Richard Torbjørn Sanders

Collaborations, Semantic Interfaces and Service Goals: a way forward for Service Engineering

Thesis for the degree doktor ingeniør

Trondheim, March 2007

Norwegian University of Science and Technology
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Telematics

NTNU
Innovation and Creativity

NTNU
Norwegian University of Science and Technology

Thesis for the degree doktor ingeniør

Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Telematics

© Richard Torbjørn Sanders

ISBN 978-82-471-1495-7 (printed version)
ISBN 978-82-471-1500-8 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2007:68

Printed by NTNU-trykk

Preface

A central theme of this thesis is services. In daily use we think of a service as something that an organization or system provides to the public. What “something” is can have wildly different interpretations, and whether a service is involved or not can be food for lawyers; in 2001 the US won the “banana-war” against the EU by winning the argument that providing bananas was a service, not a product, thus placing the “banana distribution service” under the GATS rules[1]. The result was that the EU banana-regime, favouring some of the poorest banana-producing countries, was ruled by the WTO to be illegal.

Banana distribution services are not the topic of this thesis; telecommunication services are. Most people take services such as telephone calls for granted. They have after all been around for over a century; most of us grew up with phones at home, now most of us carry one in our pocket as well. Exactly because of the latter - that increasingly complex machines fit into our pockets - there is a risk that the future ahead of us might turn messy. Because not all of what comes need necessarily be to the better. Much of what is commonly called progress is arguably the opposite.

In the western world we are encouraged to define ourselves in our role as consumer. We are supposed to delight in choice; between different phones with different colours and ringing tunes, between different operators, between different services like SMS and MMS. We are not supposed to long back to the days when phones were rationed, and the state ran the service with all-knowing human operators at the switchboards. But what will we do the day our mobile phone crashes while downloading some fancy context-sensitive media application at it’s own behest, so that we can’t make or receive good old telephone calls? What if the terminal runs on fuel cells so we can’t even pull the trick we resort to when our gadgets fail: pull out the plug, re-boot, and hope it works?

The context of our work is exactly such a brave new world, where our computer-controlled devices try to sort things out by themselves, based on what is going on around them. About discovering new service opportunities, and negotiating with other devices about what great things they can do together - for our sake. We don’t necessarily wish to push or pull the world in that direction; but it seems this is the way things are going, whether we like it or not. Our concern is that there will be some quality out there in the future, not just quantity of functionality, which seems to be the name of the game in other ICT fields.

1. See for instance http://www.globalexchange.org/campaigns/bananas/guardian031501.html

Our contribution aims at ensuring at least some level of quality in the services we one day will use. Just like good old phones used to work 99,999% of the time, so should our devices in the future. We hope our ideas will contribute to this to some degree.

We believe this thesis suggests things that can bring the world a little further. If it does, it does so by “standing on the shoulders of giants”[2]. One of these giants is my advisor, Rolv Bræk, whom I have known and admired since I in 1985 became acquainted with his outstanding work with Structure-Oriented Modelling - SOM. I still admire him very much; his ability to recognise and extract the essence when all seems chaotic, his mild stubbornness, and his gift for expressing his thoughts in pictures like no other person I know.

Another shoulder upon which I stand firmly is my good friend and colleague for many years, Jacqueline Floch. She showed the way, by commencing on and completing her own PhD, and by doing such excellent work. I have relied heavily on the foundation she has laid; I think it is solid, and my somewhat more esoteric work remains grounded partly thanks to her. In addition she has been a great partner in discussions about the subject matter - and a wonderful person to have as a friend.

The SIMS project I am currently leading is the latest in a series of interesting projects that have contributed with relevant and interesting discussions. Other projects include *Avantel*, with Ericsson and Telenor, the *SISU* project that produced *TIMe - The Integrated Methodology,* and the *Norse* project that started while I was at Stentofon. Many of these projects have revolved around the same set of people. Among them I in particular extend sincere thanks to Birger Møller-Pedersen and Øystein Haugen, most recently for discussions about the details in UML 2.0 - and of course for the wonderful scientific work they have put into TIMe, SDL, MSC and UML. The same goes to Geir Melby, who has been instrumental in making things happen since SISU started in 1988.

