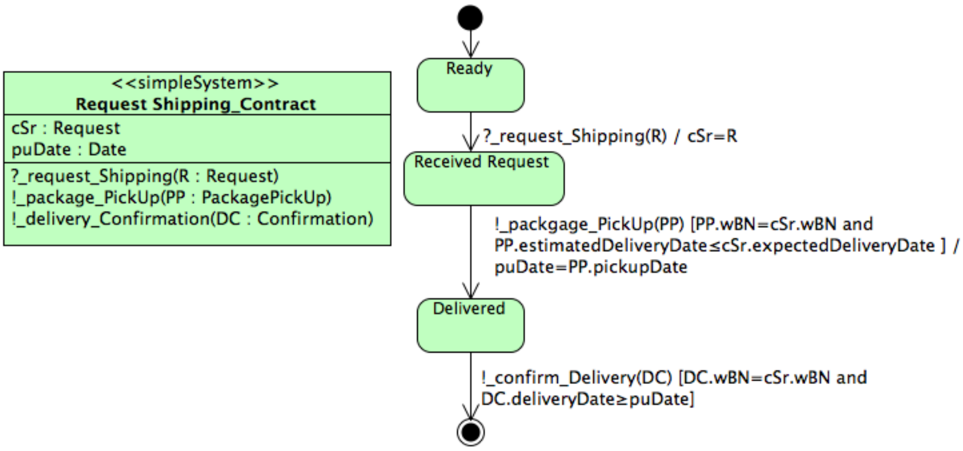

<<simpleSystem>>
Request Shipping_Contract
cSr : Request
puDate : Date
?_request_Shipping(R : Request)
!_package_PickUp(PP : PackagePickUp)
!_delivery_Confirmation(DC : Confirmation)
Ready
?_request_Shipping(R) / cSr=R
Received Request
!_packgage_PickUp(PP) [PP.wBN=cSr.wBN and
PP.estimatedDeliveryDate≤cSr.expectedDeliveryDate] /
puDate=PP.pickupDate
Delivered
!_confirm_Delivery(DC) [DC.wBN=cSr.wBN and
DC.deliveryDate≥puDate]

Figure 7.16: Request Shipping service: contract (constructive style)

Service Get Ship Status

Fig. 7.17 shows the interface of the service Get Ship Status. The service provides a means to require the status of a shipment by message ?_get_ShipmentStatus(W:WaybillNumber) and it can respond the result in !_shipmentStatus(SS:ShipmentStatus).

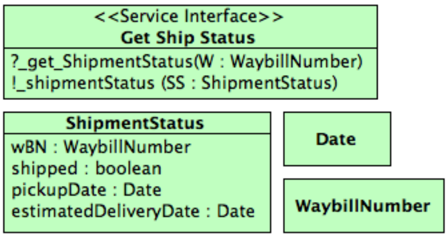


Figure 7.17: Get Ship Status service: interface

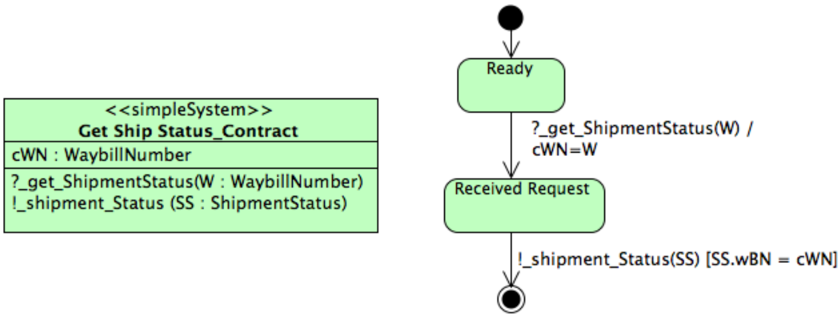

<<simpleSystem>>
Get Ship Status_Contract
cWN : WaybillNumber
?_get_ShipmentStatus(W : WaybillNumber)
!_shipment_Status (SS : ShipmentStatus)
Ready
?_get_ShipmentStatus(W) /
cWN=W
Received Request
!_shipment_Status(SS) [SS.wBN = cWN]

Figure 7.18: Get Ship Status service: contract (constructive style)

The semantics is not very interesting and we do not describe the realm of the service, since that should include all the situations of the ships and of the sea and of the harbors, etc., so we should not present it.

7.5.1.3 Participant models

The models of the three types of participants of Dealer Network (Dealer, Manufacturer, and Shipper) are shown in Fig. 7.19, to be more precise in such figure we show only the three type diagrams defining the three corresponding dynamic types stereotyped by ≪Participant≫, while for simplicity we do not duplicate the models of the services that they offer and use, since they have been already presented in the part about the local services of Dealer Network. We do not add any other information on these three participants, since we do not know anything other on them (e.g., if they are monolithic or structured, and what is their behaviour). CASL4SOA allows also these kind of specifications

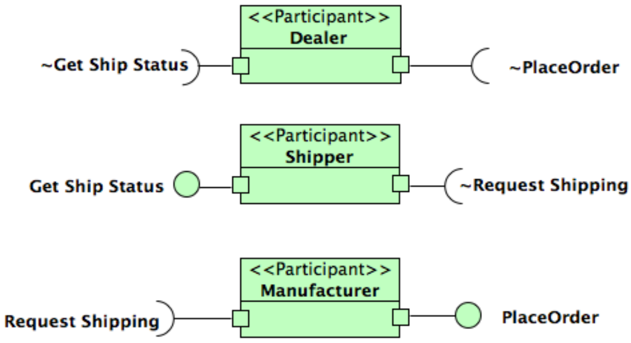


Figure 7.19: Dealer Network service system: participant models

7.5.2 Dealer Network Model in CASL4SOA: property-oriented style

The CASL4SOA Dealer Network model made following the property-oriented style has the same structure of the constructive one presented in subsection 7.5.1, the only different parts are the definitions of the contracts of the three services (Place Order, Get Ship Status and Request Shipping) and of the semantics of Place Order. In this case they are defined by means of sets of constraints, i.e., set of temporal logic formulas. We present these contracts and this semantics in Fig. 7.20, 7.22, 7.23, and 7.21 respectively.

Fig. 7.20 presents the contract of service Place Order in property oriented style. The first formula expresses that whenever (in_any_case) the service receives a quote request, it will always (always) respond with a corresponding quote. The second formula expresses that whenever the service receives an order request, it will always respond with a corresponding order status which includes the order status

```
in_any_case always
   ?_request_Quote(QR) ⇒
      ∃Q:Quote • (Q.quoteRequest=QR ∧ eventually !_quote(Q))

in_any_case always
   ?_place_Order(O) ⇒
      ∃OS:OrderStatus • (OS.orderID=O.orderID ∧eventually !_order_Status(OS))
```

Figure 7.20: Place Order service: contract (property-oriented style)

```
in_any_case always
   ?_request_Quote(QR) ⇒ ∃ Q:Quote • (Q.quoteRequest=QR ∧ eventually !_quote(Q))

in_any_case always
   ( ?_place_Order(O) ∧ O.quantity≤ stock) ⇒
      ∃ OS:OrderStatus • (OS.orderID=O.orderID ∧ OS.status=confirmed ∧
         eventually (!_order_Status(OS) ∧ stock=stock@pre-O.quantity))

in_any_case always
   ?_place_Order(O) ∧ O.quantity>stock ⇒
      ∃ OS:OrderStatus • (OS.orderID=O.orderID ∧ OS.status=cancelled ∧
         eventually !_order_Status(OS))

in_any_case always
   ∃ K:int • (K+ stock≥ 0 ∧ eventually stock = stock@pre +K)
```

Figure 7.21: Place Order service: semantics (property-oriented style)

that may be either confirmed or cancelled. The correspondence is guaranteed by Q.quoteRequest=QR.

In the property-oriented style, the semantics of service Place Order is represented by a set of formulas (see Fig. 7.21). The first formula is the same as the first one in the service contract (see Fig. 7.20), because the quotation does not depend on the service realm. The second formula expresses that if the ordered quantity is less than or equal to stock, then the order will be confirmed, and an order status with attribute status equal to confirmed will be sent to the buyer; stock=stock@pre-O.quantity expresses that the stock will be reduced. Otherwise, as expressed by the third formula, if the ordered quantity is greater than stock, the order will be cancelled, and an order status with attribute status equal to cancelled will be sent to the buyer. The last formula states that the stock can always be modified adding or removing goods.

Fig. 7.23 shows the contract in property style of service Get Ship Status. It contains formula expressing that whenever the service receives a request for status of a

```
in_any_case always
   ?_request_Shipping(R) ⇒
      (eventually ∃ PP:PackagePickUp •
         (PP.wBN=R.wBN ∧
          PP.estimatedDeliveryDate ≤ R.expectedDeliveryDate ∧
          !_package_PickUp(PP))
          ∧
          eventually ∃ DC:DeliveryConfirmation •
             (DC.wBN=R.wBN ∧
              DC.deliveryDate ≥ PP.pickupDate ∧ !_confirm_Delivery(DC))
      )
```

Figure 7.22: Request Shipping service: contract (property oriented style)

```
in_any_case always
   ?_get_ShipmentStatus(W) ⇒
      eventually ∃ SS:ShipmentStatus • (SS.wBN=W ∧ !_shipment_Status(SS))
```

Figure 7.23: Get Ship Status service: contract (property oriented style)

shipment, it will respond the information that exists.

7.6 Casl4Soa Model of Office System

In this section, we model the Office System using Casl4Soa, first following the constructive style and later following the property-oriented one. The description of the Office System case study has been given in Sect. 3.2.

The Office System is a service system, and thus is a special structured participant without ports for offering and consuming services. The Office System Casl4Soa model, in both styles, is thus a participant model consisting of:

- the definition of a dynamic type stereotyped by ≪ServiceSystem≫ corresponding to the Office System and named Office System, by means of a type diagram including such type; Office System is a special case of a Casl-Mdl structured dynamic type (thus the definition of this type will also express the service architecture of the Office System);

- the models of the local services, that are Print, Check Italian, Check French, Check English and Publish on Web;

- the models of (the types of) its subparticipants, that are PrintingCenter, EnglishCenter, ItalianCenter, FrenchCenter, WebPublisher, and OfficeComponent.

7.6.1 Casl4Soa constructive model of Office System

7.6.1.1 Office System

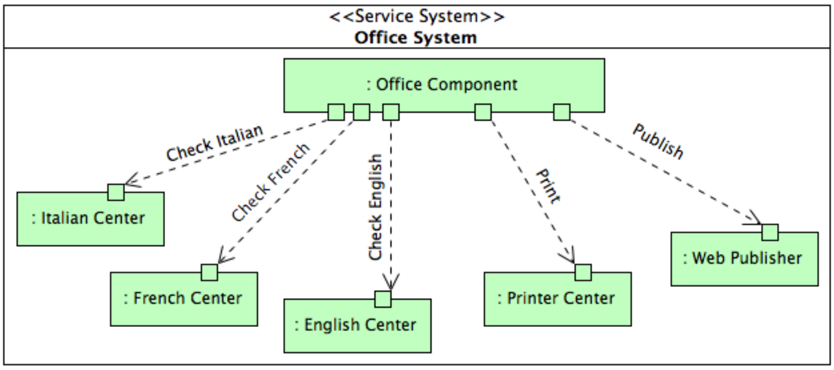


Figure 7.24: **Office System: service architecture**

The type Office System is defined by the type diagram shown in Fig. 7.24. It is a structured system that is made participants of six different types providing and consuming five different services. The definition of the Office System type gives also the architecture of the service system Office System, showing which (roles for) participants of the various types use which services provided by which other (roles of) participant types. The fact that a participant uses a service provided by another one is shown by means of a dashed arrow going from the user to the provider and labelled by the name of the service. For example, we can see that OfficeComponent uses the Print service provided by the PrintingCenter, also it is clear that in this service system the participants typed by OfficeComponent uses the services provided by the participants of all the other types, and that the latter do not interact among them.

7.6.1.2 Local Service Model

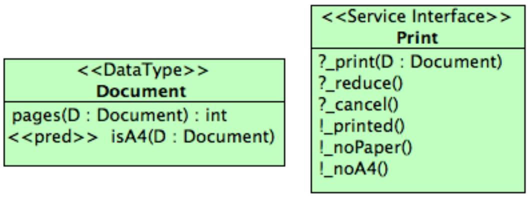


Figure 7.25: Print service interface

Service Print The interface of service Print is shown in Fig. 7.25. Notice that also the datatypes defining the parameters of the in and out messages are defined in this type diagram, for example Document defines the printable documents; it contains an operation pages and a predicate isA4 returning respectively the number of pages of the document and the indication if the size of its pages is equal to A4.

Fig. 7.26 presents the contract of the service Print by means of a simple system and of an associated interaction machine. The interaction of this simple system are exactly the interactions appearing in the interface of the service define din Fig. 7.25, which correspond to the messages received and sent by the service; all of them appear at least on a transition of the interaction machine.

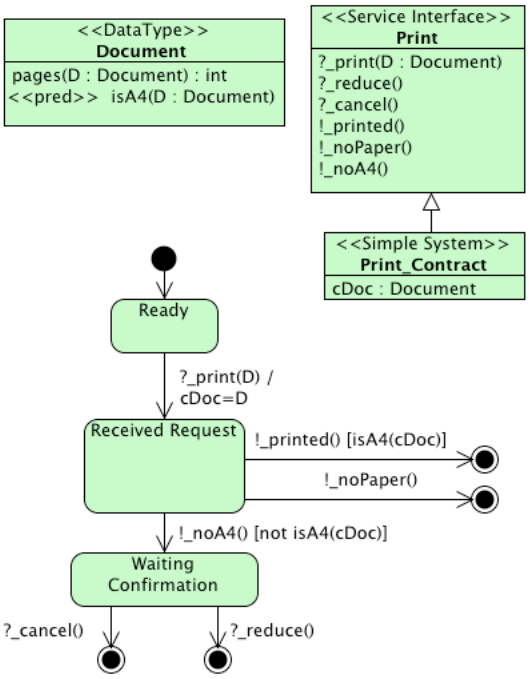


Figure 7.26: Print service contract (constructive style)

The realm of the service Print is described by the paper quantity available, modelled by the attribute paper in the simple system Print_Semantics in Fig. 7.27.

The semantics of the service Print shown in Fig. 7.27 is defined by means of a simple dynamic type and an associated interaction machine. We can see now that the reason for getting the message !_noPaper; moreover, if we get the message !_noA4 we know that there is however enough paper to print the document (we can have a different service that after having received the message reduce may inform you that there is not enough paper to print). Moreover, we also know that each time a document is printed, the paper quantity will be decreased exactly by the number of its pages, and this is modelled by the effect / paper = paper - pages(cDoc) of the corresponding transition (again a different service may consume an extra page printing a forefront with the information on the data and the time of the printing).

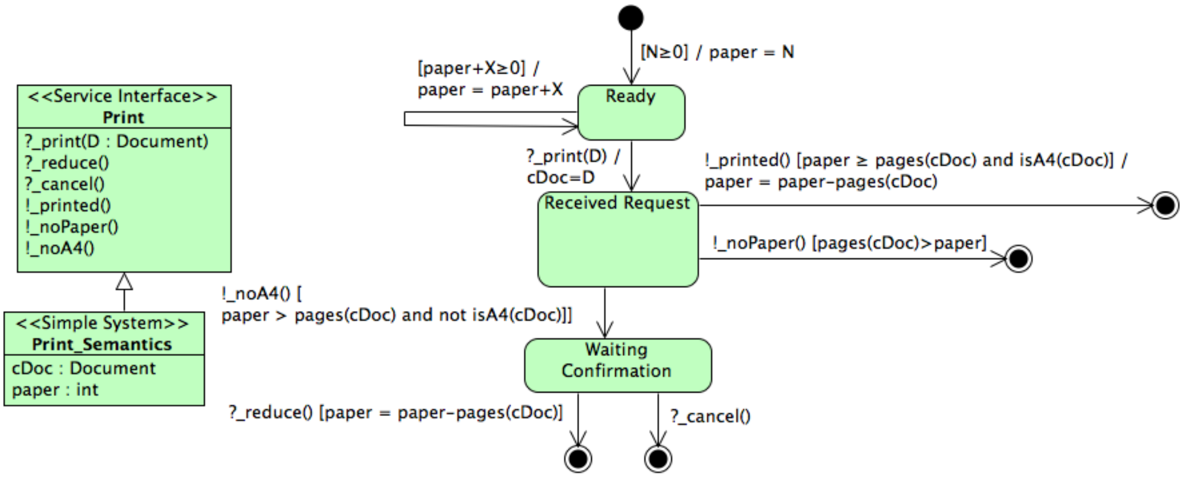


Figure 7.27: Print service semantics (constructive style)

Services Check Italian Here we consider only the service Check Italian, the models of Check French and of Check English are similar.

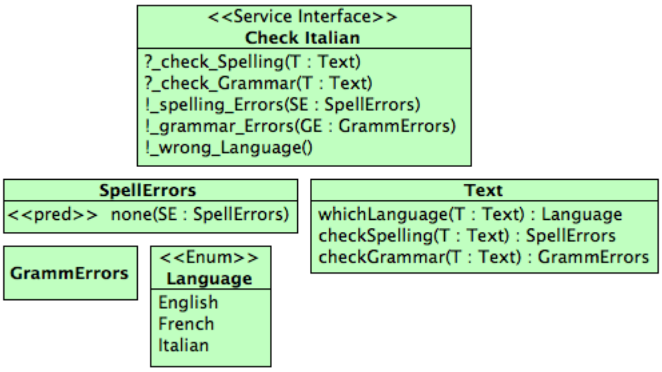

<<Service Interface>>
Check Italian
?_check_Spelling(T : Text)
?_check_Grammar(T : Text)
!_spelling_Errors(SE : SpellErrors)
!_grammar_Errors(GE : GrammErrors)
!_wrong_Language()
SpellErrors
<<pred>> none(SE : SpellErrors)
Text
whichLanguage(T : Text) : Language
checkSpelling(T : Text) : SpellErrors
checkGrammar(T : Text) : GrammErrors
GrammErrors
<<Enum>>
Language
English
French
Italian

Figure 7.28: Check Italian interface

Fig. 7.28 shows the interfaces of service Check Italian. The service offers two types of checking: the *spelling* and the *grammar* checking, that may be required by using the two input interactions of the service: ?_check_spelling and ?_check_Grammar. The content and the structure of the text is defined by the datatype Text, the spelling and grammar errors by the datatypes SpellErrors, and GrammErrors. The language of a text is detected by the operation whichLanguage of Text. The return value of this operation can be one of the three values: English, French and Italian, which are listed in the enumeration type Language.

In the service contract in Fig. 7.29, there are five final states illustrating the five results provided by service Check Italian in all possible cases. Before to check the

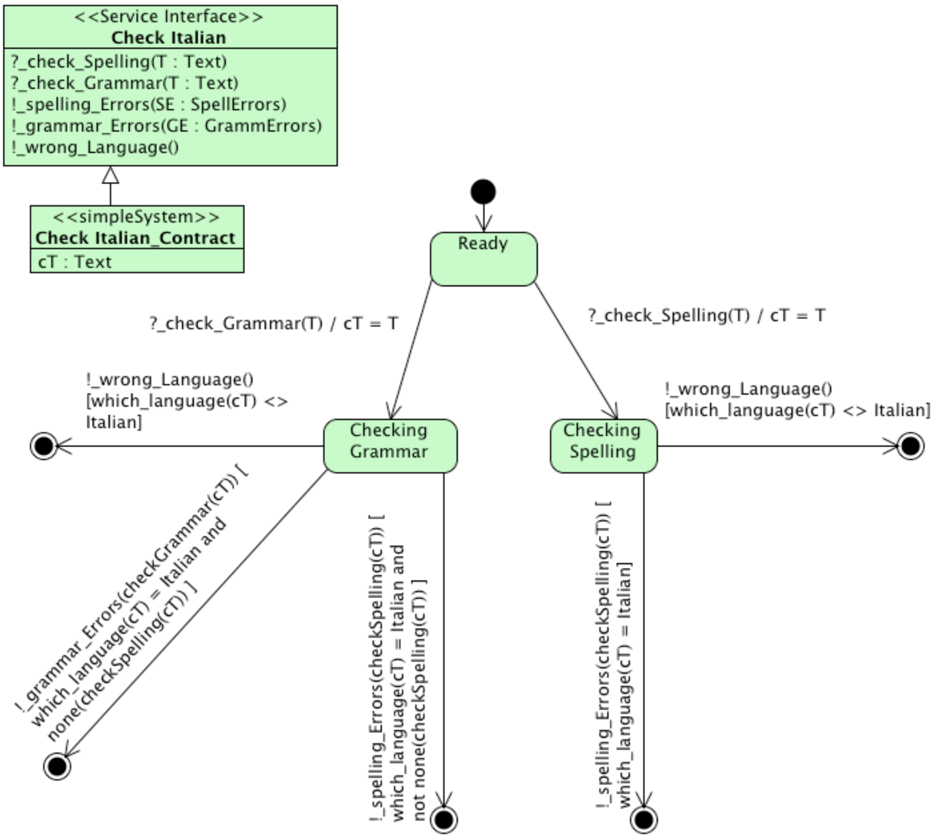

<<Service Interface>>
Check Italian
?_check_Spelling(T : Text)
?_check_Grammar(T : Text)
!_spelling_Errors(SE : SpellErrors)
!_grammar_Errors(GE : GrammErrors)
!_wrong_Language()
<<simpleSystem>>
Check Italian_Contract
cT : Text
Ready
?_check_Grammar(T) / cT = T
?_check_Spelling(T) / cT = T
!_wrong_Language() [which_language(cT) <> Italian]
Checking Grammar
Checking Spelling
!_wrong_Language() [which_language(cT) <> Italian]
!_grammar_Errors(checkGrammar(cT)) [which_language(cT) = Italian and none(checkSpelling(cT))]
!_spelling_Errors(checkSpelling(cT)) [which_language(cT) = Italian and not none(checkSpelling(cT))]
!_spelling_Errors(checkSpelling(cT)) [which_language(cT) = Italian]

Figure 7.29: Check Italian contract (Constructive Style)

spelling or the grammar, the service will check if the submitted text is written in Italian. If not, the service will send the message !_wrongLanguage. If the service finds some errors in the text, it will return the list of the found errors. If there is no error, this list will be empty. If there are spelling errors in a text required to be grammar checked, the service will return the spelling errors. It means that the service performs the grammar check if only if there are no spelling errors in the text.

We do not give the semantics of service Check Italian, since this service does not depend on a realm: we assume that the correctness of the spelling and of a language does not depend on any changeable aspects of the real world, and moreover this service does not modify anything. If it is relevant to precisely specify how the spelling and grammar checks are made, it is possible to enrich the definition of the datatype Text by operation definitions or by constraints.

Service Publish on Web Fig. 7.30 presents the interface of the service Publish on Web, whereas its contract is shown in Fig. 7.31. When a page published on the web, then the following conditions are satisfied: the page is in HTML format, the URL that the user provides is correct and its server is available at this time. In order

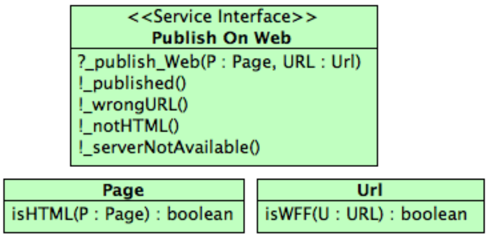


Figure 7.30: Publish on Web service interface

to check whether the page is written in HTML and the provided URL is correct or not, two operations isHTML(P) and isWFF(U) are defined in the datatypes Page and URL respectively.

If the page is not in HTML, the service will communicate !_notHTML(), whereas if it is in HTML but the URL is ill-formed the service will send another error message. Finally, if both the document is in HTML and the URL is correct, the service may still send the error message about the server of the URL being not available.

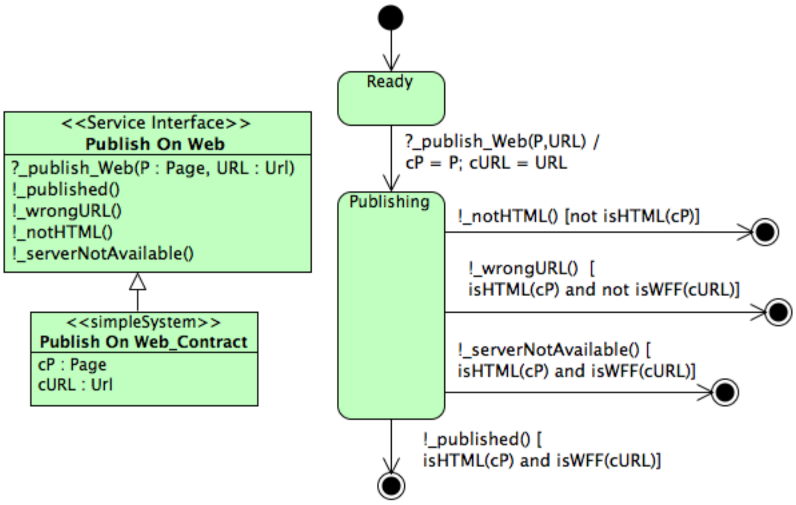


Figure 7.31: Publish on Web service: contract (constructive style)

The semantic view of service Publish on Web concerns the availability of the Web servers, and thus this is its realm, and it is modelled by the datatype Web The operations up and down modify the web status making a server available and not available respectively, whereas the predicate on checks if a server associated with a url is available.

