NTNU

Norwegian University of
Science and Technology

Coordination Patterns for Reactive Services

Urooj Fatima

Master of Telematics - Communication Networks and Networked Services (2 year)

Submission date: July 2010

Supervisor: Rolv Bræk, ITEM

Co-supervisor: Humberto Nicolás Castejón, Telenor R&I

Norwegian University of Science and Technology
Department of Telematics

Problem Description

In [1] a service engineering approach was proposed. In this approach the services of a distributed reactive system are modeled in two steps: service structure is first modeled using UML 2 collaborations (i.e. services are described as collaborations between roles); thereafter, service behavior is modeled as a choreography of sub-collaborations (i.e. sub-services) using UML 2 activity diagrams. The service models obtained in this way can then be used to synthesize role behaviors in the form of state machines. Two related problems need still to be addressed:

1. The behavior of each system component has to be designed as a composition of the roles played by the component. During this design process extra functionality may need to be added in order to coordinate such roles (e.g. in the case a component may play simultaneous roles in concurrent sessions of a service). Can this be done in a modular and systematic way?

2. In real life, it is common to improve systems by adding new functionality to their services (e.g. we may want to improve a basic call service by adding the possibility to transfer calls). Can new functionality be incrementally added to an existing service model? And can the behavior of system components be extended with such functionality in a modular way?

Both of the above problems deal with adding new/extra functionality to an existing model, although for different purposes. The student should initially focus on problem 1 and study it in detail. Based on the modeling and design of various service examples the student should extract general rules and patterns that may be applied to address such problem. If time allows, and with the experience gained from the resolution of problem 1, the student should present initial thoughts and examples of how to address the problem 2.

[1] H. N. Castejón, "Collaborations in service engineering: Modeling, Analysis and Execution", PhD thesis, Norwegian University of Science and Technology (NTNU), 2008

Assignment given: 15. February 2010
Supervisor: Rolv Bræk, ITEM

Abstract

In our everyday life we come across countless *reactive systems*. These are the systems that engage in stimulus-response behaviour. The development of distributed reactive systems is quite complex. Being able to rapidly develop and incrementally deploy new services, while avoiding interactions with existing ones, is a challenging task.

In this thesis, service examples of a distributed reactive system are modeled using the service engineering approach proposed by Humberto Nicolás Castejón in his PhD thesis; *Collaborations in Service Engineering: Modeling, Analysis and Execution.* First, services are modeled as collaborations between roles. Thereafter, the behaviour of each system component is designed as a composition of the roles it plays in the different services.

In many cases, a component may be requested to participate simultaneously in several occurrences of the same service, or of different services (e.g. a UserAgent representing a user in a telecommunication system may be requested to participate in several phone calls simultaneously). To address such problems, extra coordination functionality is introduced in this thesis to coordinate the roles or role instances that a system component may play at a given point in time. For this, another role is defined, which is external to the service roles and serves to coordinate role binding. This role is named as 'Controller'. The Controller is designed to keep track of the resource status, assign the resource if it is free, and if it is not then respond to the service invitation requests according to the preferences of the actors that receive them.

Depending upon how the Controller performs the coordination functionality, some coordination patterns have been investigated. Apart from modeling the service from scratch and investigating the coordination patterns for it, it has been explored in this thesis how this coordination functionality can be added into an existing service model in a modular way. Some solutions are discussed but these are the initial thoughts which can be further explored in depth. The general structure of the coordination patterns has also been identified, which further strengthened the generality of the coordination patterns.

Preface

This master's thesis is written at the Department of Telematics in the Norwegian University of Science and Technology (NTNU) during the Spring Semester, 2010.

First of all, I would like to thank my supervisor Humberto Nicolás Castejón and Professor Rolv Bræk for their timely advices, continuous support and encouragement. From the point of origin to the point of completion of my thesis, I never found a single moment when my questions were not answered, not even on the weekends. They are blessed with very kind personalities. Special thanks to them for keeping my spirits up in the difficult moments and guiding me through the tricky tasks. I am also thankful to them for providing me valuable comments and suggestions during report writing. I have learnt a lot under their inspiring guidance.

I am also grateful to the Higher Education Commission of Pakistan (HEC) for funding my studies in Norway.

I am thankful to my Mother, Salma Nasim and Sister, Nida Batool for being there always for me to keep my moral high, regardless of being thousands of miles away from me. I would also like to thank my friends who supported me and encouraged me whenever I felt alone.

I hope that the reader will find my work interesting, and that my contributions in this thesis can play a part for the further research.

_____

Urooj Fatima
Trondheim, July 2010

Contents

List of figures

Abbreviations

AR-P	**A**ssign and **R**eject – by **P**olling
AR-SU	**A**ssign and **R**eject – by **S**tatus **U**pdate
AW-SU	**A**ssign and **W**ait – by **S**tatus **U**pdate
SDL	**S**pecification and **D**escription **L**anguage
UML	**U**nified **M**odeling **L**anguage

Chapter 1: Introduction

This chapter presents the motivation for this thesis and the problem to be solved. An outline of the thesis is also provided.

1.1 Motivation

In our everyday life we come across countless *reactive systems*. They are everywhere, from digital watches, microwave ovens and robots to telecommunication systems, information systems and complex industrial plants. One can observe that common to all of these is "the notion of system responding or reacting to external stimuli" [HP85]. *Reactive systems* are "the systems that engage in stimulus-response behaviour" [Wie03]. The concept of *reactive systems* encompasses many other systems, including real-time systems, embedded and control systems. They all share a fundamental characteristic: once they are switched on, they enforce a certain desirable behaviour on their environment [Wie03].

On the other hand, *transformational systems* accept inputs, perform transformations on them and produce outputs, and may prompt the user from time to time to provide extra information [HP85]. They do not leave the system in a significant state after performing terminating computations. Unlike transformational systems, *reactive systems* continuously interact with their environment and are repeatedly prompted by the environment and must be designed from the structure of the environment.

When reactive systems are distributed, "they consist of separated autonomous components that may take independent initiatives, operate concurrently and interact with each other and the environment in order to provide services" [Cas08]. Being able to rapidly develop and incrementally deploy new services, while avoiding interactions with existing ones, is a challenging task. These systems often follow a peer-to-peer structure. During the construction of the design models of such systems, the possibility of having simultaneous occurrences of the services they provide, and the need to coordinate them,

have to be addressed. Thus, the challenge for the designer of such *distributed reactive systems* is to avoid undesired interactions between the system components and to ensure correct system behaviour. It is a difficult task, but interesting as well.

1.2 Problem to be solved

A service engineering approach is proposed in [Cas08] for modeling services of a distributed reactive system. The behaviour of each system component is designed as a composition of the roles played by that system component in different services. During this design process extra coordination functionality may need to be added between different roles or role instances, incase a component may participate in multiple concurrent occurrences of the same service or of different services. The first goal of this thesis is to investigate patterns for coordination mechanisms. This will be done by modeling service examples and extracting general patterns. The second goal of this thesis is to present initial thoughts for improvement of systems by adding new functionality to their services. Both of these goals deal with adding new/extra functionality to an existing model, although for different purposes.

1.3 Report Outline

This thesis is structured as follows:

- **"Chapter 1: Introduction"** presents the motivation behind this thesis. It gives an overview of the report and the problem to be solved.

- **"Chapter 2: Background"** describes the theoretical background for the work done in this thesis.

- **"Chapter 3: Methodology for Service Development"** narrows down the problem to a set of tasks, and explains the system engineering approach followed by [Cas08] for the development of reactive systems, which this thesis is based on.

- **"Chapter 4: Coordination Patterns"** presents patterns for coordination of the service roles played by a system component.

- **"Chapter 5: Applying Coordination Patterns to an Existing Service Model"** presents some initial thoughts on adding new/extra functionality to an existing service model.

- **"Chapter 6: Discussion and Conclusion"** concludes the thesis by summing up the results, discussing them and proposing future work.

Chapter 2: Background

This chapter introduces the theoretical background regarding the concepts which are relevant to the work done in this thesis. It begins by describing the notion of services and components. Then it explains the service, design models and collaboration-oriented approach recognized and followed by [Cas08]. It highlights shortly the area of our concern in design process. In the end, it discusses design patterns in general and their significance which is obviously related to our problem.

2.1 Services and Components

The concepts; service and component, are interlinked. Therefore, we will discuss them in an interleaving fashion in this section.

The concept of "service" is used widely in our daily lives. The literature provides many informal definitions of the term "service" pertaining to different domains. We will have a flavour of some of them as follows:

- "a service is a meaningful set of capabilities provided by or over a network to its different players, including end users, network providers, and service providers" [MTJ93].
- "a service is the functionality an object provides" [CV93].
- "a (software) service is a set of functions provided by a (server) software or system to a client software or system, usually accessible through an application programming interface" [BKM07].
- "a (telecommunication) service is offered by an administration to its customers in order to satisfy a specific telecommunication requirement"[KK98].

Service is also identified as a set of features; where feature is defined as *"an incremental unit of functionality"* [BDC+89, Zav01]. According to [KK98], telecommunication services can be divided into two major groups; *bearer services* and *teleservices*. A *Bearer*

service provides the capability for the transmission of signals between user network interfaces. A *teleservice*[1] provides the complete capability, including terminal equipment functions, for communication between users according to protocols established by agreement between administrations.

By having a glance at these definitions of service, we can deduce that a single entrenched definition of service does not exist. [KBH09] uses a general definition of service which captures most common uses of the term "service". The definition is:

"*A service is an identified functionality aiming to establish some effects among collaborating entities*" [KBH09].

[KBH09] further elaborates the definition of service by characterizing the service into following properties:

- Services are functionalities; they are behaviors performed by entities.
- Services imply collaborations; it makes no sense to talk about a service unless at least two entities collaborate.
- Service behavior is cross-cutting; it involves coordination of two or more entity behaviors to fulfill a certain task.
- Service behavior is partial; it is to be composed with all the other services provided by the system to obtain a complete behaviour of the system.

We talked about entities that collaborate with each other to fulfill a specific task (i.e. the goal of the service). These entities are the "(system) components". Therefore, a component may participate in different services. So generally, the behaviour of services is composed from partial component behaviours, while component behaviour is composed from partial service behaviours [KBH09, CBB07]. This results in two axes of decomposition as depicted in Figure 2.1. This definition binds services to system components but there is not a one-to-one relationship between services and system components [Cas08]. As recognized by [Cas08], separation between services and

[1] In this thesis, we will use the general term "service" in place of *teleservice*.

components can be obtained by using the concept of *service role* which is defined as *the part that a system component plays in a service*. Therefore, the final definition of service which we will use in this thesis is:

"A service is an identified functionality with value for the service users, which is provided in a collaboration among service roles played by system components and/or service users" [Cas08].

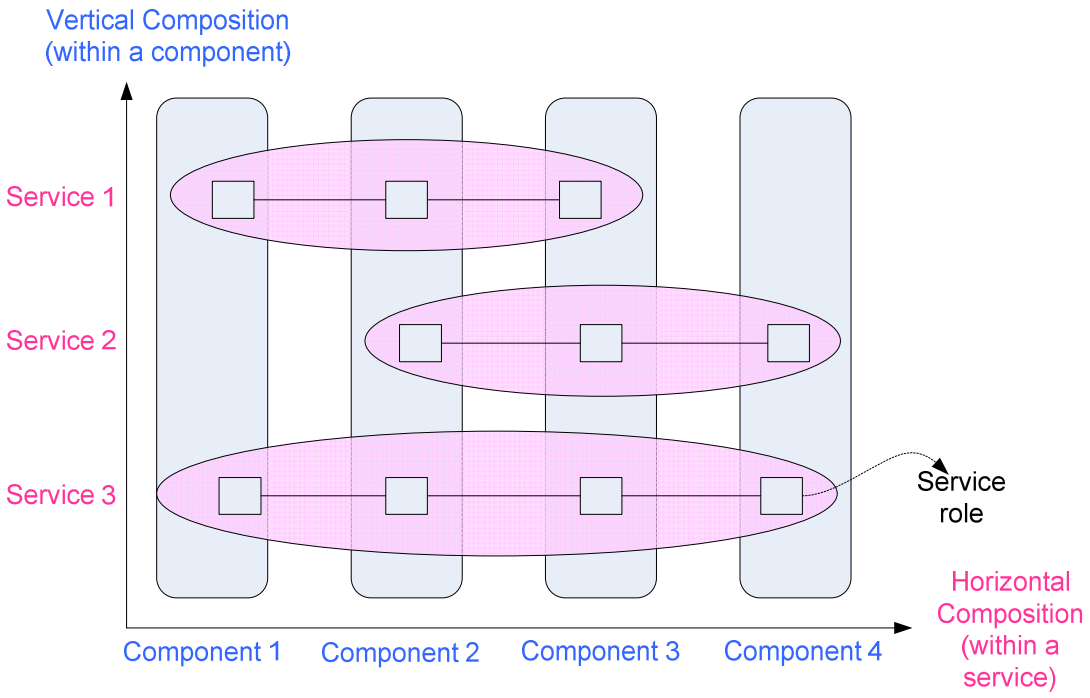


Figure 2.1: Services as collaboration among service roles played by system components [Cas08, KBH09]

This definition enables us to specify services independent of particular system designs or implementations. Moreover, in this definition service is emphasized as a functionality which is provided by the collaboration among service roles.

2.2 Service and Design Models

We discussed in section 2.1 about the two axes of decomposition depicted in figure 2.1. [KBH09] named these axes as *collaborative* axis (decomposing the system functionality into services and focusing on collaborations) and *component* axis (decomposing the

system functionality into components, defining complete behaviour of each). Because the component axis defines systems and system behaviours completely, traditionally, this axis was emphasized more. Moreover, component behaviour can be modeled as communicating state machines (as supported by SDL and UML) which define reactive behaviour of components in a precise and human-understandable manner, can be automatically analyzed and can serve as input for automatic code generation [Cas08, KBH09]. But, as discussed in section 2.1; there is not a one-to-one relationship between components and services. Consequently, in order to understand how services work, the joint behaviour of several components must be considered. Modeling reactive systems and describing complete component behaviour from end-user requirements (which are usually not given from the point of view of individual components) is challenging [Cas08]. Therefore, there is a need to understand collaborative behaviour of components and to model services independent of particular component designs. For this, a service-oriented modeling approach is given by [Cas08] which is shown in figure 2.2. In this approach, service models describing services explicitly are first created. Thereafter, they are used for the synthesis of the behaviour of the individual system components.

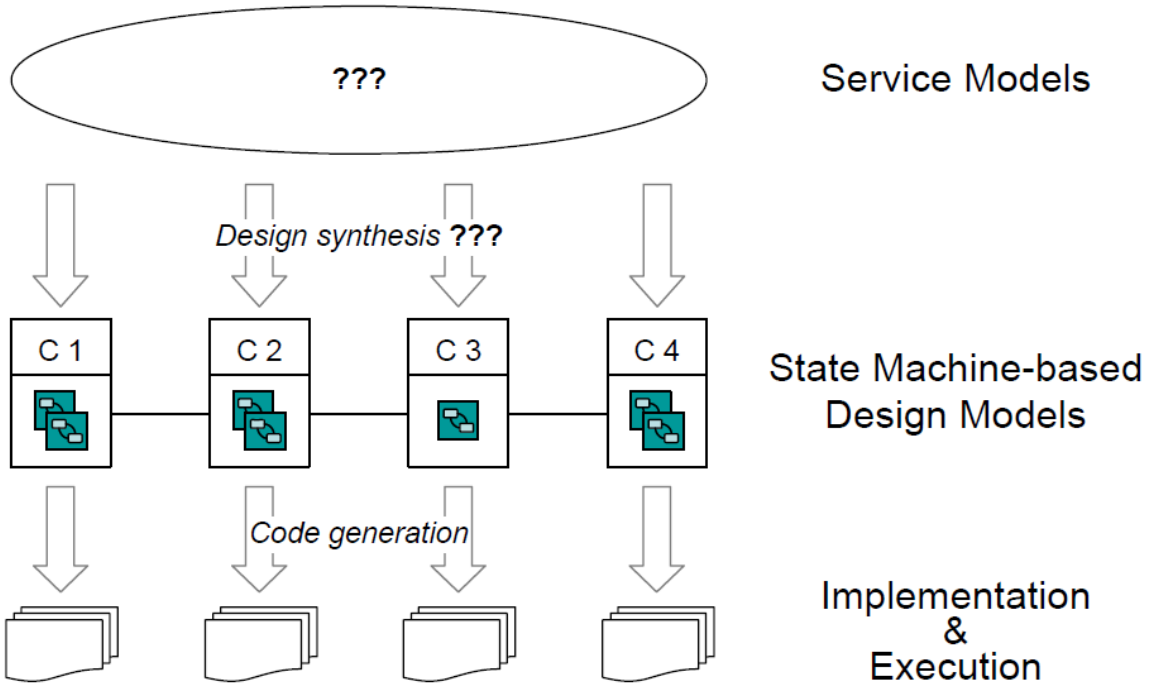


Figure 2.2: Service-oriented development: Service models are first created and then used to synthesize behaviour of individual system components [Cas08]

Interaction diagrams, for instance UML sequence diagram [OMG09], are a common solution to express collaborative behaviour using messages which are exchanged between components. But this solution is applicable for limited scenarios only and contains other drawbacks mentioned in [CBB07]. Therefore, interaction diagram may not be a good choice to model services completely. UML 2.0 collaborations[2] and activity diagrams are utilized by [Cas08] as useful mechanisms to model service behaviours more completely. Let us have a brief introduction of what these useful mechanisms are.

2.2.1 UML 2.0 Collaborations

[OMG09] defines a collaboration as "*a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality*". This definition explains the reason why [Cas08, KBH09 etc] found UML collaborations as a promising candidate for service modeling i.e. UML collaborations successfully model the notion of service described in section 2.1. The roles represent the partial objects that interact with each other to achieve a joint task. UML 2.0 collaborations are structured classifiers and can have any kind of behavioral descriptions associated [OMG09]. Figure 2.3 depicts the graphical notation for the UML 2.0 collaboration. Associations among roles are represented by 'connectors'. These connectors specify the communication paths that must exist between the participating instances. Relationships among roles and connectors inside a collaboration are meaningful in that context only.

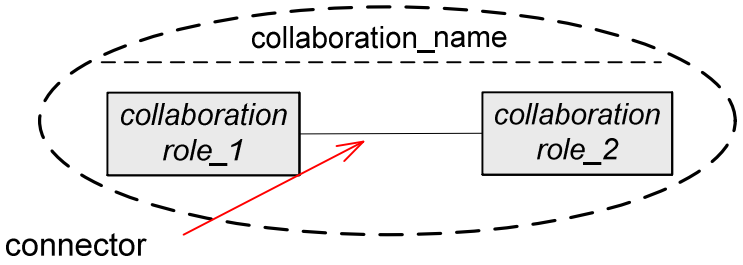


Figure 2.3: Graphical notation for UML 2.0 collaboration: Connector represents association among collaboration roles

[2] UML 2.0 collaborations should not be confused with UML 1.x collaboration diagrams, which are now called communication diagrams.

A *collaboration use* can appear within the definition of a larger collaboration. In other words, a collaboration can have other smaller collaborations. In this context, the roles of the smaller (sub-)collaboration are bound to the roles of the larger (containing) collaboration by means of a *collaboration use* [RJB05, Cas08]. This topic is discussed in detail in the chapter 3 (section 3.2.1). Thus, UML 2.0 collaborations support the concept of service composition as well. Moreover, they are particularly useful as a means for capturing standard design patterns [OMG09].

The structural decomposition of collaborations may result in elementary collaborations (i.e. collaborations that are not further decomposed into sub-collaborations). These collaborations are simple enough to be defined by UML sequence diagrams [Cas08]. To define the global execution ordering of the sub-collaborations, [Cas08] makes use of a 'choreography graph'. For this, UML 2.0 activity diagrams are utilized (with some extensions in notation). We will come to this again in chapter 3.

2.2.2 Collaboration-Oriented Development

A collaboration-oriented approach has been proposed by [Cas08] where the main structuring units are collaborations, for the reasons discussed in section 2.2.1. The key elements of this approach are depicted in figure 2.4 and explained below:

- *Service models are used to separately specify the global behavior of services. UML collaborations are used to provide a structural framework for these models. Sequence diagrams are used to describe the behavior of elementary collaboration and choreography graphs for specifying the global behavior of composite service collaborations.*
- *Design models describe the system structure, as well as the complete local behavior of each system component type. Asynchronously communicating state machines are used at this level.*
- *Implementations consist of executable code that is automatically generated from the design models.*
- *Execution platforms are the systems where software processes are executed to provide services.*

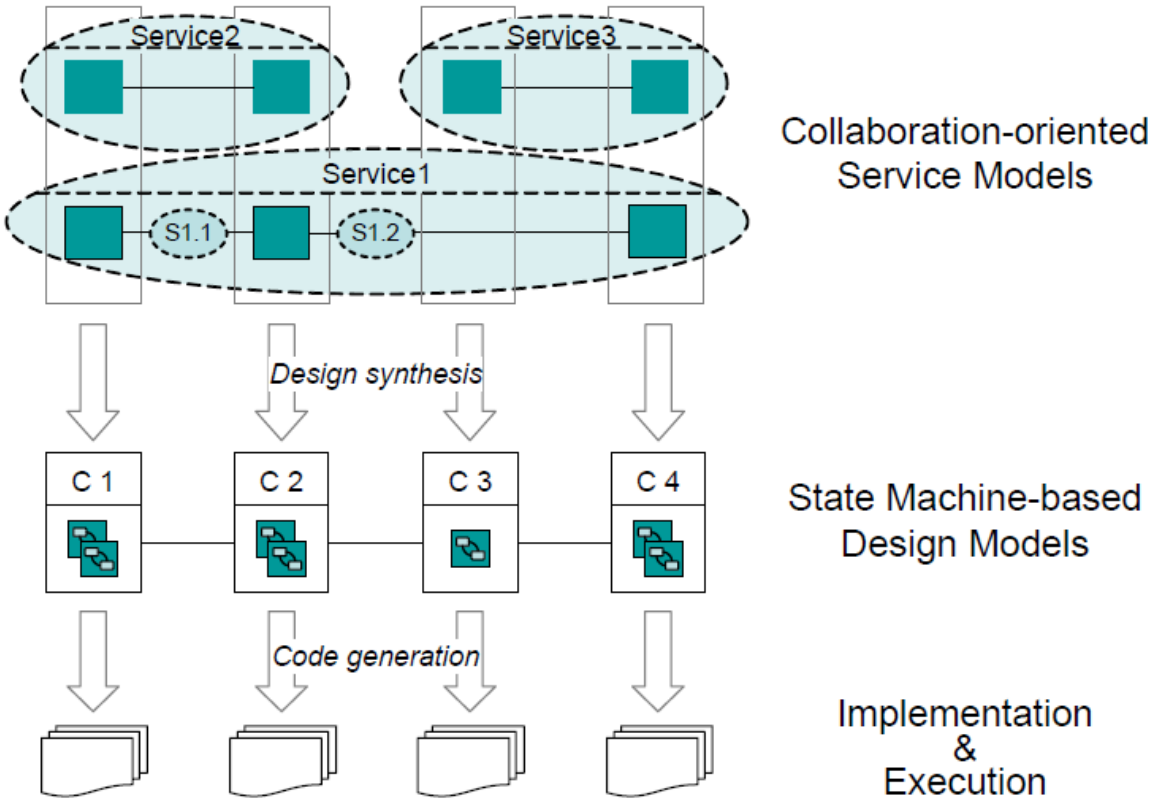


Figure 2.4: Collaboration-Oriented Development [Cas08]

2.3 What are Design Patterns?

During the construction of a design model we have to deal with the possibility of having multiple concurrent occurrences of a service running in the system. At this point, there is a need for coordination mechanism between different role instances. In this thesis, some coordination patterns are identified. Let us have a glance at what design patterns generally are and why we need them.