I thank my fellow doctoral students at the Department of Telematics; Humberto and Frank have particularly been good discussion partners, while Shanshan, Fritjof, Haldor and Arne Lie have been great company, as have the other colleagues at the department. Cyril Carrez, whom I got to know while in Paris and who later joined NTNU, provided me with lots of comments, and scrutinized my work meticulously. Fancy a Frenchman correcting the language of a semi-native Englishman! Thanks, Cyril!

Of course it took a pair of true Englishmen to correct the finer language details; thanks to Norman and Anne for their meticulous work.

I thank the members of the SIMS project, in particular those who initially bought into the ideas we presented to them in the late Autumn of 2004, and to those who have grappled with the concepts since - largely based on this unpublished PhD thesis.

I must also thank The Money. It was Telenor that financed the PhD grant and half of the additional expenses for my stay in Paris, so thanks to Oddvar Hesjedal and Oddvar Risnes. Thanks to SINTEF for their generosity in keeping me up to my SINTEF salary, and for their flexibility during my stay in Paris. In particular thanks to Eldfrid Øfsti Øvstedal, who supported me all along. Also to Tore Dybå for pioneering the generous SINTEF PhD deals, and for being my first role model. Thanks to the Department of

2. As Isaac Newton wrote in a letter to his colleague Robert Hooke in 1676

Telematics; first to Peder Emstad for giving me the best financial offer one could get, to Randi Flønes for making difficult things easy, to the boys (Asbjørn and Pål) for offering good computing services, and to all the other friendly people who work at the department.

I would also like to thank the Machine; by that I mean the inner workings of NTNU and of SINTEF, for doing their job without me noticing it, so I could get on with mine.

In particular I thank Elie Najm for being such a wonderful person, for inviting me to ENST Paris, for lending me much of his office space for what turned out to be 9 months. Thanks for the fun discussions about modelling, politics, war and religion. Thanks also to friends and colleagues there, Arnaud Bailly in particular, but also Cyril - and for their role (sic) in bringing Happy Hour - le “Zappy” - to Trondheim.

Lastly I wish to thank Jorunn for her support at a crucial point when I couldn’t begin writing, and my children, Ingeborg and Andreas, for offering encouraging cheers when I had a good day, and comforting compassion on more useless days, of which there were many.

It has been said that the average thesis gets read 2,5 times. That supposedly includes the committee. Most of my friends and colleagues will of course not read it, even though they in due respect and friendship might file it in their bookshelf. They may even have read this far, before giving up on understanding “magic interfaces” and “service trolls”. I will not be insulted if they do not read further. But I do hope that some of the ideas herein are taken up in some way by future ICT practitioners. Not that I think that ours should become a household word (“My new phone is Sanders’ enabled!”), but because we believe the core ideas can make some things a little better. However, we are aware that there is much more left to be done. Our contribution is only a fraction what is needed to make future communication services work seamlessly across network and service provider boundaries.

Trondheim, March 2007

Richard Torbjørn Sanders

Abstract

The challenges faced when designing and deploying convergent telecommunication services are well know to practitioners. While good tools and methods are available for designing, implementing and testing system parts that run on a single computer, there is a lot to be desired when it comes to the cross-cutting aspects of services that require cooperation between several software components distributed over several machines.

Our work addresses the latter issues.

Firstly it defines services in a more general way than much of the current thinking, exemplified by Service Oriented Architecture; we consider a service to involve a collaboration between objects, rather than an interface to an object. This means that a larger set of problems can be addressed by services, due to a more general definition.

Secondly, we present the core concept of *semantic connectors* as a reusable modelling construct. Semantic connectors, and the pair of *semantic interfaces* they define, are used as elements in the composition of more complex services, without any limitations regarding the number of objects participating in the service.

Semantic connectors can be designed and validated as separate entities. Semantic interfaces are small state machines, and opposite semantic interfaces must adhere to *basic safety properties*, entailing that all output from one interface must be accepted as input in the opposite interface, and vice-versa, states taken into consideration.

The “semantic” aspect of semantic connectors is expressed by what we call *service goals*, which characterize events or states that are desirable to achieve over the semantic connector. Service goals are used to check whether collaborations between objects can achieve their intentions; if one can prove that service goals never can be achieved, then the collaboration has no useful purpose. We call this a validation of *basic liveness properties*, which comes as an add-on to the validation of basic safety properties.