7.6.1.3 Participants

Six participant types of Office System are collected in the type diagram in Fig.7.33. They are represented by six ≪Participant≫ classes having ports typed by interfaces

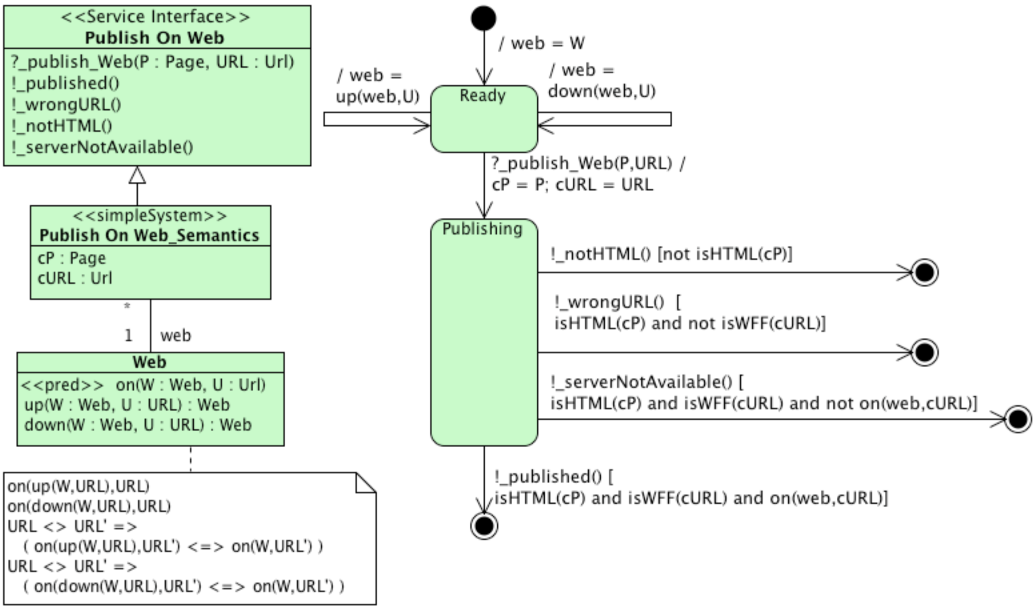

<<Service Interface>>
Publish On Web
?_publish_Web(P : Page, URL : Url)
!_published()
!_wrongURL()
!_notHTML()
!_serverNotAvailable()
<<simpleSystem>>
Publish On Web_Semantics
cP : Page
cURL : Url
*
1
web
Web
<<pred>> on(W : Web, U : Url)
up(W : Web, U : URL) : Web
down(W : Web, U : URL) : Web
on(up(W,URL),URL)
on(down(W,URL),URL)
URL <> URL' =>
(on(up(W,URL),URL') <=> on(W,URL'))
URL <> URL' =>
(on(down(W,URL),URL') <=> on(W,URL'))
/ web = W
/ web = up(web,U)
/ web = down(web,U)
Ready
?_publish_Web(P,URL) / cP = P; cURL = URL
Publishing
!_notHTML() [not isHTML(cP)]
!_wrongURL() [isHTML(cP) and not isWFF(cURL)]
!_serverNotAvailable() [isHTML(cP) and isWFF(cURL) and not on(web,cURL)]
!_published() [isHTML(cP) and isWFF(cURL) and on(web,cURL)]

Figure 7.32: Publish on Web service semantics (constructive style)

of the services they provide and consume. ≪Participant≫ Office Component is the participant type which only consumes services, thus all its ports are typed interfaces shown by the "cup" notation. Meanwhile, other participant types are providing participants which their ports are typed interfaces shown by the "lollipop" notation, for instance, ≪Participant≫ Printer Center has got the port Print representing for service Print that it provides.

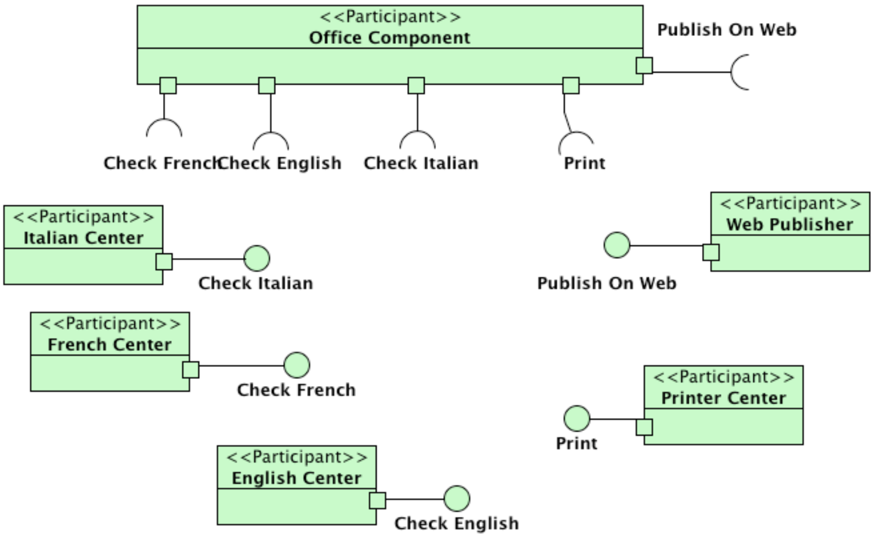


Figure 7.33: Office System participants models

7.6.2 Property-oriented Casl4Soa Model of Office System

In this section, we model Office System using Casl4Soa in a property-oriented way, giving only the contracts and the semantics of the different services, since all the other parts are the same as in the constructive mode in Sect. 7.6.1.

```
in_any_case always
       (isA4(D) ∧ pages(D) ≤ paper ∧ ?_print(D)) ⇒
              eventually (!_printed() ∧
                     (paper = N ⇒ next paper = N-pages(D))
in_any_case always
       (not isA4(D) ∧ pages(D) ≤ paper and ?_print(D)) ⇒
              eventually (!_noA4() ∧
                     eventually ((?_reduce()∧ (paper = N ⇒ next paper=N-pages(D)))
                                  ∨ ?_cancel() ))
in_any_case always
       (pages(D) > paper ∧ ?_print(D)) ⇒ eventually !_noPaper()
in_any_case always
       ∃ X.X>0 ⇒ eventually paper=paper+X
```

Figure 7.34: **Print service: semantics (property oriented style)**

The semantics in property oriented style of service Print is characterized by formulas that are more precise than those in the contract. They describe not only the possible transitions of the system but also what justifies the answer, what is true or not, and the effects of the requests. For example, in Fig. 7.34, the second formula expresses that whenever the service receives printing request ?_print(D), meanwhile the paper is not in A4 format not isA4(D) and the printer has enough paper (pages(D) ≤ paper), it will communicate that the paper is not in A4 !_noA4() and either cancel the printing ?_cancel() or receive the page reducing from the user ?_reduce(). If the user reduces the pages, the effect of this interaction is that paper will be decreased paper = N - pages(D). The last formula is defined in order to guarantee that the printer always will be refilled with paper.

The semantic view of service Publish on Web concerns the availability of the Web server. The datatype Web built in Fig. 7.32 can be also regarded as the Internet in general. The operation up(W,P) that returns the Web type expresses the available status of the Web server, otherwise the operation down(W,P) expresses an unavailable status. It is important to note that these statuses are temporal for an actual Web server at specific moment. Later on we shall use those operations to define the semantics in property oriented style for this service in Fig. 7.37.

Fig. 7.37 defines the properties of the Web server in two formulas that express the conditions for the Web server to be available. The Web server is available if and

```
in_any_case always
        ?_check_Spelling(T) ∧ whichLanguage(T)<>Italian ⇒
                eventually !_wrongLanguage()
in_any_case always
        ?_check_Spelling(T) ∧ whichLanguage(T)=Italian ⇒
                eventually !_spelling_Errors(checkSpelling(T))

in_any_case always
        ?_check_Grammar(T) ∧ whichLanguage(T)<>Italian ⇒
                eventually !_wrongLanguage()

in_any_case always
        ?_check_Grammar(T) ∧ whichLanguage(T)=Italian ∧ none(checkSpelling(T)) ⇒
                eventually !_grammar_Errors(checkGrammar(T))

in_any_case always
        ?_check_Grammar(T) ∧ whichLanguage(T)=Italian ∧ not none(checkSpelling(T)) ⇒
                eventually !_spelling_Errors(checkSpelling(T))
```

Figure 7.35: Check Italian: contract (property oriented style)

```
in_any_case always
        (isHTML(P) ∧ isWFF(U) ∧ ?_webPublish(P,U)) ⇒
                eventually (!_published() ∨ !_serverNotAvailable())
in_any_case always
        (not isHTML(P) ∧ ?_webPublish(P,U)) ⇒
                eventually (!_notHTML())
in_any_case always
        ( isHTML(P) ∧ not isWFF(U) ?_webPublish(P,U)) ⇒
                eventually (!_wrongURL())
```

Figure 7.36: Publish on Web service: contract (property oriented style)

```
on(up(W,U),U)
U ≠ U' ⇒
        on(up(W,U),U')⇔ on(W,U')
not on(down(W,U),U)
U ≠U' ⇒
        on(down(W,U),U')⇔ on(W,U')
```

Figure 7.37: Publish on Web service: semantics (property oriented style)

only if it will turn back to the status up after it was in status down before, while the URL can be different. It means that if the Web server is available, it will not maintain the down status for ever. Therefore at last the page will be published in this case.

7.7 Tool support

7.7.1 Requirements - What we need from a tool?

There are model elements in the graphic diagrams in Casl4Soa that full respect of the UML2 standard, they are:

1. Interface: illustrated by means of UML class diagram
2. Contract in Constructive Style: illustrated by means of UML State machine diagram
3. Semantic in Constructive Style: illustrated by means of UML State machine diagram
4. Participant Model: illustrated by means of UML Component diagram

The Casl4Soa profile declares the stereotypes below:

- ≪Service Interface≫
- ≪simpleSystem≫
- ≪Participant≫

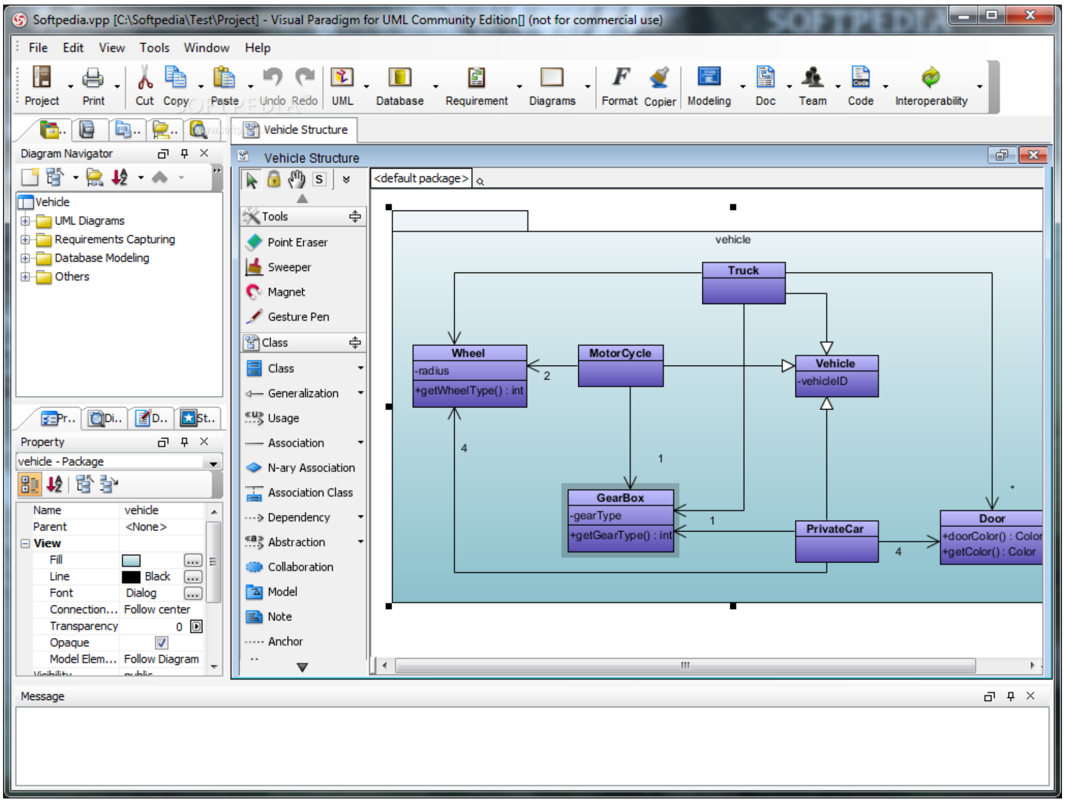
The tool supporting Casl4Soa is required to have capabilities followings:

- Create Casl4Soa Profile, in which, defining new stereotypes of Casl4Soa.
- Provide graphic diagram editors particularly supporting UML2, in which, it is possible to define new names of diagrams according to Casl4Soa
- Provide a text-based editor for the View in Property Oriented style as inspired by Casl4Soa.
- Export created diagrams into images.

7.7.2 Existing Tools - What we can do with existing tools?

Here, we do not consider commercial tools, instead we focus on free and open source tools. There are various UML tools supporting our aims, however for non-commercial versions, they all have some limitations. We find that there are two tools that are stable and nearly fully support the CASL4SOA profile. They are Visual Paradigm and Papyrus.

Visual Paradigm Community Edition (free version 8.3) Visual Paradigm is a UML design tool and UML CASE tool designed to aid software development. It has a good working environment, which facilitates viewing and manipulation of the modeling project. This is not an open source tool but there is free version. Link: http://www.visual-paradigm.com/



vehicle - Package

Name	vehicle
Parent	<None>
View	
Fill	
Line	Black
Font	Dialog
Connection...	Follow center
Transparency	0
Opaque	☑
Model Elem...	Follow Diagram

Figure 7.38: **Visual Paradigm version 8.3**

Usefulness:

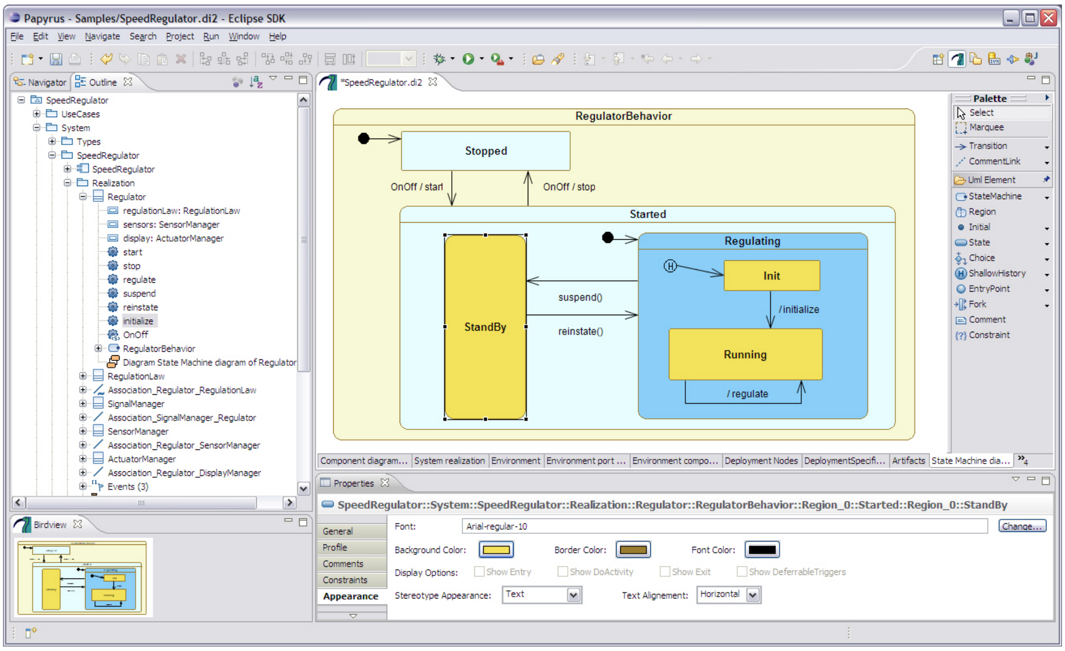
- Agility and flexibility in building graphic diagrams.
- Supply powerful customization mechanism, so we can define new name for the diagrams, define new stereotypes that we propose.
- We can use Visual Paradigm to create our five graphic diagrams.

Limit:

- Cannot creating Profile.
- Cannot export diagrams to images.

- No support for text-based editor for the View in Property Oriented style as inspired by Casl4Soa.

Papyrus UML (version 1.12.3, stand-alone) Papyrus is a dedicated tool for modelling within UML2. This open source tool is based on the Eclipse environment and is licensed under the EPL (Eclipse Public License). Link: http://www.papyrusuml.org/



SpeedRegulator::System::SpeedRegulator::Realization::Regulator::RegulatorBehavior::Region_0::Started::Region_0::StandBy

Font: Arial-regular-10

Background Color: 🟨 Border Color: 🟫 Font Color: ⬛

Display Options: □ Show Entry □ Show DoActivity □ Show Exit □ Show DeferrableTriggers

Stereotype Appearance: Text Text Alignement: Horizontal

Figure 7.39: **Papyrus UML version 1.12.3**

Usefulness:

- We can create Casl4Soa profile.
- We can build our five graphic diagrams.
- Support for exporting diagrams to images (extra plugin needed).

Limit:

- We cannot name the diagrams as we would need to.
- No support for text-based editor for the View in Property Oriented style as inspired by Casl4Soa.

7.7.3 Solution for Casl4Soa tool

The advantage of existing tools are that they offer various possibilities for model-based development and customization mechanism, however it seems that we cannot create a Casl4Soa project with only one of them. The “Open source” is a possible solution for us. To satisfy above requirements, we can integrate and extend the Eclipse plug-in Papyrus to build precise supporting environment for Casl4Soa. Where we can:

- Create Casl4Soa Profile
- Build seven diagrams of Casl4Soa metamodel
- Export created diagrams into images.

Conclusion: The work presented in this chapter is the development of a modelling method using formal notations called Casl4Soa (based on Casl-Ltl [45]) and Casl-Mdl [9]). This method is also designed over the SOA paradigm presented in Chapter 4. The form of the Casl4Soa models is presented together with the constraints defining the well-formed models. Besides giving the guidelines for building the models following this method, the use of this modelling method is also illustrated by the application on Dealer Network and Office System case studies (presented in Chapter 3).

Chapter 8

Service System Design

Contents

This chapter presents a method for designing a service system being built to support a business. This method follows the Model Driven approach [51, 22]. The starting points are a business model (defined in Chapter 5), and the description of some existing services (modelled using the UML following the PreciseSoa method presented in Chapter 6), and the target is a UML model of the design of a service system, again modelled following the indications of PreciseSoa.

The method is organized in five phases that are summarized in Fig. 8.1, starting with Place the System, followed by four transformation phases, i.e., Eliminate Useless Parts, Simplify Tasks, Operationalize Tasks, and Introduce Services, in order to transform a business model into the model of a design of a service system. A set of transformation patterns is provided to support those four transformation phases. To help the developer works, a common template is defined in Sect. 8.1 to describe all the transformation patterns.

The first phase, Place the System, presented in Sect. 8.2, requires to place the service system being built over the modelled business.

In phase Eliminate Useless Parts (Sect. 8.3) the parts of the model not covered by the placement will be eliminated.

In phase Simplify Tasks (Sect. 8.4), the tasks will be decomposed into smaller tasks. After this phase, the tasks are suitable to become operation calls to business entities (objects of entity classes).

In phase Operationalize Tasks (Sect. 8.5), the tasks will be transformed into the operation calls of the business entities.

In phase Introduce Services (Sect. 8.6), the entities of the business are transformed into participants of the service system, several new services are introduced, and the behaviour of the various participants will be defined in terms of service calls.

Since it follows the Model Driven approach, our design method will show how to transform the model of the business till to reach the model of the designed service system.

The resulting service system design model will include also some extra parts to show how the various processes of the business have been realized.

The various phases of the transformation activity cannot be fully automatized, since the designer has to make many choices during the transformation.

We recall that, here, the term “entity” is used for describing entities taking part in a business. However, in a service system, we use the term “participant” to denote the entities taking part in several services having the roles as service providers and service consumers. Thus, the entities of a business will be transformed into participants of a service system.

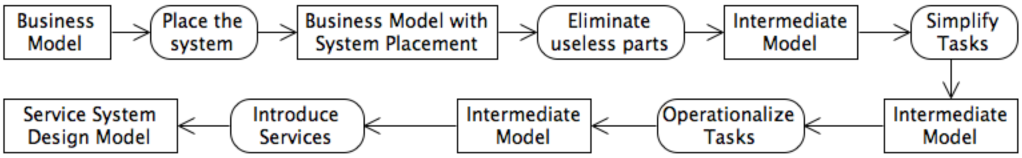


Figure 8.1: Service System Design Method Phases

The application of our design method is shown in Sect. 8.7, in which we applied some phases on the model of Dealer Business (adopted in Sect. 5.4).

Finally, to validate a designed service system, in Sect. 8.8, we propose a mechanism using UML activity diagram to model the realization of business processes in that system.

8.1 Transformation patterns

Our design method provides a set of transformation patterns to support the four phases, i.e., Eliminate Useless Parts, Simplify Tasks, Operationalize Tasks, and Introduce Services. These transformation patterns are about transforming UML models having particular forms and built using particular profiles. The transformation patterns have been inspired by the well-known design patterns[1].

A common template is built to describe all transformation patterns, and it is given here as following:

Transformation pattern: Name
Motivation: motivate the pattern
Description: informal description of the transformation provided by the pattern
Subjects: list of the parts of the model subjects of the transformation
Preconditions: conditions of the pattern subjects that they must be satisfied before to apply the pattern
What to do: schematic description of the transformation
Example: an illustrative example of application of the pattern

A transformation pattern describe a transformation on UML models; precisely, each transformation pattern describes how to transform an intermediate model into another intermediate model.

We call intermediate models all the various models produced during the various steps of the application of our method, they are of a kind of hybrid between the

[1] http://en.wikipedia.org/wiki/Software_design_pattern

business models and the service system models, using the profiles used in both these kinds of models and having the views part of both these two kinds of models.

The patterns will help the transformation, but obviously some steps may be done also without using a pattern. Indeed, there may be some modifications that can be done without using the patterns, for instance, changing the name or the definition of some datatypes, transforming two parameters of an operation in one, etc.

The transforming of a model refers to the work of adding, modifying, and eliminating some parts of this model. We may have to add some new parts to the model, to modify others, and to eliminate some other parts. Thus, for denoting the adding and the deleting of elements in a model, we use the mathematical symbols, i.e., + and -; e.g., ClassDiag + C denotes adding a class Cto the class diagram ClassDiag. We assume that these operations have the following properties:

A+A = A, it means that there is no duplication of the same elements in the result of the operation, and

(A-B)+B = A

(A+B = B+A

For illustrating of the transformation of a model defined by a pattern (in the section **What to do**), we sketch the state of this model before and after the transformation as following:

Model before | ⇒ | Model after

For illustrating of the replacement of an element by another element in a model, we denote such replacement as following:

M[e'/e]

where M denotes a model, e denotes an element that is replaced, and e' denotes the new element that replaces e in M.

Lest us give here an example of a transformation pattern that is described following this defined template:

Transformation pattern: Eliminating unused class

Motivation: To eliminate a class not used in any part of the model and outside the placement.

Description: An unused class is eliminated, and any association that will result pending after the class elimination is also eliminated from the model.

Subjects:

C, a class in the Static View

Ass, the set of the associations having Cas one of their ends

Preconditions:

Cis an unused class

What to do:

SV		SV - C - Ass
TV		TV - C - Ass
BPOD	⇒	BPOD
BPM		
BPMs		BPMs

Example: An example of application of this pattern is given in Sect. 8.3

8.2 Place the System phase

In this section, we present the Place the System phase (the first phase in our design method). We denote the system placement using a closed line on some selected views of the business model (cf. Chapter 5) in order to enclose every elements that the service system will support. The closed line is considered as a boundary of the service system marked on the business model. We give also some constraints on how to place an element outside/inside/crossing the closed line, and also on how to reflect the placement of an element to those related with it. The result of the Place the System phase is a *Placement Model.*

We present now in Sect. 8.2.1 in a detailed way the form of a placement model, and then in Sect. 8.2.2 we give some guidelines to drive the placement activity.

8.2.1 Placement Models

A placement model has the form of a Business Model (as introduced in Chapter 5) where the parts of the business that will be supported by the Service System to build (shortly SSys) are marked. Precisely the markings are:

- one closed line over the Business Process Overview Diagram,
- one closed line over the Static View,
- one swimlane named as the placed system (called the *system swimlane*) on each activity diagram presenting the Behaviour View of a business process,
- and one closed line over the Task View.

Placement on Business Process Overview Diagram In the Business Process Overview Diagram (BPOD) there are two icons, they represent the business processes and the participants. During the placement the developer has to decide which processes will be supported/partly supported/unsupported by the SSys. The placement on the BPOD is marked by means of a closed line. The icon of a supported/partly supported/unsupported business process will be placed inside/crossing/outside the closed line.