Design patterns were first described for civil engineering. In that context, Christopher Alexander says: *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* [GHJ+94]. Although, Christopher Alexander illustrates this concept with reference to civil engineering, it is still applicable for object-oriented design patterns.

[GHJ+94] defines design patterns for object-oriented systems; "*A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems*".

The main purpose of defining such patterns is to make designs more flexible, elegant, and ultimately reusable. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them [GHJ+94]. Nevertheless, design patterns help designers to choose among design alternatives that best suites their system requirements.

We have discussed design patterns in this section to learn about the advantages and need of patterns. In this thesis, we have not followed any particular standard schema for defining our coordination patterns. We will learn in the next chapters that there are more powerful mechanisms for reusability as compared to patterns (for example; inheritance).

Chapter 3: Methodology for Service Development

This chapter discusses the 'problem description' and narrows down the problem to a set of tasks. It explains the system engineering approach followed by [Cas08] for the development of reactive systems, which is the basis of this thesis. The approach is explained with the help of a service example, which will be used as running example throughout the rest of the thesis.

3.1 Problem Description

In [Cas08] a service engineering approach was proposed. In this approach the services of a distributed reactive system are modeled in two steps: service structure is first modeled using UML 2 collaborations (i.e. services are described as collaborations between roles); thereafter, service behavior is modeled as a choreography of sub-collaborations (i.e. sub-services) using UML 2 activity diagrams. The service models obtained in this way can then be used to synthesize role behaviors in the form of state machines. At that point, two related problems need to be addressed:

1. The behavior of each system component has to be designed as a composition of the roles played by the component. During this design process extra functionality may need to be added in order to coordinate such roles (e.g. in the case a component may play simultaneous roles in concurrent sessions of a service). *Can this be done in a modular and systematic way?*

2. In real life, it is common to improve systems by adding new functionality to their services (e.g. we may want to improve a basic call service by adding the possibility to transfer calls). *Can new functionality be incrementally added to an existing service model? And can the behavior of system components be extended with such functionality in a modular way?*

3.2 The System Engineering Approach Followed

In the proposed modeling approach of [Cas08], the behaviour of a system is first decomposed in terms of the services that the system provides. Figure 3.1 illustrates this service engineering approach[3]. Let us first have a brief overview of the five main activities involved in this iterative approach.

1. ***Service modeling:*** *A service model is created for each individual service to be provided by the system under development. Each service is modeled as a UML collaboration defining the structure of roles needed for the service, and its decomposition into elementary collaborations. The complete behaviour of elementary collaborations is specified with sequence diagrams, while the global behaviour of the service collaboration is described with a choreography graph describing the execution ordering of its sub-collaborations.*

2. ***Realizability analysis:*** *Each service model is analyzed in search of realizability problems. The aim is to ensure that the service model does not imply behaviors that are not explicitly specified, but that may arise in the design model.*

3. ***Service role synthesis:*** *For each service model, the local behaviours of its service roles are automatically synthesized in the form of state machines. The choreography graph and sequence diagrams are used as input for the synthesis.*

4. ***System composition:*** *The system structure is specified in terms of system components (with type and multiplicity) and their relationships. The complete behaviour of each system component type is designed as a composition of the service roles that it may play. To determine the correct way of coordinating the role behaviours, an analysis of potential interactions is necessary (see next activity).*

[3] The contents of this service engineering approach are taken as extracts from [Cas08]. For more details, the reader is referred to [Cas08].

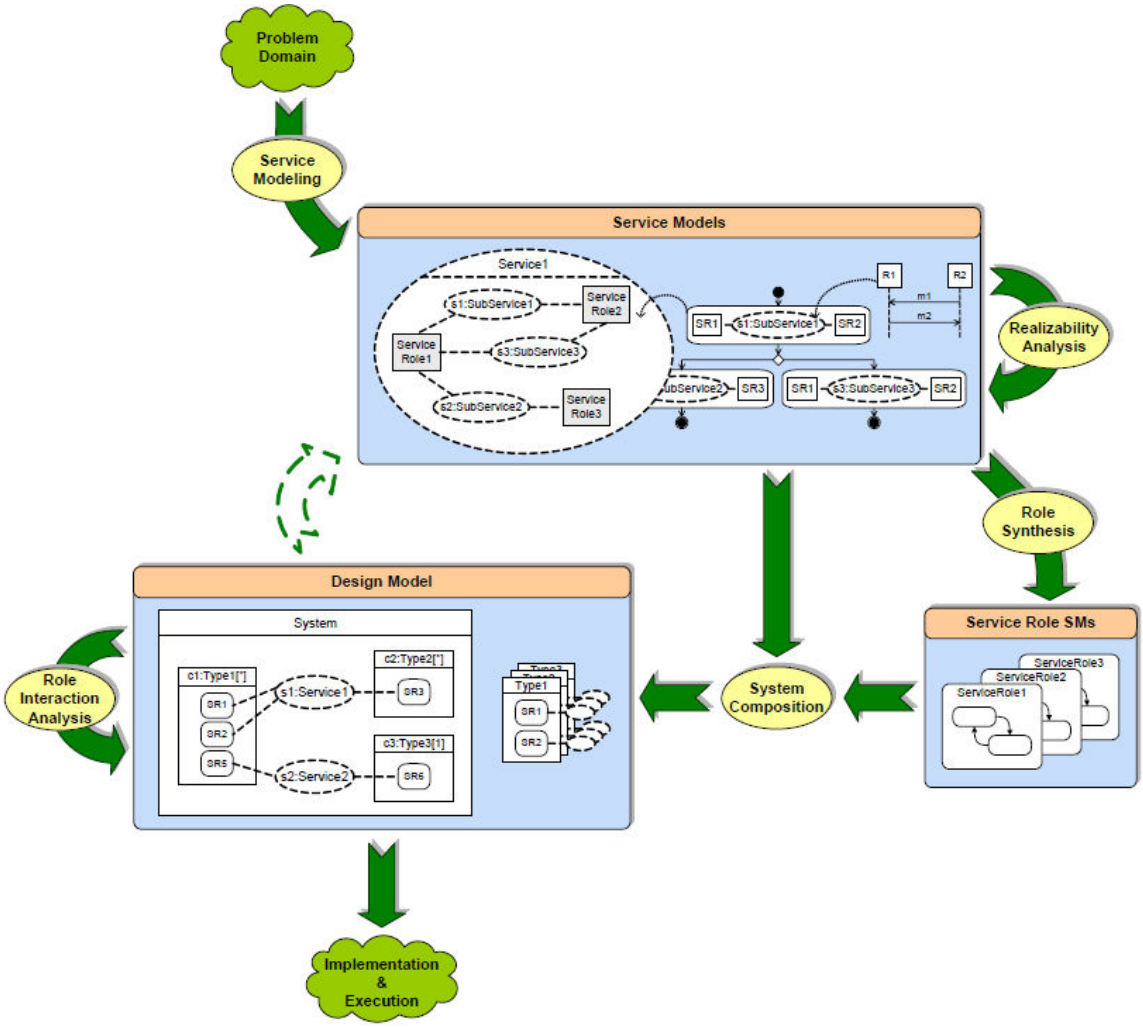


Figure 3.1: Service Engineering Approach [Cas08]

5. ***Role interactions analysis:*** *A system component may simultaneously participate in different service collaborations, as well as in several occurrences of the same service collaboration. We analyze whether undesired interactions may arise between the roles that the system component may play in simultaneous service collaborations. The results of this analysis will dictate how role behaviors should be coordinated and composed into system component behaviors (see previous activity).*

   The final goal of this activity and the previous one is to design system components so that they can play appropriate roles in each service collaboration they participate in, without undesired interactions with other running roles. This can

be seen as a problem of dynamic role binding, and policies may be defined to govern such role binding.

The focus of our work is to deal with the last two activities discussed above (see chapter 4). In this chapter, we will discuss the processes of service modeling and system composition.

3.2.1 Service Modeling

In this section, we introduce and model a *SimpleChat* service, which will be used as a running example throughout the rest of the thesis.

Example Service: Simple Chat

SimpleChat is a kind of Instant Messaging (IM) service. According to Wikipedia, "*Instant Messaging is a form of real-time direct text-based communication between two or more people using personal computers or other devices, along with shared software clients".* To keep the example simple and easy to understand, only two communicating entities are considered (a Chatter and a Chattee) i.e. the functionality of conferencing between many people is omitted. This is why we said in the beginning that it is a *kind of* Instant Messaging.

The specification of a service collaboration can be divided into five steps:

1. Identification of the roles needed to provide the service.
2. Identification of the sub-collaborations in which the service roles may engage.
3. Structural composition of the sub-collaborations identified in the previous step.
4. Description of the global service behavior by specifying the order in which the sub-collaborations should be executed.
5. Description of the behavior of each sub-collaboration.

Each of these steps is explained next.

Identification of service roles

We have to identify the service roles needed to provide the service. This is related to the problem domain and the logical architecture of the service execution environment. Each role specifies the properties and behaviour that a component should have in order to participate in a single occurrence of the service under specification.

At the early stages of service modeling it makes little sense to identify the implementation level objects that will participate in providing the service. One should rather focus on identifying actors representing domain entities such as users involved in the service (e.g. a UserAgent representing a user). Moreover, one should focus on identifying the properties and behaviour that those actors shall have, and specify these as roles, which may later be bound to system components in different ways [CBB07].

In our *SimpleChat* service, we can identify two roles:

- *Chatter*: The Chatter role is the one that can initiate a chat session.
- *Chattee*: The Chattee role is the one that receives the chat invitation.

A component (e.g. UserAgent) playing the Chatter role can initiate chat sessions, a component playing the Chattee role can receive chat invitations, and a component playing both roles can initiate sessions and receive invitations

Identification of sub-collaborations

It is discussed earlier in section 2.2.1 that decomposing a service collaboration results in more manageable sub-collaborations whose behaviour can be completely described with sequence diagrams. This decomposition can be achieved by thinking of the global phases that the service goes through.

Several phases are involved in the *SimpleChat* service. They can be described as follows:

a) *Chat Invite*[4]: Chatter will send a chat invitation request to the actor playing the role of Chattee.

[4] Here it is assumed that the Chattee exists and can be addressed in the chat invite i.e. the role is statically bound to the role actor.

b) *Chat Session*: When the actor playing the role of Chattee accepts the invitation, the actor playing the Chatter role initiates a chat session (between the Chatter and Chattee) where text messages can be exchanged.

c) *Disconnect*: The chat session will be ended if either the Chatter or the Chattee disconnects.

These phases are identified as separate sub-collaborations associated with the interface between Chatter and Chattee (see figure 3.2). Their behaviour can be described by sequence diagrams. For example, figure 3.2 shows the *ChatInvite* sub-collaboration and figure 3.4 shows its sequence diagram.

Definition of collaboration structure

The service is structurally modeled as a UML collaboration, describing the structure of roles taking part in the sub-collaborations identified in the previous step. Each sub-collaboration is represented as a collaboration use and its sub-roles are bound to the composite-roles of the main service collaboration i.e. *SimpleChat*. For example, in Figure 3.2 the sub-role 'cr' from *ChatSession* is bound to the composite-role Chatter of *SimpeChat*. Figure 3.2 depicts the collaboration structure for *SimpleChat* clarifying the initiating roles (filled circles) and terminating roles (filled squares). Note that this notation for initiating and terminating roles is not standard UML.

Collaboration choreography construction

At this point, we know the sub-collaborations in which service roles must participate in order to provide the service. We do not know the order in which the sub-collaborations of *SimpleChat* should be executed, so that their global, joint behaviour matches the intended behaviour for the service collaboration. This global behaviour of *SimpleChat* can be defined by specifying the overall ordering of its sub-collaborations – the so-called *choreography.* The choreography of the *SimpleChat* collaboration is shown in figure 3.3.

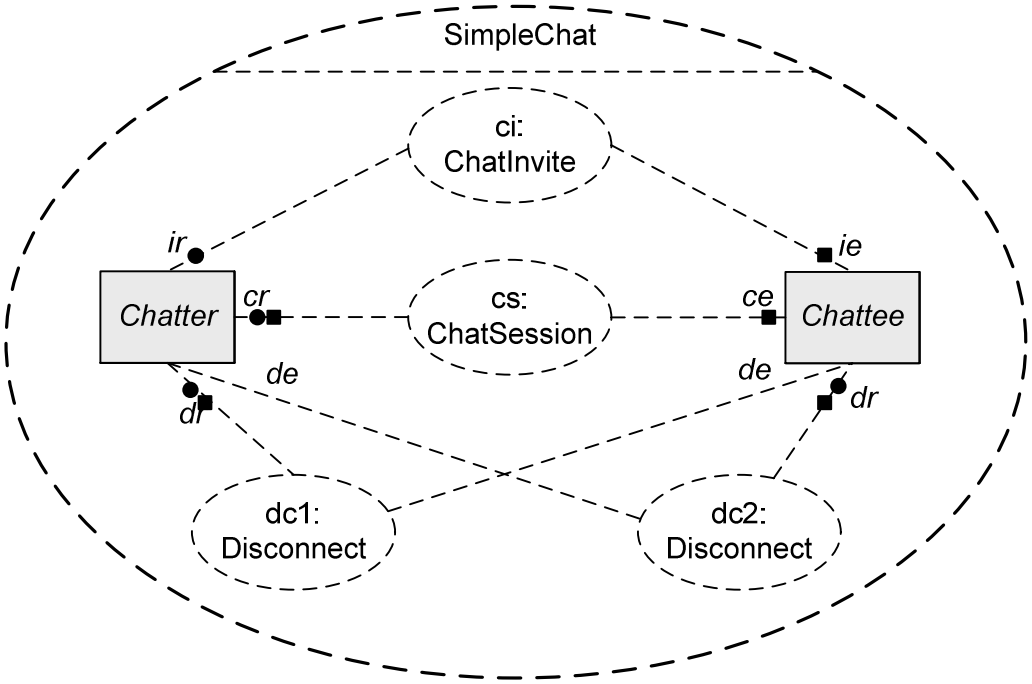


Figure 3.2: a *SimpleChat* collaboration

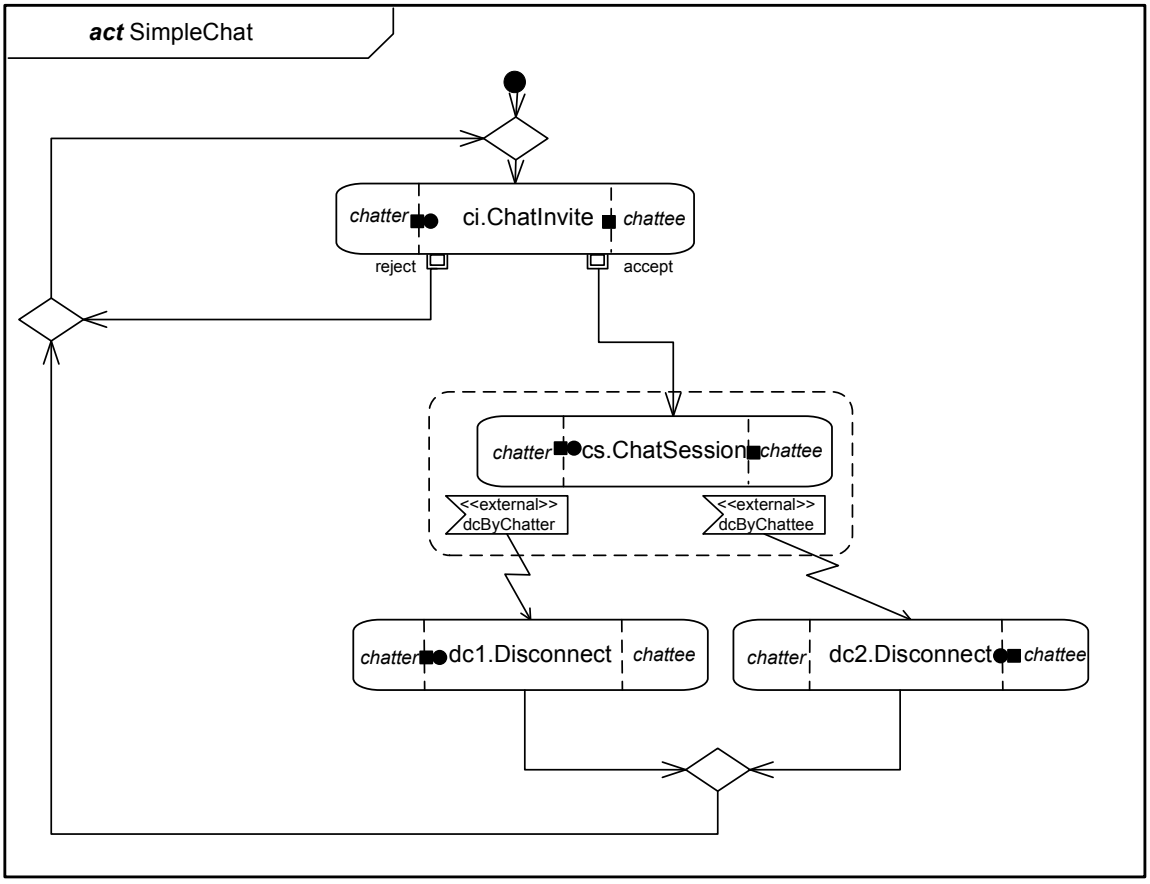


Figure 3.3: Choreography for the *SimpleChat* Collaboration

UML activity diagrams are utilized by [Cas08] for choreography description. The use of the UML activity diagrams is syntactically correct, but the semantics deviates from the standard in the following points [CBB07]:

- *In a collaboration with several initiating roles, the different initiating roles may start the execution of their part of the collaboration independently of one another, and therefore at different times. Similarly, the terminating sub-actions of a collaboration may be executed at different times.*
- *Control flow edges between different activities have the meaning of weak sequencing[5] (unless explicitly specified as strong sequence[6]).*

In choreography, it is important to show participation of roles in the collaborations and whether they are initiating or termination roles. [Cas08, CBB07] utilized the UML concept of *Partition* that may be used to indicate parts that participate in activities. In collaborations the parts are roles and thus [Cas08, CBB07] used partitions to represent roles.

Specification of elementary collaboration behaviour

One needs to describe the behaviour of each of the sub-collaborations. Some sub-collaborations may be composed of other smaller sub-collaborations. In that case, their behaviour would also be given by choreography. UML 2.0 sequence diagrams are proposed by [Cas08] to specify the complete behaviour of elementary sub-collaborations, where each role in the collaboration will be represented as a lifeline. Continuations may be used to identify states in the collaboration or in the role behaviors. These could then be used to relate the sequence diagram behavior with the pins of activities in the choreography graph (see [Cas08] for details). The sequence diagram for *ChatInvite* sub-collaboration of *SimpleChat* is shown in figure 3.4.

[5] Weak sequencing of two sub-collaborations C1 and C2, basically requires each composite role in C2 to be completely finished with C1 before it may initiate participation in C2 [CBB07].

[6] Strong sequencing between two sub-collaborations C1 and C2, requires C1 to be completely finished, for all its roles, before C2 can be initiated [CBB07].

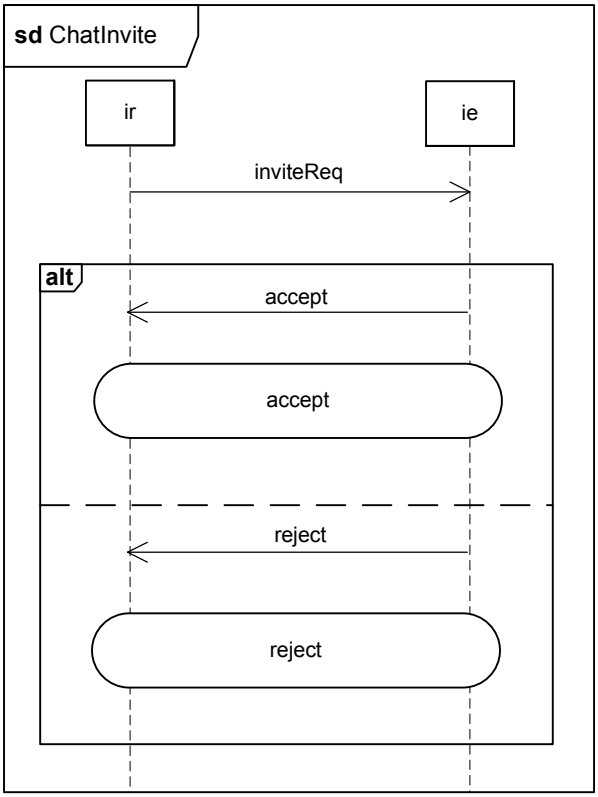


Figure 3.4: UML 2.0 sequence diagram for *ChatInvite* sub-collaboration of *SimpleChat*

3.2.2 System Composition

In service modeling, the focus was on specification of the services offered by the system under development. In the system composition phase the focus is on designing the complete behaviour of each of the system components. In order to design the behaviour of a system component it is necessary to know:

- *the service collaborations in which the component participates, and the role(s) it plays in them;*
- *whether the component may simultaneously participate in multiple occurrences of a given service collaboration; and*
- *whether the roles played by the component may interact in unexpected ways if executed concurrently.*

In the following we will discuss the first issue. The remaining two issues will be discussed in the next chapter (chapter 4), which needs to be addressed as part of this thesis.