Furthermore, the behaviour of services composed of semantic interfaces can be characterised by what we call *goal sequences*. These provide an overview of the service logic, focusing on the achievement of goals of the constituent semantic connectors. *Collaboration goal sequences* facilitate the validation of composite service structures, and can be used to derive *role goal sequences* that describe relationships between goals of semantic interfaces. *Actor goal sequences* describe the ability of objects to play semantic interfaces, taking sequences of goals into consideration.

In addition to a compositional approach to service design and validation, our work also shows how semantic interfaces can facilitate service discovery at runtime. By comparing semantic interfaces and actor goal sequences of potentially connected actors, it is possible to determine what *services opportunities* are available between peers, both in general and at any specific time, i.e. dependent on the current context. Context dependent service discovery is expected to increase in importance as the number and diversity of terminal types, signalling and transport networks, and service providers increase.

Our work builds upon the work of other researchers, but represents a new approach. To the best of our knowledge, our suggestions are new and original. Hence there is little to compare with in terms of other initiatives, although there is a large volume of work that constitutes our point of departure.

We use the second version of the Unified Modeling Language, UML 2, as a modelling language. We make original and innovative use of some of the new constructs in UML, in particular *collaborations* and *collaboration uses,* as well as *interaction overviews*. This may be of interest to the UML community at large, not only to designers of convergent services. The ability to model cross-cutting concerns using UML is of general importance.

Finally, we note that this work is being used as a basis for ongoing research and evaluation work, most notably by the SIMS project funded by the European Commission, and by a number of doctoral students at the Norwegian University of Science and Technology.

Table of contents

List of figures

List of definitions

List of method rules

1
Introduction

1.1 Motivation and background

The motivation behind this thesis is twofold, and has been a driving force throughout our career as consultant, designer, researcher and teacher.

One is to come to grips with the essence of the problems we are trying to solve when we design and deploy systems. Without a profound understanding of this, we are likely to create new problems, not solve the existing. For instance, the essential needs of a user of a service may be to get in contact with a doctor. In the days of manually switched exchanges you could simply pick up any phone and ask for the doctor - and get connected[1]. Perhaps one day we will be able to do that again.

Modelling the essential services and needs rather than the accidental technology of handsets, phones and networks is a motivating force. This is why we have sought *high-level descriptions of services, focusing on the collaborating parties and their goals*, rather than dwelling on the technology used to perform services.

Another motivation has been a constant striving towards more professional development processes. Good systems design is still too much of an art, and not enough of a craft. We have constantly been on the lookout for better tools, better methods, better ways of tackling the design process, building quality into product designs from the beginning.

Furthermore, we acknowledge that the technological advances imply that more control over services will take place on networked terminals rather than on centralized switches. Hence we have sought mechanisms that *enable devices to discover and validate service opportunities on a peer-to-peer basis, possibly on-demand.* We hope to contribute to retaining the time-proven quality level of telecom services even in a future ad hoc setting.

1.1.1 Experience from the battlefield of service development

The main motivation for our work stems from experiencing the complexity of designing state machines capable of coordinating conflicting initiatives from multiple sources. This is the nature of systems design experienced by designers of real-time systems rich on behaviour - exemplified by the intercom exchanges we worked on at Stentofon.

1. The operator might know who and where your doctor was, and would connect you to that extension.

Designing complex state machines that coordinate multiple initiatives is so complex that it challenges the limitations of the human brain. It is no easy task to solve such challenges - one day you may have a clear understanding of the parallel tasks being controlled by the state machine, but when you return half a year later even you yourself can make fatal mistakes when altering or adding a small function. Not to speak of the risks involved in subjecting it to a novice designer inexperienced in the design of such systems.

Attempting to divide and conquer this challenge is what lies behind the factoring out of interactions in *semantic connectors*, and (re)using them in composite structures.

An observation made while working as a systems designer at Stentofon was that of the market people negotiating service functionality with customers, while struggling with the service designers with their own ideas of customer needs. Could there be a better way of making conceptual models of services, consisting of modules that could be put together in different ways? And described in a way that customers, market people and designers can agree on? These considerations lie behind *service goals* and *goal sequences* to express relationships between the semantic connectors. In addition, it is desirable to validate that components indeed can reach goals when interacting.

1.1.2 Inspiration from the world wide web paradigm

Telecom systems have traditionally been statically designed, with detailed knowledge about devices, networks and basic services deeply embedded in the architecture and code of the systems. Even the Intelligent Networks architecture has a built-in Basic Call Model, plus pre-defined trigger points that dictate much of the functional possibilities for subsequent tailoring of services.