A participant icon is placed inside or outside but not crossing the closed line. A participant is placed inside the closed line if it takes part in supported or partly supported business processes and its activities are supported by the SSys, otherwise it is placed outside the closed line. The decisions about which business process to support will drive the decisions about which of the other parts of the business to support.

Placement on Static View We recall that there are three types of participants represented by UML classes in the Static View (cf. Sect. 5.2), they are the business workers, the systems and the business objects. A business worker class and a system class are placed inside/outside the closed line in the Static View if all/none of their activities are executed by the SSys. A business object class is placed inside/outside if it is handled/not handled by the SSys.

Placement on Behaviour View of business processes A swimlane will be used in a Behaviour View of a business process to show that this business process is supported/partly supported by the SSys. If a business process is supported/partly supported by the SSys, then all/part of elements in the activity diagram of its Behaviour View are placed inside the system swimlane. The fork nodes and the join nodes are the elements that may be placed inside/outside/crossing the system swimlane. The decision nodes and the merge nodes are placed either inside or outside but not crossing the system swimlane. An action will be placed inside/outside the system swimlane if it is/is not executed by the SSys.

Placement on Task View The elements of a Task View may be placed inside/outside the closed line over it. A task class will be placed inside/outside the closed line if it is/is not executed by the SSys (i.e., the task is inside the system swimlane of a business process supported by the SSys). In the Task View, besides the task classes, there are some elements previously modelled in the Static View, they are kept in the same positions with respect to the closed lines in both views.

Table 8.1 shows the well-formedness constraints on the placement models.

– There is no element crossing the closed lines in the Static View and the Task View.

– There is no action node, decision node, or merge node crossing the system swimlane in a Behaviour View.

– At least one action node is placed inside or crossing the border of the system swimlane, except for the unsupported business processes.

– If a decision node is inside the system swimlane, then all leaving arcs and the reached nodes are placed inside the system swimlane.

– A merge node of a corresponding decision node is placed accordingly to what was done with that decision node.

– A join node of a corresponding fork node is placed accordingly to what was done with that fork node.

– If a business process is supported, then:

    * all parts of its activity diagram must be inside the system swimlane,
    * all classes/datatypes typing its participants must be inside the closed line in the Static View,
    * all task classes typing its tasks must be inside the closed line in the Task View.

– If a business process is partly supported, then:

    * at least one action node must be outside and at least one action node inside the system swimlane,
    * all classes/datatypes typing its participants that are enclosed by the system swimlane must be inside the closed line in the Static View,
    * all task classes typing its tasks inside the system swimlane must be inside the closed line in the Task View.

– If a class/datatype is inside the closed line in the Static View, then:

    * all classes and datatypes typing its attributes and all classes connected with it by the associations must be enclosed by the closed line in the Static View,
    * it must be inside the closed line in the Task View if it presents in the Task View.

– If a task class is inside the closed line in the Task View, then:

    * all classes related on it by associations and all classes/datatypes typing its attributes must be inside the closed line in the Task View,
    * corresponding action node must be inside the system swimlane.

Table 8.1: Placement model: Well-formedness constraints

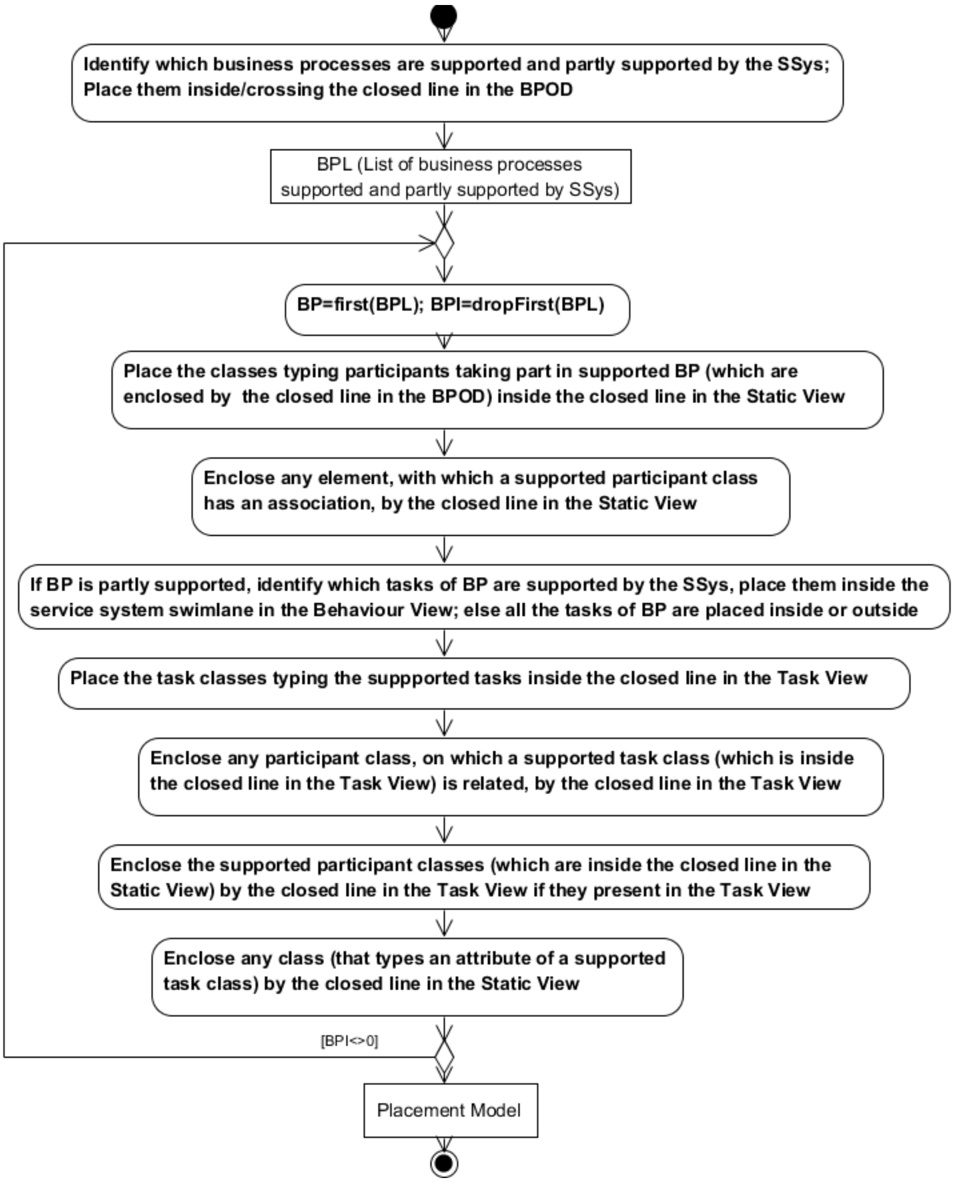

Identify which business processes are supported and partly supported by the SSys; Place them inside/crossing the closed line in the BPOD
BPL (List of business processes supported and partly supported by SSys)
BP=first(BPL); BPl=dropFirst(BPL)
Place the classes typing participants taking part in supported BP (which are enclosed by the closed line in the BPOD) inside the closed line in the Static View
Enclose any element, with which a supported participant class has an association, by the closed line in the Static View
If BP is partly supported, identify which tasks of BP are supported by the SSys, place them inside the service system swimlane in the Behaviour View; else all the tasks of BP are placed inside or outside
Place the task classes typing the suppported tasks inside the closed line in the Task View
Enclose any participant class, on which a supported task class (which is inside the closed line in the Task View) is related, by the closed line in the Task View
Enclose the supported participant classes (which are inside the closed line in the Static View) by the closed line in the Task View if they present in the Task View
Enclose any class (that types an attribute of a supported task class) by the closed line in the Static View
[BPl<>0]
Placement Model

Figure 8.2: Flow of steps in the placement on a Business Model

8.2.2 How to Place the System

We present by the activity diagram in Fig. 8.2 how to proceed to perform the Place the System.

- Spent to discuss

The first activity of the diagram shows that the placement starts at the BPOD, a closed on the BPOD in such a way that it encloses all processes supported by SSys, crosses all processes partly supported by the SSys, encloses the participants taking part in those processes (i.e., enclose all participants involved in a supported process, and enclose some (at least one) participant involved in a partly supported process).

The next placement is made over the Static View. In the Static View, a closed line is placed in such a way that it encloses all classes typing participants taking part in the supported processes (that are enclosed by the closed line in the BPOD), and encloses any element with which a participant class has an association.

The placement is made for the classes in the Task View is described in the fifth activity of the diagram.

The last two activities of the diagram is to verify the enclosing of related elements of a supported element. Those activities includes following steps:

- Enclose any participant class, on which a supported task class in the Task View is related, in the closed line in the Task View .

- Enclose the participant classes (which are supported in the Static View) in the closed line in the Task View if they are in the Task View.

- Enclose any class (that types an attribute of a supported task class inside in the Task View) in the closed line in the Static View.

8.3 Eliminate Useless Parts phase

The aim of this phase is to eliminate all useless "*parts*" from a placement model; obviously the useless parts are those that are not interested by the placed system.

The term "*part*" may refer to: an element in a diagram (e.g., a class), and to a diagram (e.g., Static View) in the model. A part is considered *useless* if only if:

– it is outside the placement marking,

– and it is not used to define or to type any other part,

– and it has no association with any other class if it is a class, it has no reference to any other part if it is a datatype, and any change on it does not impact other part if it is a diagram.

The list of the patterns of this phase is shown in Table 8.2, and the template used to describe them has been presented in Sect. 8.1.

	Eliminate Useless Parts phase patterns
1	Eliminating unsupported business process
2	Eliminating useless placement marking on BPOD
3	Eliminating useless placement marking on Static View
4	Eliminating useless placement marking on Task View
5	Eliminating unused class

Table 8.2: Patterns for Eliminate Useless Parts

In this phase the developer has to handle models whose form has been defined in Fig. 5.1 (c.f., Sect. 5.2), thus made of one Static View, one Task View, one Business

Process Overview Diagram, and a number of Business Process Models. Such models may be schematically denoted as follows:

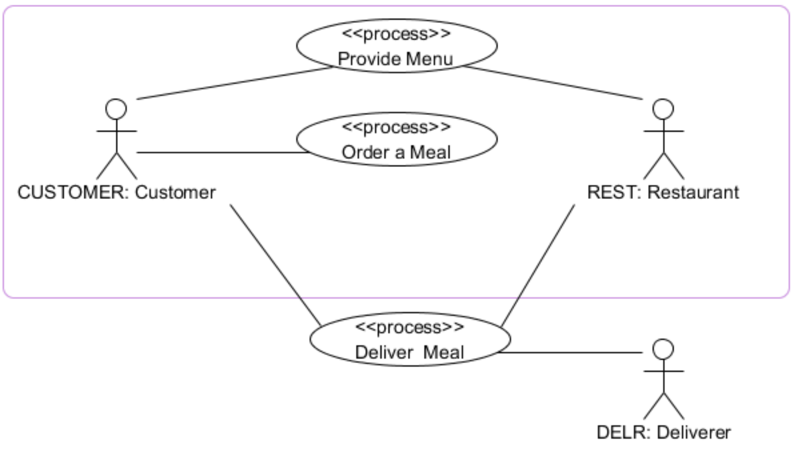
SV
TV
BPOD
BPMs

where the Static View is denoted by SV, the Task View is denoted by TV, the Business Process Overview Diagram is denoted by BPOD, and BPMs denotes the set of the business process models.

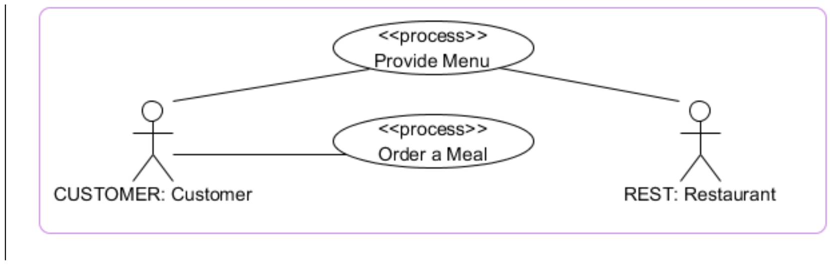
Transformation pattern: Eliminating unsupported business process
Motivation: The business processes not supported by the SSys are useless, and thus may be eliminated by the model
Description: An unsupported business process is eliminated from the model
Subjects:
– BP, a business process in the BPOD
– BPM, the Business Process Model of BP
– E$_1$, . . . , E$_n$, the entities associated only with BPin the BPOD
Preconditions:
BPis outside the placement in BPOD
What to do:

SV		SV
TV	⇒	TV
BPOD		BPOD - BP - E$_1$ - . . . - E$_n$
BPMs		BPMs - BPM

Example:



⇒



The BPOD in the example above (on the left of '⇒') is derived from the model of business Meal Delivery, which has three business processes: Provide Menu, Order a Meal, and Deliver Meal. The closed line in this BPOD represent the placement of Meal Ordering System on this business, and this service system does not support process Deliver Meal. Hence, after the application of this pattern (see the BPOD on the right of '⇒'), process Deliver Meal is eliminated, and the entity DELR that is associated only with this process is eliminated as well.

Transformation pattern: Eliminating useless placement marking on BPOD
Motivation: To eliminate the placement marking, i.e., the closed line representing the placement of a service system in the BPOD, which has all elements inside it
Description:

A useless placement marking is eliminated from the BPOD.

Subjects:

BPOD

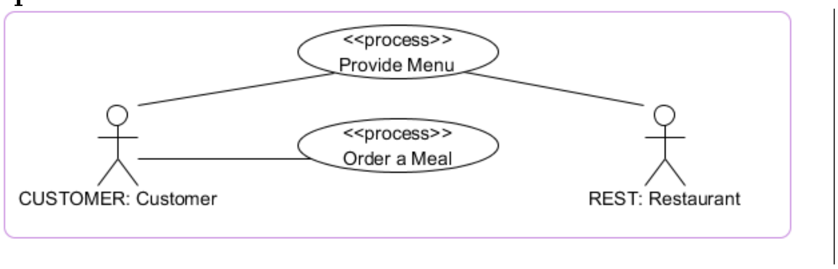
Preconditions:

All business processes are inside this closed line.

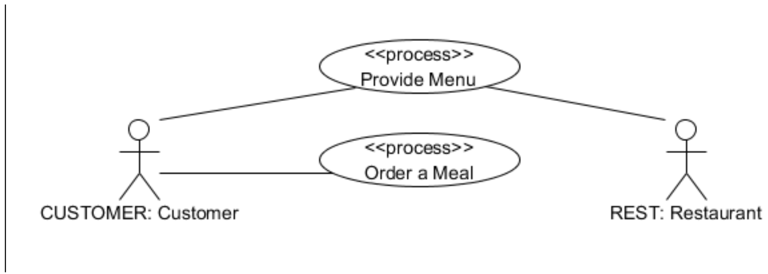
What to do:

SV		SV
TV		TV
BPOD	⇒	BPOD - *closed line*
BPMs		BPMs

Example:



⇒



The closed line in BPOD above represent the placement of Meal Ordering System on business Meal Deliver. Two processes of this business are all supported by system Meal Ordering System, hence they are all inside the closed line. The closed line here is useless, hence it is eliminated from the model, see the BPOD in the right of ⇒.

Transformation pattern: Eliminating useless placement marking on Static View
Motivation: To eliminate the placement marking, i.e., the closed line representing the placement of a service system in the Static View, which has all elements inside it

The content of this pattern is similar to the pattern Eliminating useless placement marking on BPOD, in which the objective part is the closed line in SV that all classes are inside it.

Transformation pattern: Eliminating useless placement marking on Task View
Motivation: To eliminate the placement marking, i.e., the closed line representing the placement of a service system in the Task View, which has all elements inside it

The content of this pattern is similar to the pattern Eliminating useless placement marking on BPOD, in which the objective part is the closed line in TV that all classes and task classes are inside it.

Transformation pattern: Eliminating unused class
Motivation: To eliminate a class not used in any part of the model and outside the placement.
Description: An unused class is eliminated, and any association that will result pending after the class elimination is also eliminated from the model.
Subjects:

C, a class in the Static View

Assbe the set of the associations having Cas one of their ends

Preconditions:

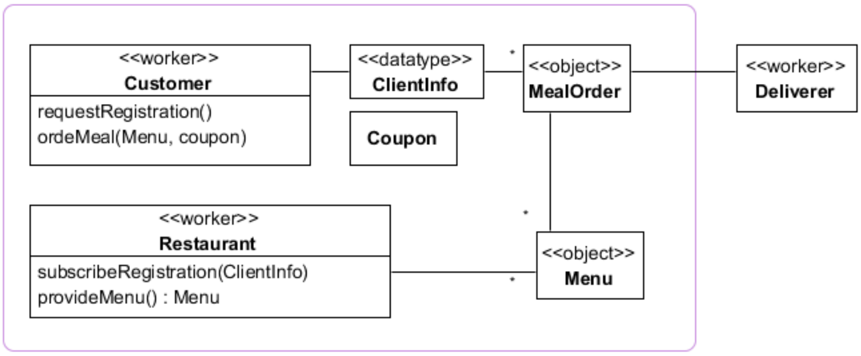
Cis an unused class

What to do:

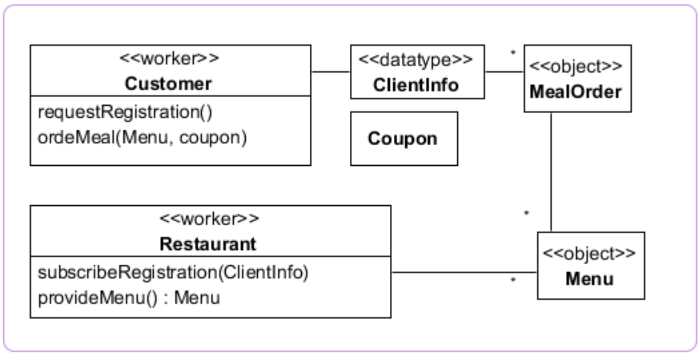
SV		SV - C - Ass
TV		TV - C - Ass
BPOD	⇒	BPOD
BPMs		BPMs

Example:

The class Deliver is unused in the Static View, in the Task View, in any Business process model, and in the BPOD of the model of the Meal Delivery business. It is eliminated from the Static View, the association between this class with another class is eliminated as well.



⇒



8.4 Simplify Tasks phase

The list of the patterns supporting this phase (only one) is shown in Table 8.3. The template used to describe these transformation pattern has been defined in Sect. 8.1.

The Simplify Tasks phase transforms a business model into another business model,

	Simplify Tasks phase patterns
1	Decomposing task

Table 8.3: Patterns for Simplify Tasks

thus also the patterns used in this phase transforms a business model into a business model, that may be schematically presented as follows:

SV
TV
BPOD
BPMs

where the Static View is denoted by SV, the Task View is denoted by TV, the Business Process Overview Diagram is denoted by BPOD, and BPMs denotes the set of the business process models.

Transformation pattern: Decomposing task
Motivation: The tasks must be decomposed into smaller tasks till they have a granularity corresponding to call a service functionality
Description: The behaviour of a task is realized by combining other (smaller) tasks, thus it must be eliminated from the model replacing each of its instances by such combination, moreover its class will be eliminated from the Task View
Subjects:
– TC, a task class whose participants and parameters are $\mathsf{P}_1, \ldots, \mathsf{P}_n$
– TVF, a task view fragment (introducing the new smaller tasks needed to realize the behaviour of TC)
– $\mathsf{ADF}(\mathsf{X}_1, \ldots, \mathsf{X}_n)$, an activity diagram fragment describing how the behaviour of $\mathsf{TC}<\mathsf{P}_1 \leftarrow \mathsf{X}_1, \ldots, \mathsf{P}_n \leftarrow \mathsf{X}_n>$ will be realized by using the new tasks
Preconditions:
$\mathsf{ADF}(\mathsf{X}_1, \ldots, \mathsf{X}_n)$ is an activity diagram fragment including an initial node, a number of action nodes that are instances of task classes, and a finale node. $\mathsf{X}_1, \ldots, \mathsf{X}_n$are all free variables appearing in ADF.
What to do:

$$\begin{array}{l|c|l}
\mathsf{SV} & & \mathsf{SV} \\
\mathsf{TV} & & (\mathsf{TV} - \mathsf{TC}) + \mathsf{TVF} \\
\mathsf{BPOD} & \Rightarrow & \mathsf{BPOD} \\
\mathsf{BPMs} & & \mathsf{BPMs}[\mathsf{ADF}(\mathsf{E}_1, \ldots, \mathsf{E}_n) \ / \ \mathsf{TC}<\mathsf{P}_1 \leftarrow \mathsf{E}_1, \ldots, \mathsf{P}_n \leftarrow \mathsf{E}_n>]
\end{array}$$

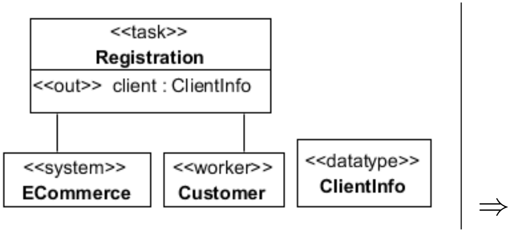
Example: The task class Registration is decomposed into a composition of three task classes RequestRegistration, SubscribeRegistration, ConfirmRegistration. An instance of such class, e.g., Registration<CUST, ECOM, CINFO> is transformed into activity diagram AVF(CUST, ECOM, CINFO).

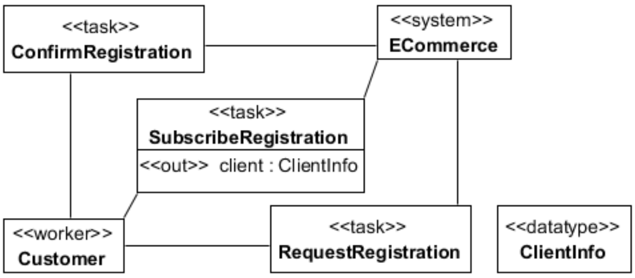
The subjects are:

- ≪task≫ class Registration (shown as a part in the Static View in the first picture below)

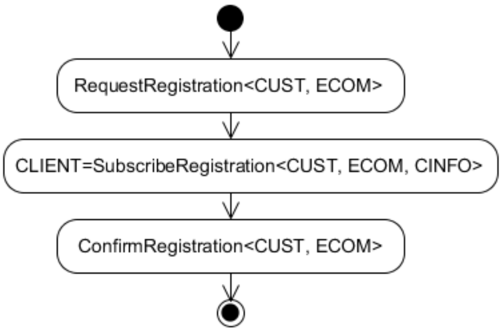
- A fragment of the Task View includes three ≪task≫ classes RequestRegistration, SubscribeRegistration, and ConfirmRegistration (shown in the second picture below)

- Activity diagram ADF(CUST, ECOM, CINFO) describes how the behaviour of Registration<CUST, ECOM, CINFO> is realized by using three new tasks (shown in the third picture)





ADF(CUST, ECOM, CINFO) is shown in the following picture:


RequestRegistration<CUST, ECOM>
CLIENT=SubscribeRegistration<CUST, ECOM, CINFO>
ConfirmRegistration<CUST, ECOM>

8.5 Operationalize Tasks phase

The aim of this phase is to transform the decomposed tasks into operation calls to a business entity (later, they will become operations in the interface of a service). There are two types of decomposed tasks, i.e., binary task and unary task, having two and one business entities taking part in it, respectively. The list of the patterns of this phase is shown in Table 8.4, and they are presented following the template defined in Sect. 8.1.

Operationalize Tasks phase patterns	
1	Operationalizing binary task
2	Operationalizing unary task
3	Eliminating empty task view

Table 8.4: Patterns for Operationalize Tasks

The Operationalize Tasks phase transforms a business model into another business model, thus also the patterns used in this phase transform a business model into a business model, that may be schematically presented as follows:

SV
TV
BPOD
BPMs

where the Static View is denoted by SV, the Task View is denoted by TV, the Business Process Overview Diagram is denoted by BPOD, and BPMs denotes the set of the business process models.