System diagram

The system diagram is essentially a UML structured class with inner parts, where the structured class represents the system itself, and the internal parts represent the system components, with type and multiplicity. The services provided by the system are represented by collaboration uses defining appropriate binding of service roles to system components. Figure 3.5 shows the system diagram for the *InstantMessagingSystem* consisting of multiple UserAgents that can behave both as Chatters and as Chattees to provide a *SimpleChat* service.

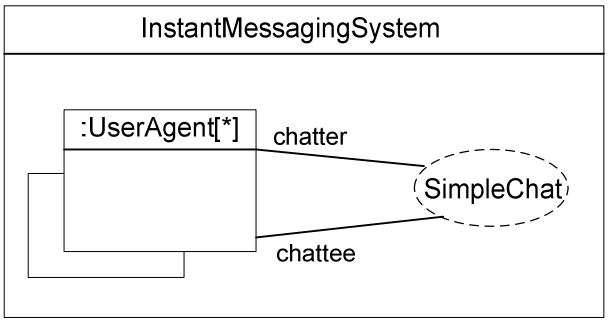


Figure 3.5: System diagram for the *InstantMessagingSystem*

Chapter 4: Coordination Patterns

In this chapter we present patterns for coordination of the service roles played by a system component, which is the first task of this thesis. The service example (*SimpleChat*) introduced and modeled in chapter 3, will be used in this chapter as a tool to identify and specify the coordination patterns.

4.1 Simultaneous Collaboration Occurrences and Unexpected role Interactions

During service modeling phase (chapter 3), the focus was on one isolated occurrence of the service. But during the design process, we have to deal with the possibility of having multiple concurrent occurrences of the service running in the system and the roles played by the system component may interact in unexpected ways if executed concurrently.

As depicted in figure 3.5 (chapter 3), in the final system (*InstantMessagingSystem*) many service occurrences may coexist. Sometimes, a *UserAgent* may be requested to simultaneously participate in several occurrences of the *SimpleChat* collaboration. For example, a *UserAgent* may be requested to join an occurrence of *SimpleChat* when it is already participating in another occurrence of that collaboration. Therefore, extra coordination functionality must be added to coordinate the role behaviours.

4.1.1 Extra Coordination Functionality

Extra coordination functionality can be modeled by defining another role for each actor (*UserAgent*). This role is named as *Controller* and it is external to the service roles (i.e. Chatter and Chattee roles in *SimpleChat* example). Depending upon the service requirements, the Controller role can perform the coordination functionality for different *UserAgents* (i.e. it behaves as a central controlling entity for different agents). This case will be discussed in detail in section 4.3 with the help of a service example. However, the

basic functionality of the Controller will remain the same for both the cases. In this section, the functionality of Controller is described in detail by extending our *SimpleChat* service example.

In our example service, *Controller* will act as a coordinator between the Chatter and Chattee roles and their respective agents/actors. Since Controller is coordinating among the roles played by one actor or different actors, it must have some basic knowledge about the actor and its preferences, for instance, whether an actor can have one instance of a particular role or many. Consider the example of *SimpleChat*. It is earlier discussed in section 3.2.1 that Chatter sends the chat invitation directly to the actor playing the role of Chattee. Now, the Chatter role will send the invitation request to the Controller of the actor playing the role of Chattee. Controller is responsible for keeping track of the status of its UserAgent playing the role of Chattee and to respond the Chatter whether it can be connected to the Chattee or not[7]. If the Chattee can be connected to the Chatter i.e. when Chattee is available, Controller will accept the invitation by the Chatter and as a result the Chatter will initiate the chat session. Hence, the responsibility of the Controller is to create a possible communication session between the two parties (roles).

The responsibility of the Controller can be seen as:

- To keep the resource status of the UserAgent.
    - By Polling
    - By Status Update
- Busy handling
    - By Reject
    - By Wait (queuing)

Depending upon how the Controller fulfills its responsibilities, some coordination patterns are proposed.

1. **A**ssign and **R**eject - by **P**olling (AR-P)
2. **A**ssign and **R**eject - by **S**tatus **U**pdate (AR-SU)
3. **A**ssign and **W**ait - by **S**tatus **U**pdate (AW-SU)

[7] It is assumed here that an instance of Chattee role is always running (statically bound to the role actor), so we need to know whether it is busy or not.

The naming convention followed for coordination patterns is explained below:

- First part describes how the Controller responds to an invitation i.e. *Assign* the resource (Chattee) to the Chatter when Chattee is available and *Reject* the invitation or putting into *Wait* when Chattee is busy.
- Second part describes how the Controller keeps the resource status i.e. by *Polling* or by *Status Update.*

The *SimpleChat* example is used just as a tool to identify and specify coordination patterns. Therefore, to keep the coordination patterns flexible we have assumed that the actor (*UserAgent*) can allow multiple instances of the roles (Chatter/Chattee).

These coordination patterns are discussed in detail in the next sections.

4.1.1.1 Assign and Reject – by Polling (AR-P) pattern

In the AR-P pattern, the Controller *Assigns* the resource (Chattee) to the Chatter if the Chattee is available (*Assign* sub-collaboration) and *Rejects* the invitation request sent by the Chatter (*Reject* sub-collaboration), if the Chattee is busy. Figure 4.1 shows the UML collaboration structure of AR-P coordination pattern. The session collaborations (*ChatSession*, *Disconnect*) remain invariant over the pattern[8].

Next question is how the Controller keeps track of the status of the Chattee in the AR-P pattern. In AR-P, the Controller polls the Chattee (or set of Chattees) to learn whether it is available or busy by means of *GetStatus* sub-collaboration. The term 'Polling' is defined as *the continuous checking of other programs or devices by one program or device to see what state they are in, usually to see whether they are still connected or want to communicate*; a definition from *whatis.com*. So, as the name of the pattern indicates, in this case the Controller will check the status of the Chattee every time the Chatter sends a chat invitation request (*ChatInvite* sub-collaboration). If the Chattee is busy, the invitation request will be simply rejected. However, if the Chattee is available,

[8] The session collaborations will remain invariant over all the patterns including the AR-P pattern.

the Chatter will be assigned the available Chattee, and Chatter will initiate the chat session.

The *UserAgent* may allow multiple instances of the roles associated with *SimpleChat.* Thus, if the Controller is designed using the AR-P pattern then it must poll all the Chattee roles of a *UserAgent* and assign the first available Chattee to the Chatter. Consequently, designing the Controller using the AR-P pattern is not an efficient solution. Or in other words, the presence of Controller is not meaningful if it is designed using the 'Polling' option. Besides, the Chattee may be dynamically created and thus, not possible to be polled.

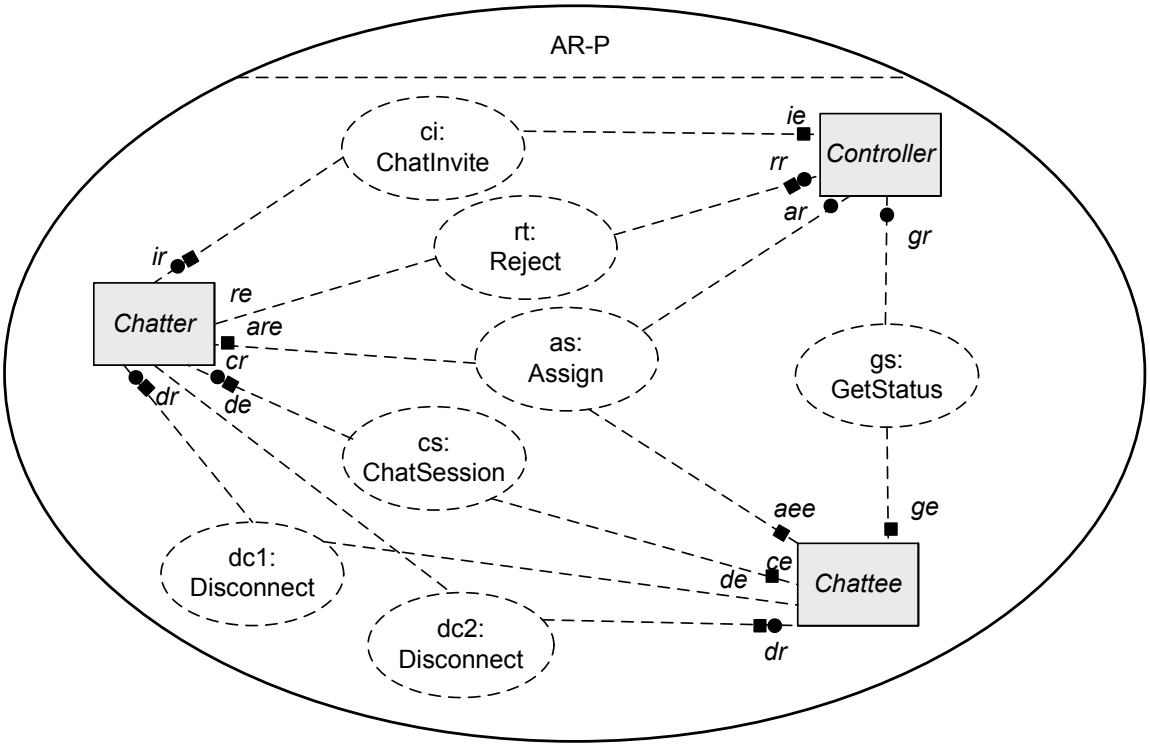


Figure 4.1: UML collaboration of the AR-P pattern

Extension in Semantics of Collaboration choreography construction

Figure 4.2 elaborates the choreography of the *SimpleChat* collaboration shown in figure 4.1. The semantics of the choreography used in [Cas08] deals with one instance of a service. In this thesis, these semantics are extended to represent multiple instances (if it is allowed by the actor). For this, the sign of multiplicity [*] is added in the partitions (used

to represent the roles). The possibility of showing global operations (in the sense: not-yet-localized) in the flows of choreography is also introduced (e.g. *status=chatteeStatus()* in figure 4.2). Since, there can be multiple instances of roles (Chatter/Chattee) active at a given time, so it is assumed that the flows of choreography can have multiple tokens flowing in them, at any given time i.e. one token per instance.

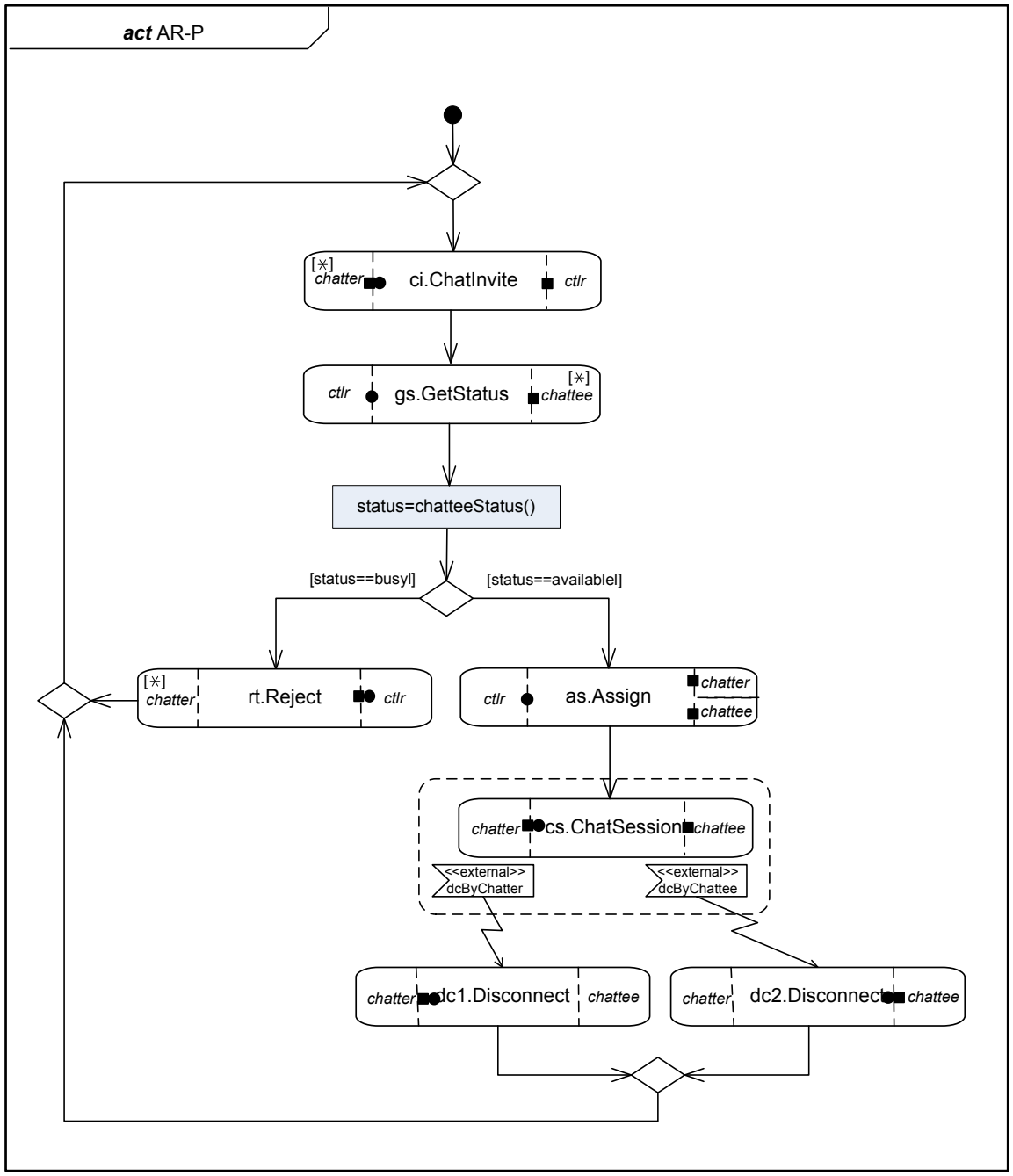


Figure 4.2: Choreography for the AR-P collaboration

Elementary Collaboration Behaviour

As discussed in section 3.2.1 (Service Modeling; chapter 3), the behaviour of each of the sub-collaborations should be described in order to complete the service specification. Some sub-collaborations may further be composed of smaller sub-collaborations. Their behaviour should be given by a choreography. For example, the *Assign* sub-collaboration is further composed of two sub-collaborations (discussed later in this section). [Cas08] proposes to use UML sequence diagrams for the description of collaborative behaviour. Figures 4.3-4.7 depict the behaviour of each of the sub-collaborations of AR-P shown in figure 4.1 (except *Assign*; which is discussed later).

Each role of a sub-collaboration is represented as a lifeline in the sequence diagram. Local actions are represented as rounded rectangles along the lifeline of the corresponding role (for example, *remove(chatteeList, chatteeId)* is the local action which is performed by the 'qe' role in *ChatteeQuit* sequence diagram as shown in figure 4.13).

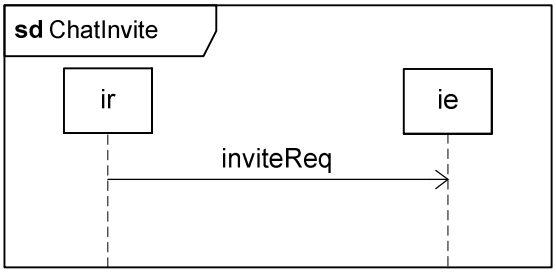


Figure 4.3: Sequence diagram of *ChatInvite*[9] sub-collaboration of AR-P

[9] The behaviour of *ChatInvite* sub-collaboration of AR-P is not same as *ChatInvite* sub-collaboration of *SimpleChat* shown in figure 3.4; chapter 3. The names of these sub-collaborations are kept same for the sake of simplicity. They should not be confused with each other. For the rest of the thesis, *ChatInvite* sub-collaboration will refer to the coordination patterns only.

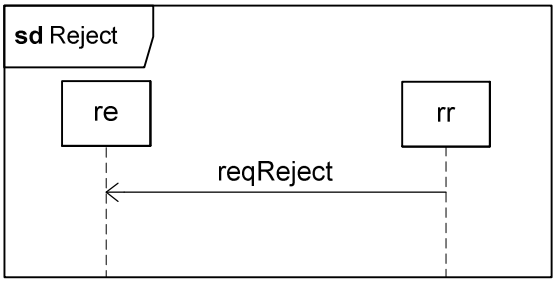


Figure 4.4: Sequence diagram of *Reject* sub-collaboration of AR-P

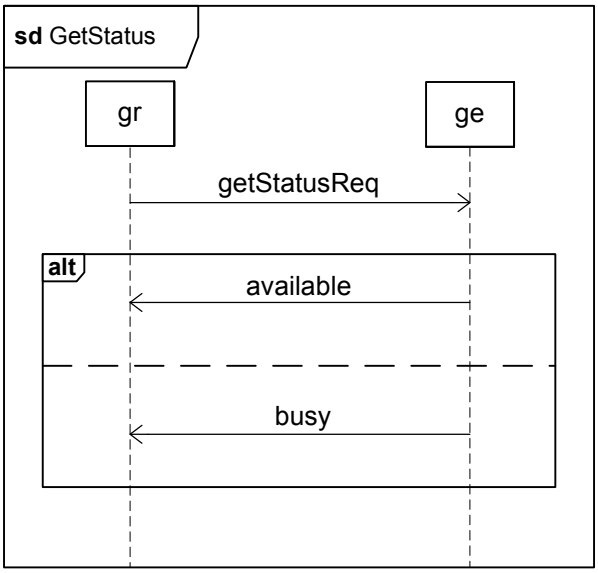


Figure 4.5: Sequence diagram of *GetStatus* sub-collaboration of AR-P

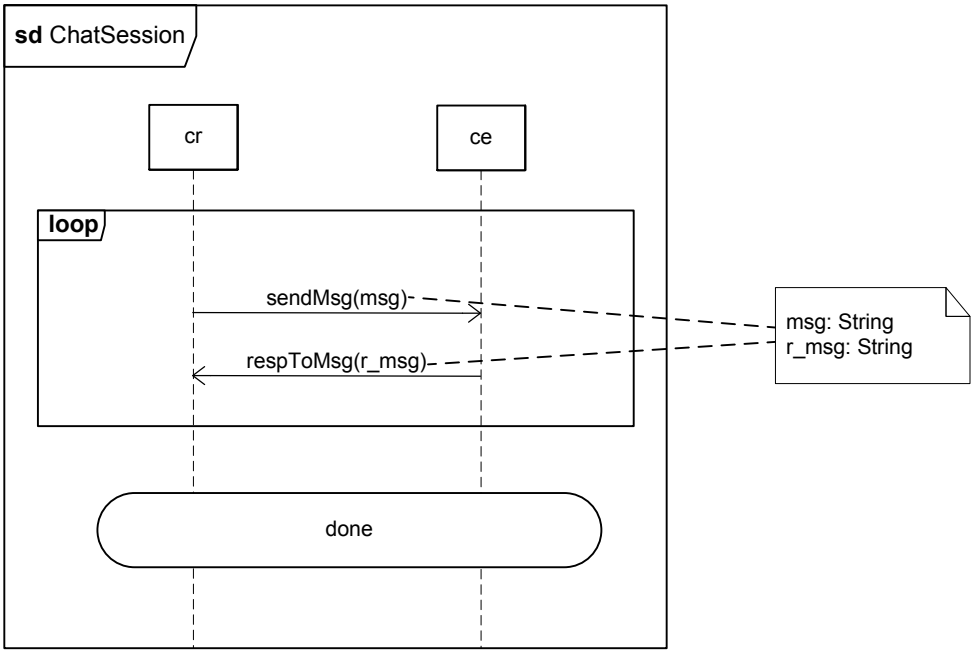


Figure 4.6: Sequence diagram for *ChatSession* sub-collaboration

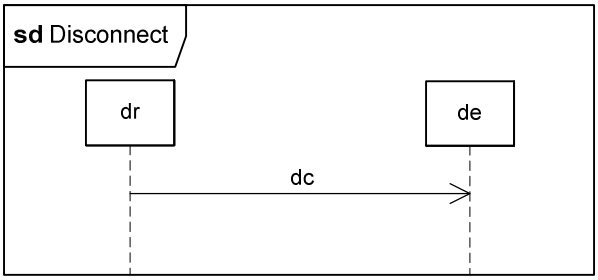


Figure 4.7: Sequence diagram of *Disconnect* sub-collaboration

The *Assign* sub-collaboration

The *Assign* sub-collaboration is further composed of two sub-collaborations; *Seize* and *Grant,* as shown by the UML collaboration of *Assign* in figure 4.8. As it can be seen from the choreography graph of the AR-P pattern (figure 4.2), *Assign* is executed when the Chattee is available for the Chatter. The available Chattee is seized for the Chatter and then granted to the Chatter in *Seize* and *Grant* sub-collaborations respectively, as shown in the sequence diagrams in figure 4.10. The choreography of *Assign* is shown figure 4.9. After the assignment of the Chattee to the Chatter, the *ChatSession* can be initiated by the Chatter.