In the world wide web, the client-server model is based on a “dumb” browser client not really knowing much about the web sites the user visits. Here the server provides the content that determines the subsequent operation of the client. This means that the server at any time can be changed, without having to worry about what clients have down-loaded before; the next time the client accesses the server, they will get the new functionality.

We have found inspiration from this. We envision mechanisms to discover *service opportunities* on the fly, as they arise. And, taking this a step further, seeking to upgrade terminal functionality while in use, i.e. *role learning*. Thus, for instance, new service capabilities in a callee’s terminal can result in new features being made available to the caller. Such “automated” deployment of services is not supported by traditional telecom systems.

1.1.3 Inspiration from earlier research work

For many years we have had the privilege of working with very skilled researchers; people who have created new system design languages, methodologies and tools, people who have been capable of distinguishing between the essential problems and the accidental shortcomings of technology.

Most important when starting this work was the contribution towards plug-and-play services of Jacqueline Floch. The validation mechanisms of her thesis [Floch 2003] address safety properties, ensuring that errors will not occur when peers interact. We saw that role validation could be enhanced to include basic liveness properties, i.e. validating that something good should be achievable as a result of the cooperation between peers. This lead to our contribution of *progress labels* and consequentially to the formulation of *semantic connectors* with *semantic interfaces*.

The modelling of services using UML2 collaborations was suggested earlier by Øystein Haugen and Birger Møller-Pedersen [ARTS 2003]. Their initial verdict was not without reservations, as the binding of roles to parts of classifiers has some restrictions. However, the shortcomings of our initial attempt at modelling services using UML association classes forced us to reconsider the use of UML2 collaborations. The work of Haugen and Møller-Pedersen inspired us to pursue the matter further.

However, our greatest inspiration has been provided by the systems engineering perspective of Rolv Bræk. His view on role modelling [Bræk 1999] underpins our approach. In addition, state orientation, which stems from the early days of structure-oriented modelling [SOM 1981], was a particular inspiration: could this be put to renewed use in modelling and validation? We have adopted some of this thinking in our suggestions for modelling collaboration behaviour.

1.2 The client-server versus the peer-to-peer paradigm

Some essential characteristics of peer-to-peer systems in the telecom domain compared to client-server systems in the computing domain need to be recognized in order to appreciate the problems we are addressing, and to put our contributions into perspective. There are fundamental differences between their communication models and service concept.

Many practitioners within the computing domain consider systems and services according to the client-server paradigm. Such systems are characterized by passive objects responding to operations, communication by procedure calls, and one-way interfaces supporting request-response interactions, as illustrated in Figure 1.1.

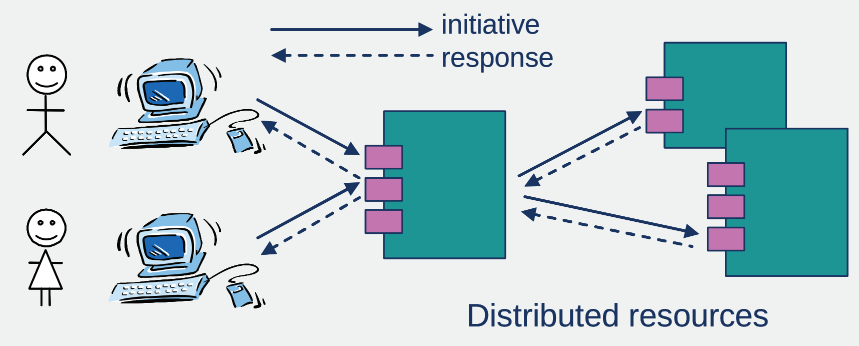


Figure 1.1 : The client-server paradigm [Bræk and Floch 2004]

Essentially, only one side of the communication will ever take the initiative to communicate. This is the paradigm behind CORBA and, more recently, Web-services and what has been coined as *service-oriented computing* or a *service-oriented architecture* (SOA).

Telecommunication systems, on the other hand, are characterized by the peer-to-peer paradigm, where active objects with collaborating behaviour communicate via signals over two-way interfaces in symmetrical interactions, where communication initiatives may be taken simultaneously from several directions, see Figure 1.2.