Transformation pattern: Operationalizing binary task
Motivation: To transform the instances of a binary task, i.e., a task that has exactly two entities taking part in it, into operation calls to business entities that may be later transformed into calls to functionalities of services.
Description: Any instance of the binary task becomes an operation call of an entity class, that will be added in that entity class, and the task class is eliminated from the task view
Subjects:
– TCB, a task class that has exactly two parameters typed by entity classes, P_1: CE_1 and P_2: CE_2, and n attributes typed by datatypes $\mathsf{P'}_1$: $\mathsf{DT}_1, \ldots,$ $\mathsf{P'}_n$: $\mathsf{DT}_n (n \geq 0)$
– opTCB(CE_2, $\mathsf{DT}_1, \ldots, \mathsf{DT}_n$), an operation
Preconditions:
What to do:

$\mathsf{SV} + \mathsf{CE}_1 + \mathsf{CE}_2$		$\mathsf{SV} + (\mathsf{CE}_1 + \mathrm{opTCB}(\mathsf{CE}_2, \mathsf{DT}_1, \ldots, \mathsf{DT}_n)) + \mathsf{CE}_2$
TV		TV - TCB
BPOD	$\Rightarrow$	BPOD
BPMs		$\mathsf{BPMs}[\mathsf{E}_1.\mathrm{opTCB}(\mathsf{E}_2, \mathsf{E'}_1, \ldots, \mathsf{E'}_n)$ /
		$\mathsf{TCB}<\mathsf{P}_1 \leftarrow \mathsf{E}_1, \mathsf{P}_2 \leftarrow \mathsf{E}_2, \mathsf{P'}_1 \leftarrow \mathsf{E'}_1, \ldots, \mathsf{P'}_n \leftarrow \mathsf{E'}_n>]$

Example:

The subjects are:

- Binary ≪task≫ class OrderMeal

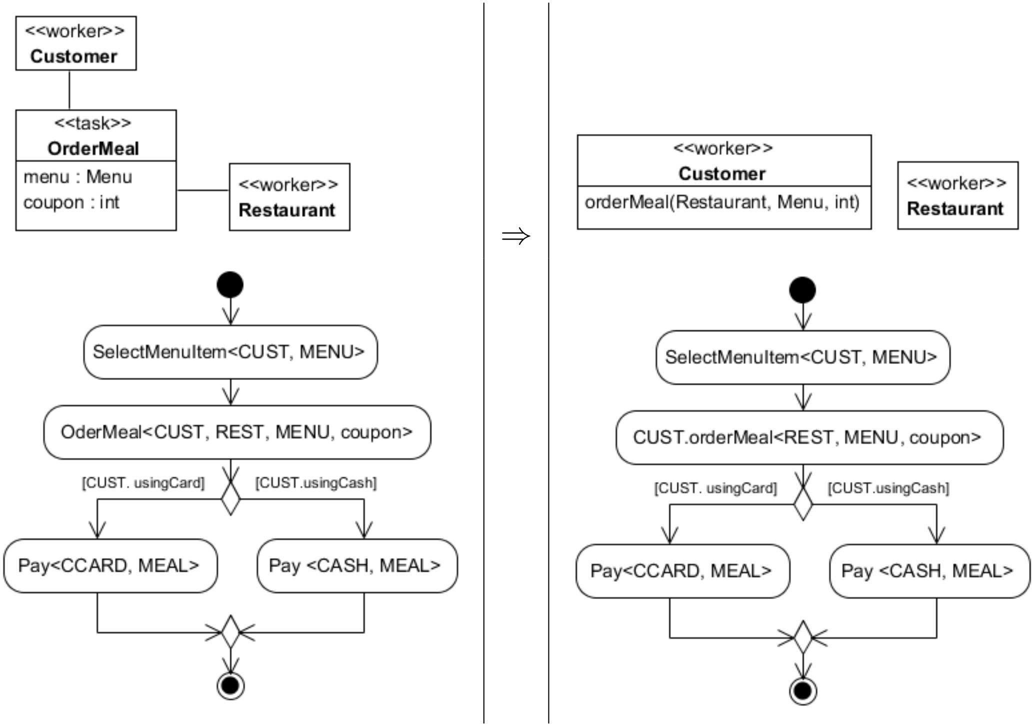
- Operation orderMeal<Restaurant, Menu, int>

- An instance of task class OrderMeal in the Behaviour View, e.g., OrderMeal<CUST, REST, MENU, coupon>

- Operation call CUST.orderMeal<REST, MENU, coupon>

The operationalizing binary task class OrderMeal is shown in the following pictures. It is transformed into operation orderMeal(Restaurant,Menu,int) of ≪worker≫ class Customer. The transformation results in adding operation orderMeal<Restaurant, Menu, int> to ≪worker≫ class Customer, in which, the attributes of task class OrderMeal are transformed into parameters of this operation, and participant Restaurant taking part in task OrderMeal is indicated by the parameter Restaurant in this operation.

Task OrderMeal<CUST, REST, MENU, coupon> is transformed into the operation call CUST.orderMeal<REST, MENU, coupon>.



Transformation pattern: Operationalizing unary task

Motivation: To transform the instances of a unary task, i.e., a task that has exactly one entity taking part in it, into operation calls to a business entity that may be later transformed into internal activities of the business entity.

Description: The instances of a unary task become operation calls, that will be added in that entity class, and the task class is eliminated from the task view

Subjects:

– TCU, a task class that only one parameter typed by an entity class, P: CE, and n attributes typed by datatypes P_1: DT_1, ..., P_n: DT_n(n $\geq$ 0)

– opTCU(DT_1, ..., DT_n) be an operation

Preconditions:

What to do:

$$\begin{array}{l|c|l} \mathsf{SV} + \mathsf{CE} & & \mathsf{SV} + (\mathsf{CE} + \mathrm{opTCU}(\mathsf{DT}_1, \ldots, \mathsf{DT}_n)) \\ \mathsf{TV} & & \mathsf{TV} - \mathsf{TCU} \\ \mathsf{BPOD} & \Rightarrow & \mathsf{BPOD} \\ \mathsf{BPMs} & & \mathsf{BPMs}[\mathsf{E}.\mathrm{opTCU}(\mathsf{E}_1, \ldots, \mathsf{E}_n) \ / \ \mathsf{TCU}<\mathsf{E}, \mathsf{E}_1, \ldots, \mathsf{E}_n>] \end{array}$$

Example:

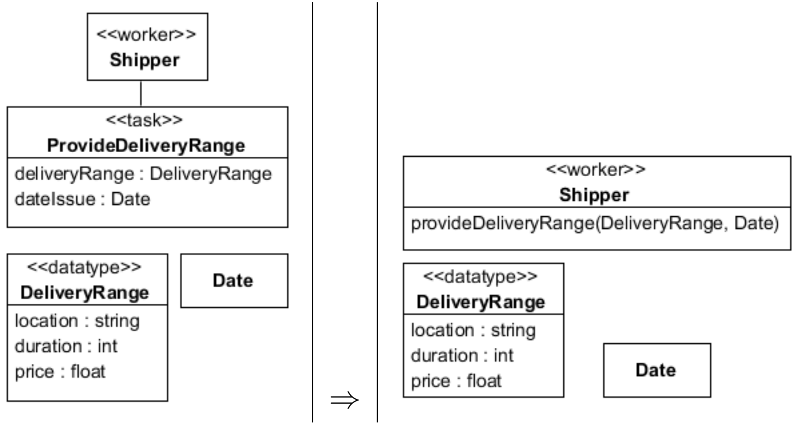
The subjects are:

- Unary ≪task≫ class ProvideDeliveryRange

- An instance of ProvideDeliveryRange, e.g., ProvideDeliveryRange<DRANGE, now>

- Operation call SHIPPER.provideDeliveryRange<DRANGE,now>

Operationalizing the unary task class ProvideDeliveryRange is shown in the following pictures. The transformation results in adding operation provideDeliveryRange(DeliveryRange, Date) to ≪worker≫ class Shipper. In a Behaviour View, the transformation results in transforming task ProvideDeliveryRange<DRANGE, now> into operation call SHIPPER.provideDeliveryRange<DRANGE,now>.



Transformation pattern: Eliminating empty task view

Motivation: To eliminate an empty task view (i.e., having no elements inside) from the model

Description:

The empty Task View is eliminated from the model

Subjects:
TV, the Task View of the model
Preconditions:
TV does not include any class
What to do:

SV		SV
TV		
BPOD		BPOD
BPMs	⇒	BPMs

8.6 Introduce Services phase

The aim of this phase is to expose and define the participants and the services of the designed service system. The participants are derived from the business entities in such a way that a large part of the operations of these entities are transformed into service calls. The services are exposed from a set of operations between particular business entities. The list of the patterns of this phase is shown in Table 8.5. The template for describing those transformation patterns has been presented in Sect. 8.1.

	Introduce Services phase patterns
1	Introducing empty service architecture
2	Transforming a system class into a participant
3	Transforming an object class into a datatype
4	Transforming a unique object into a participant
5	Transforming an object class with several instances into a participant
6	Transforming a worker class into a participant
7	Combining two participants
8	Splitting a participant into two or more participants
9	Making a participant to provide a new service
10	Adding a participant providing already available services
11	Eliminating an empty Static View

Table 8.5: Patterns in Introduce Services

The Introduce Services phase transforms a model into another model, till such model is a service system model. The models produced during this phase will be built using both the profile for the business models (see Chapter 5) and for the service system models (see Chapter 6) and will have the views required by both kinds of models. Thus a model transformed in this phase will have the following form:

SV

BPOD

BPMs

SA

SMs

PMs

where the Static View is denoted by SV, the Task View is denoted by TV, the Business Process Overview Diagram is denoted by BPOD, BPMs denotes the set of the business process models that will be at the end reduced to activity diagrams showing how the various processes have been realized in the service system, the Service Architecture is denoted by SA, the Service Models are denoted by SMs, and the Participant Models are denoted by PMs.

Transformation pattern: Introducing empty service architecture
Motivation: A service system model must include a service architecture, so initially an empty one will be inserted in the model
Description: An empty service architecture is added to the model.
Preconditions:
There is no service architecture in the model.
What to do:

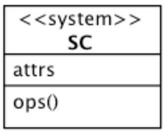
SV		SV
		EmptySa, a service architecture named as the system without any role for participants (and thus without any indication of service usage)
BPOD	⇒	BPOD
BPMs		BPMs
SMs		SMs
PMs		PMs

Transformation pattern: Transforming a system class into a participant
Motivation: A system has to be wrapped to be introduced in a service system, this pattern introduces that wrapper
Description: A participant is added to model the wrapper of the system class[2], it will have the same operations as the system class; reference to the system class

[2] A system class is a class stereotyped by ≪system≫

in the model will be replaced by a reference to the wrapper class

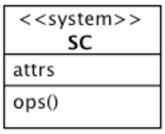
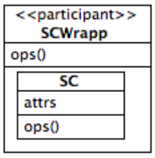
Subjects:



, a system class appearing in SV

Preconditions:

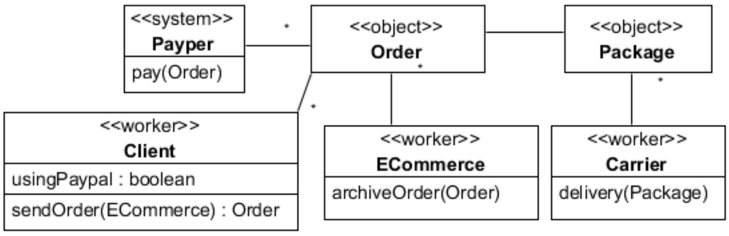
What to do:

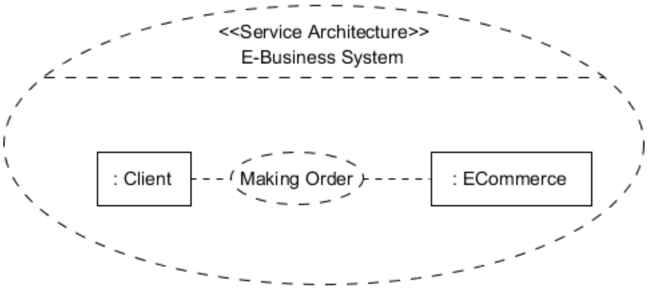
SV		SV - 
SA		SA plus a part modelling a role for participants typed by SCWrapp
BPOD	⇒	BPOD where each occurrence of SC has been replaced by SCWrapp
BPMs		BPMs where each occurrence of SC has been replaced by SCWrapp
SMs		SMs
PMs		PMs + 

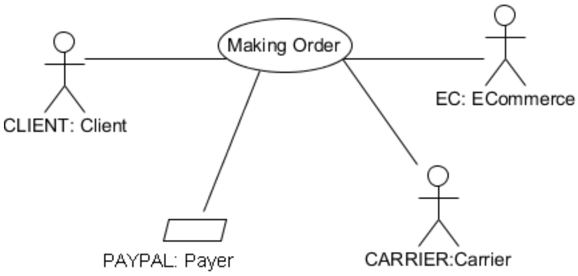
Example:

The subject is:

- ≪system≫ class Payper

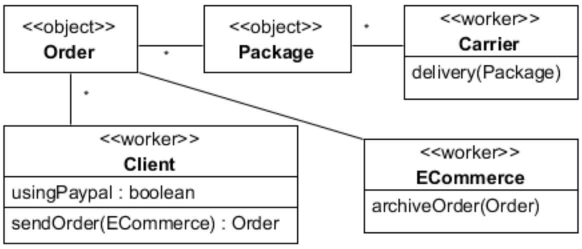


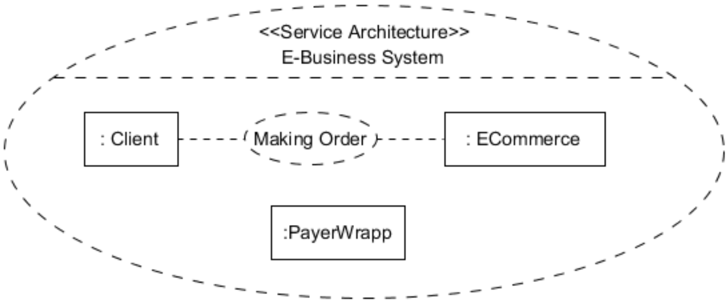


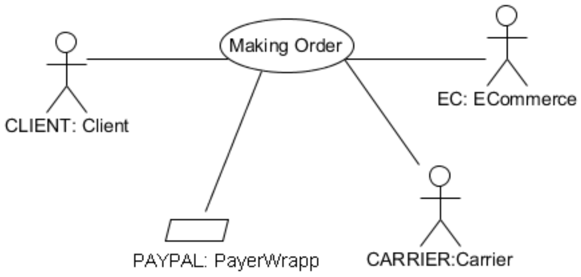


⇒

Making Order Service Model
E-Business System Participant Model

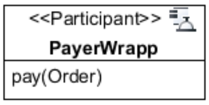






Making Order Service Model

E-Business System Participant Model +



Transformation pattern: Transforming an object class into a datatype

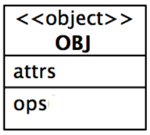
Motivation: Whenever an object class[3] has all the characteristics of the datatypes, e.g., all its operations are queries and no direct check for equality is ever made using the "=" combinator, it should be transformed into a datatype (this will help the transition to the service paradigm, since the service message parameters must be datatypes).[4]

Description: The object class is replaced by a datatype definition having the same name, same attributes and same operations and the same associated constraints and method definitions.

Subjects:

[3] A class stereotyped by ≪object≫

[4] Sometimes in a business model they have been considered business objects because they had a relevant role in the business.

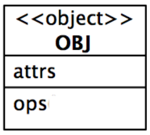
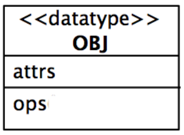


, an object class

Preconditions:

OBJ is essentially a datatype, i.e., all its operations are queries, and no equality check for elements typed by OBJ (exp = exp', with exp and exp' having type OBJ) appears in the model

What to do:

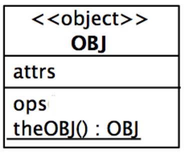
SV + 		SV + 
SA		SA
BPOD	⇒	BPOD
BPMs		BPMs
SMs		SMs
PMs		PMs

Transformation pattern: Transforming a unique object into a participant

Motivation: To handle the unique instance of an object class we introduce a participant that will provide a service (some services) to access to such instance

Description: The object class OBJ and its unique instance will be transformed into a participant class, a role with multiplicity one typed by OBJ will be added to the service architecture,

Subjects:

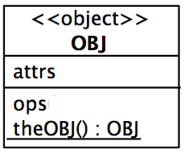


, an object class with a unique instance denoted by the static operation theOBJ (that is defined is following the singleton design pattern)

Preconditions:

There exists at most one instance typed by OBJ in the model, and thus a non-static operation of OBJ has neither a parameter typed by OBJ nor the return type equal to OBJ

What to do:

SV + 

SA

BPOD

BPMs

SMs

PMs

⇒

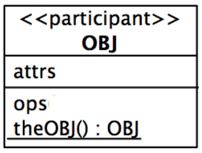
SV

SA plus a part modelling a role for a participant typed by OBJ

BPOD

BPMs where each occurrence of OBJ is replaced by theOBJ

SMs

PMs + 

Example:

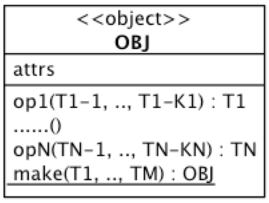
The case of an object class modelling the regulation of an organization, which may be changed at any time, but where only the last version should be recorded, and the date of the last change together with the name of anyone has ever contributed to the document must be recorded too.

Transformation pattern: Transforming an object class with several instances into a participant

Motivation: To handle many business objects of a given type we should introduce a participant that will then provide a service (some services) to handle and to store them.

Description: The object class OBJ will be transformed into a standard class having an extra attribute containing an indentifier, and a new participant model OBJ-s will be added to the (able to store and handle all the instances of OBJ referring to them them using their identitifers), a part with multiplicity one typed by OBJ-s will be added to the service architecture, all operations calls on objects typed by OBJ will be replaced by calls to the unique participant typed by OBJ-s having as an extra parameter the identifier of such objects.

Subjects:



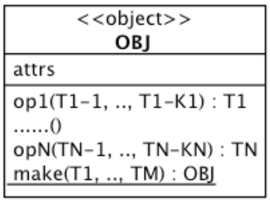
, an object class where make is the constructor operation for creating new instances of the class

Preconditions:

There exists several different instances typed by OBJ in the business model

What to do:

SV +



SA

BPOD

BPMs

SMs

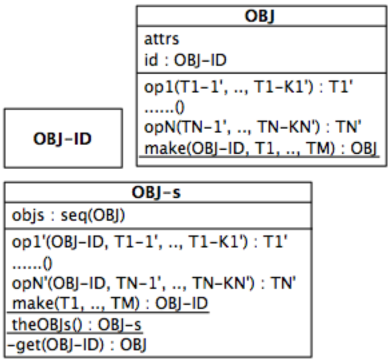
PMs

⇒

SVwhere each call of an operation of OBJ e.op(e_1, . . . , e_n) is replaced by theOBJs.op'(e,e_1, . . . , e_n)

SMs

PMs +


OBJ-ID
OBJ
attrs
id : OBJ-ID
op1(T1-1', .., T1-K1') : T1'
......()
opN(TN-1', .., TN-KN') : TN'
make(OBJ-ID, T1, .., TM) : OBJ
OBJ-s
objs : seq(OBJ)
op1'(OBJ-ID, T1-1', .., T1-K1') : T1'
......()
opN'(OBJ-ID, TN-1', .., TN-KN') : TN'
make(T1, .., TM) : OBJ-ID
theOBJs() : OBJ-s
-get(OBJ-ID) : OBJ

where given a type T, T' denotes T if T is different from OBJ, otherwise it denotes OBJ-ID;

the operations op1', . . . , opN' are defined as follows

os.op'(oid,e1, . . . , eN) = op(get(oid,os),e1, . . . , eN);

os.make(e1, . . . , eN) =

newI = newId(os); os.objs = os.objs.append(make(newId(os),e1, . . . , eN); return newI

SA plus a part modelling a role for participants typed by OBJs

BPOD

BPMs where each call of an operation of OBJe.op(e_1, . . . , e_n) is replaced by theOBJs.op'(e,e_1, . . . , e_n), and OBJ has been replaced by OBJ-ID

Example: The case of an object class modelling the orders in an e-commerce business

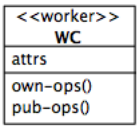
Transformation pattern: Transforming a worker class into a participant

Motivation: For each business worker we should introduce a participant in the service system allowing s(he) to use the services provided by the other participants to perform hers/his activities; this participant should provide a GUI for allowing the worker to request the various services

Description:

The worker class is replaced by a participant, whose internal architecture is made by the worker as human-being, and a software subsystem interacting with she/him by means of a GUI and at the same time using the services offered by other participants

Subjects:

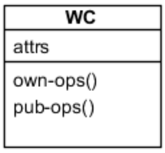
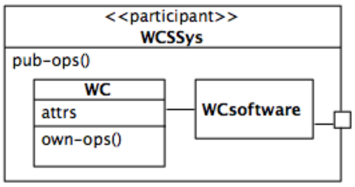


, a worker class

Preconditions:

The operations in own-ops are all operations of the class WC that are called only by the elements of the class WC itself, and do not have any argument nor the result typed by an entity class

What to do:

		
SV +		SV where each occurrence of WC is replaced by WCSyss
SA		SA plus a part modelling a role for a participant typed by WCSyss
BPOD	⇒	BPOD
BPMs		BPMs where each occurrence of WC as argument of an operation in pub-ops() is replaced by WCSyss
SMs		SMs
PMs		PMs +  <<participant>> WCSSys pub-ops() WC attrs own-ops() WCsoftware

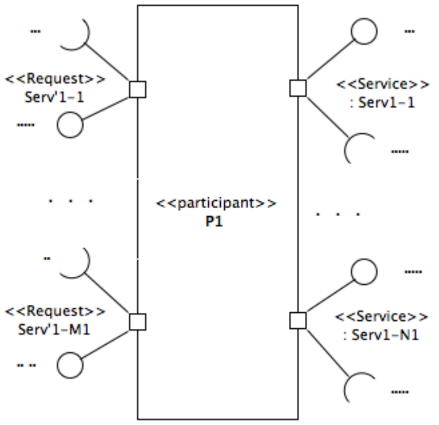
Transformation pattern: Combining two participants

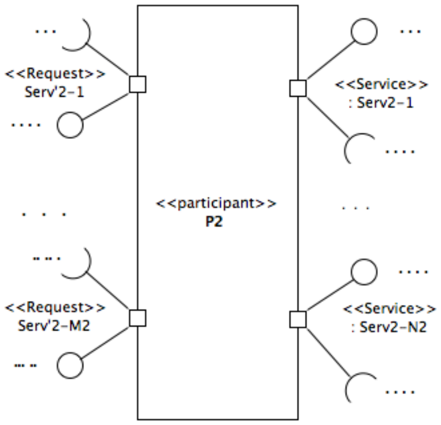
Motivation: To rearrange logically the service system, two participants may be combined together to become a unique one

Description:

A new participant (model) is introduced, it will offer and consume all the services offered and consumed by the two combined participant (model)s; its behaviour will be the parallel composition of the behaviours of the two combined participants

Subjects:





xxx the name of the combination of P1 and P2

Preconditions:

No service is offered by P1 and consumed by P2 or offered by P2 and consumed by P1

xxx is different both from P1 and P2

What to do:

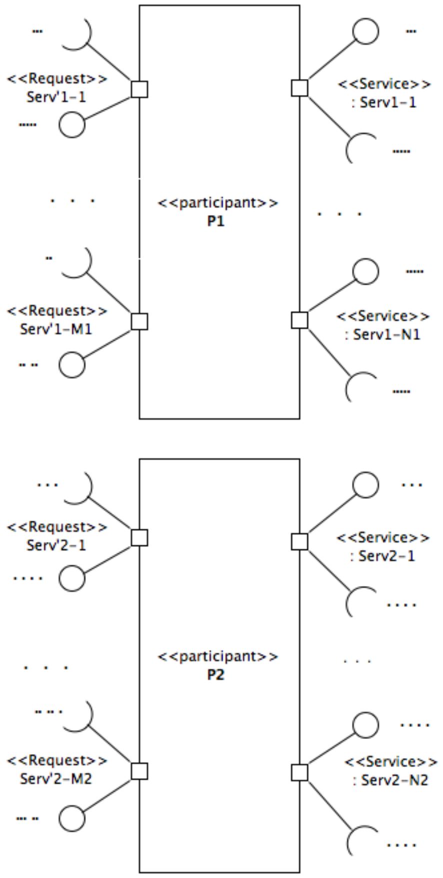
SV

SA

BPOD ⇒

BPMs

SMs



PMs +

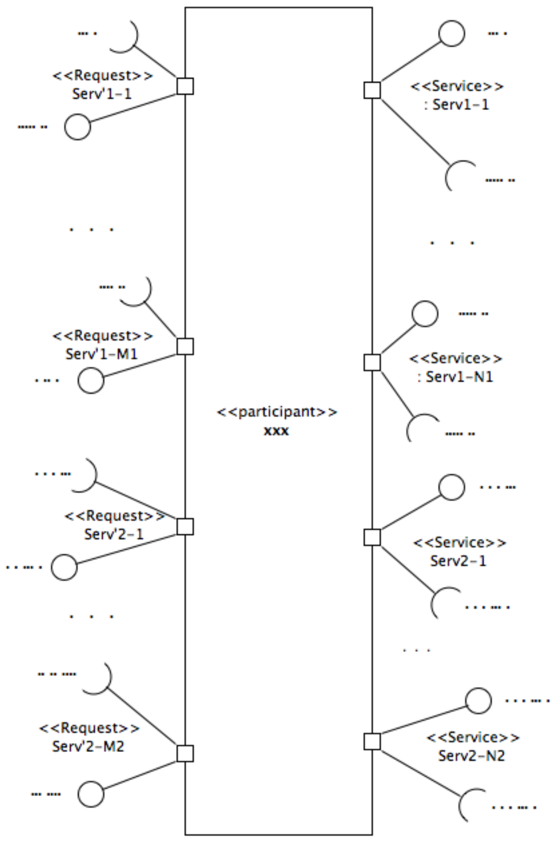
SV

SA where the parts typed by P1 and P2 are merged in a unique part typed by xxx

BPOD where each occurrence of P1 and of p2 is replaced by xxx

BPMs where each occurrence of P1 and of p2 is replaced by xxx

SMs



PMs +

Transformation pattern: Splitting a participant into two or more participants

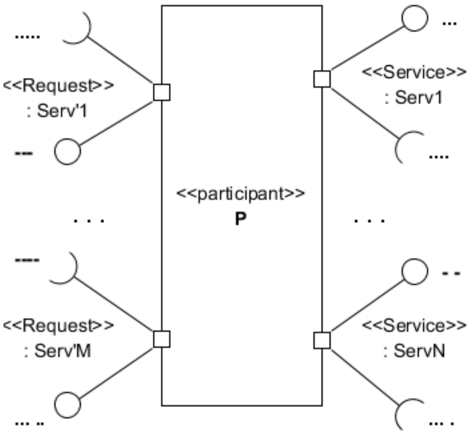
Motivation: To rearrange logically the service system, a participant may be split into two or more participants.