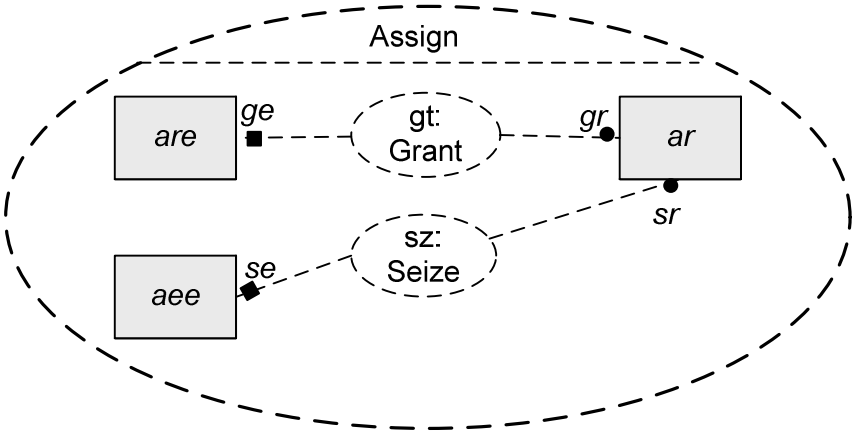


Figure 4.8: UML Collaboration of *Assign*

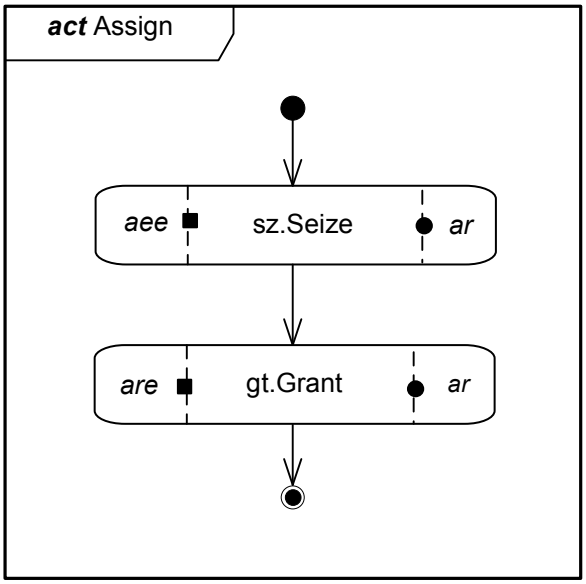
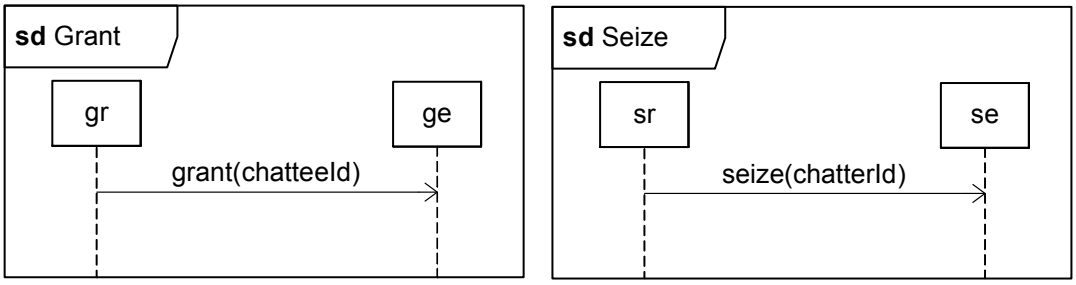


Figure 4.9: Choreography for the *Assign* sub-collaboration



(a) (b)

Figure 4.10: Sequence diagrams of *Seize* (a) and *Grant* (b) sub-collaborations of *Assign*

4.1.1.2 Assign and Reject – by Status Update (AR-SU) pattern

In the AR-SU pattern, the Controller does not ask Chattee for its current status as in AR-P pattern. Instead, Chattee takes the autonomous initiative[10] to update the Controller about its status whenever it becomes available (*Update* sub-collaboration). The Controller maintains a Chattee list (*chatteeList*) (which is the same concept of List in Java, supported by Jdk 1.5 and higher versions). In this list, the Controller inserts the Chattee which sends the *update* message of its availability. When Chatter sends an invitation request, the Controller checks this list. This function is performed by the operation *chatteeId=chatteeStatus(chatteeList)* shown in the choreography graph (figure 4.12). If the list is empty it means no Chattee is available, all are busy i.e. *chatteeId = = Null*. If the Chattee is available (*chatteeId ! = Null*) and when an invitation request is received (*ans ! = Null*), the Controller gets one Chattee from the list and assigns it to the Chatter (*Assign* sub-collaboration). The operation *ans=isChatterThere( )* checks whether any invitation request is received. The details of the *Assign* sub-collaboration is already discussed in section 4.1.1.1. If all the Chattees are busy i.e. *chatteeList* is empty, then the Controller rejects the invitation request (*Reject* sub-collaboration) as in the AR-P pattern. The UML collaboration of the AR-SU pattern is shown in figure 4.11.

When a Chattee makes itself available via the *Update* sub-collaboration, the Controller checks whether an invitation request has just been received (*ans=isChatterThere*). If no invitation request is received (*ans = = Null*), the Controller sends a message to the Chattee to wait (*ChatteeWaiting* sub-collaboration). While in the *ChatteeWaiting* sub-collaboration, the Chattee can opt to quit waiting (*ChatteeQuit* sub-collaboration), if it decides not to wait more. If this happens, the Chattee will be deleted from the *chatteeList* by the Controller (*remove(chatterId , chatteeList)* shown in figure 4.13 (c)). If the Chattee continues to wait, then whenever the Chatter sends an invitation request, the *ChatteeWaiting* sub-collaboration will be interrupted and Chattee will be assigned to the Chatter by the Controller. It is illustrated in the choreography graph in figure 4.12 that if

[10] Here we assume active Chattees that may take initiatives. This will be more relevant for multiple agents than single agents. We will come to multiple agents' discussion later in section 4.3.

the *UserAgent* allows multiple roles then only one instance will be interrupted (flows of choreography have one token per instance flowing in them; discussed earlier in section 4.1.1.1). The Chatter will then be able to initiate the *ChatSession*. The execution order of the sub-collaborations of AR-SU collaboration is illustrated in the choreography graph shown in figure 4.12. The behaviour of the sub-collaborations of AR-SU can be better understood by the sequence diagrams in figure 4.13[11] which also show the local actions performed by the sub-roles.

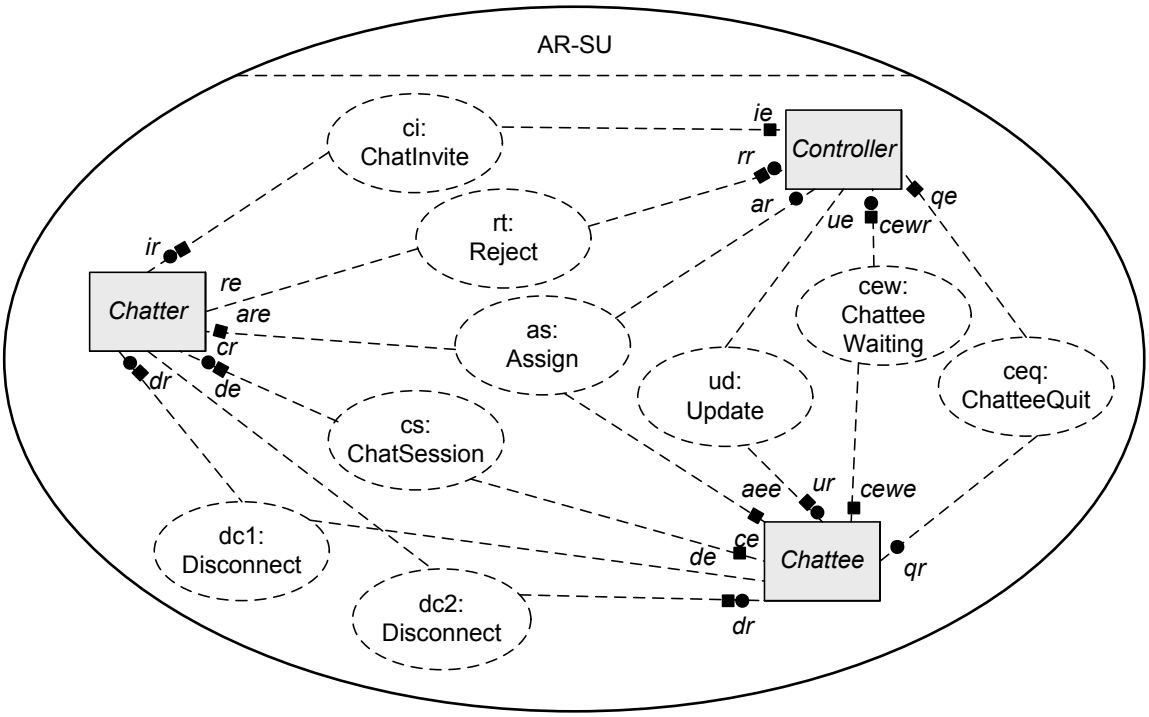


Figure 4.11: UML collaboration of the AR-SU pattern

[11] The sequence diagrams of the sub-collaborations which are new in the AR-SU pattern are illustrated in figure 4.13. The sequence diagrams which are similar to those in the AR-P pattern are already depicted in section 4.1.1.1.

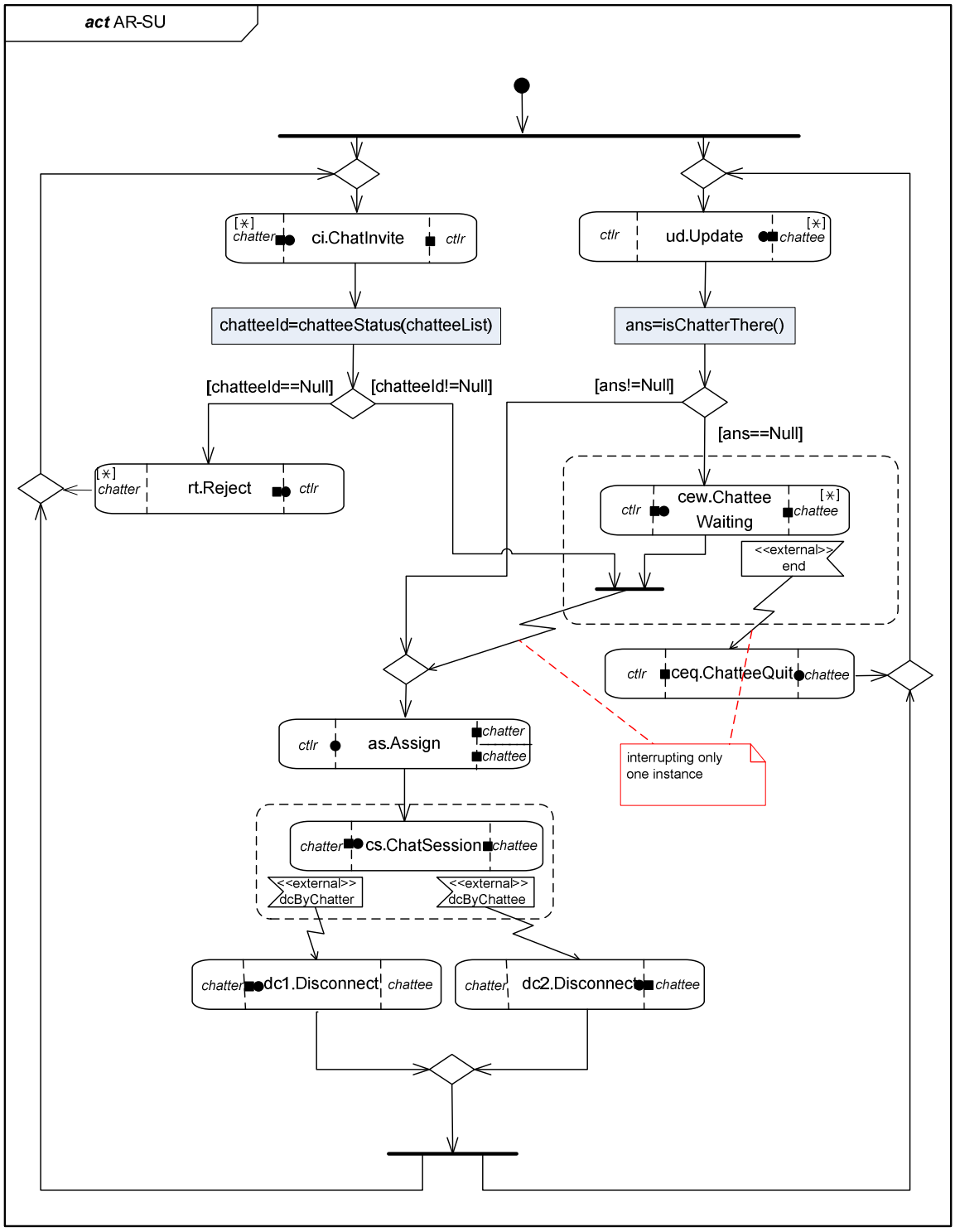
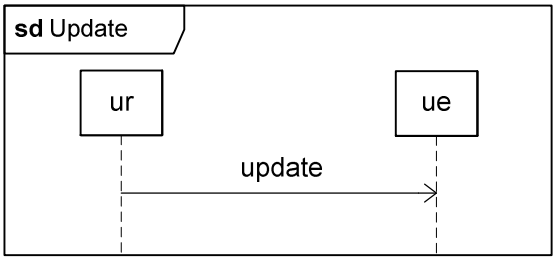
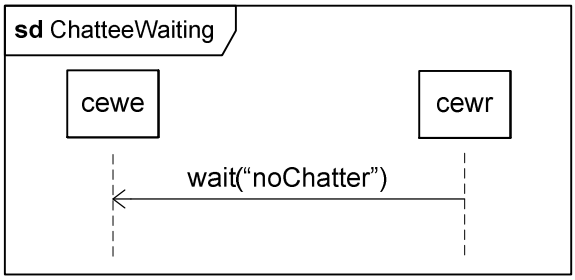


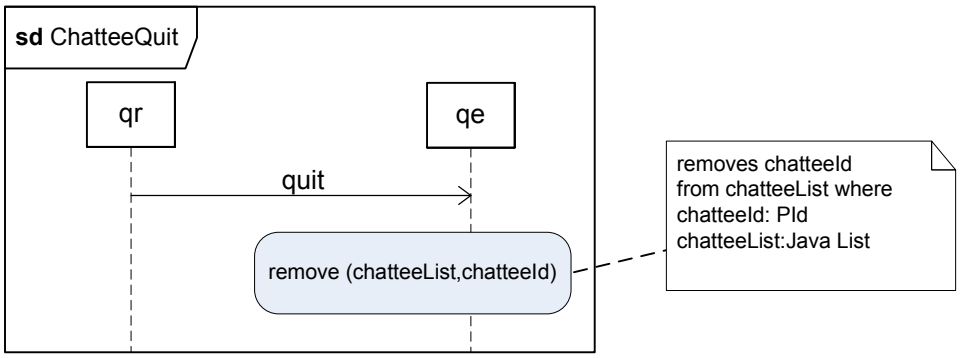
Figure 4.12: Choreography for the AR-SU collaboration



(a)



(b)



(c)

Figure 4.13: Sequence diagrams for sub-collaborations of AR-SU pattern
(a) *Update* (b) *ChatteeWaiting* (c) *ChatteeQuit*

4.1.1.3 Assign and Wait- by Status Update (AW-SU) pattern

In this pattern, if the Chattee is not available, the Controller puts the Chatter on waiting (*ChatterWaiting* sub-collaboration). The Controller maintains a *chatteeList* of available Chattee roles (see section 4.1.1.2 for details). If this list is empty, the Controller inserts the chatter ID into a waiting queue (*waitQ*) and communicates to the Chatter its current position in the queue. Meanwhile, the Chatter can opt to quit the waiting option (*ChatterQuit* sub-collaboration). In that case, the Chatter will be deleted from the waiting queue by the Controller (*remove(waitQ , chatterId)*), as shown by the sequence diagram of *ChatterQuit* sub-collaboration in figure 4.17. Whenever a Chatter leaves the queue, either because it decided not to wait any more, or because it was assigned a Chattee when it becomes available, the Controller updates the position of the other Chatters in the queue, and informs them about their new position. The UML collaboration of the AW-SU pattern is shown in figure 4.14. The behaviour of the *ChatterWaiting* sub-collaboration is specified by the sequence diagram shown in figure 4.16.

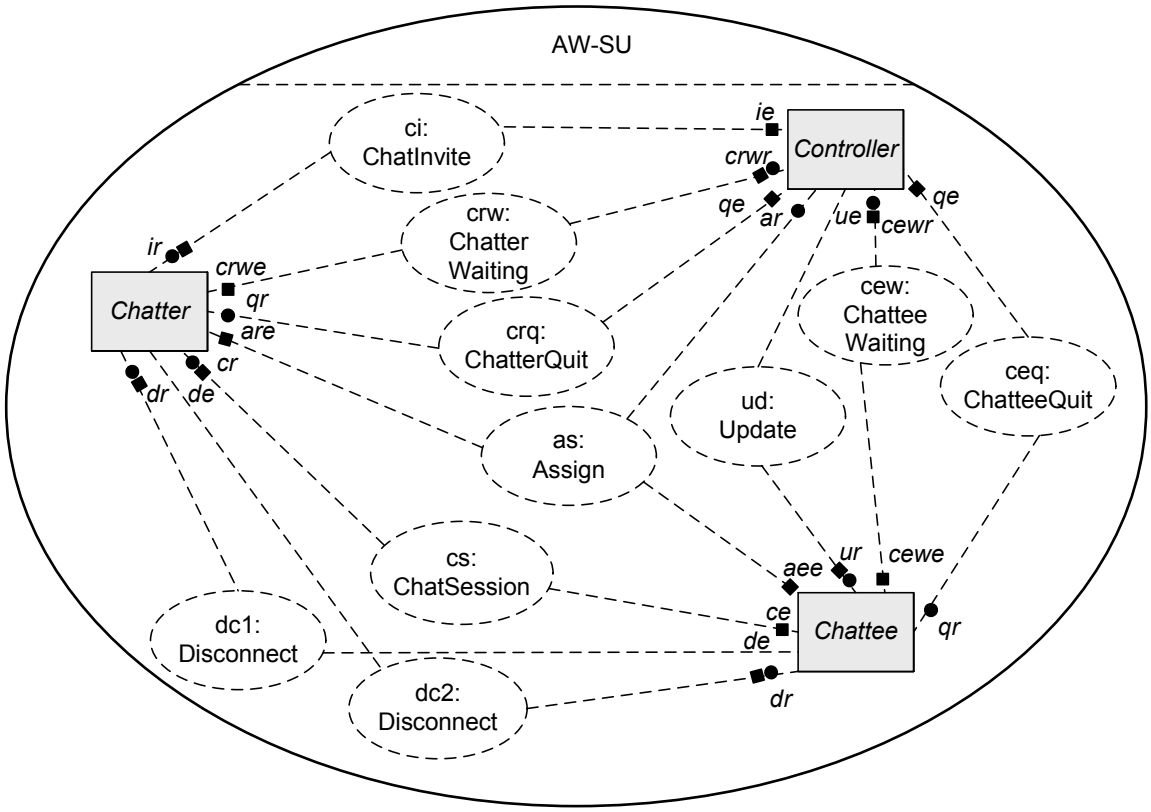


Figure 4.14: UML collaboration of the AW-SU pattern

The concept of waiting queue is the same as Queue in Java (supported by Jdk 1.5 and higher versions). In our design, the Controller is the entity which maintains the waiting queue. However, another possibility could be that the actor allows local waiting queues to be maintained by each role that it plays. But in that case, we may need to modify the behaviour of the original roles. The former solution, where the Controller maintains the waiting queue and there is no need to modify the original Chattee role, is preferred here.

We have discussed how the Controller responds to the Chatter when the Chattee is available and when it is not. We now discuss how the Controller keeps track of the resources (Chattee), is important to be analyzed specifically when the Chatter is put on waiting.

As we know, the Controller may keep track of the resource status in two ways:

- by Polling
- by Status Update

The difference between these alternatives has already been discussed in section 4.1.1.1 and 4.1.1.2 in connection with the AR-P and AR-SU patterns. The first alternative (i.e. 'Polling') is not recommended for the case where Chatters are placed on a waiting queue. The reason is explained below.

Let us suppose that we have designed the Controller selecting the 'Polling' option. The Controller will recognize the status of the Chattee by means of *GetStatus* sub-collaboration, which is invoked only once for a particular chat request (*inviteReq*). If the Chattee is busy, Controller will request the Chatter to wait and it will be in the *ChatterWaiting* sub-collaboration with the Chatter. In this case, unfortunately, Controller will keep the Chatter in the *ChatterWaiting* sub-collaboration forever, unless the Chatter opts to quit. Because, by using the Polling for this design, once the Controller is in *ChatterWaiting* collaboration with the Chatter, Controller cannot be updated by the Chattee if it becomes available. Hence, there is no way to identify the availability of the Chattee for the Controller. Consequently, *ChatterWaiting* collaboration will never lead to the *Assign* and *ChatSession* sub-collaborations. Another possibility to design the

Controller using the 'Polling' option, is to poll the Chattee at regular intervals when Chatter is in *ChatterWaiting* sub-collaboration with the Controller, but it is not an efficient solution. Therefore, this demonstrates the polling is undesirable in most cases.

When the Chatter is to be put on wait then, the second alternative (i.e. by Status Update) works better. As discussed earlier in section 4.1.1.2, in this case, the Chattee takes the autonomous initiative to update the Controller about its status whenever it becomes available (*Update* sub-collaboration). So the Controller always has the current status information of the Chattee. When the Chattee becomes available, the Controller extracts the Chatter from the head of the waiting queue, *ChatterWaiting* collaboration is interrupted for one instance and Controller assigns the Chattee to the Chatter. Accordingly, the Controller updates the queue number to other Chatters in the waiting queue.

If the Chattee is available and there is no Chatter waiting in the queue i.e. *waitQ* is empty (shown by the operation *chatterId=chatterWaiting(waitQ)* in choreography graph figure 4.15), then the Controller sends a message to the Chattee to wait (*ChatteeWaiting* sub-collaboration). While in the *ChatteeWaiting* sub-collaboration, the Chattee can opt to quit (*ChatteeQuit* sub-collaboration), if it decides not to wait more. If this happens, the Chattee will be deleted from the *chatteeList* by the Controller. The sequence diagrams of *ChatteeWaiting* and *ChatteeQuit* is already shown in figure 4.13 in section 4.1.1.2 for the AR-SU pattern. *ChatteeWaiting* will be interrupted when a chat request is received, and the Controller will assign the Chattee to the Chatter (details of *Assign* sub-collaboration are already discussed in section 4.1.1.1).