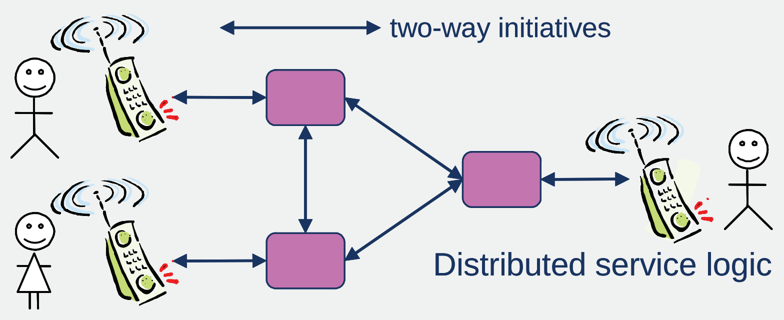


Figure 1.2 : The peer-to-peer paradigm [Bræk and Floch 2004]

The behavioural complexity of peer-to-peer systems is by nature greater than that of client-server systems, as simultaneous and possibly conflicting initiatives must be handled by each peer.

While ICT convergence is gradually bridging these views, it can be argued that peer-to-peer is the most general paradigm, while the client-server paradigm can be seen as a special case [Bræk and Floch 2004].

Services in computing systems are often viewed as a computation or information processing operation accessed via an interface. Such a service is provided by an object or component. This is the principle underpinning service-oriented computing, see e.g. [Singh and Huhns 2005][2].

In contrast to this, the peer-to-peer services of the telecom domain result from the collaboration between several actors (active objects or components), see Figure 1.3.

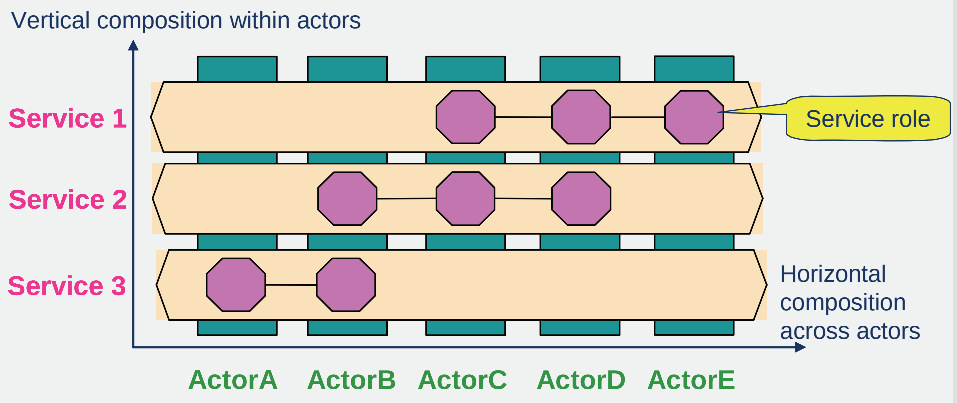

Vertical composition within actors
Service 1
Service 2
Service 3
Service role
Horizontal composition across actors
ActorA
ActorB
ActorC
ActorD
ActorE

Figure 1.3 : Horizontal and vertical composition of services [Bræk 2004]

2. However, even within this domain the definition of the term *service* is not unanimous, see e.g. [Jones 2005].

We address convergent services. Such services combine communication control services with aspects of information services. We do not specifically address application services in the computing domain, nor transport services within the telecom domain.

We consider a service to be a collaboration between service roles played by actors. Complex convergent services typically involve more than two actors. An actor is typically capable of playing several service roles, both simultaneously and/or alternately. Providing a service means executing a successful collaboration between roles.

In this problem area there are a number of well-known issues.

- Telecommunication services are becoming increasingly complex as network and terminal diversity increases, and as more information aspects are added. There are greater expectations from users and service providers for services to offer personalization and context awareness, and to operate across network and service provider boundaries. Hence it is desirable that services are well-formulated and well-understood by all stakeholders. Formal descriptions that readily lend themselves to human understanding are a key to this end. Formal validation is also increasingly important to achieve service quality, but easy-to-use validation tools seem to be out of reach for many practitioners;
- The languages and techniques traditionally used to design telecom systems focus on describing protocols and control behaviour by means of state machines and asynchronous signalling. However, systems and service modelling at the requirement stage tend to use textual descriptions with informal diagrams, supplemented by informal models such as Use Case diagrams. Added precision is possible using activity diagrams and sequence diagrams. A challenge remains to find better ways to model services separately in ways that are precise, readable and compositional, i.e. supportive of the succeeding design and composition of actors (system components);
- There are also growing expectations from users and network operators that services should be discovered dynamically on-the-fly as soon as they are deployed, implying that new functionality is propagated to users in a semi-automated, ad hoc fashion.