Description: Some new participants (participant models), X1...Xn ($n \geq 2$), are introduced to replace a participant P (participant model P) in the model.

The services offered and consumed by participant Pwill be consumed and provided by participants in the set of participants X1...Xn ($n \geq 2$), in such a way that a service is consumed or offered by an unique partcicipant.

The composition of the behaviours of the new participants will be the behaviour of participant P.

Subjects:



X1. . . Xn are participants split from P.

Names of consumed and provided services of each participant in X1. . . Xn.

Preconditions:

No service is offered or consumed both by Xi and by Xk ($i \neq k$, $1 \leqslant i \leq n$, $1 \leqslant k \leq n$).

X1. . . Xn are different from each other and different from P.

All services consumed and provided by Pare consumed and provided fully by set of new participants X1. . . Xn.

What to do:

SV

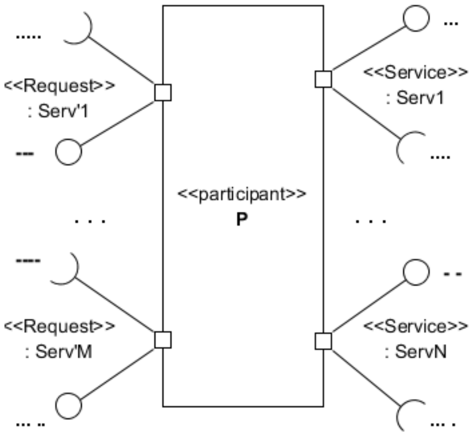
SA

BPOD

BPMs

SMs

PMs +



⇒

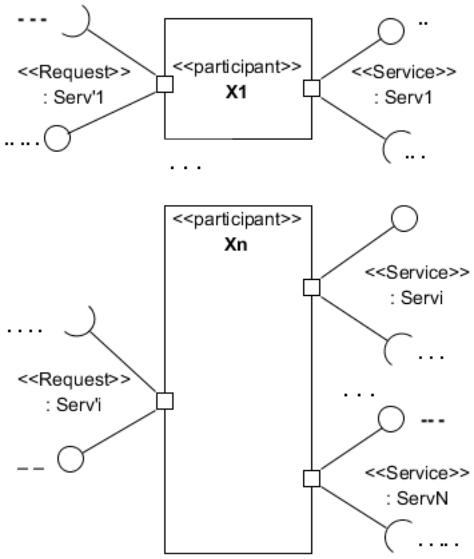
SV

SA where the part typed by P are split into n parts typed by X1 . . . Xn in such a way that part Xi ($i = 1 \ldots n$) is bounded to service(s) that participant Xi consumes and provides

BPOD where each occurrence of P is replaced by X1 . . . Xn

BPMs where each occurrence of P is replaced by a participant in set of X1 . . . Xn

SMs



PMs - P +

Transformation pattern: Making a participant to provide a new service

Motivation: A service of the system being built must be provided by a participant. Thus, for introducing a new service, a participant will be indicated as a service provider.

Description: A new service is introduced, and a participant is indicated to provide this service.

Subjects:

- P, a participant (model) having a service port in PMs

- S, a service (model). Let pI be the provider interface of service S, and cI be the consumer interface of service S

Preconditions:

S is different from the names of all the existing services in the model

What to do:

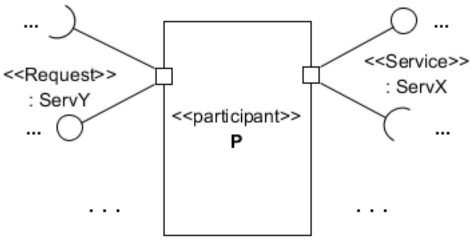
SV

SA

BPOD ⇒

BPMs

SMs



PMs +

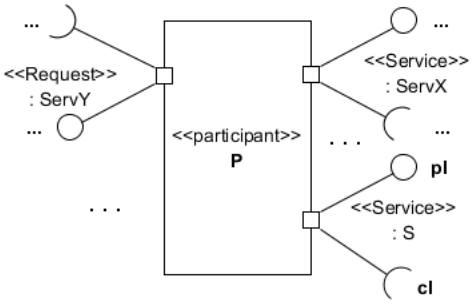
SV

SA + a *collaboration use* representing S that is bounded to the part typed by P

BPOD

BPMs

SMs + S



PMs +

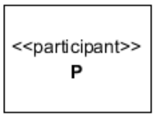
Transformation pattern: Adding a participant providing already available services

Motivation: Already available service(s) may be incorporated in the system. After it is modelled following PreciseSoa method (see 6), a participant must be added to provide it.

Description: A participant has service ports for providing available services is added to the model.

Subjects:

– S1. . . Sn, n services (service models) in SMs. Let pli be provider interface of service Si (i=1. . . n), and cli be consumer interface of service Si (i=1. . . n).

-  , a participant class to be added.

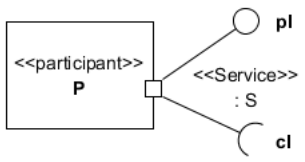
Preconditions:

P is different from the names of other participants of the system.

What to do:

SV
SA
BPOD ⇒
BPMs
SMs
PMs

SV + any related class needed to define P
SA + a part typed by P and is bounded to S
BPOD
BPMs
SMs
PMs +



Transformation pattern: Eliminating an empty Static View
Motivation: To eliminate an empty Static View from the model.
Description: An empty Static View is deleted from the model
Subjects:
SV, the static view in the model
Preconditions:
SV is an empty class diagram.
What to do:

SV		
SA		SA
BPOD	⇒	BPOD
BPMs		BPMs
SMs		SMs
PMs		PMs

In the following sections, we will present the applications of various design phases defined above on a case study, in particular, we will design Dealer Networking System supporting Dealer Business.

8.7 Applying design method to case study Dealer Network

We use the model of the Dealer Business (adopted in Sect. 5.4) to show an example of application of our design method.

TO DO: Change 'SOAS' in the figures to 'Dealer Networking System'.

8.7.1 Applying Place the System

Placement on the Business Process Overview Diagram

The diagram in Fig. 8.3 illustrates the placement over the Business Process Overview Diagram of the model of Dealer Business. Business process Buying goods is placed crossing the closed line because it is partly supported by the Dealer Networking System (the system being built to support Dealer Business).

The payment in this process is executed by means of an existing application represented by participant PAY of type Payment. The participants, e.g., SHIPPER, DEALER, and MANUFACTURE are placed inside the closed line because they take part in Buying goods process and their activities are executed by the Dealer Networking System. Process Handling reputation information is placed outside the closed line because the Dealer Networking System does not support this process.

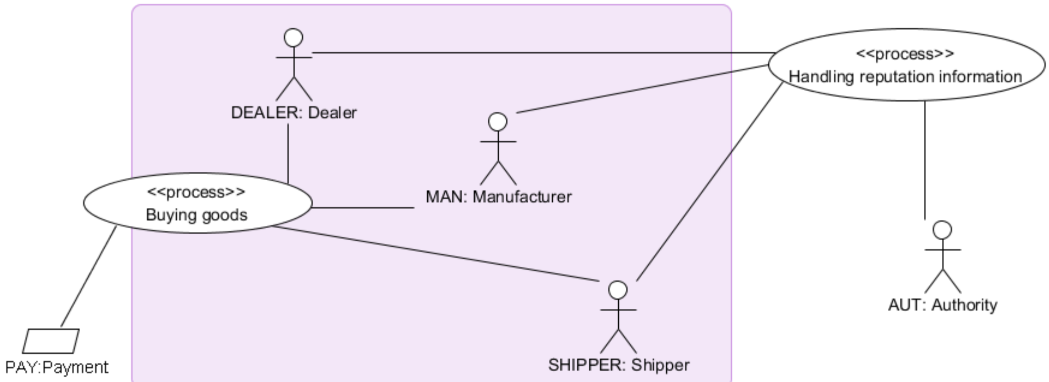


Figure 8.3: Placement Dealer Networking System on Business Process Overview Diagram of business model Dealer Network

Placement on the Static View

The placement on the Static View of the model of Dealer Business is presented by the diagram in Fig. 8.4. There are two classes placed outside the closed line, i.e., class Invoice because the invoice generated in process Buying goods is not created by the Dealer Networking System, and class Payment because it represents an existing application that is not built in the Dealer Networking System.

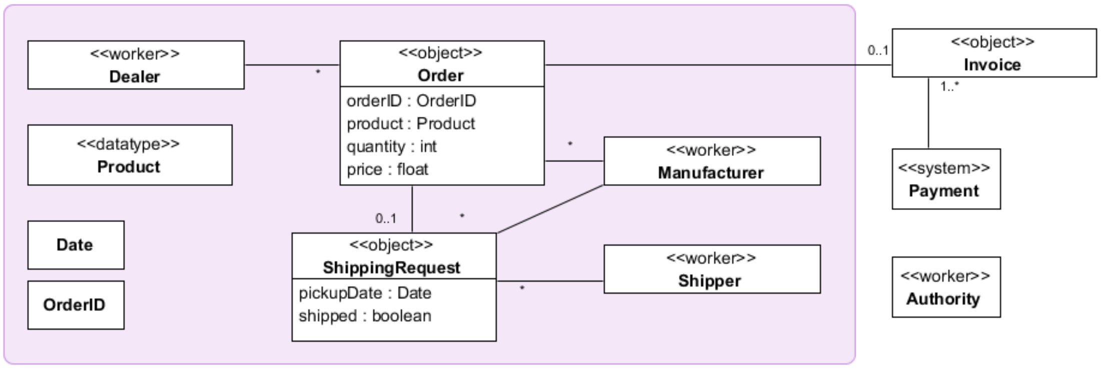


Figure 8.4: Placement Dealer Networking System on Static View of business model Dealer Network

Placement on the Behaviour View of Buying goods process

The activity diagram including a swimlane named Dealer Networking System in Fig. 8.5 is the placement on the behaviour view of process Buying goods, one of processes of business Dealer Network. All the activities placed inside the swimlane are executed by the Dealer Networking System.

There are two activities placed outside of the service system swimlane, i.e., SendInvoice and PayInvoice that are supposed to be performed by an existing payment application, not by the Dealer Networking System.

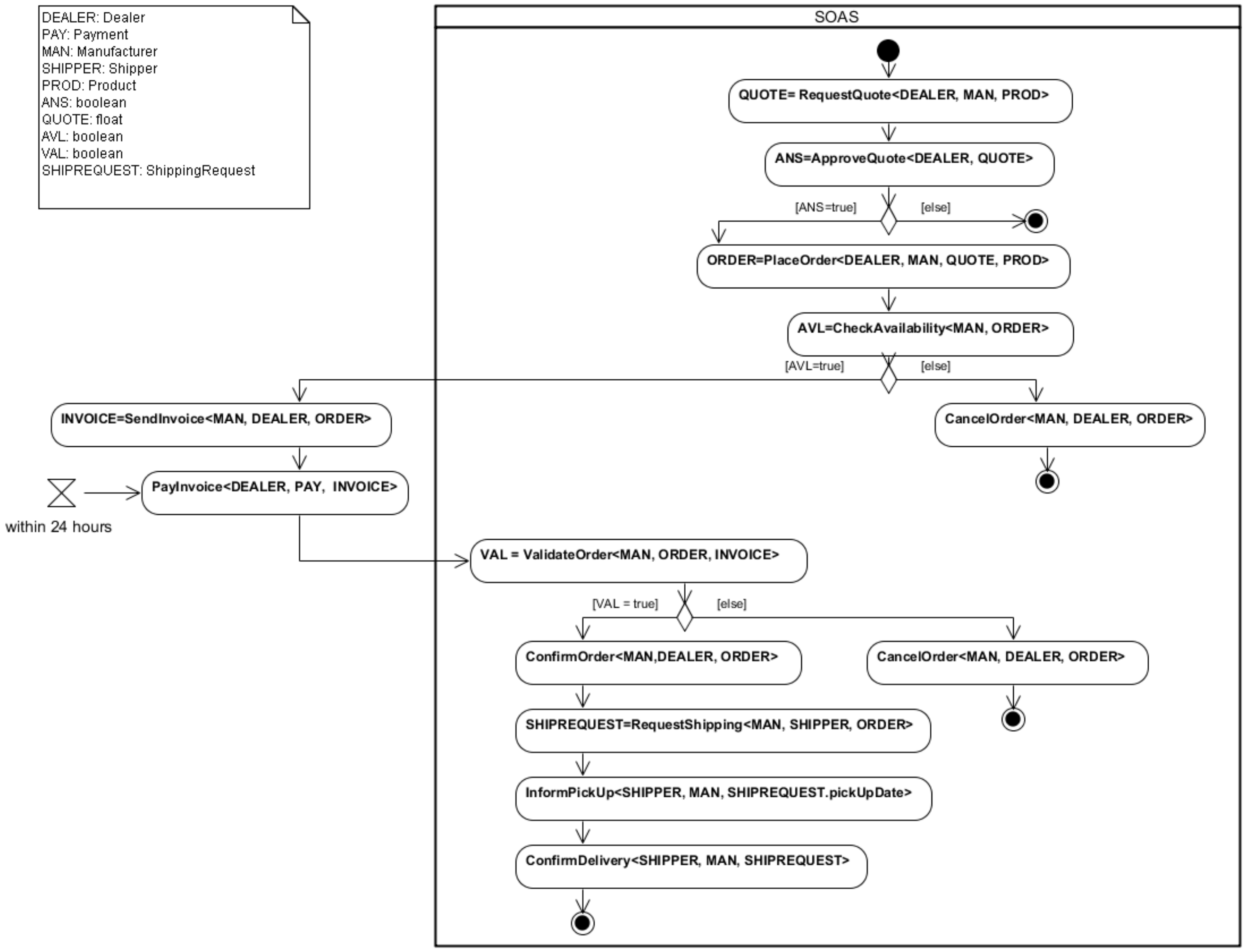

DEALER: Dealer
PAY: Payment
MAN: Manufacturer
SHIPPER: Shipper
PROD: Product
ANS: boolean
QUOTE: float
AVL: boolean
VAL: boolean
SHIPREQUEST: ShippingRequest
SOAS
QUOTE= RequestQuote<DEALER, MAN, PROD>
ANS=ApproveQuote<DEALER, QUOTE>
[ANS=true]
[else]
ORDER=PlaceOrder<DEALER, MAN, QUOTE, PROD>
AVL=CheckAvailability<MAN, ORDER>
[AVL=true]
[else]
INVOICE=SendInvoice<MAN, DEALER, ORDER>
CancelOrder<MAN, DEALER, ORDER>
PayInvoice<DEALER, PAY, INVOICE>
within 24 hours
VAL = ValidateOrder<MAN, ORDER, INVOICE>
[VAL = true]
[else]
ConfirmOrder<MAN,DEALER, ORDER>
CancelOrder<MAN, DEALER, ORDER>
SHIPREQUEST=RequestShipping<MAN, SHIPPER, ORDER>
InformPickUp<SHIPPER, MAN, SHIPREQUEST.pickUpDate>
ConfirmDelivery<SHIPPER, MAN, SHIPREQUEST>

Figure 8.5: Placement on Behaviour View of process Buying goods in business model Dealer Network

Placement on the Task View

After the placement over the BPOD, the Static View, and the behaviour views of business processes, we make the placement on the Task View. In Fig. 8.6, we can see that two classes outside the closed line the Static View (cf. Fig. 8.4) and two tasks outside the system swimlane in the behaviour view of Buying goods process (cf. Fig. 8.5) are placed again outside the closed line in the Task View. It means that those classes are not supported by the Dealer Networking System.

8.7.2 Applying Eliminate Useless Parts

In this section, we present the execution of the phase Eliminate Useless Parts, and apply some the relative patterns.

Fig. 8.7 shows the BPOD of the model of Dealer Business model with placement

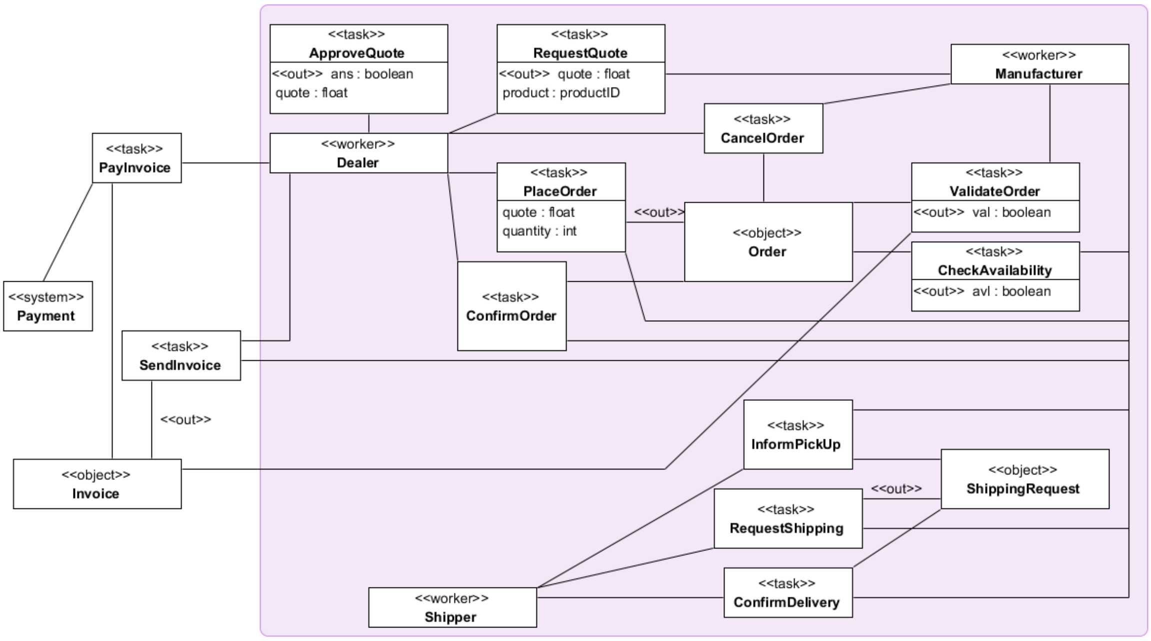


Figure 8.6: Placement Dealer Networking System on the Task View of business model Dealer Network

marking (see Fig. 8.3) after applying the patterns Eliminating unsupported business process, in which the process Handling reputation information and the business entity Authority are eliminated from the model. Process Handling reputation information is placed crossing the closed line of the placement, thus we can not apply the pattern Eliminating useless placement marking on BPOD, so the placement marking is not eliminated.

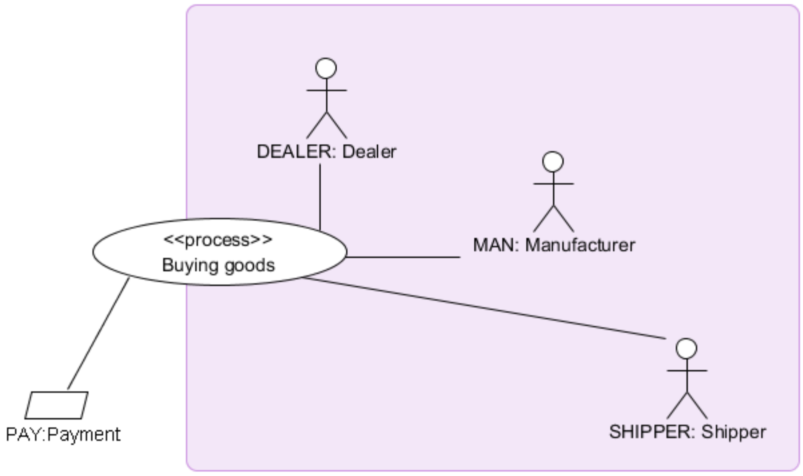


Figure 8.7: Dealer Business model BPOD transformed 1

Fig. 8.8 shows the Static View of the model of the Dealer Business with placement marking (see Fig. 8.4) after applying the patterns Eliminating useless placement marking on Static View and Eliminating unused class, in which three classes Invoice, Payment, and Authority are eliminated, their associations are also eliminated from the model.

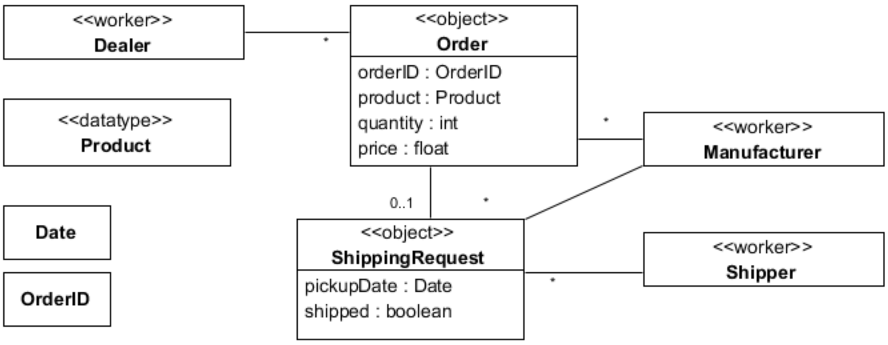


Figure 8.8: Dealer Business model: Static View transformed

Fig. 8.9 shows the BPOD of the Dealer Business model after applying the patterns Eliminating unused class on the Static View (see Fig. 8.8), in which the entity PAY: Payment is eliminated from the model.

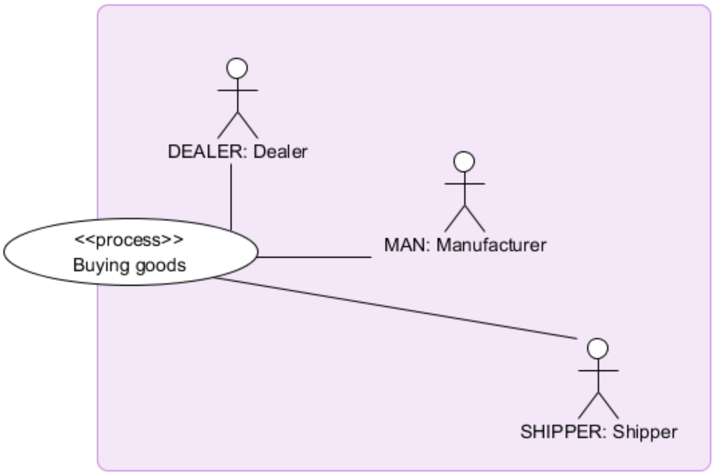


Figure 8.9: Dealer Business model: BPOD transformed 2

Fig. 8.10 shows the Task View of the model of Dealer Business with placement marking (see Fig. 8.6) after applying the patterns Eliminating useless placement marking on Task View, and Eliminating unused class, in which some task classes and entity classes that are not enclosed by the service system are eliminated from the model, i.e., PayInvoice, Payment, SendInvoice, and Invoice.

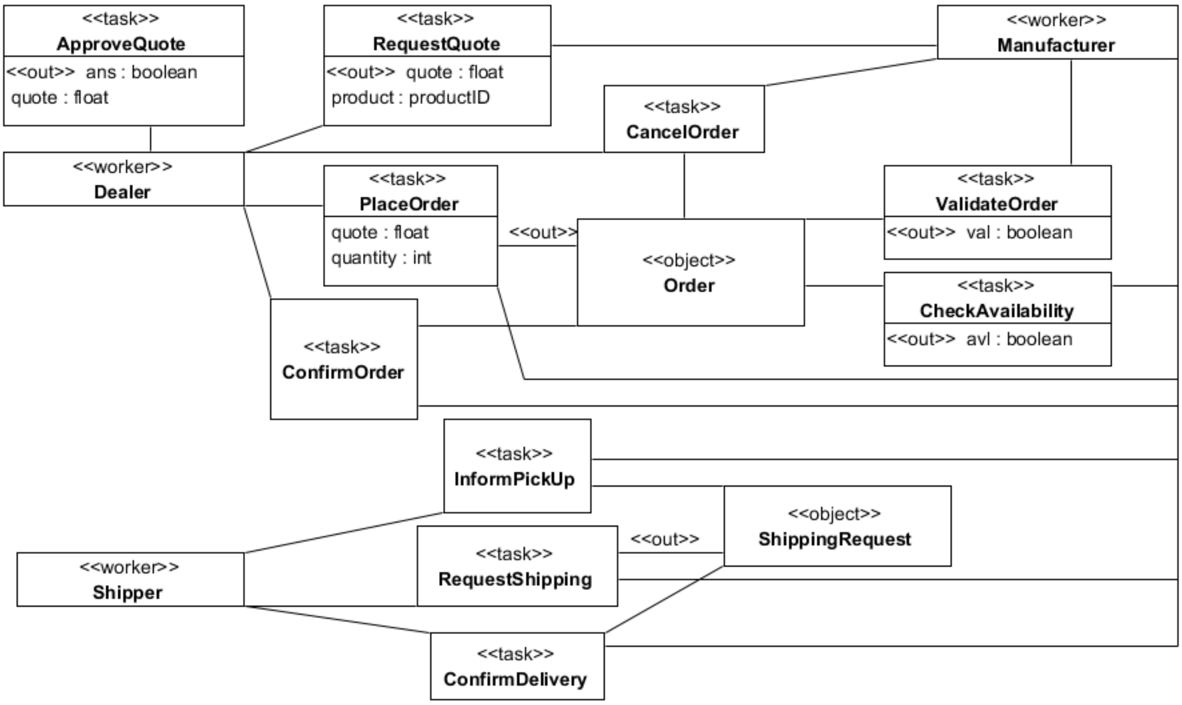


Figure 8.10: Dealer Business model: Task View transformed

8.7.3 Applying Simplify Tasks

Before the application:

The task class RequestQuote in the Task View of a business, its instance is Quote=RequestQuote<DEALER, MAN, PROD> in a behaviour view of a business process model.