The above discussion is illustrated by the choreography of the AW-SU collaboration in figure 4.15. The choreography is showing the global operations along with the execution order of AW-SU sub-collaborations. For the detailed behaviour of sub-collaborations, see the sequence diagrams in figure 4.16 and 4.17. The sequence diagrams for *ChatInvite*, *Update*, *ChatteeWaiting*, *ChatteeQuit* can be found in sections 4.1.1.1 and 4.1.1.2.

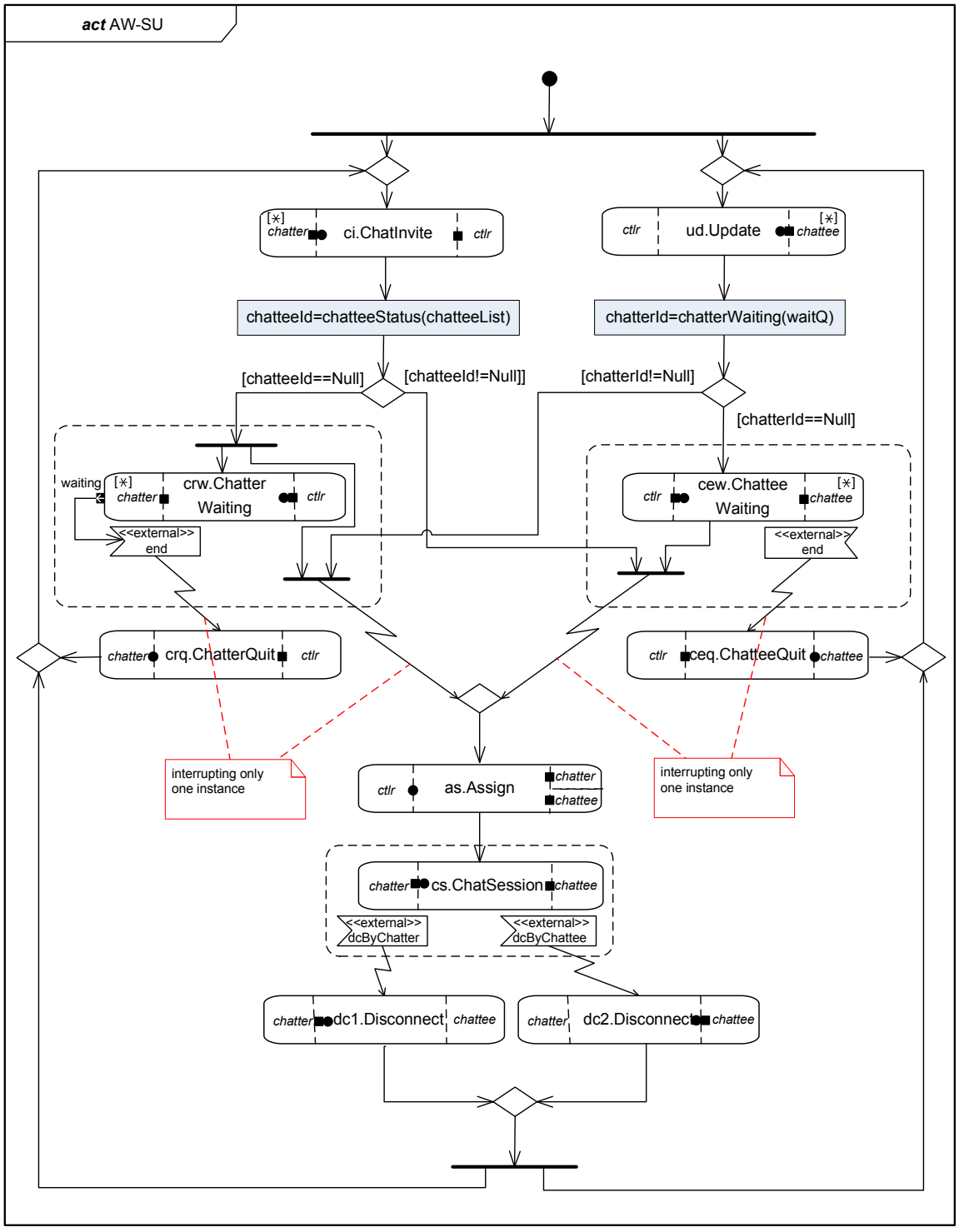


Figure 4.15: Choreography for the AW-SU collaboration

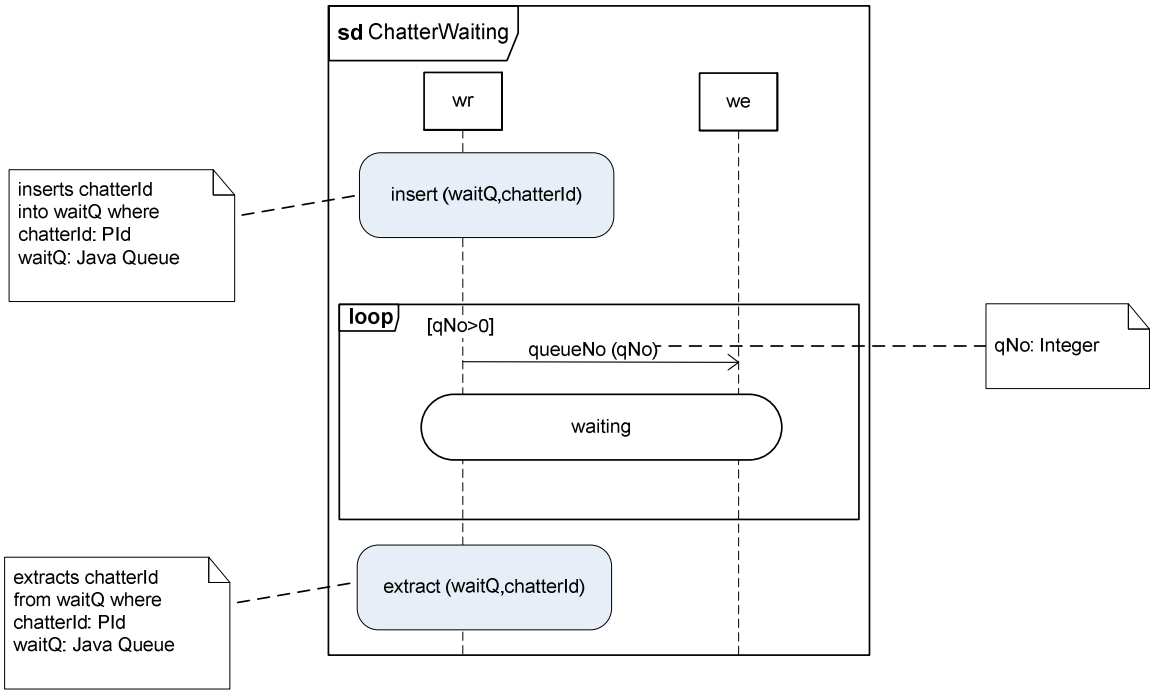


Figure 4.16: Sequence diagram for *ChatterWaiting* sub-collaboration of AW-SU pattern

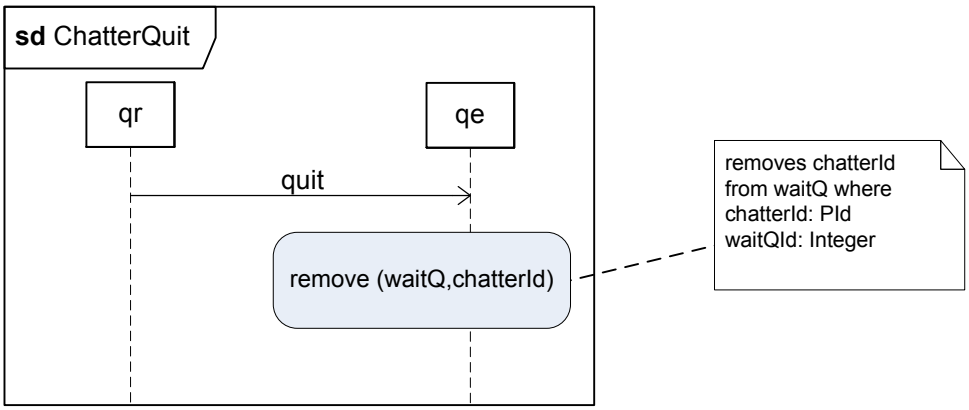


Figure 4.17: Sequence diagram for *ChatterQuit* sub-collaboration of AW-SU pattern

4.2 System Diagram (with Controller role)

The system diagram including the Controller role in addition to Chatter and Chattee roles is shown in figure 4.18. In this new system diagram, we bind the Controller role to an appropriate Component i.e. *UserAgent.* Compare this system diagram (in figure 4.18) with the system diagram presented earlier in section 3.2.2 (figure 3.5).

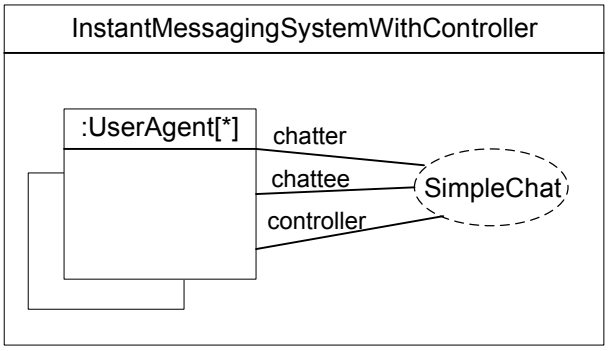


Figure 4.18: System Diagram with Controller role

There can be two possibilities depending upon the preference of each actor (*UserAgent*):

- single role per *UserAgent*
- multiple roles per *UserAgent*.

If single role per *UserAgent* is preferred, then the design of the Controller can be seen as a special case of multiple roles per *UserAgent* with the Controller maintaining only one available Chattee in the *chatteeList.* Therefore, the coordination patterns discussed for multiple roles per *UserAgent* can be used without modifications for single role per *UserAgent* case.

4.3 Controller role as Coordinating Entity for Different Agents with Multiple Roles

We discussed in section 4.1.1 that we modeled extra coordination functionality by defining another role (Controller role) for each actor. In the *SimpleChat* service example, it is required that the actor playing the role of the Chatter contacts a specific actor that

plays the role of Chattee. For example, if a user wants to contact John and he is not available, then the availability of any other person cannot serve one's purpose. However, there are services examples, where the Sender role (e.g. Chatter role in *SimpleChat*) does not need to contact any specific actor among the actors which play the Receiver role (e.g. Chattee role in *SimpleChat*). In other words, the actor playing the Sender role can take advantage of the availability of any of the actors instead of a specific actor. We will discuss a service example with this requirement in section 4.3.3.

In this section, we will discuss the Controller role which will perform the coordination functionality for different *UserAgents* with multiple roles (or it will behave as a central controlling entity for different *UserAgents*). That is; instead of having a Controller role for each *UserAgent*, all the *UserAgents* will be controlled by a central Controller. We will discuss this design approach first for our example *SimpleChat* to keep the understanding simpler. That is; how to design SimpleChat with Controller as a central entity for different *UserAgents* with multiple roles. Then later in section 4.3.3 we will discuss this solution for other examples for more clarity.

When the Controller will play the role as a central entity then it maintains an inventory of all *UserAgents* it is controlling. For this design approach, we bind the Controller role to a new component *Controller* instead of *UserAgent.* The system diagram is shown in figure 4.19. Compare this system diagram with the system diagram presented in figure 4.18.

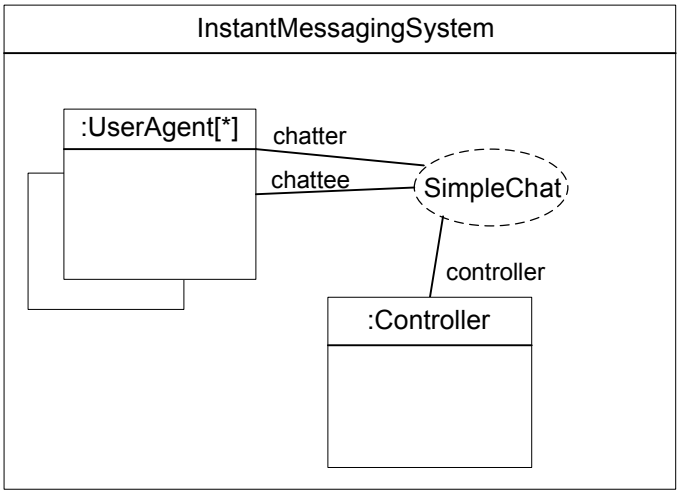


Figure 4.19: System diagram of *InstantMessagingSystem* where Controller role is bound to a new component

Now we will look at how the Controller as a central entity can affect our coordination patterns. We will discuss the patterns one by one.

4.1.1 AR-P pattern with Controller as a Central Entity

We discussed in detail in section 4.1.1.1 that in the AR-P pattern, the Controller *Assigns* the resource (Chattee) to the Chatter if the Chattee is available (*Assign* sub-collaboration) and *Rejects* the invitation request sent by the Chatter (*Reject* sub-collaboration), if the Chattee is busy. The Controller polls the Chattee to learn whether it is available or busy by means of the *GetStatus* sub-collaboration. Figure 4.1 shows the UML collaboration structure of the AR-P coordination pattern.

If a Chatter wants to chat with a specific actor playing the role of Chattee, it has to tell the Controller which *UserAgent* it wants to contact. For this purpose, Chatter has to send additional information i.e. *chatteeName, chatteeId* along with the invitation request as shown in the sequence diagram of *ChatInvite* in figure 4.20 (compare it with figure 4.3). Accordingly, the Controller will poll the respective *UserAgent* playing the Chattee role.

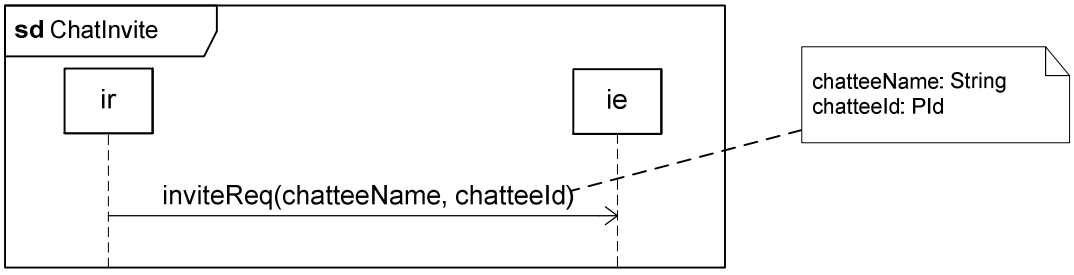


Figure 4.20: Sequence diagram of *ChatInvite* when the Controller role is central for different *UserAgents*

4.1.2 AR-SU pattern with Controller as a Central Entity

It is discussed earlier in section 4.1.1.2 that in the AR-SU pattern, Chattee takes the autonomous initiative to update the Controller about its status whenever it becomes available (*Update* sub-collaboration). The Controller maintains a Chattee list

(*chatteeList*). In this list, the Controller inserts the Chattee which sends the update message of its availability. But now the Controller is responsible for different *UserAgents*. As we know, the Chatter wants to chat with a specific actor playing the role of Chattee. To manage this, the Controller has to maintain separate *chatteeList* for each *UserAgent* containing the available Chattees for that *UserAgent*. When Chatter sends an invitation request containing the *chatteeName and chatteeId* as shown in figure 4.20, the Controller checks the corresponding *chatteeList* by using this *chatteeId* as index to indicate the specific actor (*UserAgent*) *chatteeList.* If the corresponding chatteeList is empty, the invitation request is rejected or if that *chatteeList* has any Chattee available it will be assigned to the Chatter. It is important to mention here that the basic AR-SU pattern will remain the same as discussed in section 4.1.1.2, except the minor changes we just discussed in this section.

4.1.3 AW-SU Pattern with Controller as a Central Entity

As discussed in section 4.3.2, the Controller as central entity has to maintain separate *chatteeList* for each *UserAgent*. When a Chatter sends an invitation request, the Controller checks the corresponding *chatteeList.* If the list is empty, the Controller inserts the chatter ID into a waiting queue (*waitQ*) and sends its current position in the queue. It is important to mention here that the basic AW-SU pattern will remain the same as discussed in section 4.1.1.3, except the minor changes we discussed in section 4.3.2 regarding the *chatteeList* and some other issues when Controller is central to different *UserAgents*. These issues are discussed next.

Some questions arise when we consider the design of Controller as a central entity; how it will maintain waiting queues for a number of *UserAgents*? Should there be only one waiting queue for all the *UserAgents* or separate waiting queues for each *UserAgent*? Which solution is more efficient? In our service example *SimpleChat*, the Chatter wants to chat with a specific *UserAgent* playing the role of Chattee. Hence, it is desirable for *SimpleChat* service that the Controller should maintain separate waiting queues for each *UserAgent* so that the Chatters waiting for a specific *UserAgent* can be inserted into the waiting queue reserved for that *UserAgent*. But, there are other service examples where it

will be preferred to maintain one waiting queue for all the *UserAgents*. These examples are discussed next.

Controller maintaining one waiting queue (as central entity for different *UserAgents*)

The AW-SU pattern discussed so far (with the Controller as a central entity for different *UserAgents*), is based on our *SimpleChat* example. Because the Chatter wants to chat with a specific Chattee, the Controller has to maintain separate waiting queues for each *UserAgent*. However, there are service examples that require only one waiting queue. One such example is the telemedicine consultation service, or *TeleConsultation* service, that can be found in [CBB07, Cas08]. The service is described as follows:

"*A patient is being treated over an extended period of time for an illness that requires frequent tests and consultations with a doctor at the hospital to set the right doses of medicine. Since the patient may stay at home and the hospital is a considerable distance away from the patient's home, the patient has been equipped with the necessary testing equipment at home and a terminal with the necessary software. The patient will call the hospital on a regular basis to consult with a doctor and have remote tests done. A consultation may proceed as follows:*

1. The patient uses the terminal to access a virtual reception desk at the hospital and to request a consultation session with a doctor assigned to this kind of consultation.

2. If no doctor is available, the patient will be put on hold, possibly listening to music, until a doctor is available. If the patient does not want to wait he/she may hang up (and call back later).

3. When a doctor becomes available while the patient is still waiting, the doctor is assigned to the patient.

4. A voice connection is established between the patient terminal and the doctor terminal allowing the consultation to take place.

5. During the consultation the doctor may perform remote tests using the equipment located at the patient's site and a central data logging facility located at the hospital. The doctor evaluates the results and advises the patient about further treatment. Either the doctor or the patient may end the consultation call.

6. After the consultation call is ended, the doctor may spend some time updating the patient journal and doing other necessary work before signaling that he/she is available for a new call. The doctor may signal that he/she is unavailable when leaving office for a longer period, or going off-duty".

In the *TeleConsultation* service we can identify two roles, the patient and the doctor, that behave partly independently of each other and may take uncoordinated initiatives to initiate activities that involve the other. A third role, the virtual reception desk, serves to coordinate these initiatives. Figure 4.21 shows a collaboration describing the structure of the *TeleConsultation* service.

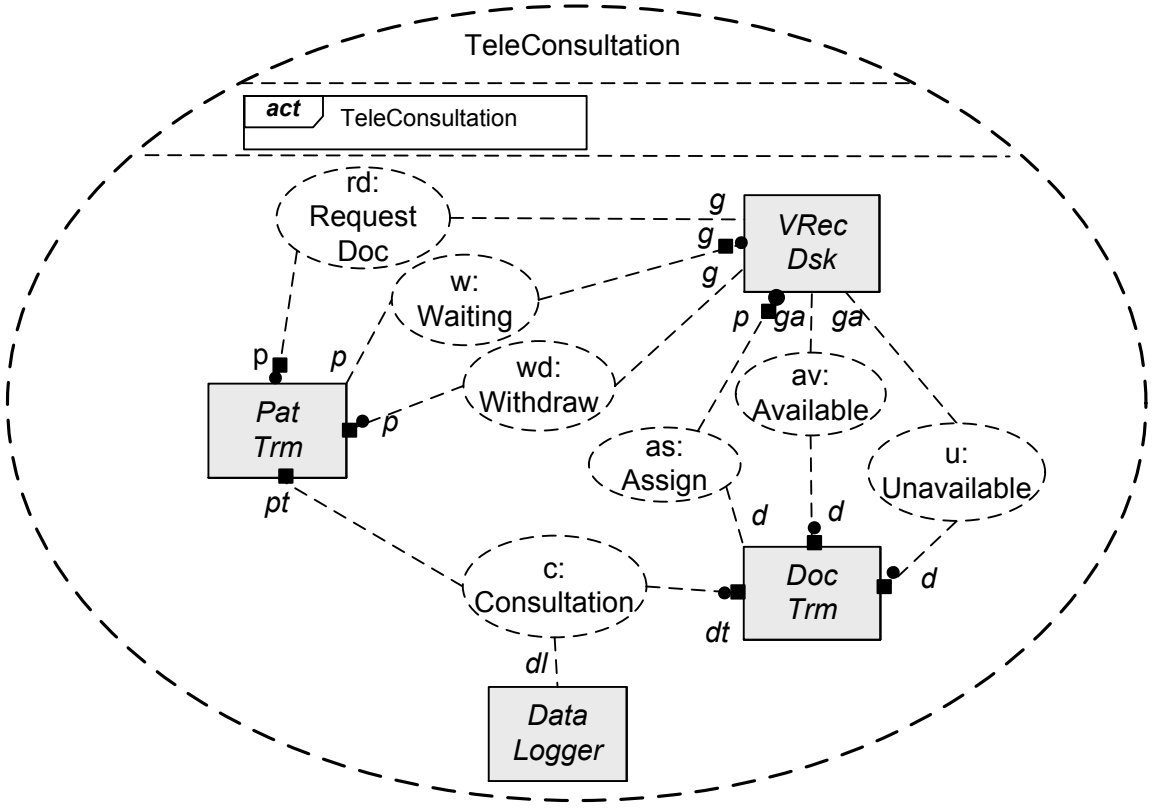


Figure 4.21: Roles and sub-collaborations in the *TeleConsultation* service [CBB07]

If we compare this example with our *SimpleChat* service example that uses AW-SU pattern (shown in figure 4.14), we can say that virtual reception desk role is similar to the Controller, with a minor difference as explained next. The Doctor role (*DocTrm* in figure 4.21) updates its status (available/unavailable) to the virtual reception desk role (*VRecDsk* in figure 4.21). Since, in this example, the Patient role (*PatTrm* in figure 4.21) wants to

get remote assistance from any of the doctors i.e. not from any specific doctor, the *VRecDsk* role can assign any of the available doctors through the *Assign* sub-collaboration as shown in the figure 4.21. Let us assume that *VRecDsk* inserts the *PatTrm* role into a waiting queue if no doctor is available. Since, it does not matter which doctor should be assigned, *VRecDsk* does not need to maintain separate waiting queues for each doctor. It will maintain a single waiting queue for all the doctors in which all the *Patients* waiting for the availability of any of the doctors will be inserted. The same argument holds for maintaining one list for all the available doctors. Hence, it is unlike the *SimpleChat* example where the Controller needs to maintain separate waiting queues for each Chattee because Chatter wants to chat with a specific actor playing the role of Chattee.