After the application:

The task Quote=RequestQuote<DEALER, MAN, PROD> is decomposed into two tasks RequestQuote<DEALER, MAN, PROD> and Quote<MAN, DEALER, PROD, QUOTE>, in which, the first participant appear in each task is required to do this task.

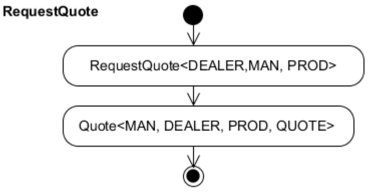

RequestQuote
RequestQuote<DEALER,MAN, PROD>
Quote<MAN, DEALER, PROD, QUOTE>

Figure 8.11: Decomposing a task in the Business Process Model of process Buying goods

8.7.4 Applying Introduce Services phase

Application of pattern Introducing empty service architecture

The current model includes Dealer Business System Placement Model and an empty Service Architecture illustrated in Fig. 8.12.

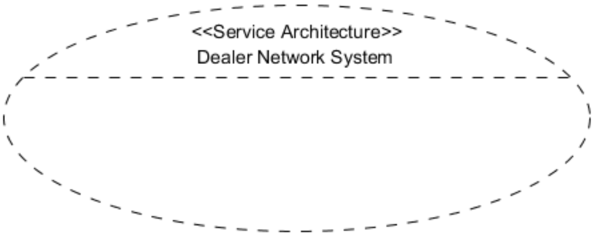


Figure 8.12: Introducing empty service architecture of Dealer Network System

8.8 Validation of designed system

Modelling realization of a business process in a service architecture is a quality validation of this service architecture.

There are two contexts for validating the realization of a business process on a designed service system: a *service system context* and a *structured participant context.*

Business process realization in "service system context" The services architecture of a service system may have one or more associated business processes models.

There are two cases in modelling a business process: modelling a "generic" business process attached to a services architecture, and modelling a "concrete" business process attached to a configuration of a service system.

A business process B in service system context will be modelled by an activity diagram defined as follows:

- A swimlane is used for any participant taking part in the business process.

  If B is a "generic" business process attached to a service architecture, the swimlanes are used for the roles of the participants. The swimlane of a participant is named in form: r:P, where r is the role typed by P, and P is a participant type;

  If B is a "concrete" business process attached to a configuration, the swimlanes are used for the instances of the participants. The instances correspond to the ones in the configuration. The swimlane of participant P is named in form: P:Ptype, where P is a participant in the configuration and typed by Ptype.

- All activities that a participant realizes should be put in its swimlane. The activities of a participant P should be named in the following form: q::Serv.operation(parameter) where if P is consumer of service Serv then q is a participant that provides service Serv, and operation is an operation of the provided interface of service Serv. This operation corresponds to the call from P to q; if P is provider of service Serv then q is a participant that consumes service Serv, and operation is an operation of the required interface of service Serv. This operation is sent from P to q.

- The condition expressions of decision nodes are built on operation parameters.

Constraints: The flow of the activities in the activity diagram relative to a service must be coherent with its service contract.

Business process realization in "participant context" A structured participant having inner participants and using internal services may be accompanied by one or more business processes to specify how this participant provide services.

The business process model of such structured participant may be the same as before but with extra swimlanes for the inner participants. Those extra swimlanes are used to group the activities of inner participants that are coherent with service contracts of the participant architecture.

8.8.1 Dealer Network business processes

The activity diagram in Fig. 8.13 illustrates a business process on the models of Dealer Network designed in Sect. 6.2 following precise SoaML method. In this business process, dealer d uses service Place Order to request the quote of the product he wants to buy from two manufacturers m1 and m2. He will place an order for this product from the manufacturer who offers a better price. In the case that the price of quote Q1 from manufacturer m1 is lower than or equal to the price of quote Q2 from manufacturer m2, he might place an order to manufacturer m1 by using service Place Order. In turn, he could place an order to manufacturer m2. The behaviours of dealer d are the actions belonging to his swimlane, i.e, m1::PlaceOrder.request_Quote(QR) or m2::PlaceOrder.place_Order(O). The first decision node depicts the point where the dealer can choose the best price for the product offered by different manufacturers. The actions of manufacturer m1 after he receives a place order request are sending confirmation to the dealer and sending shipping request to the shipper that he chooses if the order is confirmed. If the order is not confirmed, the process will come to an end. The shipper s1 is chosen by manufacturer m1, whereas the shipper of manufacturer m2 is s3. The shippers s1 and s3 will send the confirmations to the corresponding manufacturer for a fulfilled order when the shipment delivered. Their actions are illustrated on their swimlanes. In result, a complete process for the case that a dealer can buy and receive a product with the best price is depicted from the initiate state of the diagram to the final states coming out of the actions of the shippers.

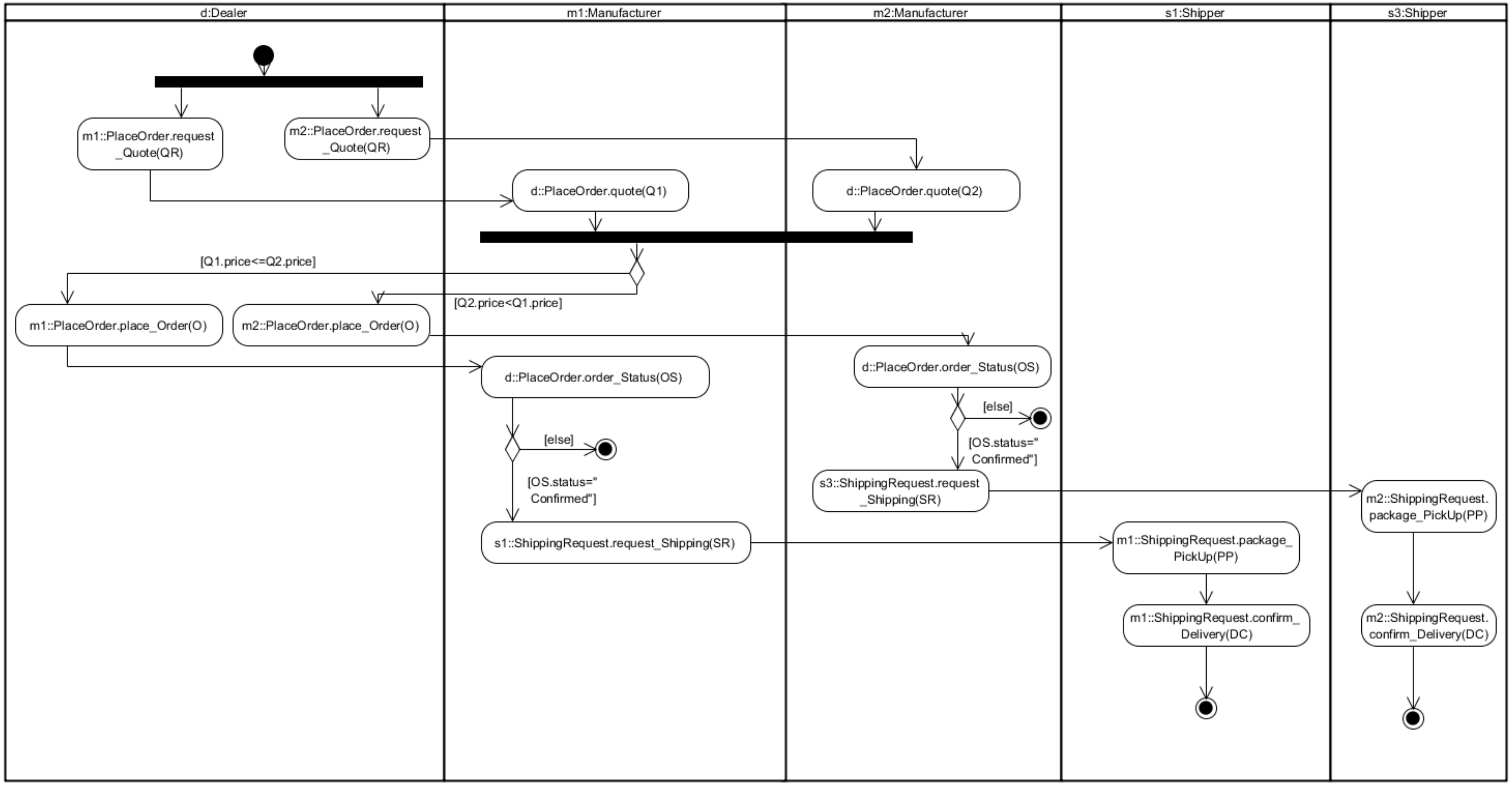

d:Dealer
m1:Manufacturer
m2:Manufacturer
s1:Shipper
s3:Shipper
m1::PlaceOrder.request _Quote(QR)
m2::PlaceOrder.request _Quote(QR)
d::PlaceOrder.quote(Q1)
d::PlaceOrder.quote(Q2)
[Q1.price<=Q2.price]
[Q2.price<Q1.price]
m1::PlaceOrder.place_Order(O)
m2::PlaceOrder.place_Order(O)
d::PlaceOrder.order_Status(OS)
d::PlaceOrder.order_Status(OS)
[else]
[OS.status=" Confirmed"]
[else]
[OS.status=" Confirmed"]
s3::ShippingRequest.request _Shipping(SR)
s1::ShippingRequest.request_Shipping(SR)
m2::ShippingRequest. package_PickUp(PP)
m1::ShippingRequest.package_ PickUp(PP)
m1::ShippingRequest.confirm_ Delivery(DC)
m2::ShippingRequest. confirm_Delivery(DC)

Figure 8.13: **Dealer Network Business Process 1**

Conclusion

We have introduced a method for designing a service system. It includes five phases, starting from placement phase to a set of four transformation phases in order to transform a business model into the model of a design of a service system.

In Place the System phase, we denote the placement using a closed line on some selected views of the business model in order to enclose every elements that the service system will support. The closed line is considered as a boundary of service system marked on the business model. We provides the rules and the constraints on how to place an element outside/inside/crossing the closed line and also on how to reflect the placement to other elements relating with this element. Following the guidelines describing how to do the placement, the user can adopt a set of steps to build a placement model for a modelled business in particular, and this is also the input of the later phases in general. The next four phases are considered as transformation phases, they are Eliminate Useless Parts, Simplify Tasks, Operationalize Tasks, and Introduce Services. The result of the application of each transformation pattern is an *intermediate model* . The intermediate models are UML models built using both the profile for modelling a business (see Sect. 5.2) and the profile for modelling the design of a service system (see Chapter 6).

We have illustrated our design method by modelling the Dealer Business.

Chapter 9

Comparison and evaluation of the methods

We spend this chapter to make a comparison between two our proposed approaches for modelling a service oriented system, i.e., PreciseSOA (c.f., Chapter 6) and Casl4Soa (c.f., Chapter 7), and to distinguish our works to other related works in the same field of interest.

The difference between the two approaches, PreciseSOA and Casl4Soa, is essentially the level of precision (and thus the expressiveness) both at the level of constructs and at the semantic level. Table 9.1 summarizes their main differences.

For example, the presence of a service semantic view in the Casl4Soa service model allows to express the meaning/effect of the functionalities offered by the modelled service. Casl4Soa provides the possibility to produce property-oriented style models, that may be more abstract than the constructive ones, but where all the nuances of the contract and the semantic view of a service may be expressed using temporal logic formulas.

The difference entailed by the object orientation (in UML-based, while Casl4Soa is not O-O) shows in the *service interface* modelling. First of all, the exchange of messages with a service is expressed in UML-based by operation calls whereas Casl4Soa uses pairs of matching elementary interactions. According to our precise approach, a service interface in UML-based is composed of the provided and used interfaces,[1] that belong to the provider and the consumer of the service, respectively: the provided interface offers the operations to be called by a service consumer, and

[1] which are UML interfaces

the used interface those to be called by the service provider.

	Casl4Soa	**PreciseSOA**
object oriented	no	yes
visual	yes	yes
semantics	formal semantics via translations into CaslLtl	as for the UML (informal)
datatypes	user defined with attributes or generators	user defined with attributes
dynamic elements	dynamic systems based on labelled transition systems	objects and active objects
communication mechanisms	simultaneously execution of pairs of matching elementary interactions	object-oriented messages (i.e., operation calls)
types of dynamic elements	dynamic types: elementary interactions, attributes or generators	O-O classes: operations, attributes
interfaces for dynamic types	set of elementary interactions	(provided or required) set of operations
constructive definition of the behaviour	interaction machines (reactive, autonomous and proactive behaviour)	state machines (reactive behaviour only)
logics	many sorted first-order branching time temporal logic with edge formulas (L2)	many sorted first-order L1) but (quantifications only over finite sets
constraints	logical properties using L2	invariants and pre/post conditions for operations using L1
structured dynamic elements	defined by communication structure of subsystems in terms of connectors	defined by structured classes and collaborations
behaviour of structured dynamic elements	- logical properties - interaction diagrams	sequence diagram
tools	reuse of UML tools	UML tools
formal analysis	possible via the corresponding CaslLtl specifications but not viable for non toy cases	only for tiny subsets and very micro examples
model quality	partly guaranteed by the notation	partly guaranteed by our precise approach

Table 9.1: Service modelling: comparison of Casl4Soa and PreciseSOA notations

The two methods are also quite different in modelling the service contract. The collaboration used to illustrate the contract in UML-based just depicts the role types of the service provider and consumer whereas in Casl4Soa the roles of the service provider and of the service consumer are implicit. In Casl4Soa, the service contract is precisely modelled by means of an interaction machine that define all the possible traces of message exchanges between the service provider and the service consumer.

Differently, in UML-based the service contract is modelled by means of sample traces, represented by sequence diagrams.

For both notations, models may be developed using UML editors supporting the use of profiles. For instance, the diagrams in this thesis are made with Visual Paradigm UML Editor.[2]

The big difference, between the notations used in the two approaches is that Casl4Soa is based on proprietary concepts and constructs coming from the algebraic specification languages, better well-known in the academy, whereas UML-based is based on object-oriented concepts and the UML constructs, widespread in the industry.

If in Casl4Soa we model also how the realm changes dynamically along the time, while now in UML we do not consider these aspects. For example, it is not modelled using PreciseSOA the fact that the quantity in stock of a product may decrease/increase freely (c.f., service Place Order in Sect. 6.2.2.1), while this features is modelled in the Casl4Soa version (c.f., service Place Order in Sect. 7.5.1.2).

In our opinion, both approaches, with their specific characteristics, are useful in proposing a precise framework for service modelling. Indeed, when someone has to decide which method to use to model a service, one should balance the wanted formality and expressiveness, and the context in which the models will be produced and read.

In terms of modelling services for the design of SOA systems, several proposals using UML have been developed, e.g, UML4SOA developed in Sensoria project [38, 35, 20]. The idea for introducing a semantic view of a service, not present in other approaches, was prompted by an interesting report of the Sensoria project [4].

SRML a formal semi-visual notation proposed in the Sensoria project [19, 52] (c.f., Section2.3). SRML [18] is based on a different view about SOA requiring e.g., to distinguish the interactions among modules (components) into business protocols, layer protocols and interactions protocols, whereas in Casl4Soa and in UML-based there is a unique type (the contract between who offer a service a who uses it).

[2] www.visual-paradigm.com

In [53], the definition of SOA is extended and a lightweight formal framework for capturing SOA main components is proposed, based on a critical assessment of existing design formalization techniques in object and component-oriented programming domains.

The precision and the formal semantic models in our Casl4Soa do more than editing the models, they are constructed with the same goal of other formal approaches in specification, analysis, and even checking the syntax of models. A formal approach given in [7, 6] (c.f., Section 2.3) uses priced-timed automata and address the formal verification of functional, timing and resourcewise correctness.

At last, the protocol of the services that is specified at precise level of our models shall be effective in refactoring services to maintain service systems for future change. This makes agility, one of the most important capabilities of SOA-based system, to be taken in to account.

The quality and the correctness of the proposed approaches is evaluated by the application on case-studies presented in the thesis. However, we need to extend the case-studies using their variants to verify the compatible of our approaches.

Chapter 10

Conclusion and future work

Conclusion

The goal of the research presented in this thesis is to develop precise methods for the modeling and the designing of services and service systems.

First of all, we introduce our view on services and service oriented systems (c.f., Service Systems Conceptual Model in Chapter 4). This view is the basis of our work in the building proposed methods. Particularly in our view, a *service system* is considered as a special case of a participant, it is a *structured participant* neither offering nor requiring services. Under this view, we have the agility in the modelling and designing of complicated service systems.

At the first aim, in Chapter 5, we have defined a method for defining and modelling a business and its processes in such a precise way to avoid any misunderstanding and unclear point, to help the readers. This method use the UML notation, also a specific profile of the UML. A set of guidelines helping the users of our method are provided. We have then applied the method to model a case study, the dealer network business.

The following aims, that are to model and to design a service system being built to support a modelled business, are obtained in later chapters.

For modelling a service system, we have used the UML notation and extended Casl-Mdl to propose two different profiles, called PreciseSOA and Casl4Soa. While PreciseSOA applies to a more widespread notation and Casl4Soa to a formal one, both has been designed to be used when following the SOA paradigm, and are in accord with our view on services and service systems presented in Chapter 4. The choice of using one or the other can be made given the context in which the models are to be developed and used. PreciseSOA method has been inspired by SoaML, a standard OMG profile to architect and model SOA solutions. For this method, we provided a metamodel defining the structure of the service model, named *UML Service System model*, equipped with well-formedness constraints so as to guide the use of the notations. This leads us to select a subset of UML constructs, which relieves the modellers from the work of choosing which one to model the concepts of a service system, i.e., *participant*, *service contract*, *service interface*, *service architecture*. CASL4SOA has been developed as a profile of CASL-MDL, in which, we used stereotyped CASL-MDL constructs summarized in Appendix A.3 to define the new CASL4SOA constructs. We proposed two kinds of CASL4SOA service models, constructive and property-oriented. In a constructive model, the behavioural aspects of the services and of the participants are expressed by means of the interaction machines, whereas in a property oriented model, such aspects are expressed by means of first-order temporal logic formula. We have illustrated the application of those two modelling methods on two case-study.

For design a service system being built to support a business, in Chapter 8, we presented a method following the Model Driven Approach. The method provides a set of transformation patterns to transform a model of the business where the system to be developed has been placed, i.e., the parts of the business to be supported have been marked (see Sect. 8.2) into a service system design model that will include also some extra parts to show how the various processes of the business have been realized. To help the designer activities, we provided many transformation patterns, covering the most frequent cases, inspired by the well-known design patterns[1]. The transformation activity is organized in five phases, see Fig. 8.1, and cannot be fully automatized, since the designer has to make many choices during the transformation. The first phase, presented in Sect. 8.2, require to place the service system being built over the modelled business. We denoted the placement using a closed line as a boundary of service system marked on the business model. We have proposed the rules and the constraints on how to place an element outside/inside/crossing the closed line and also on how to reflect the placement to other elements relating with this element. The second phase *Elimination of useless parts* (Sect. 8.3) is used to eliminate the parts of the model not covered by the placement. In the third phase *Task simplification* (Sect. 8.4), the tasks will be decomposed into smaller tasks. In the fourth phase *Task operationalization* (Sect. 8.5), the tasks will be transformed into the operation calls of the business entities. And in the last phase *Introducing services* (Sect. 8.6), the entities of the business are replaced by the participants of

[1] http://en.wikipedia.org/wiki/Software_design_pattern

the service system and several new services are introduced. We have illustrated our design method on several small examples (attached in the description of each pattern), and a case-study.

Finally, to validate the quality a designed service system, in Sect. 8.8, we propose a mechanism using UML activity diagram to model the realization of a business process in the architecture of this service system.

Future work

We have identified several possible directions that our research could follow in the future. The current approach has been validated by the application on case studies, however it is needed to verify and investigate more well-formedness constraints in each methods.

In the future, we plan to investigate possibility for implementing a service system by defining transformations from the design model of a service system into a running system coded using WSDL and BPEL and existing services.

Our plan is to provide a tool suitable not only for modelling but also checking correctness of the constraints on the elements by OCL. We may take the advantage of existing tools that allow to define OCL constraints and verify them on the models, such as Papyrus of Eclipse[2]. We intend to use some frameworks from Eclipse, i.e., EMF[3] and GFM[4] to develop a stand alone environment for supporting our proposed approach to SOA system development ranging from modelling business to transforming it into design model of a service system. Furthermore, this tool should support model-driven engineering by being integrated with existing technologies.

[2] http://www.eclipse.org/papyrus/

[3] http://www.eclipse.org/modeling/emf/

[4] http://www.eclipse.org/modeling/gmp/

Appendix A

Appendix

A.1 UML

A.1.1 Classifier

Classifier is an abstract metaclass which describes set of instances having common features. A feature declares a structural or behavioral characteristic of instances of classifiers.

For instance, classifier are classes, interfaces, enumerations, associations, datatypes.

A.1.2 Interface

Interface provides an entry point that consumers use to access the functionality exposed by the application. Each software service has an associated interface that it presents to the consumers.

In the UML, interfaces (see "Interface" Appendix A.1.4) are model elements that define sets of operations that other model elements, such as classes, or components must implement.

Difference between interface and class (see "Class" in Appendix A.1.4): An interface includes only operations.

A.1.3 Datatype

A datatype is a type that its instances are identified only by their value. A datatype may contain attributes to support the modeling of structured data types.

Notation: A datatype is denoted using the rectangle symbol with/without keyword ≪datatype≫ or ≪Datatype≫, the rectangle contains the name of the datatype, and may contain a number or also none of attributes.

Constraints:

A datatype must have a name.

There is no association (c.f. Appendix A.1.4) towards or from a datatype.

A.1.3.1 Primitive Type

A primitive type is a datatype which represents atomic data values, i.e. values having no parts or structure.

Standard UML primitive types include: *Boolean*, *Integer*, *String*, *Float*, and *Real*.

A.1.3.2 Enumeration

An enumeration is a datatype whose values are enumerated in the model as user-defined enumeration literals.

Notation: An enumeration is shown using a rectangle with the keyword ≪enum≫ or ≪enumeration≫. The rectangle has three compartments: upper compartment is used to place the name of the enumeration, a compartment listing the attributes for the enumeration is placed below the name compartment, the compartment placed below the attribute compartment is used to list the operations for the enumeration.

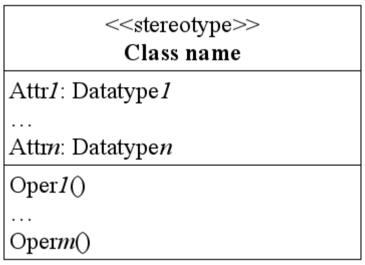
A.1.4 Class Diagram

Description A class diagram is a type of static structure diagram. It represents static aspect of a system by showing the its classes (their attributes and operations), interfaces, associations and generalizations.

Elements

Class A class is a definition of objects that share given structural or behavioral characteristics. A class comprises a name, a number of attributes, and a number of operations.

Notation: A class is shown as a rectangle containing class name and optionally with compartments separated by horizontal lines. The rectangle may have three compartments: upper compartment is used to place the name of the class and optionally with a stereotype, a compartment listing the attributes of the class is placed below the name compartment, the compartment placed below the attribute compartment is used to list the operations of the class or other members of the class.



Attribute An attribute is a typed value attached to each instance of a class.

Operation An operation is a function that can be performed by instances of a class. An operation may have return type in case it returns a value.

Interface An interface is a classifier that declares of a set of coherent public features and obligations. An interface specifies a contract. Any instance of a classifier that realizes the interface must fulfill that contract and thus provides services described by contract.

Notation: An interface is using a rectangle symbol with the keyword ≪interface≫ preceding the name. The rectangle has two compartments: upper compartment is used to place the name of the interface, a compartment listing the operations for the interface is placed below the name compartment.

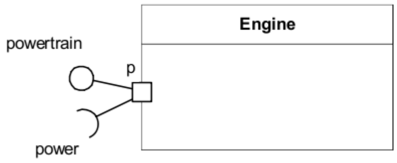
Provided interface Interfaces realized by a classifier are its provided interfaces, and represent the obligations that instances of that classifier have to their clients. They describe the services that the instances of that classifier offer to their clients.

Notation: Interface participating in the interface realization dependency is shown as a lollipop (i.e., a ball or a circle), labeled with the name of the interface and attached by a solid line to the classifier that realizes this interface.

Required interface Required interface specifies services that a classifier needs in order to perform its function and fulfill its own obligations to its clients. It is specified by a usage dependency between the classifier and the corresponding interface.

Notation: The usage dependency from a classifier to an interface is shown by representing the interface by a cup (i.e., a half-circle), labeled with the name of the interface, attached by a solid line to the classifier that requires this interface.

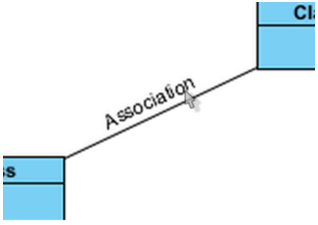
Example Figure below shows the notation for port `p`, it is a port on the *Engine* class. The provided interface of port `p` is *powertrain*, and the required interface of port `p` is *power*.



Relationships among instances

Association An association is a relationship between the members of two classes.

Notation: An association is drawn as a solid line between two classes.

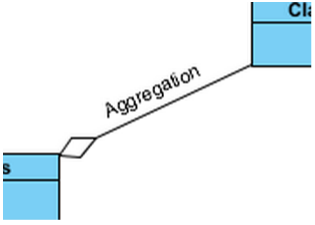


Association name An association may have a name. The name can be left empty.

Association end An association has two ends. Each end is the name a role, and is used to refer to the associated object (further described in A.1.4.1).

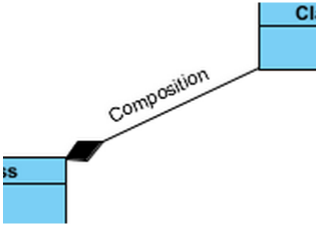
Aggregation (Shared association) An aggregation is an association represents a shared ownership relationship. A aggregation is a "weak" form of aggregation when part instance is independent of the composite in such a way that: the same (shared) part could be included in several composites, and if composite is deleted, shared parts may still exist.

Notation: An aggregation is shown as binary association decorated with a hollow diamond as a terminal adornment at the aggregate end of the association line.



Composition An association represents a whole-part relationship. A composition is a "strong" form of aggregation when part instance is independent of the composite in such a way that: part could be included in at most one composite (whole) at a time, and if a composite (whole) is deleted, all of its composite parts are "normally" deleted with it.

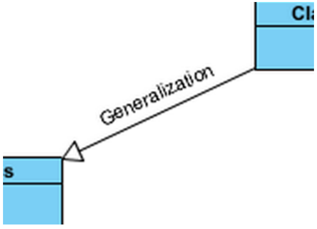
Notation: Composite aggregation is depicted as a binary association decorated with a filled black diamond at the aggregate (whole) end.



Relationships among classes

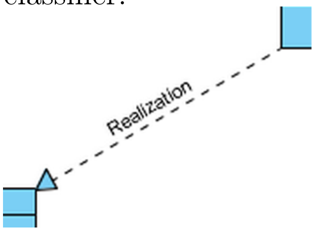
Generalization The specific class inherits part of its definition from the *general* class.

Notation: A generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier.



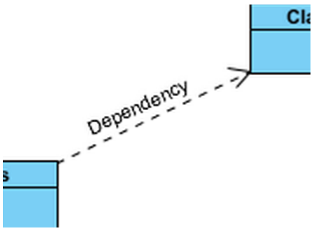
Realization A realization is a specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other represents an implementation of the latter (the client).

Notation: A generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier.



Dependency A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).

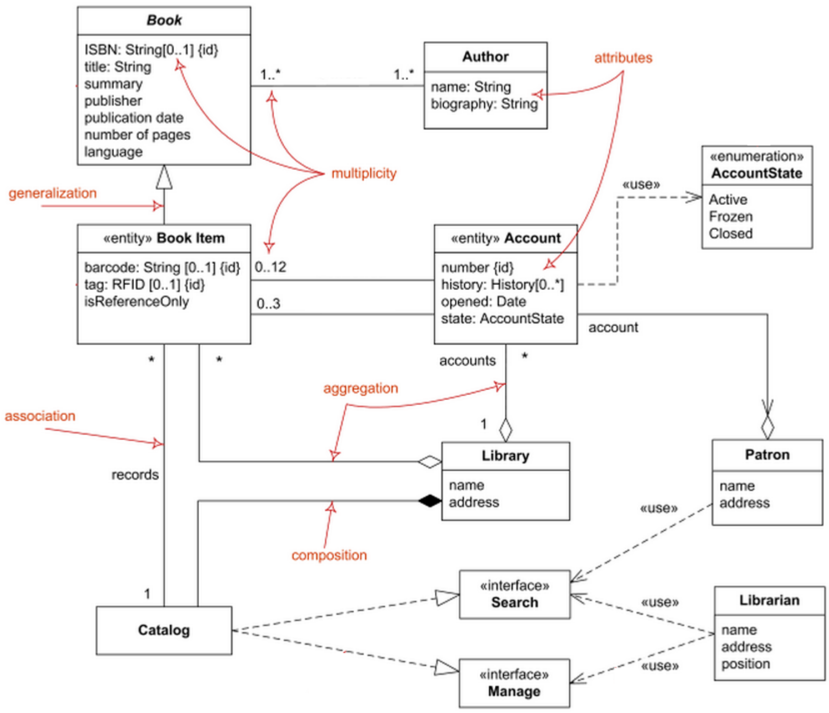
Notation: A dependency is shown as a dashed arrow pointing from the client (dependent) at the tail to the supplier (provider) at the arrowhead. The arrow may be labeled with an optional stereotype and an optional name. The definition or implementation of the dependent classifier might change if the classifier at the arrowhead end is changed.



Multiplicity Multiplicity is a definition of cardinality, i.e., number of elements, of some collection of elements by providing an inclusive interval of non-negative integers to specify the allowable number of instances of described element. Multiplicity is used to indicates how many of the objects at this end can be linked to each object at the other.

Multiplicity interval has some lower bound and (possibly infinite) upper bound. Lower and upper bounds could be natural constants or constant expressions evaluated to natural (non negative) number. Upper bound could be also specified as asterisk '*' which denotes unlimited number of elements. Upper bound must be greater than or equal to the lower bound.

Some typical examples of multiplicity: 0. . . 0/0: Collection must be empty; 0. . . 1: No instances or one instance; 1. . . 1/1: Exactly one instance; 0. . . */*: Zero or more instances; 1. . . *: At least one instance; m. . . n: At least m but no more than n instances



Example

Constraints

A class must have a name.

The classes must have different names.

An aggregation cannot involve more than two classes.

A.1.4.1 Association end

An association end identifies the entity type on one end of an association and the number of entity type instances that can exist at that end of an association. Association ends are defined as part of an association; an association must have exactly two association ends. Navigation properties allow for navigation from one association end to the other.

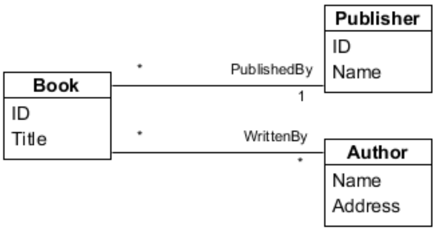


Figure A.1: Assocation end example

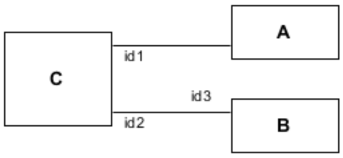


Figure A.2: Generic association end

Association end could be owned either by end classifier, or association itself. Association ends of associations with more than two ends must be owned by the association.

A.1.4.2 Distinguish parameters of kind *in*, *out*, and *inout*

The parameter is of kind **in** when it must be set to a value before calling the operation. We use *in* parameter to pass values to an operation. Inside the operation, it will acts like a constant and cannot be assigned a value.

The parameter is kind **out** when it does not need to be set before calling the operation and it often contains the result. We use **out** parameter to return values to the caller. Inside the operation, an **out** parameter acts like a variable. We can change its value, and reference the value after assigning it. We must pass a variable, not a constant or an expression to an **out** parameter.

The parameter is kind of **inout** when it must be set to a value before calling the operation, but once the execution of the parameter is completed, the same parameter will be holding the last updated valued of it. We use an **inout** parameter to pass initial values to an operation and returns updated values to the caller. It can be assigned a value and its value can be read.

A.1.5 Use case Diagram

Description A *use case diagram* is used to visualize high level functions or requirements of a system, i.e., what a system is supposed to do (but not how to do). A use case diagram contains primarily actors and use cases. Actors are entities that interact with the system, while *use cases* are a means for capturing the requirements of a system. A use case diagram is a specialization of Class Diagram such that the classifiers shown are restricted to being either *Actors* or *Use Cases.*

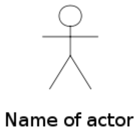
Elements

Actor An actor models a type of role played by an entity, e.g., a user or any other system, that interacts with the system.

An actor is external to the system.

In a use case diagram, an actor icon may represent roles played by human users, external hardware, or other systems.

Notation: An actor is represented by "stick man" icon with the name of the actor below the icon.

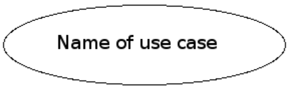


Use Case A use case is the specification of a set of actions performed by a system, which result in value for one or more actors of the system.

A use case is a kind of behaviored classifier that represents a declaration of an offered behavior.

In a use case diagram, a use case icon represent the actions performed by one or more actors in the pursuit of a particular goal.

Notation: A use case is shown as an ellipse containing the name of the use case. An optional stereotype keyword may be placed above the name.



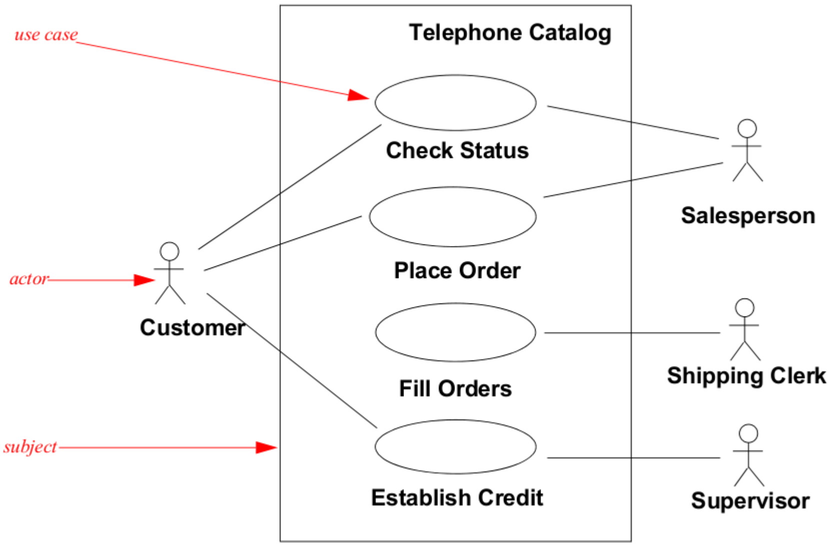
Association An *association* indicates that an actor takes part in this use case.

Notation: An association relationship between an actor and a use case is shown by a solid line, i.e., a binary arrow.

Dependency relationship A *dependency* indicates that the design of the source depends on the design of the target.

Notation: An dependency relationship between use cases is shown by a dashed arrow. The arrow may be labeled with an optional stereotype keyword.

Example The use case diagram below (derived from [25]) shows a set of use cases used by four actors, i.e., Customer, Saleperson, Shipping Clerk, and Supervisor, of a system, i.e., Telephone Catalog, that is optionally represented by a rectangle.



Constraints

An actor must have a name.

An actor can only have associations to use cases.

A use case must have a name.

A use case cannot have a association to any other use case.

A use case cannot include use cases that directly or indirectly include it.

An association between an actor and a use case must be binary.

A.1.6 Object Diagram

Description An object diagram shows a view of the structure of the system at some point in time. It acts as a test case for the class diagram. An object diagram consists of objects in some states and the links among them (a link is an instance of an association in a class diagram).

Elements

Object

An object is an instance of a class. It is represented by a rectangle. An object has a name and the values of the attributes of the corresponding class that need to be captured. A name of an object is underlined and in the form of “X:Y”, where X is name of the object and Y is name of the corresponding class. The name of the object is optional. It is not mandatory to give a value to all the attributes.

Relationships among instances

See “Relationships among instances” form Appendix A.1.4.

Example

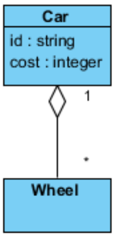


Figure A.3: Class Diagram Example

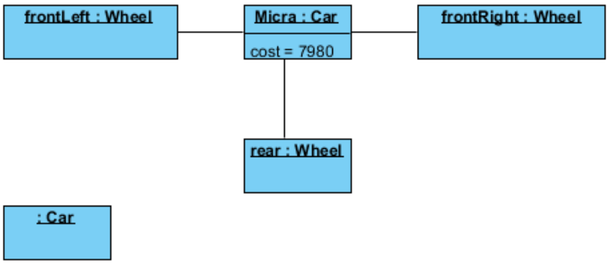


Figure A.4: Object Diagram Example

Constraints

Each object is corresponding to an instance of a class in a class diagram.

Each link is corresponding to an instance of an association in a class diagram.

The value of attribute of object must have the type required for such attribute in corresponding class.

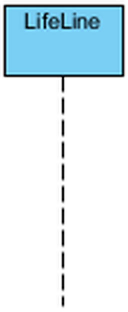
The numbers of instances are not limited.

A.1.7 Sequence Diagram

Description A Sequence Diagram models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case.

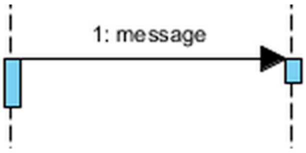
Elements

Lifeline A lifeline represents an individual participant in the interaction.

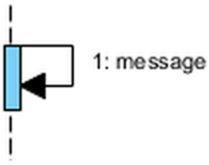


Notation:

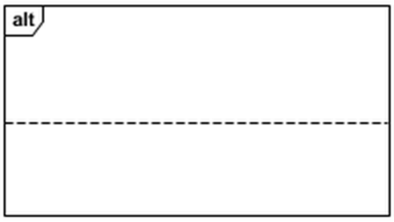
Message A message defines a particular communication between Lifelines of an Interaction.

Notation: 

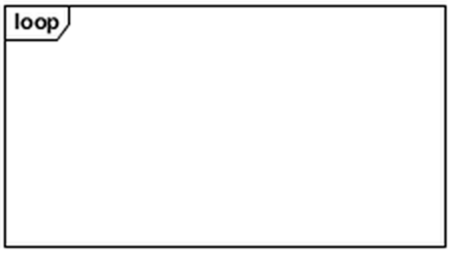
Self Message A self message is a kind of message that represents the invocation of message of the same lifeline.

Notation: 

Alternative Combined Fragment An alternative combined fragment represents a choice of behavior. At most one of the operands will be chosen.

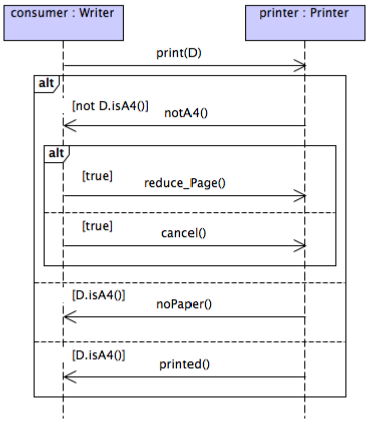
Notation: 

Loop Combined Fragment A loop combined fragment represents a loop. The loop operand will be repeated a number of times.

Notation: 

Example

The sequence diagram in the example below describes the sequence of messages that are exchanged between two participants: Writer and Printer.



Constraints A lifeline must have a name.

If the message is an operation call having a return value, e.g., oper(): TypeT, then the message must have the form of x=oper(), in which x is a variable of type TypeT.

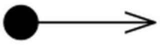
A.1.8 Activity Diagram

Description An activity diagram is a connected oriented graph made of *activity nodes* and *activity edges* (connecting a pair of activity nodes). The dynamics of an activity diagram is then defined in terms of control tokens flowing in and out of the activity nodes moving along the activity edges. There are different kinds of activity nodes and edges. Set of constructs to build the activity diagram representing the behaviour of a business process presented in [46] is recalled below, together with some rules needed to be conformed when one creates an activity diagram.

Elements

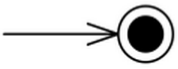
Initial node

An initial node is a control node at which flow starts when the activity is invoked. An activity may have more than one initial node.

Notation: 

Final nodes

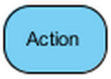
An activity may have more than one activity final node. The first one reached stops all flows in the activity.

Notation: 

Action nodes

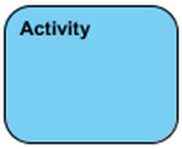
An action node describe what will be done in the process modelled by the activity diagram, they may be denoted using either the UML actions (more or less the usual statements of an object oriented programming language) or with a name (just an identifier).

Notation: Action is shown as a round-cornered rectangle with action name in the center.



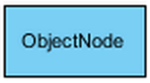
Activity An activity specifies the coordination of executions of subordinate behaviors that are modeled as activity nodes connected by activity edges. An activity may include flow of control constructs (e.g., decision, merge nodes, etc.,).

Notation: Activity is shown as a round-cornered rectangle with activity name in the upper left corner and nodes and edges of the activity inside the border.



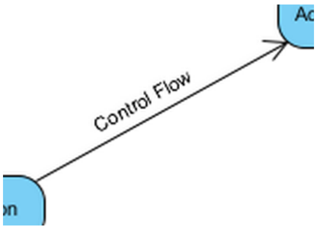
Object nodes

An object node is an activity node that indicates an instance of a particular classifier, possibly in a particular state, may be available at a particular point in the activity.

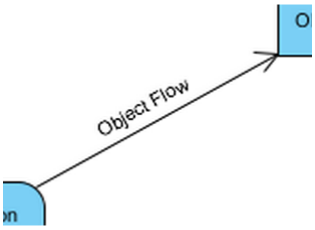
Notation: 

Control flow edges

A control flow edge connects two nodes and depicts the flowing of a control token from the first to the latter. The action nodes may have any number of ingoing and outgoing control flow edges, whereas the initial nodes may have only outgoing edges and the final nodes only ingoing edges.

Notation: 

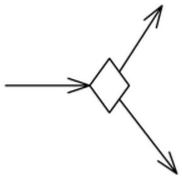
Object flow edges An object flow edge is an edge that can have objects or data passing along it.

Notation: 

Decision nodes

A decision node accepts tokens on an incoming edge and presents them to multiple outgoing edges. Which of the edges is actually traversed depends on the evaluation of the guards on the outgoing edges.

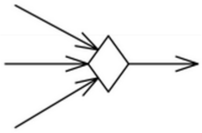
Notation:



Merge node

A join node is a control node that synchronizes multiple flows. A join node has multiple incoming edges and one outgoing edge.

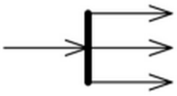
Notation:



Fork nodes

A fork node is a control node that splits a flow into multiple concurrent flows. A fork node has one incoming edge and multiple outgoing edges.

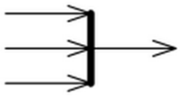
Notation:



Join nodes

A join node is a control node that synchronizes multiple flows. A join node has multiple incoming edges and one outgoing edge.

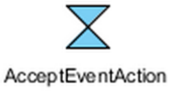
Notation:



Time and accept events

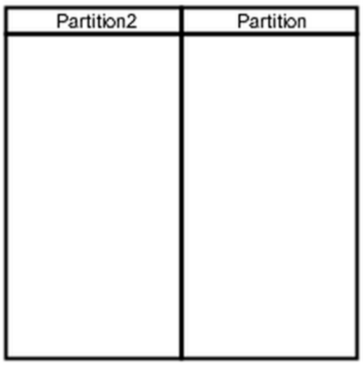
If the occurrence is a time event occurrence, the result value contains the time at which the occurrence transpired. Such an action is informally called a wait time action.

Time and accept events may be used in the activity diagrams to model business process where some tasks are activated by the reaching of some specific time or by the receiving of some messages (i.e., operation calls). The events may be seen as nodes with exactly one outgoing edge and either one or zero ingoing edge. The time event (represented by the hourglass symbol) may be seen as a special activity that will deliver a control token on the outgoing edge at the time annotating it, whereas an accept event will do the same after receiving the message annotating it. The events without ingoing edge are assumed to be always active and can deliver the token many times, whereas those with the ingoing edge are activated upon the receiving of the control token (and will be able to deliver at most one token).

Notation: 

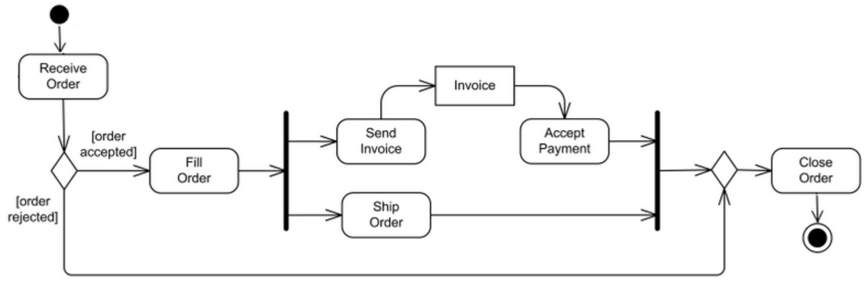
Activity partition (Swimlane) An activity partition is a kind of activity group for identifying actions that have some characteristic in common. Partitions divide the nodes and edges to constrain and show a view of the contained nodes.

Notation: Activity partition may be indicated with two, usually parallel lines, either horizontal or vertical, and a name labeling the partition in a box at one end. Any activity nodes and edges placed between these lines are considered to be contained within the partition.



Example

Figure below shows an example of business flow activity of Order processing.



Constraints

An activity diagram has one and only one *initial node*, and any number, also none, of *final nodes*.

A join node must have exactly one outgoing edge and at least two ingoing edges, when there will be a token arriving on all of the ingoing edges, only one token will be displaced on the outgoing edge.

A fork node must have exactly one ingoing edge and at least two outgoing edges; a token arriving by the ingoing edge will be displaced on all the outgoing edges.

A decision node must have exactly one ingoing edge and at least two outgoing edges, each of them labelled by a guard (a condition) without side effects; a token arriving by the ingoing edge will be displaced on only one outgoing edge with the true guard. The special guard "else" is a shortcut for the conjunction of the negations of the guards of the other edges, thus it will be selected whenever all the other guards are false. A merge node must have exactly one outgoing edge and at least two ingoing edges; a token arriving by any of the ingoing edges will be displaced on the unique outgoing edge. A *merge node* and a decision node can be combined at the visual level by allowing them to share the same node symbol. The guard of the decision node must be defined.

A.1.9 Composite Structure Diagram

Description Composite structure diagram visualizes the internal structure of a class or collaboration. Its primary purpose is to explain how a system works.

Elements

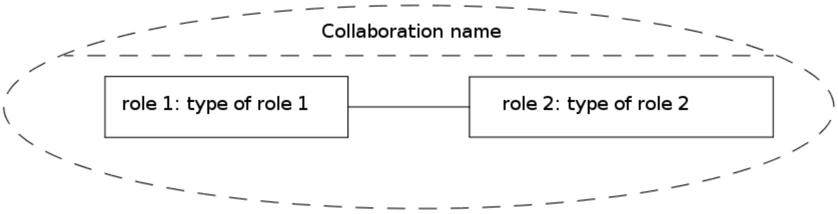
Connector A connector specifies a link that enables communication between two or more instances. This link may be an instance of an association.

Notation: A connector is shown as a solid line.

Collaboration A collaboration defines a set of cooperating entities to be played by instances (its roles), as well as a set of connectors that define communication paths between the participating instances. The cooperating entities are the properties of the collaboration.

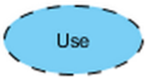
A collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality.

Notation: A collaboration is shown as a dashed ellipse icon containing the name of the collaboration. The internal structure of a collaboration as comprised by roles and connectors may be shown in a compartment within the dashed ellipse icon.



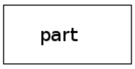
Collaboration Use A collaboration use represents the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration.

Notation: A collaboration use is shown by a dashed ellipse containing the name of the occurrence, a colon, and the name of the collaboration type. For every role binding, there is a dashed line from the ellipse to the client element, and the dashed line is labeled on the consumer end with the name of the provider element.



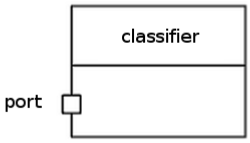
Part A part represents a set of instances that are owned by a containing classifier instance.

Notation: A part is shown as a rectangle containing the name of the instance.