For the sake of simplicity and to keep the understanding of the central Controller design simple, let us stick to our *SimpleChat* example and assume (temporarily) that the actor playing the Chatter role does not require chatting with a specific actor playing the Chattee role. For example, contacting a user named John can serve the purpose instead of contacting another user named Cristian. With this assumption, it can be clearly understood that the design of the Controller as central entity (for different UserAgents with multiple roles) will not require any modifications or additional operations as discussed in sections 4.3.1, 4.3.2 and 4.3.3. We just need to bind the Controller role to a new component instead of binding it with the *UserAgent*. See the next section for details.

4.4 When to use which solution

We have discussed the Controller design with three possibilities:

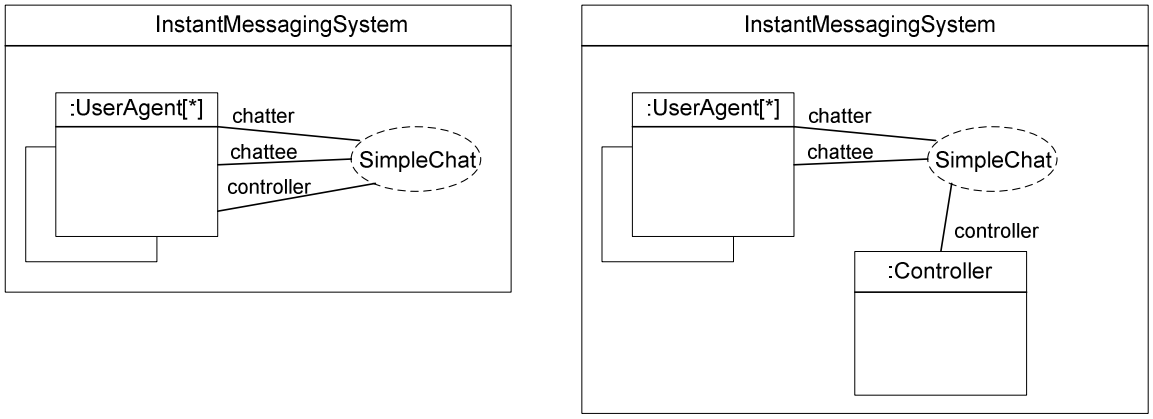
- Single role per agent
- Multiple roles per agent
- Multiple roles-different agents

These alternative solutions have advantages and disadvantages depending on which service they will be applied to. We discussed the Controller design for *SimpleChat* with two options; the Controller maintaining a single waiting queue and the Controller maintaining separate waiting queues for each UserAgent. The latter approach introduces

a processing power overhead for maintaining and managing different waiting queues, which will affect the computation speed. However, it is the service requirements that put the constraints on which design to choose. When to use which solution is explained as follows:

- When there are several alternate resources and the Chatter may want to contact an actor playing the role of the Chattee, but it does not matter which actor (*UserAgent*) (as in the case of the *TeleConsultation* service example), then one may prefer to follow the system diagram shown in figure 4.22 (b), where the Controller is bound to a new component and it is the coordinating entity for different *UserAgents* with multiple roles.

- When Chatter wants to chat with a particular actor playing the role of Chattee, then a Controller role will be defined for each actor (*UserAgent*). The system diagram for this approach is shown in figure 4.22 (a). The Controller will maintain a single waiting queue for that specific actor playing the role of Chattee. Another example where we should use this approach is phone call service. The Caller wants to talk to a specific actor (*UserAgent*) playing the role of Callee, Controller cannot assign any of the available *UserAgents* playing the role of Callee.



(a) (b)

Figure 4.22: Comparison of System diagrams (a) two possibilities: single role per agent and multiple roles per agent (b) multiple roles - different agents

4.5 Coordination within an Actor (UserAgent)

In our coordination patterns, we have focused on dealing with external communication i.e. between actors. However, the Controller should also be made responsible to coordinate for the internal communication within an actor.

A system component plays one or more initiating roles whose execution is triggered by an external event. Then, it is possible that a system component that is already participating in an occurrence of a service collaboration, tries to initiate a new collaboration occurrence in response to an external event. For example, let us consider that in the *SimpleChat* service, single instance of roles (Chatter/Chattee) is possible per *UserAgent*. The Chatter role is the initiating role whose execution is triggered by an external event (end-user). One cannot control the actions of an end-user. The event can happen while the *UserAgent* is already busy playing the role of the Chatter in another *SimpleChat* occurrence. In this case, the Controller needs another decision policy in order to respond to this event. The decision will then be dependent on *whether the Chatter role already exists or not.* If it already exists within the *UserAgent*, then the Controller responds according to the coordination patterns identified in previous sections i.e. either by rejecting the end-user initiative or putting it on wait. The basic structure of the coordination patterns will remain the same as already discussed in this chapter, only the decision policy of the Controller will be different.

4.6 Whether to Relay the Invitation Request or Not?

In our coordination patterns, we have considered that it is the responsibility of the Controller to keep the track of the resources and to respond back to the Chatter depending upon the availability of the resource. If the Chattee role is busy, the Controller either rejects the chat invitation request or gives the option of waiting to the Chatter. But if the Chattee is available, the Controller assigns the Chattee to the Chatter by seizing the Chattee and granting it to the Chatter. In other words, the Controller itself decides to accept the chat invitation request by sensing the availability of the Chattee.

There is a possibility to design the Controller to relay the invitation request to the Receiver role (Chattee). As a result, the Chattee role itself responds to the Chatter for its invitation request. But this approach kills the real purpose of the Controller, which is designed to respond to the autonomous events. It is defined external to the service roles and serves to coordinate role binding by dynamically binding the roles to actors during execution. Therefore, relaying of the invitation request by the Controller is not recommended for the coordination patterns proposed in this chapter.

Chapter 5: Applying Coordination Patterns to an Existing Service Model

In this chapter, we will focus on the second task of this thesis i.e. how to improve systems by adding new/extra functionality to an existing service model. This chapter presents initial thoughts to address this problem by taking an example service. The coordination patterns identified in this thesis (presented in chapter 4) are considered in this chapter as an extra/new functionality which we want to apply to an existing service model. The explanation is given for the AR-SU pattern as a case study. We start by identifying general structure in coordination patterns.

5.1 Identification of General Structure in Coordination Patterns

We have discussed in detail the coordination patterns in chapter 4, using the *SimpleChat* service example. From these patterns we can see that the Controller role is involved in the coordination patterns until the chat invitation request is processed by the assignment of Chattee to the Chatter (*Assign* sub-collaboration) in the case of successful connection, or in other words, until the 'actual' service starts between the two parties. This gives the indication that we can extract coordination patterns from the *SimpleChat* service example which are generic to a broad range of services.

Let us look at how this extraction is possible. We will take the case of the AR-SU pattern. As shown in figure 5.1, it is understandable that the sub-collaborations *ChatSession* and *Disconnect* are specific to the *SimpleChat* service, and the sub-collaborations *ChatInvite, Reject, Assign*, *Update, ChatteeWaiting and ChatterQuit* are generic to all services. We can easily draw a separation line between general structure and service specific structure as depicted in figure 5.1. Consequently, the general AR-SU pattern will become as depicted in figure 5.2. To be general, Chatter and Chattee roles are renamed as

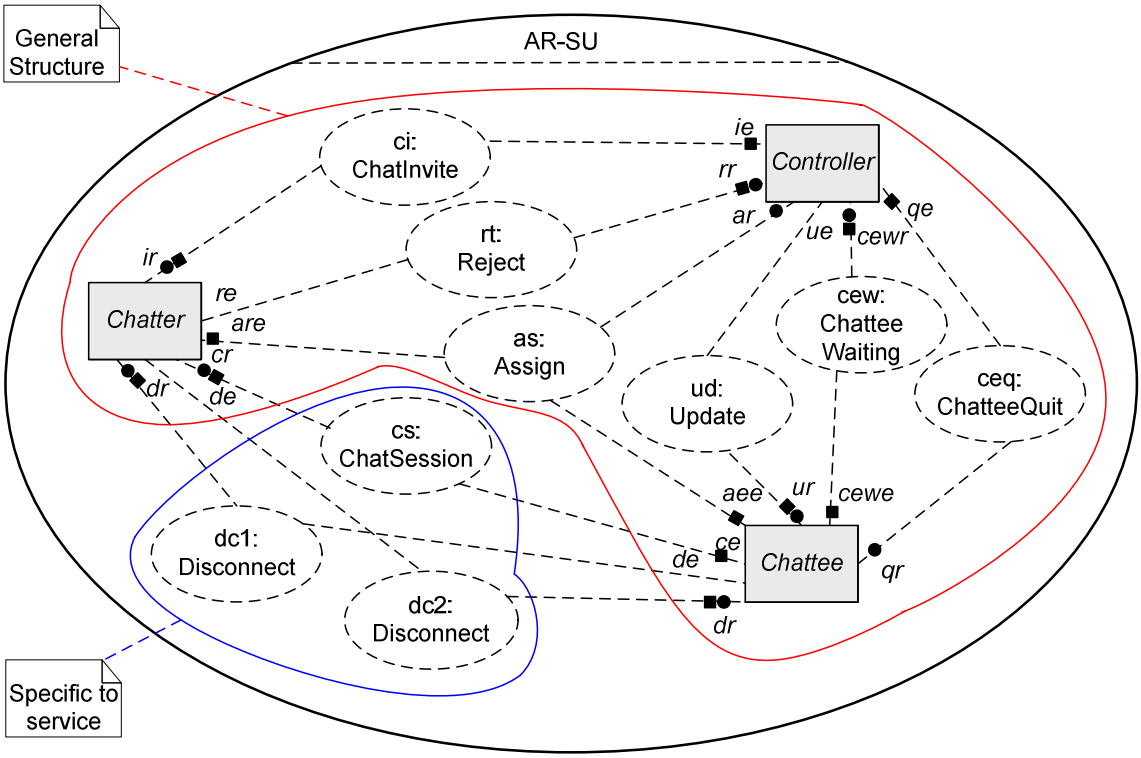


Figure 5.1: UML collaboration of the AR-SU pattern depicting the separation between General Structure and Service Specific Structure

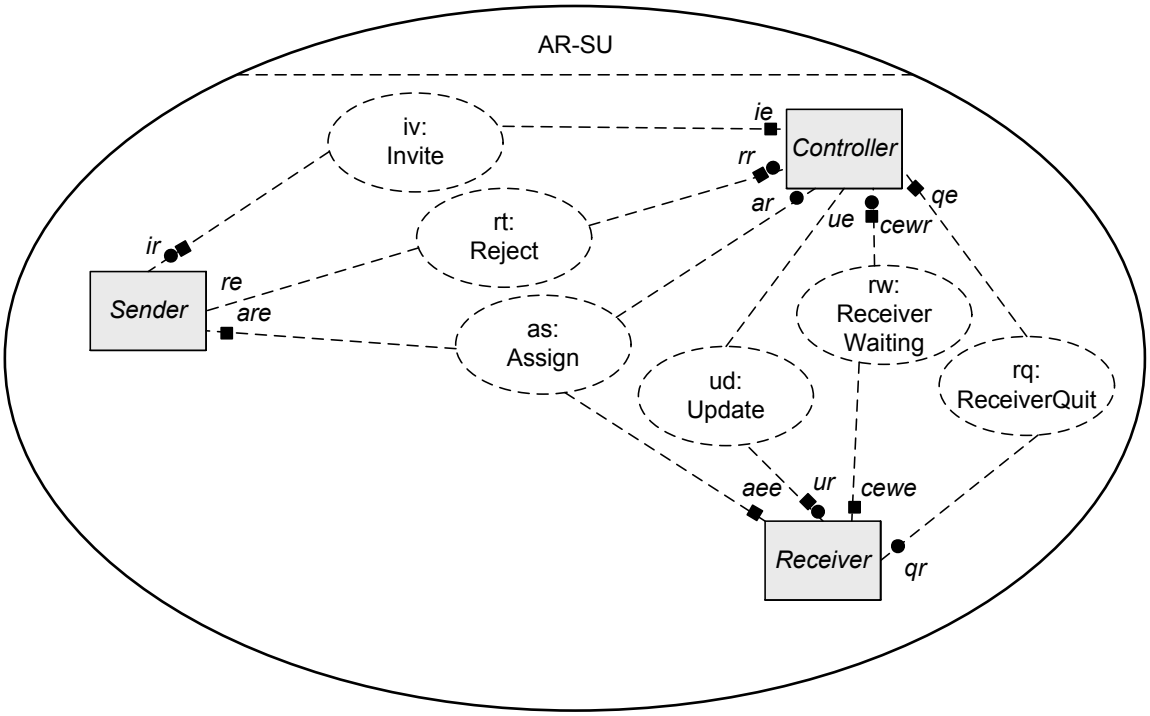


Figure 5.2: General AR-SU pattern; Chattee role and Chatter role are renamed as Sender and Receiver. Similar renaming can be observed in the sub-collaborations

'Sender' and 'Receiver'. Similarly, the *ChatInvite* sub-collaboration is renamed as *Invite*, *ChatteeWaiting* as *ReceiverWaiting* and *ChatteeQuit* as *ReceiverQuit*.

5.2 Applying the coordination pattern into an existing model

The next question is "how an existing service will use the coordination patterns as an additional functionality". In the following we will discuss some of the potential opportunities for reuse that UML 2.0 offers:

- UML 2.0 generalization relationship
- UML 2.0 templates
- UML 2.0 *extend* relationship

Another solution of the above cited question is to exercise 'service composition' as worked out by [Ros09]. We will discuss how we can apply these solutions with their pros and cons.

5.2.1 By Using UML 2.0 Generalization Relationship

[OMG09] defines generalization as "*a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier"*.

The service models which need to have the functionality of coordination among roles should specialize from a coordination pattern, for instance from the AR-SU. By specializing a general classifier, all the properties of the general classifier are inherited by the specialization but any redefinable element of the general classifier may be either replaced or extended and declared as **{redefined}** in the specialization [HPW03]. An entity that cannot be redefined in specializations is declared as **{final}** in the general

classifier [OMG09]. Now let us see how we can use the UML 2.0 generalization relationship to reuse our coordination patterns in existing service models.

Consider for example a *BasicPhoneCall* service as shown in figure 5.3. This collaboration consists of two roles, Caller and Callee, involved in sub-collaborations; *CallSession* and *CallEnd*. The roles of these sub-collaborations are bound to Caller and Callee. For instance, caller (cr), disconnecter (dr) and disconnectee (de) roles are bound to the Callee role of the containing collaboration (*BasicPhoneCall*). The execution order of the sub-collaborations involved is defined by the choreography graph in figure 5.4.

If the *BasicPhoneCall* service needs to have the coordination functionality of the AR-SU pattern, we can create a new *BPCwithAR-SU* collaboration that specializes the AR-SU pattern and extends it with the functionality of *BasicPhoneCall.* For that, we redefine the Sender and Receiver roles and extend them with the behaviour defined by the Caller and Callee roles. Because, we do not want the Controller to be redefined, it is declared as **{final}** in the AR-SU pattern (see figure 5.5). The inherited aspects are represented by dashed lines which differentiate them from the extension added in the classifier [HPW03]. Since, dashed lines are also used to represent the ellipse of UML 2.0 collaborations, we use a solid line (shown in red in figure 5.5) inside the dashed lines of collaboration to represent the extensions in specialization. The specialized *BPCwithAR-SU* service is shown in figure 5.5. It can be seen that Sender and Receiver roles are *redefined* in *BPCwithAR-SU.* Inheritance is not defined in UML for activity diagrams, however it is defined for state machines (which are used to model behaviour like activity diagrams). We have worked out a way to define inheritance for activity diagrams. In order to represent the elements that are inherited in activity diagrams, we used dashed lines in combination with solid lines such that dashed lines appear inside solid lines (see figure 5.6). We avoid using *only* dashed lines because they are used to represent interruptible regions in activity diagrams. The choreography for the *BPCwithAR*-SU collaboration is shown in figure 5.6. It can be seen in figure 5.6 that the Sender and Receiver roles are *redefined* only for the *BasicPhoneCall* collaboration.

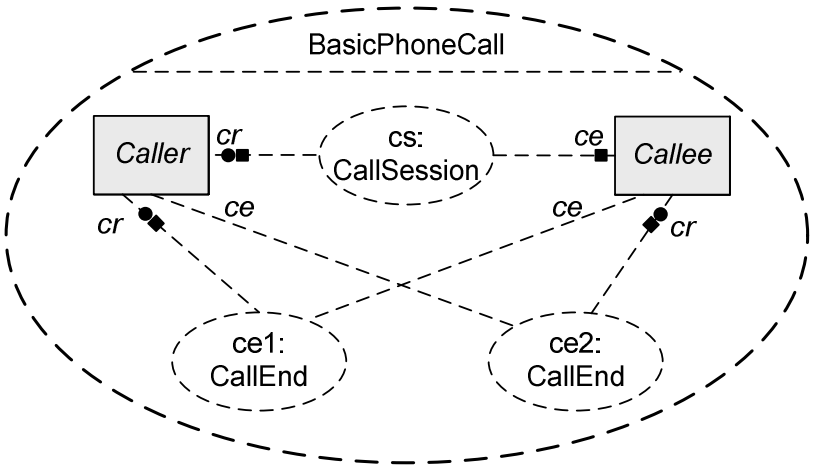


Figure 5.3: UML collaboration of BasicPhoneCall service

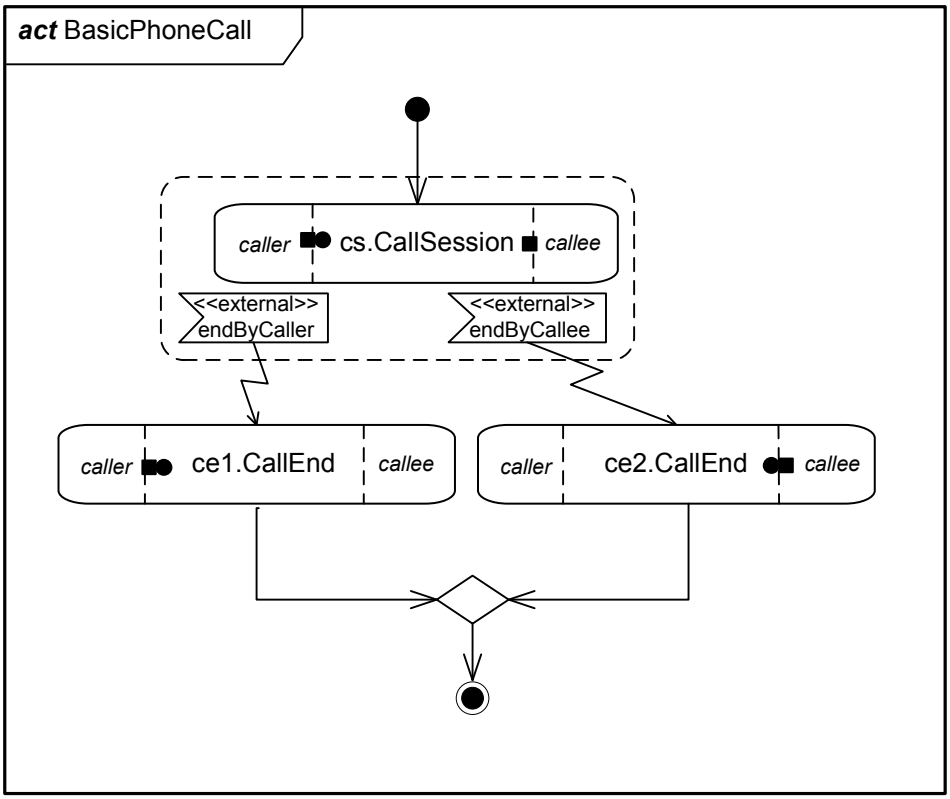


Figure 5.4: Choreography of the *BasicPhoneCall* collaboration

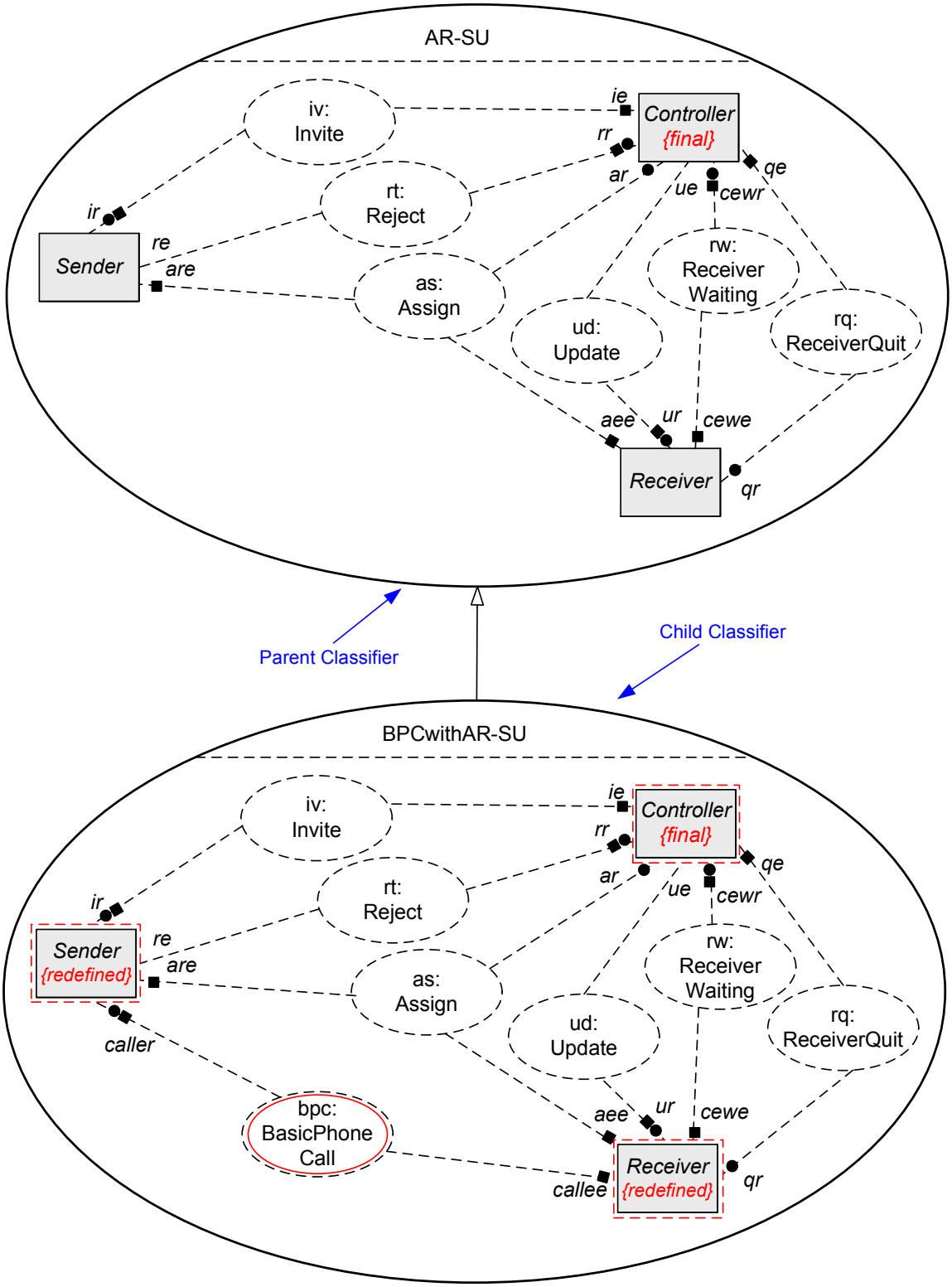


Figure 5.5: UML 2.0 Generalization relationship: *BPCwithAR-SU* collaboration is specialized from general AR-SU collaboration

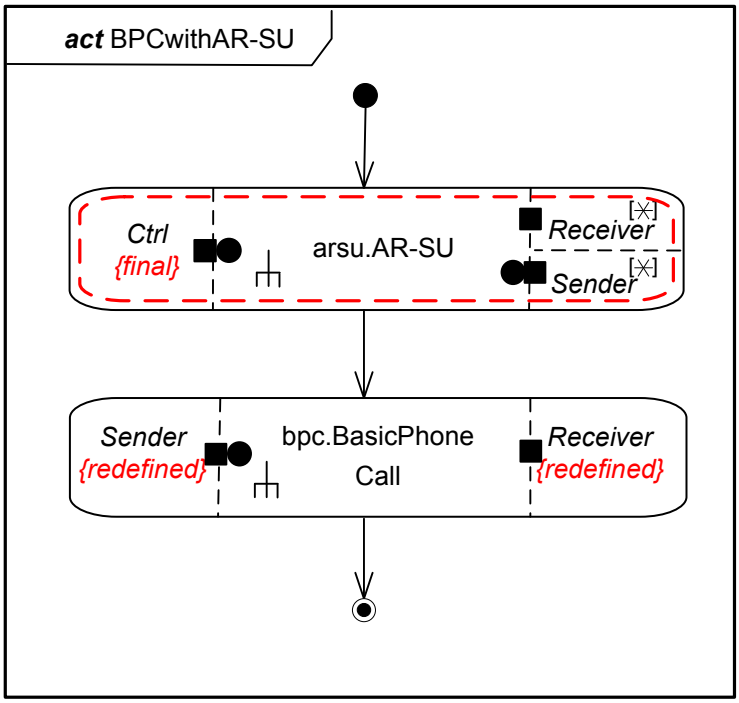


Figure 5.6: Choreography for *BPCwithAR-SU* collaboration: The symbol ⫛ represents a complex activity

Pros and Cons

The *BPCwithAR-SU* service which is specialized from the AR-SU pattern, surely contains the intended behaviour. This solution has advantage over UML 2.0 *extend* relationship and UML 2.0 templates because of the following reasons;

1. By specializing a general classifier, we cannot only add new properties but can also redefine the existing ones.

2. It is guaranteed that the Sender role and Caller role are played by the same actor. Similarly, Receiver role and Callee role are played by the same entity (this is not certain when we use UML 2.0 *extend* relationship; it is discussed later in section 5.2.3).

3. Unlike UML 2.0 templates, we do not need to introduce an *UndefinedService* parameter (we will discuss it in section 5.2.2).

The redefinition of elements is advantageous if the properties are added, but if the existing behaviour of the Sender and Receiver role is replaced then it will be detrimental. So additional constraints are required which ensure that the redefinition will result in the addition of properties but not in replacement. For this, it must be ensured that the Controller role should be declared as **{final}.** The major drawback of generalization

relationship is that it is not completely modular i.e. we have to modify the general model for additional functionalities. For example, the Sender role of AR-SU pattern should be modified in accordance with the behaviour defined by the Caller role of *BasicPhoneCall*.

5.2.2 By Using UML 2.0 templates

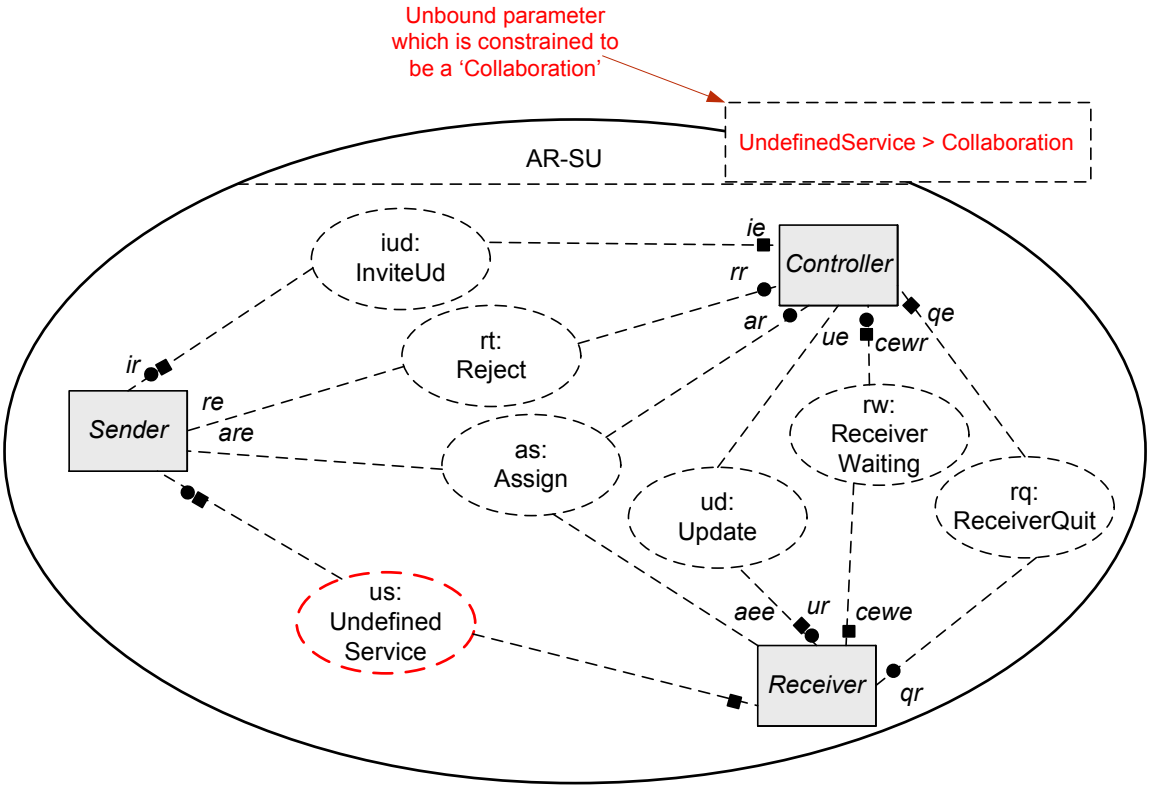
[RJB05] defines template as "*the descriptor of an element with one more unbound parameters"*. Typically, the parameters are the classifiers[12] but they can be operations and packages as well. UML templates are not directly instantiatable as they have unbound parameters. The templates must be instantiated by binding the unbound parameters to actual values. The unbound parameters are substituted by the actual values. UML templates allow us to create UML elements (classifiers, packages) to work with other UML elements when we do not know specifically what those elements are.

We will discuss how UML templates can be used to solve our problem. Collaborations (a UML classifier) also support the ability to be defined as templates [OMG09]. Figure 5.7 illustrates how to use UML template concept for applying the coordination patterns to an existing service. It shows the concept by using the example of AR-SU coordination pattern as a UML collaboration template and *BasicPhoneCall* service (earlier shown in figure 5.3) as the binding collaboration. To represent the UML collaboration as a template, a dashed rectangle is used at the upper right corner of the collaboration as shown in figure 5.7(a). In the rectangle there is a place holder for the unbound parameter. Since, we are now using the AR-SU coordination pattern as a UML collaboration template, a sub-collaboration is introduced (*UndefinedService*) between the Sender and Receiver roles which is not defined. The *UndefinedService* is used as an unbound parameter that is constrained to be a UML collaboration (the symbol '>' defines a constraint which is placed just after the unbound parameter; constraints are optional [OMG09]). When an existing service model uses the AR-SU template, it will be called "derivation". In our example, the AR-SU template is being used by *BasicPhoneCall_AR-SU* collaboration (in other words the *BasicPhoneCall* collaboration

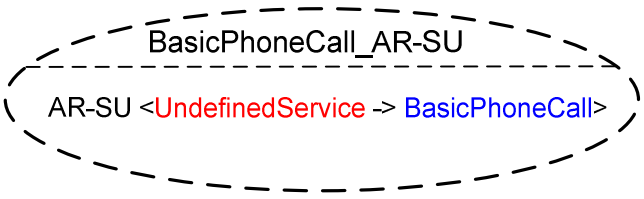
[12] [RJB05] defines classifier as *"a model element that describes behavioral and structural features" (e.g. class, collaboration).*

is derived from the AR-SU template). The *UndefinedService* unbound parameter is substituted by the *BasicPhoneCall* collaboration

The choreography of the derived *BasicPhoneCall_AR-SU* collaboration (bound collaboration) is shown in figure 5.8. It can be observed that *BasicPhoneCall* is substituted in place of *UndefinedService* collaboration which is the unbound parameter defined in the AR-SU collaboration template already shown in figure 5.7(a).



(a)



(b)

Figure 5.7: (a) AR-SU template declaration (b) Bound collaboration resulting from binding the *BasicPhoneCall* to *UndefinedService* parameter of AR-SU template

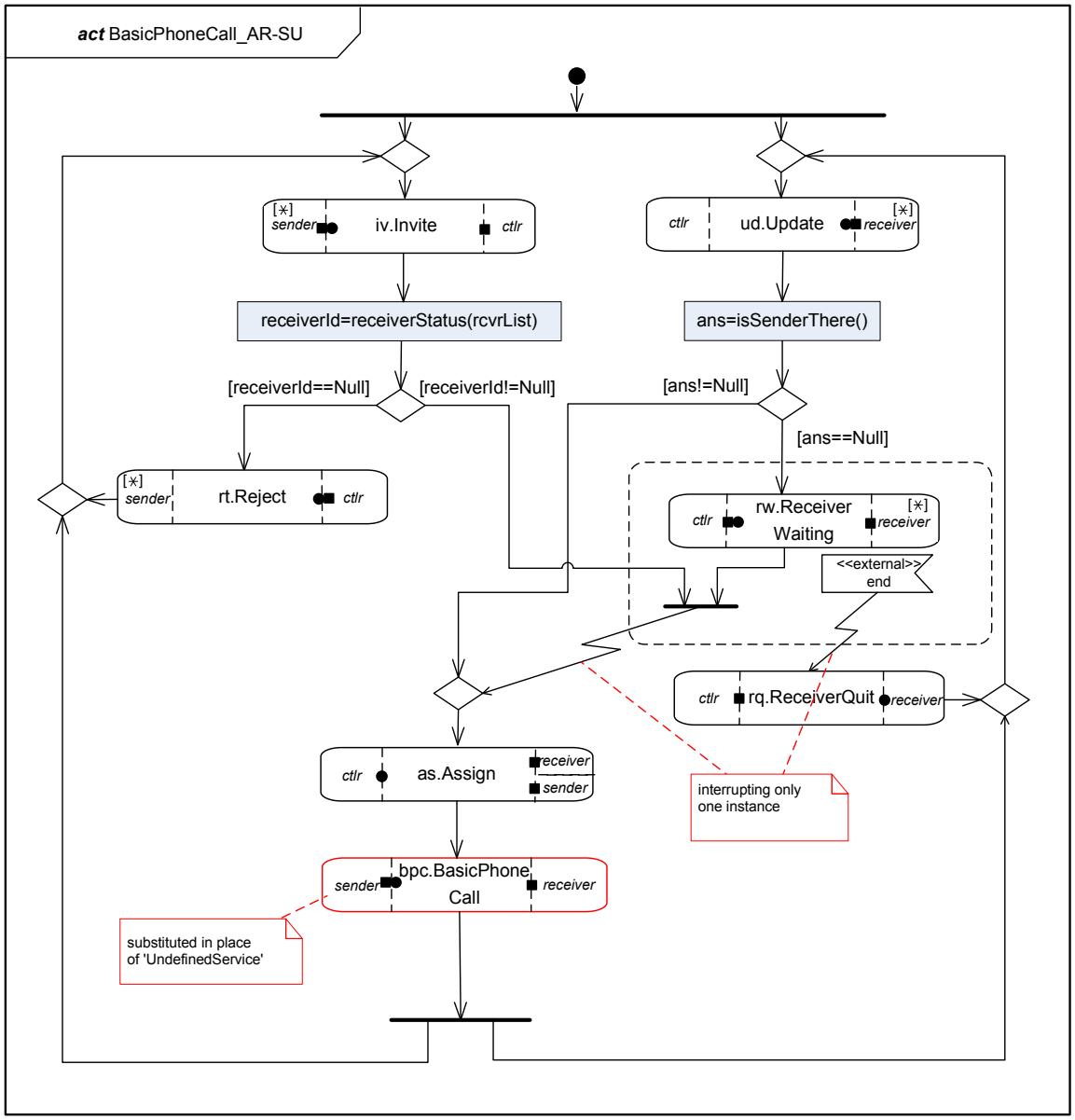


Figure 5.8: Choreography of the bound collaboration *BasicPhoneCall_AR-SU*. The *BasicPhoneCall* is substituted in place of *UndefinedService* collaboration parameter of the AR-SU template.

Pros and Cons

UML templates allow us to provide abstractions. It is another way to introduce a new functionality in an existing service model, keeping the essence of modularity. Unlike *extend* relationship, templates allow to bind the roles. But in UML templates the binding of the role means complete substitution of the role. For example, Sender role can be declared as an unbound parameter which can be defined to be substituted by the actual

Caller role of *BasicPhoneCall* service. But the derived *BasicPhoneCall_AR-SU* service will not behave as it is intended to be. As the Sender role is completely replaced by the Caller role, the derived *BasicPhoneCall_AR-SU* service will discard the presence of the Controller and that will result in the original *BasicPhoneCall* service (as if it was not derived). If we see the other way round, Sender role cannot be declared as an unbound parameter because it is already bound in sub-collaborations of AR-SU.

The UML templates has an advantage over the *extend* relationship. It is obvious that Caller role will be played by the Sender role and Callee role will be played by the Receiver Role (this is not obvious in *extend* relationship; discussed in next section). First reason; the *UndefinedService* is an undefined sub-collaboration between Sender and Receiver roles, the *BasicPhoneCall* will replace it. Second reason ; as shown in figure 5.7(a), Sender is the role will initiate the service (*UndefinedService*) and Receiver role can only terminate the service (represented by filled circle and filled square respectively) and in *BasicPhoneCall*, Caller role can initiate the service and Callee role can only terminate.

But the concept of UML templates does not solve the dilemma of role binding without substituting the roles of the collaboration template.

5.2.3 By Using UML 2.0 'extend' Relationship

In UML 2.0 'extend' is defined as a relationship from an extending use case to an extended use case which defines how the behaviour of the extending use case can be inserted into the behaviour of the extended use case [OMG09, RJB05]. In other words, the extending use case expands the behaviour of the extended use case. The *extend* relationship contains one or more *extension points* defined in the extended use case. *Extension point* is the location within the behaviour sequence of extended use case at which additional behaviour can be inserted [RJB05]. The *extend* relationship may have a *condition* that must hold for the extension to take place when the extension point is reached. The *condition* and *extension point* can optionally be defined in a *Note* attached to the extend relationship as shown in figure 5.9 [OMG09].

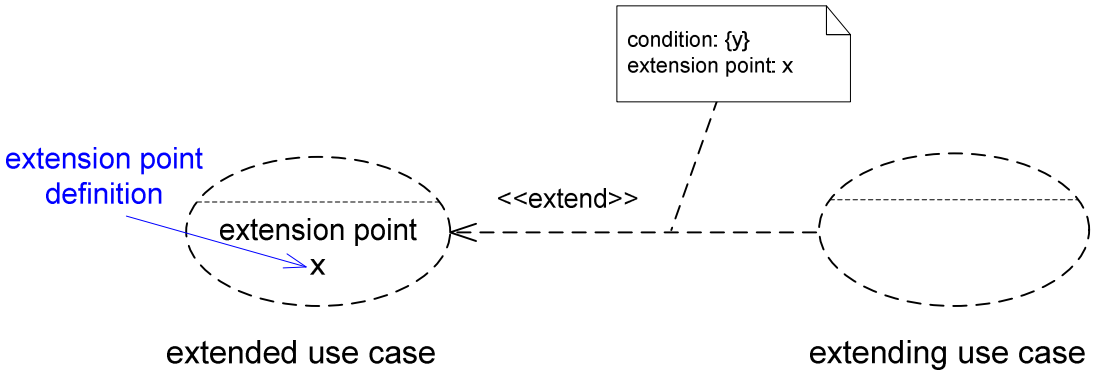


Figure 5.9: UML 2.0 *extend* relationship [OMG09]

When the *extension point* is reached in the behaviour of the extended entity, the *condition* on the extend relationship is evaluated. If it is true, then the behaviour of the extending use case is executed [RJB05].

We can comfortably say that the elegancy of *extend* relationship lies in the fact that "*the extending use case incrementally modifies the extended use case in a modular and systematic way"*. But, as we have discussed, the UML 2.0 *extend* relationship is defined for use cases in the standard. As far as modularity is concerned, UML 2.0 *extend* relationship stands in a better position as compared to the UML 2.0 generalization relationship and UML 2.0 templates. With this spirit, we are going to present *extend* relationship for UML 2.0 collaborations with additional profiling.

First, we will discuss that how to define the *extension point* for UML 2.0 collaborations. The solution is simple; we know that the behaviour of the each of the elementary collaborations can be specified by the UML 2.0 sequence diagrams. Therefore, we can define the *extension point* as the location within the behaviour sequence of the extended collaboration at which additional behaviour can be inserted. The *condition* can be defined as reception of a certain message by an entity (collaboration role).