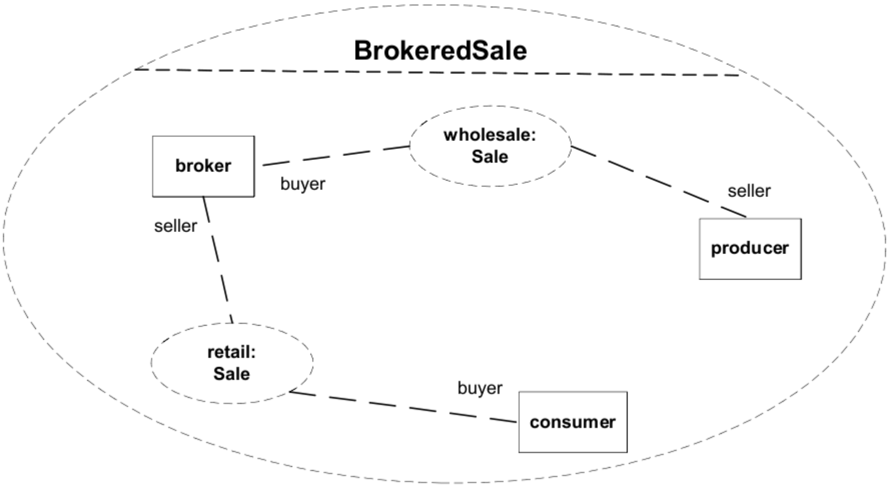
Port A port is a property of a classifier that specifies a distinct interaction point between that classifier and its environment or between the classifier, i.e., its behavior, and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier. A Port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment.

Notation: A port of a classifier is shown as a small square symbol. The name of the port is placed near the square symbol. A provided interface may be shown using the "lollipop" notation (i.e., a ball) attached to the port. A required interface may be shown by the "cup" notation (i.e., a half-circle) attached to the port.



Example

BrokeredSale is a collaboration among three roles, a *producer*, a *broker*, and a *consumer*. It consists of two occurrences of the *Sale* collaboration, i.e., wholesale: Sale and retail:Sale, indicated by the two dashed ellipses. The occurrence wholesale indicates a *Sale* in which the producer is the seller and the broker is the buyer. The occurrence retail indicates a *Sale* in which the broker is the seller and the consumer is the buyer.



Constraints

There are exact two parts binding with a collaboration use.

The required interfaces of a port must be provided by elements to which the port is connected.

When a port is destroyed, all connectors attached to this port will be destroyed also.

A.2 OCL - Constraints for UML

The OCL (Object Constraint Language) [12] is a declarative language for describing rules that apply to UML [25]. OCL is a formal specification language extension to

UML, and is used with any Meta Object Facility[1] (MOF) or meta-model. It is a precise text language providing constraint and object query expressions on any MOF model or meta-model that can not be expressed by diagrammatic notation.

In OCL 2.0, the definition has been extended to include general object query language definitions. OCL language statements are constructed in four parts:

1. a context that defines the limited situation in which the statement is valid
2. a property that represents some characteristics of the context
3. an operation that manipulates or qualifies a property, and
4. keywords that are used to specify conditional expressions.

OCL supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. It is used to express additional constraints on UML models that are very difficult to express by the graphical means provided by UML. OCL is based on first-order predicate logic but it uses a syntax similar to programming languages and closely related to the syntax of UML.

Various open-source OCL tools has been developed. These can be integrated into UML CASE tools to support precise specification of UML models beyond the pure specification of OCL expressions as strings.

OCL offers a Smalltalk-based "block" syntax for convenient definition of some kinds of functions and directly, but it does not provide corresponding type rules for this syntax.

With OCL syntax, any function f(a,b):c can be described as an object with a single method eval(a, b): c. For example: Collection(T):: select (T → Boolean): Collection(T) means "*select takes a block parameter that maps each element to a Boolean; select returns another collection of T*".

Most collection functions on Collection(T) use blocks that are Predicate(T) to do selection, Comparator(T) to do sorting, and Converter(T, T1) to do a general mapping from the collection elements. Collection(T) has associated functions, such as: c→size, c→exists(x | P), c→forAll(x | P), c→select(x | P), c→reject(x | P), c→collect(x | E).

Collection types include sequences (ordered), bags (not ordered), and sets (no duplicates, not ordered). Each of them has its own expression and associated functions, such as: Seq(T) with s→prepend (e), Bag(T) with b→intersection (c), and Set(T) with s→intersection(c).

[1] http://www.omg.org/mof/

In addition, OCL includes the expressions for some data types and associated operations such as Boolean with *and, or, if* b *then* e1 *else* e2 *endif*...; and String with s.substring(l, u), Real with <, >, >=, <=, r. max(r2), and Integer with i mod i2.

In OCL, "→" symbol means "*do not move through a level of indirection*". For example: joe.cars.size means "*collecting the size of each of Joe's car into the result*", otherwise joe.cars → size means "*just give the size of the set of Joe's cars*".

It would be more consistent to have the "." operator have the same meaning for collections and non-collection types and to use "→" to apply to each element in the collection.

List of OCL tools:

- Dresden OCL for Eclipse that supports OCL2.2
  `http://sourceforge.net/projects/dresden-ocl/`
- Eclipse MDT/OCL
  `http://www.eclipse.org/modeling/mdt/downloads/?project=ocl`
- Incremental OCL
  `http://www.lsi.upc.edu/~jcabot/research/IncrementalOCL`
- HOL-OCL - An Interactive Proof Environment for OCL
  `http://www.brucker.ch/projects/hol-ocl/`
- OCL editor with libraries
  `http://squam.info/?p=142`
- OSLO - Open Source Library for OCL
  `http://oslo-project.berlios.de`
- OCLE
  `http://lci.cs.ubbcluj.ro/ocle/index.htm`

List of UML and MDE tools that provide OCL support by different manner and power (parsing, static checking, evaluation, code generation, etc.):

- ArgoUML (Open Source)
  `http://argouml.tigris.org/`
- Borland Together
  `http://www.borland.com/us/products/together/index.html`

- Eclipse Model Development Tools (MDT)
  `http://www.eclipse.org/modeling/mdt/?project=ocl`
- ECO for Visual Studio by CapableObjects
  `http://www.capableobjects.com/ProductsServices_ECO.aspx`
- Enterprise Architect by Sparx Systems
  `http://www.sparxsystems.com.au/`
- Magic Draw UML by NoMagic
  `http://www.magicdraw.com/`
- Papyrus UML (Open Source)
  `http://www.papyrusuml.org/scripts/home/publigen/content/templates/show.asp?P=114&L=EN&ITEMID=16`
- TOPCASED (Open Source)
  `http://www.topcased.org/`

A.3 CASL-MDL

A.3.1 Type in CASL-MDL

A type may be either *predefined type* or an *entity type* which deïňĄnes a *datatype* or a *dynamic type*. The structure of Type is given in Fig.A.5 by means of a metamodel.

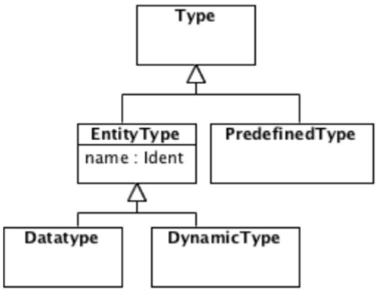


Figure A.5: Structure of Type (metamodel)

A.3.2 Datatype

A construct Datatype is used to declare new datatypes, a Datatype metamodel is presented in Fig.A.6.

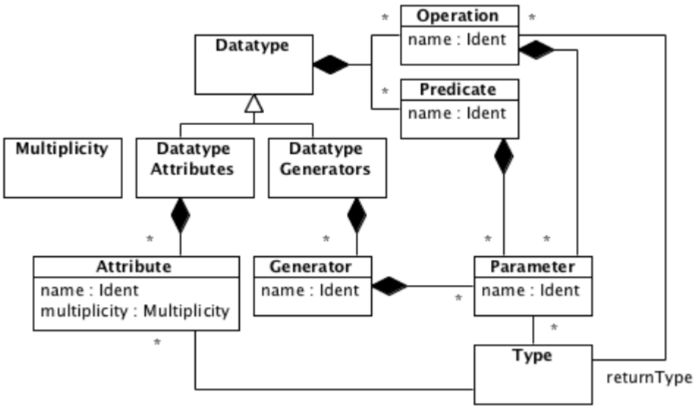


Figure A.6: Datatype structure (metamodel)

The datatypes may have *predicates* which must have at least an argument typed as the datatype itself, and *operations* that have a return type.

The structure of a datatype may be defined in two different ways, using either *generators* or *attributes*.

In case of defining the datatype values in terms of generators (see visual schematic example given in Fig.A.7): the datatype values are denoted using generators (marked by ≪gen≫). The arguments of the generators (marked by ≪pred≫) may be typed using the predefined types and the user defined datatypes and dynamic types present in the same TypeDiagram (see Sec.A.5).

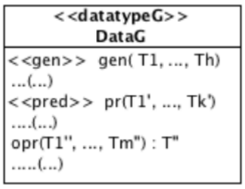

<<datatypeG>>
DataG
<<gen>> gen(T1, ..., Th)
...(...)
<<pred>> pr(T1', ..., Tk')
....(...)
opr(T1'', ..., Tm'') : T''
.....(...)

Figure A.7: Schematic datatype with generators

In case of defining the datatype values in terms of attributes (see visual schematic example given in Fig.A.8): it is similarly to the UML, an attribute attr: T of a datatype D corresponds to a CASL operation oper.attr: D → T. There is a standard generator named as the type itself having as many arguments as the attributes, but it is introduced when deïňĄning the datype by an appropriate definition.

Constraint:

The datatype used to define any notion must be defined if they are not of primitive type.

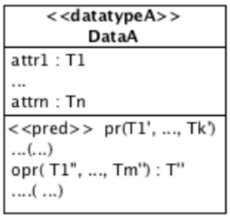


Figure A.8: Schematic datatype with attributes

A.3.3 Dynamic Type (type of dynamic system)

A dynamic systems represent any kind of dynamic entities, i.e., entities with a dynamic behaviour without making further distinctions (such as reactive, proactive, autonomous, passive behaviour, inner decomposition in subsystems), and are formally considered as *labelled transition systems.*

A *labelled transition system* (*lts*) is a triple (*State*, *Label*, $\rightarrow$), where *State* denotes the set of states, *Label* is the set of transition label, and $\rightarrow \subseteq$ *State* $\times$ *Label* $\times$ *State* is the transition relation. A triple (*s,l,s'*) $\in\rightarrow$ is said to be a *transition.*

A dynamic system is modelled by a transition tree determined by an *lts*(*State*, *Label*, $\rightarrow$) and an initial state $s_0 \in$ *State.*

A.3.3.1 Simple Dynamic Type (type of simple system)

The simple dynamic systems do not have dynamic subsystems, and the interactions of the simple systems are either of kind sending or receiving (with a naming convention !_xx and ?_yy, for sending and receiving interactions resp.) and are characterized by a name and a possibly empty list of typed parameters. These simple interactions correspond to basic acts of either sending out or of receiving something, where the something is defined by the arguments.

The visual notation for the simple dynamic types is presented in Fig.A.9.

A send act will be matched by a receive act of another simple system and vice versa, and obviously the matching pairs of interactions !_xx($v_1 \ldots v_n$) and ?_xx($v_1 \ldots v_n$). The states of simple systems are characterized by a set of typed attributes (precisely the states of the associated labelled transition system), similarly to the case of datatypes with attributes (and, as for each attribute, there is the corresponding operation). A dynamic type `DT` has

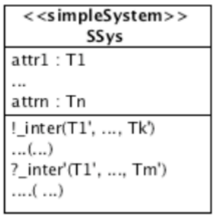


Figure A.9: A schematic Simple Dynamic Type

also an extra implicit attribute attr.id: ident_DT containing the identity of the specific considered instance; the identity values are not further detailed.

A.3.3.2 Structured Dynamic Type (type of structured system)

A structured system is characterized by its parts, or subsystems (that are in turn other simple or structured dynamic systems), and has its own elementary interactions and name.

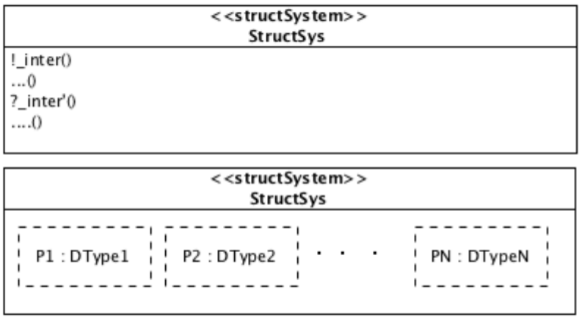


Figure A.10: A schematic Structured Dynamic Type

Fig.A.10 presents the visual syntax by the schematic structured dynamic type; its parts are depicted by the dashed boxes (in this case all of them have multiplicity one); DType1, DType2 . . . DTypeN are dynamic types (i.e., types corresponding to dynamic systems, simple or structured, defined in the same model) and P1, P2 . . . PN are the optional names of the parts. A structured dynamic type has a predefined predicate isPart checking if it has a part having a given identity.

A.3.4 Interaction Machine

An interaction machine associated with a simple dynamic type is an oriented graph, whose nodes represent the states and whose arcs represent the possible transitions of the system. Fig. shows the structures of the interaction machine.

The states are considered as interaction states corresponding to the possible stages of the activity modelled by the machine. The transitions are decorated by an interaction occurrence, a guard, and an effect. A *guard* is a boolean expression built over the attributes of the simple dynamic type and an *effect* is an action over those attributes and the free variables, if any. The form of the *effect* must be restricted to a sequence of assignments to the attributes. The transition is specified as following forms:

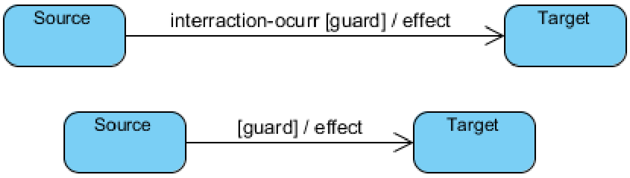


Figure A.11: **Form of a transition in Casl4Soa**

The interaction-occur may have the following forms: "$!\_inter(e_1 \ldots e_n)$", where !_inter is an elementary interaction and e_i are ground expressions built over the attributes, or "$?\_inter(X_1 \ldots X_n)$", where ?_inter is an elementary interaction and X_i are variables. A transition without interaction occurrence corresponds to some internal activity.

Constraints:

An interaction machine has only one initial state, while it may have any number , also none, of final states.

The name of the state in a interaction machine should be unique.

No transition may enter in the initial state an no transition may leave a final state.

At least a transition must leave a non final state.

A.3.5 Formula

Formula is a subset of Casl-Ltl formulas together with an extension to take interactions into account. The grammar of the formula with logic combinators is defined in Fig. A.12, where *Vi* are variable identifiers, *InterName* and *AttrName* are names of interactions and attributes respectively, and *Exp* are expressions denoting values. The logic combinators used in formulas are those of the first-order logic, together with temporal combinators (for a path formula) to address whether a property is satisfied in states of a path from a given state (*eventually, always, next*), and temporal combinators to state whether a path formula is satisfied in at least one (*sometimes*) or all paths from a state (*in_ any_ case*).

$$
\begin{array}{ll}
\mathit{Form} ::= & \mathit{Data\_Atom} \mid \neg\, \mathit{Form} \mid \mathit{Form} \;\Rightarrow\; \mathit{Form} \mid \\
 & \mathit{Form} \vee \mathit{Form} \mid \mathit{Form} \wedge \mathit{Form} \mid \\
 & \forall \mathit{Vi}.\mathit{Form} \mid \exists \mathit{Vi}.\mathit{Form} \mid \\
 & \mathit{in\_any\_case}\ \mathit{Path\_Form} \mid \\
 & \mathit{sometimes}\ \mathit{Path\_Form}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Path\_Form} ::= \mathit{Interact\_Atom} \mid \mathit{Static\_Atom} \mid \\
\quad \neg\, \mathit{Path\_Form} \mid \mathit{Path\_Form} \vee \mathit{Path\_Form} \mid \\
\quad \mathit{Path\_Form} \wedge \mathit{Path\_Form} \mid \\
\quad \mathit{Path\_Form} \;\Rightarrow\; \mathit{Path\_Form} \mid \\
\quad \forall \mathit{Vi}.\mathit{Path\_Form} \mid \exists \mathit{Vi}.\mathit{Path\_Form} \mid \\
\quad \mathit{eventually}\ \mathit{Path\_Form} \mid \\
\quad \mathit{always}\ \mathit{Path\_Form} \mid \mathit{next}\ \mathit{Path\_Form}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Interact\_Atom} ::= \mathit{InterName}(\mathit{Exp}, ..., \mathit{Exp}) \\
\mathit{Static\_Atom} ::= \mathit{AttrName} = \mathit{Exp}
\end{array}
$$

Figure A.12: **Grammar of formulas**

A.4 Profile

A *profile* in UML (Unified Modeling Language) provides a generic extension mechanism for customizing UML models for particular models for particular domains. Extension mechanisms allow refining standard semantics in strictly additive manner, so they can not contradict standard semantics.

Profiles are defined using *stereotypes*, tag definitions, and constraints that are applied to specific model elements (such as classes, attributes, operations, etc.).

A profile is a collection of such extensions that customize UML for a particular domain.

Note: It is not possible to define a standalone profile without its reference *metamodel.*

Stereotype

A *stereotype* is one type of extensibility mechanisms in the UML. They allow designers to extend the vocabulary of UML in order to create new model elements derived from existing ones, but that have specific properties that are suitable for a particular domain.

Metamodel

A *metamodel* is a special kind of model that specifies the abstract syntax of a modeling language. It can be understood as the representation of *the class of all models* expressed in that language. A *model* is a simplified representation of a certain reality. A model conforms to a language whose abstract syntax is represented by a metamodel. A metamodel is commonly represented by means of a UML class diagram (c.f., Appendix A.1.4)

List of Figures

List of Tables

Bibliography

[1] Arsanjani A., Ghosh S., Allam A., Abdollah T., Gariapathy S., and Holley K. SOMA: a method for developing service-oriented solutions. *IBM Syst. J.*, 47(3):377–396, 2008.

[2] Ali Arsanjani. Service oriented modeling and architecture. 09 Nov 2004.

[3] Egidio Astesiano, Gianna Reggio, and Filippo Ricca. Modeling business within a uml-based rigorous software development approach. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2008.

[4] L. Bocchi, A. Fantechi, L. Gonczy, and Nora. Sensoria ontology. Technical Report D1.1a, Sensoria, 2006.

[5] Roberto Bruni, Matthias Hölzl, Nora Koch, Alberto Lluch Lafuente, Philip Mayer, Ugo Montanari, Andreas Schroeder, and Martin Wirsing. A service-oriented uml profile with formal support. In *Proceedings of the 7th International Joint Conference on Service-Oriented Computing*, ICSOC-ServiceWave '09, pages 455–469, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Aida Causevic. Formal approaches to service-oriented design: From behavioral modeling to service analysis. Licentiate thesis, June 2011.

[7] Aida Causevic, Cristina Seceleanu, and Paul Pettersson. Modeling and reasoning about service behaviors and their compositions. In *Proc. of 4th Int. Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2010), LNCS 6416*, pages 82–96. Springer, 2010.

[8] Christine Choppy and Gianna Reggio. Service modelling with Casl4Soa: A well-founded approach - part 1 (service in isolation). In *Proc. Annual ACM Symposium on Applied Computing*, pages 2444–2451, 2010. Studied pages 19-28.

[9] Christine Choppy and Gianna Reggio. CASL-MDL, modelling dynamic system with a formal foundation and a UML-like notation. In *Recent Trends in Algebraic Development Techniques, Selected papers*, LNCS 7137, pages 76–97. Springer Verlag, 2012.

[10] Rational Software Corporation. Rational unified process: Best practices for software development teams. Technical report, November 2001.

[11] Dijkstra and Edsger W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[12] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML*. Addison Wesley Longman, 1999. Studied pages 689-696.

[13] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pal Krogdahl, Min Luo, and Tony Newling. *Patterns: Service-Oriented Architecture and Web Services*. April 2004.

[14] Thomas Erl. Service oriented architecture. `http://serviceorientation.com/index.php/serviceorientation/index`.

[15] Thomas Erl. *SOA Principles of Service Design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl, 2007.

[16] Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[17] Thomas Erl. *SOA Design Patterns*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2009.

[18] José Luiz Fiadeiro, Antónia Lopes, Laura Bocchi, and João Abreu. The sensoria reference modelling language. In *Results of the SENSORIA Project*, LNCS 6582, pages 61–114. Springer, 2011.

[19] Software Engineering for Service-Oriented Overlay Computers. D1.4a: Uml for service-oriented systems. `http://www.pst.ifi.lmu.de/projekte/Sensoria/del_24/D1.4.a.pdf`, 2009.

[20] Howard Foster, László Gönczy, Nora Koch, Philip Mayer, Carlo Montangero, and Dániel Varró. UML extensions for service-oriented systems. In *Results of the SENSORIA Project*, LNCS 6582, pages 35–60. Springer, 2011.

[21] S. Gorton, C. Montangero, S. Reiff-Marganiec, and L. Semini. Service-oriented computing - icsoc 2007 workshops. chapter StPowla: SOA, Policies and Workflows, pages 351–362. Springer-Verlag, Berlin, Heidelberg, 2009.

[22] Object Management Group. Model driven architecture. `http://www.omg.org/mda/index.htm`.

[23] Object Management Group. Business motivation model. `http://www.omg.org/spec/BMM/1.1/PDF/`, November 2008.

[24] Object Management Group. Business process model and notation (BPMN) 2.0. `http://www.omg.org/spec/BPMN/2.0/`, January 2011.

[25] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0*, August 2005.

[26] Object Management Group. *Unified Modeling Language: Infrastructure, version 2.0*, March 2006.

[27] Object Management Group. *Service oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services (UPMS)*, May 2012.

[28] Open Group. Service oriented architecture. `http://www3.opengroup.org/subjectareas/soa`.

[29] Open Group. What is soa. `http://www.opengroup.org/soa/source-book/soa/soa.htm`.

[30] W3C Working Group. Web services glossary. `http://www.w3.org/TR/ws-gloss/`, February 2004.

[31] Paul Harmon. Second generation business process methodologies. May 2003.

[32] Peter Herzum and Olivier Sims. *Business component factory : a comprehensive overview of component-based development for the enterprise.* J. Wiley & Sons, New York, 2000.

[33] IBM. Service oriented architecture glossary. `http://www-01.ibm.com/software/solutions/soa/glossary/index.html`.

[34] IBM. Business process execution language for web services - version 1.1. `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf`, May 2003.

[35] Nora Koch, Philip Mayer, Andreas Shroeder, and Alexander Knapp. The UML4SOA profile. Technical report, Ludwig-Maximilians-Universität München, 2010.

[36] Pal Krogdahl, Gottfried Luef, and Christoph Steindl. Service-oriented agility: Methods for successful service oriented architecture (SOA) development, part 1: Basics of SOA and agile methods. 26 Jul 2005.

[37] Fabrice Marguerie. Getting a little closer to soa. `http://madgeek.com/Articles/SOA/EN/SOA-Softly.html`.

[38] Philip Mayer, Nora Koch, and Andreas Schroeder. The UML4SOA profile. Technical report, Ludwig-Maximilians-Universitaet Muenchen, July 2009.

[39] Philip Mayer and István Ráth. The sensoria development environment. `link.springer.com/chapter/10.1007%2F978-3-642-20401-2_30`.

[40] Microsoft. Microsoft developer network library. `http://msdn.microsoft.com/library/default.aspx`.

[41] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Jolie: a java orchestration language interpreter engine. *Electron. Notes Theor. Comput. Sci.*, 181:19–33, 2007.

[42] OASIS. Reference model for service oriented architecture 1.0. `https://www.oasis-open.org/committees/download.php/19679/`.

[43] Michael P. Papazoglou and Willem Jan Van Den Heuvel. Service oriented design and development methodology. *Int. J. Web Eng. Technol.*, 2(4):412–442, July 2006.

[44] Visual Paradigm. Visual paradigm for uml 10.1 community edition. `http://www.visual-paradigm.com/download/vpuml.jsp?edition=ce`.

[45] Gianna Reggio, Christine Choppy, and E.Astesiano. CASL-LTL: A CASL extension for dynamic reactive systems version 1.0 - summary. In *Technical Report DISI-TR-03-36, DISI-Universita di Genova, Italia*, 2003.

[46] Gianna Reggio, Maurizio Leotta, Filippo Ricca, and Egidio Astesiano. Chosing the right style for modelling the business process with the UML: Complete version. Technical Report DISI-TR-12-02, 2012. DISI-University of Genova-Italy.

[47] S. Reiff-Marganiec, L. Blair, K.J. Turner, University of Stirling. Department of Computing Science, and Mathematics. *APPEL: The ACCENT Project Policy Environment/language.* Technical report (University of Stirling. Dept. of Computing Science and Mathematics). Department of Computing Science and Mathematics, University of Stirling, 2005.

[48] Mike Rosen. Where does one end and the other begin. Jan 2006.

[49] Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. REMES: A resource model for embedded systems. In *In Proc. of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. IEEE Computer Society, June 2009.

[50] David Sprott and Lawrence Wilkes. Understanding service oriented architecture. January 2004.

[51] Martin Wirsing, Matthias Hoelzl, Nora Koch, Philip Mayer, and Andreas Schroeder. Service Engineering: The Sensoria Model Driven Approach. In *Proceedings of Software Engineering Research, Management and Applications (SERA 2008)*, Prague, Czech Republic, 2008. IEEE Computer Society.

[52] Martin Wirsing and Matthias M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing.* LNCS 6582. Springer, 2011.

[53] Aliaksei Yanchuk, Alexander Ivanyukovich, and Maurizio Marchese. A lightweight formal framework for service-oriented applications design. In *ICSOC*, pages 545–551, 2005.