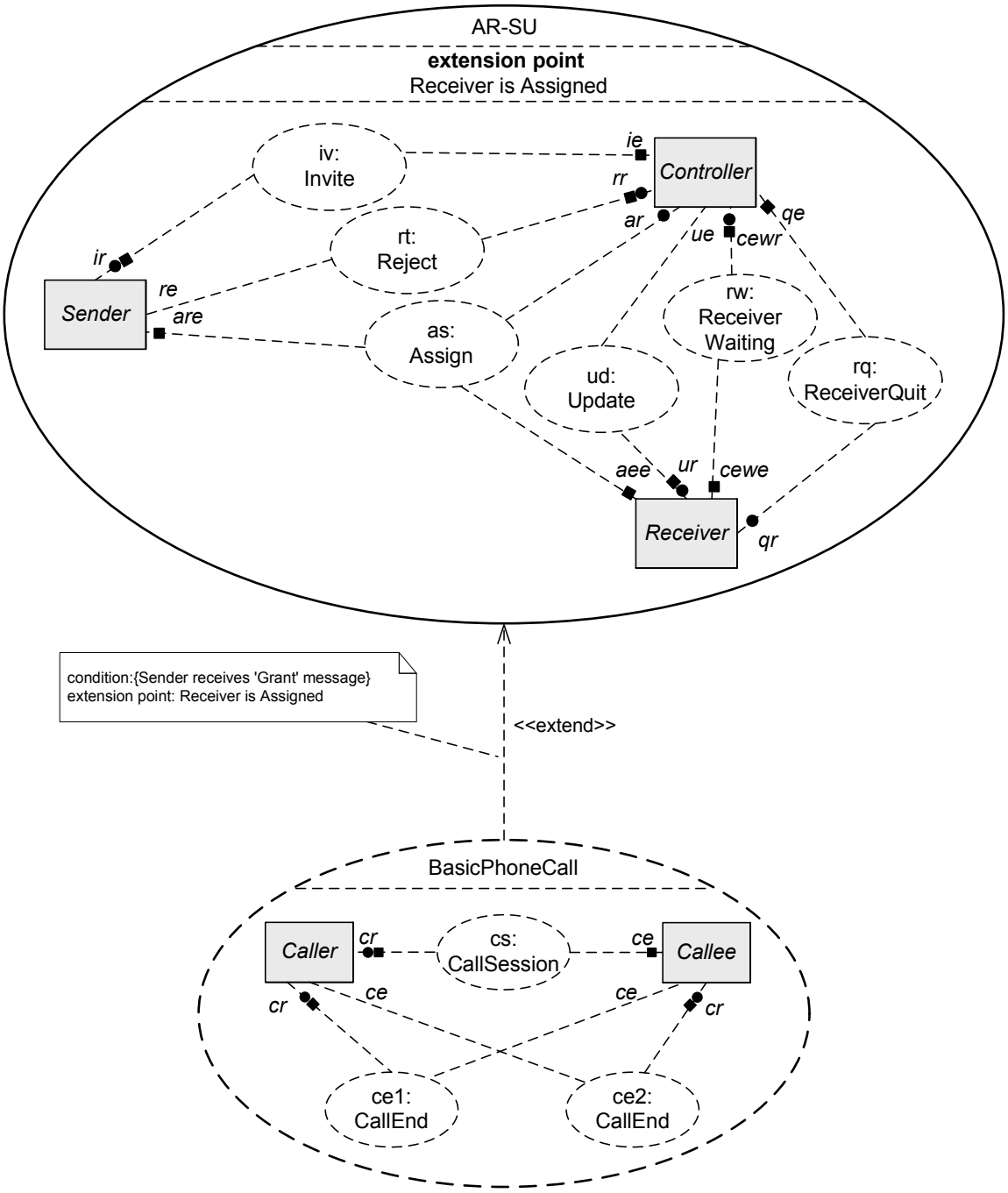


Figure 5.10: Applying coordination pattern to existing service using UML *extend* relationship

Now let us see how we can use the UML 2.0 *extend* relationship to reuse our coordination patterns in existing service models. In figure 5.10, it is illustrated how the coordination pattern AR-SU will be applied to the existing *BasicPhoneCall* service using *extend* relationship. The behaviour of the *BasicPhoneCall* service (extending collaboration) will be inserted into AR-SU coordination pattern (extended collaboration). When the *extension point (Receiver is Assigned)* is reached i.e. the Controller assigned the resource (*Receiver*) to the Sender, then the *condition* on the *extend* relationship is evaluated. The *condition* is; *Sender receives 'Grant' message* i.e. if the Sender receives *grant* message from the Controller, confirming that the *Receiver* is assigned, then the *BasicPhoneCall* service behaviour will be executed. Figure 5.11 shows the choreography of the *extend* relationship shown in figure 5.10. The choreography explains that when the *extension point* is reached in AR-SU choreography, the *condition* is evaluated which is represented by the output pin "Sender receives "Grant" message, if it is true, the choreography of *BasicPhoneCall* will be executed. If the Sender receives the "Reject" message from the Controller, then BasicPhoneCall behaviour cannot be executed.

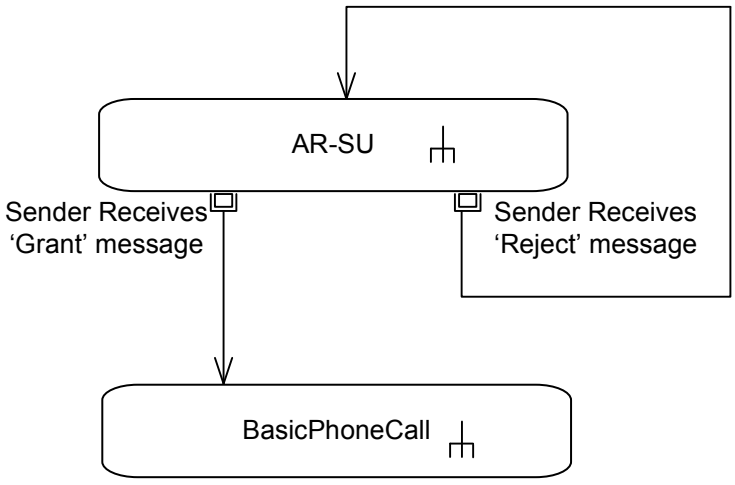


Figure 5.11: Choreography of BasicPhoneCall extending AR-SU using UML *extend* relationship. The symbol ⑂ represents a complex activity.

Pros and Cons

By using the *extend* relationship, we can conveniently apply the coordination patterns to an existing service model in a simple and modular way. Because, the only thing we have

to mention is the *extension point* (and/or *condition*) without changing or modifying anything in the system. So it assumes a simple sequential ordering.

The major drawback of this solution for applying the coordination pattern to an existing service is that we cannot ensure that the Sender role and Caller role (of *BasicPhoneCall* example) or Receiver role and Callee role (of *BasicPhoneCall* example) will be played by the same actor. Binding of the roles is not defined in *extend* relationship, since "extend" is defined for use cases in the standard.

5.2.4 By Using Service Composition

When a functionality is required that cannot be realized by the existing services, then the existing services can be combined together to fulfill the requirement [SHP03]. This is known as service composition. According to [FB03, Ros09], "*compositional design allows service developers to put service components together and reuse the individual components"*.

As said earlier in section 5.2, adding the functionality of coordination patterns to an existing service model can be addressed by the results of [Ros09] which provides the solution of our problem with service composition. The mechanism of composition of existing service model and coordination patterns is described next.

We will use the example of *BasicPhoneCall* service already under discussion. Figure 5.4 shows the UML 2.0 collaboration illustrating the *BasicPhoneCall* service. We will take the case of AR-SU pattern (shown in figure 5.2) with which we want to compose *BasicPhoneCall* service. In figure 5.12, the UML 2.0 collaboration diagram of *BasicPhoneCall(AR-SU)* service is shown. This service is a composition of the *BasicPhoneCall* and AR-SU collaborations. This new service has three composite roles Caller, Callee and Controller which have participated in the composition. The Sender, Receiver and Controller roles of AR-SU are bound to the Caller, Callee and Controller roles of *BasicPhoneCall-AR-SU* respectively. Similarly Caller and Callee roles of BasicPhoneCall are bound to the Caller and Callee roles of *BasicPhoneCall(AR-SU)* respectively. Therefore by using [Ros09] method, we can model the composition of

existing services and coordination patterns structurally. The choreography graph of the new created service *BasicPhoneCall(AR-SU)* is shown in figure 5.13.

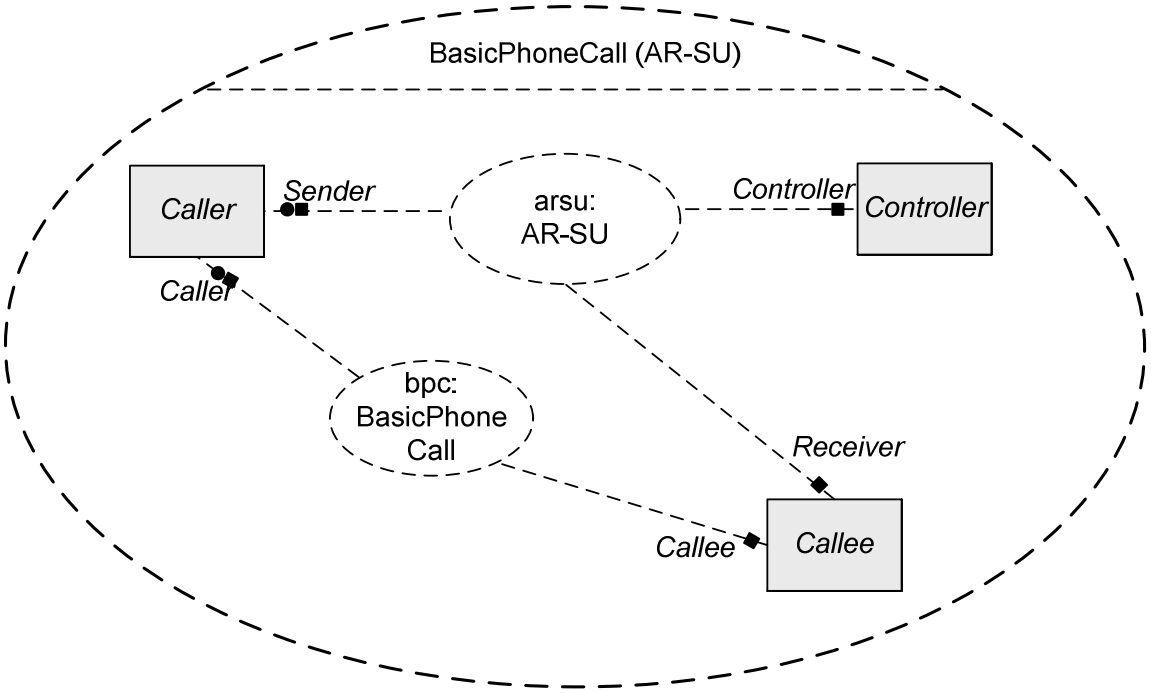


Figure 5.12: UML collaboration of *BasicPhoneCall(AR-SU);* a composition of *BasicPhoneCall* and AR-*SU* service collaborations

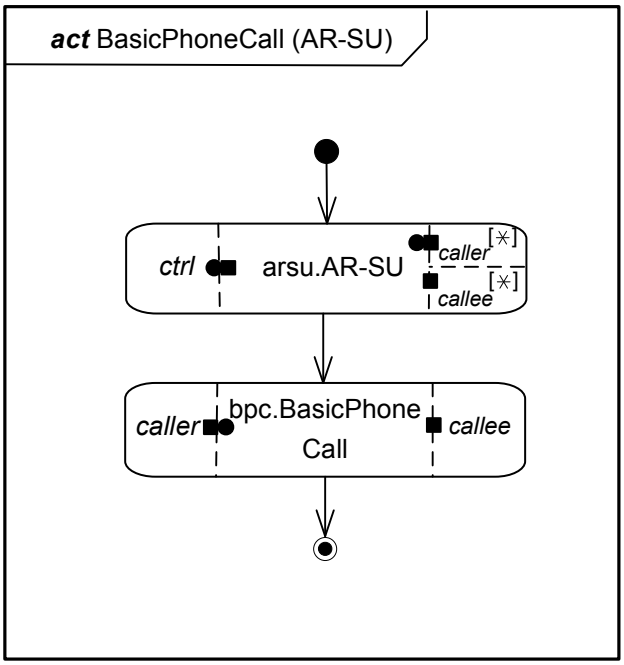


Figure 5.13: Choreography for *BasicPhoneCall (AR-SU)* collaboration

Pros and Cons

'Service composition' combines the extra functionality with an existing service model and creates a new service which contains both functionalities. The new service contains the intended behaviour. Moreover, 'service composition' is equally good as generalization relationship. It is the 'service composition' that creates new service but the drawback is that the new functionality cannot be inserted into the existing service model in a modular way. In other words, service composition is powerful but less elegant.

Cut and Paste solution

None of the four solutions we discussed, fully serve our purpose of adding extra functionality to existing services in a modular way. They have pros and cons at the same time. Generalization relationship stands in a better position among the solutions proposed in this chapter. These are the initial thoughts which can be further explored in depth. For the time being, the solution which fully serve our purpose is *Cut and Paste.* To add a new functionality into an existing service model we have to manually modify the UML collaborations and their corresponding choreography graph to achieve 100% results.

Chapter 6: Discussion and Conclusion

This chapter briefly summarizes the results achieved in this thesis. It discusses the limitations and, at the end, some future work is proposed which is based on the discussion.

6.1 Achievements

In this section, we highlight the results achieved against the tasks identified in the problem description discussed in section 3.1.

6.1.1 Identification of Coordination Patterns

An example service (*SimpleChat*) is modeled and designed using the service engineering approach proposed in [Cas08]. The service is first modeled using UML 2.0 collaborations i.e. the service is described as collaborations between roles; thereafter, the service behaviour is modeled as a choreography of sub-collaborations using UML 2.0 activity diagrams (with semantics which deviates from the standard). This service modeling is initially focused on one isolated occurrence of the service. The focus is on role behaviours. The behaviour of system component is then designed as composition of the roles it plays.

To deal with the possibility of having multiple concurrent occurrences of the service running in the system, extra coordination functionality is introduced in this thesis by defining another role which is external to the service roles and serves to coordinate role binding. This role is named as ‘Controller’. The Controller is designed to keep track of the resource status, assign the resource if it is free (i.e. not participating in another occurrence of the service collaboration), and if not then respond to the service invitation requests according to the preferences of the actors that receive them. Depending upon how the Controller performs the coordination functionality just discussed, three

coordination patterns are identified: **A**ssign and **R**eject – by **P**olling (AR-P); **A**ssign and **R**eject – by **S**tatus **U**pdate (AR-SU); **A**ssign and **W**ait – by **S**tatus **U**pdate (AW-SU).

Based on the modeling and design of our example service *SimpleChat*, the design of the Controller has been investigated and these coordination patterns have been described. Moreover, several other possibilities have been taken into account within the coordination patterns e.g. whether the Controller role is defined for each *UserAgent* (actor) or whether the Controller is defined for different *UserAgents* (actors).

The semantics of the choreography graph (proposed in [Cas08]) have also been extended to take into account the multiple concurrent occurrences of a service.

6.1.2 Applying Coordination Patterns into Existing Services

Apart from modeling the service from scratch and investigating the coordination patterns for it, it has been explored in this thesis how this coordination functionality can be added into an existing service model in a modular way. For this task, the general structure of the coordination patterns has been identified. This strengthens the generality of the coordination patterns i.e. they are not defined for a specific service but also exercised to be kept general so that they can be used in any existing service model. This exercise addresses the second requirement of the thesis i.e. *can new functionality be incrementally added to an existing service model in a modular way?*

To address the above mentioned problem, coordination functionality is considered as an additional/new functionality which we want to apply to an existing service model. Initial thoughts are presented to address this problem. This includes the concepts of: UML 2.0 generalization relationship; UML 2.0 templates; UML 2.0 *extend* relationship; and service composition [Ros09]. By using these concepts, one of the identified coordination patterns is exercised to be added into an existing service model, keeping the modularity as our first concern.

6.2 Discussion

We have discussed coordination patterns as a basic requirement for a system where components may be requested to simultaneously participate in several occurrences of a given service collaboration and also when there is contention for the actor/roles. The Controller role is the core of these patterns. It has been designed to take care of the service invitation requests and to respond to these invitation requests. If the resource is free then the Controller assigns it to the inviting entity i.e. the actor playing the Sender role. If the resource is not available (i.e. participating in another occurrence of that service collaboration), then the Controller responds to the invitation request according to the preferences of the actor that is playing the Receiver role. For this case, two major possibilities are considered: either to reject the invitation request; or to put the Sender in waiting queue. The possibility to reject the invitation request is one of the simplest approaches. It is not elegant but serves at least the purpose of handling the invitation requests in a simpler way when the resource is busy/not available. As compared to this, the second possibility, which is to put the Sender of the invitation request in a waiting queue, is a better approach to handle busy resources. The Sender is not forced to wait once it has sent the invitation request. It can opt to quit the waiting queue anytime.

In some service models, the actor playing the Sender role may want to be connected to a specific actor playing the Receiver role e.g. *PhoneCall* Service. Other services may not have this requirement e.g. *TeleConsultation* Service (where any of the available doctors can be assigned to a patient) [Cas08]. These two services have slightly different requirements. This has been addressed in this thesis by either defining the Controller role for each actor or by defining the Controller role for a set of actors.

Apart from the benefits discussed above, the coordination patterns may some times be limited to be applicable for particular situations only. For example, consider the Taxi Reservation System example in [TRS10]. This system assigns the available taxis (resources) to customers. The decision of this assignment is not only dependent on the availability of the resource (taxi) but also the location of the taxi. In our coordination patterns, the latter possibility of decision is not considered in the design of the Controller

but may easily be added by modifying the search criteria for the list of available resources (for example, by modifying *receiverId=receiverStatus(rcvrList)* in figure 5.8).

We have presented the coordination patterns with their focus on dealing with external communication i.e. between actors. However, the Controller should also be made responsible to coordinate for the internal communication within an actor. We have discussed this by considering the case in which the *SimpleChat* service can have single instance of roles (Chatter/Chattee) per *UserAgent*. Chatter is the initiating role whose execution is triggered by an external event (end-user). This event can happen while the *UserAgent* is already busy playing the role of Chatter in another *SimpleChat* occurrence. The decision of the Controller, whether to *reject* the end-user initiative or put it on *wait,* will then be dependent on *whether Chatter role already exists in UserAgent or not*.

One of the coordination patterns is AR-P (Assign and Reject - by Polling). In this pattern, the Controller polls the resource to learn about its status (free/available). When the question comes to choose among the coordination patterns identified in this thesis, then this pattern should be given the least preference to be chosen. The reason is that, in some situations, polling might not be possible. For example;

- When the Receiver role is dynamically created and thus, not possible to be polled.
- If the actor allows several service roles, then polling all of them to learn their status will overload the system with traffic.
- The decision to reject the invitation request may depend on other roles as well. (discussed in fourth para of this section).
- Waiting in a fair way will be difficult.

Moreover, ‘Polling’ destroys the elegance of the Controller and its real purpose.

We would like to mention here that the basic coordination patterns proposed in this thesis may remain the same in spite of the limitations discussed in this section. However, different decision policies can be added to the design of the Controller. These decision policies will be performed by the Controller locally. Therefore, the basic structure of the patterns will remain the same as identified in this thesis.

Coordination patterns are made general. They are not specific to any service. Therefore, some of the initial thoughts are presented in this thesis regarding how to add the coordination patterns (as a new functionality) into an existing service model, but none of the solutions proposed is completely modular. All the solutions have some drawbacks which are discussed in chapter 5. The UML 2.0 generalization relationship stands in a better position among other solutions proposed in this thesis. This can be an interesting area of further research.

6.3 Future Work

Several Controllers with different decision policies (discussed in section 6.2) can be made available in the coordination patterns for a service engineer. The service engineer will then be able to pick the Controller with that decision policy which suits his/her service requirements. Moreover, by giving different options of decision policy, he/she will be free to decide which Controller design to choose.

Apart from variation in decision policy, other features can be added to the design of the Controller in the coordination patterns. For example, security features can be added. Location awareness of the resources can be added as another feature. The coordination pattern, in which the Controller has the location awareness feature, will be able to be used for Taxi Reservation System and other similar system examples.

Besides, service engineer can be enabled to compose a new Controller design by re-using the existing Controllers with various decision policies and features. This can be done by using the basic structure of coordination patterns identified in this thesis. Moreover, other patterns can also be incorporated.

It would be interesting to explore further how a new functionality can be incrementally added to an existing service model in a modular way.

References

[BDC+89] T.F. Bowen, F.S. Dworack, C.H. Chow, N. Griffeth, G.E. Herman, and Y.J. Lin. The feature interaction problem in telecommunications systems. In *Proc. of the 7th Int'l Conf. Soft. Eng. for Telecommunications* Switching Systems *(SETSS'89)*, pages 59–62, 1989.

[BKM07] Manfred Broy, Ingolf H. Kr¨uger, and Michael Meisinger. A formal model of services. *ACM Transactions on Software Engineering and Methodology*, 16(1):5, 2007.

[Cas08] Humberto Nicolás Castejón. *Collaborations in Service Engineering: Modeling, Analysis and Execution.* PhD thesis, Department of Telematics, Norwegian University of Science and Technology, 2008.

[CBB07] Humberto Nicolás Castejón, Gregor v. Bochmann, and Rolv Bræk. Using Collaborations in the Development of Distributed Services. In *Proc. of the 14th Asia-Pacific Soft. Eng. Conf (APSEC'07)*, pages 73-80. IEEE Computer Society Press, 2007.

[CV93] E. Jane Cameron and Hugo Velthuijsen. Feature Interactions in Telecommunications Systems. *IEEE Communications Magazine*, August 1993.

[FB03] J. Floch and Rolv Bræk. Using SDL for modeling behaviour composition. In *Proc. of the 11th International SDL Forum (SDL 2003),* volume 2708 of *LNCS*, pages 36-54, 2003. Springer.

[GHJ+94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[HP85] D. Harel and A. Pnueli. *On the development of reactive systems*, volume 13 of *Nato Asi Series F: Computer And Systems Sciences*, pages 477-498, 1985. Springer.

[HPW03] Oystein Haugen, Birger Moller-Pedersen, Thomas Weigert. *UML for real: design of embedded real-time systems*, pages 53-76, Kluwer Academic Publishers, 2003.

[KBH09] Frank Alexander Kræmer, Rolv Bræk, and Peter Hermann. Compositional Service Engineering with Arctis. *Telektronikk 1*, 2009.

[KH06] Frank Alexander Kraemer and Peter Hermann. Service specification by composition of collaborations – an example. In *Proc. of the 2nd Intl. Workshop on Service Composition (SERCOMP'06)*, pages 129-133, 2006.

[KK98] Dirk O. Keck and Paul J. Kuehn. The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. *IEEE Transactions on Software Engineering*, volume 24, October 1998.

[MTJ93] John Mierop, Stefan lax, and Ronald Janmaat. Service Interaction in an Object-Oriented Environment. *IEEE Communications Magazine*, August 1993

[OMG09] Object Management Group (OMG). *UML 2.1.2 Superstructure Spec.*, February 2009.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addisen-Wesley, 2005.

[Ros09] Judith E.Y. Rossebø. *Dynamic Composition of Services – a Model – Based Approach.* PhD thesis, Department of Telematics, Norwegian University of Science and Technology, 2009.

[SHP03] Evren Sirin, James Hendler, and Bijan Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *WebSerives: Modeling, Architecture and Infrastructure workshop in International Conference on Enterprise Information Systems (ICEIS 2003)*, April 2003.

[TRS10] Taxi Reservation System. *Engineering Distributed Real Time Systems* http://www.item.ntnu.no/academics/courses/ttm4115/assignments, accessed July 21, 2010.

[Wie03] R. J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML*. Morgan Kaufmann Publishers, 2003.

[Zav01] Pamela Zave. Feature-oriented description, formal methods, and DFC. In *Proc. of the FIREworks Workshop on Language Constructs for Describing Features*, pages 11-26. Springer-Verlag, 2001.