1.3 Research problems

Several central issues have been addressed by our work:

- The modelling of services, including the modelling of service roles and service goals;
- The validation of services, with a focus on modular validation of units of service structures and service behaviour;
- Service discovery of convergent services.

Below we briefly outline each problem area.

1.3.1 Modelling services

Telecom services are characterized by complex reactive behaviour. Capturing this behaviour in models has been a challenge for several decades, and explains the advent of modelling languages and methods in the 70's[3]. While modelling structure is established in many fields of engineering and architecture, formal models of behaviour are few and mostly leave a lot to be desired in terms of expressive power and validation opportunities.

One challenge is to provide behaviour models that are easy to understand for humans, and easy to analyse by machines. A second challenge is to describe service behaviour without binding the system design and implementation unduly.

Prototyping and simulation are techniques that help people to understand the consequences of system solutions prior to their complete construction. This applies both to ICT systems and to systems in general. But such mock-ups only give a flavour of certain aspects, and generally do not lend themselves to formal validation.

Inherent to the problem lies the fact that two dimensions must be grasped.

1. One is the interactions between cooperating objects, and the relationships between these interactions. Present (and thus future) behaviour is a function of what has already happened, and the combined behaviour of cooperating entities must be captured and understood. The cross-cutting relationships between cooperating objects and their interactions are what we choose to call the *horizontal dimension* in Figure 1.3.
2. The other dimension is the composition of the objects that take part in the interactions; the *vertical dimension*. Systems are built from objects, and it is they that perform the actions of the system. So understanding and modelling them is always of great importance. The challenge is that the focus on object design takes attention away from the horizontal dimension, and one quickly loses sight of the environment in which the object acts. This often leads to errors and misunderstandings about the role that an object plays in relation to other objects.

There is a large body of work intended to help stakeholders and designers address these modelling dimensions, such as the languages in the ITU-T family, which include:

- User Requirements Notations (URN) [GRL 2003], [UCM 2003] for expressing requirements at a high level;
- Specification and Description Language [SDL-2000] and Message Sequence Charts [MSC 2004] for describing structure, behaviour and interactions between systems and components;
- extended Object Definition Language [eODL 2003] for describing the architecture and deployment of distributed system components;
- Test and Test Control Notation [TTCN 2003] for describing test suites.

These languages have matured considerably since the early beginnings [Telektronikk 4/2000]. Methodologies (such as [SDL Method 1997], [TIMe 1999]) and tools (e.g. [Telelogic], [Cinderella]) are available.

Recently the Unified Modeling Language (UML) has emerged as a leading modelling language. UML2 [UML 2.0] has included most of the expressive power of SDL and MSC,

3. The first version of recommendation Z.100 Specification and Description Language came in 1976. Methodologies like SOM were used in leading edge industrial projects from the early 1970s [Bræk 1977, 1979], applying the principles of finite state machines [Hennie 1968] to communication control systems.

albeit without the same degree of formal definition of the semantics[4]. Tools and methodologies for UML2 are expected to be found in abundance, considering that UML has obtained such a dominating position as a modelling language in the IT community.

However, good models that address both the horizontal and vertical dimensions described above are hard to come by. Informal textual descriptions and diagrams are still commonplace in the ICT domain, even in recent standardisation work (e.g. [TIPHON 2003][5]). Of the ITU-T languages listed above, for instance MSC focuses specifically on the horizontal dimension, while SDL has its strong points on the vertical axis; the combination of the two leaves many things to be desired, especially their relationship, for example what is required for an SDL system to be compatible with an MSC.

There clearly is a need to address how services best can be modelled to span both dimensions, and, given the current trends, investigate how this can be done using UML, if at all. One of our goals is to explore whether new and better ways of expressing the essential properties of convergent services are enabled by the recent advances in UML2.

1.3.2 Role modelling

The concept of describing functional properties in roles was suggested in [OORASS 1992], [OOram 1995], and was partly covered by classifier roles in UML1. A compositional approach to roles was outlined in [TIMe 1999], describing role behaviours and indicating how class behaviours could be composed of role behaviours.

[TIMe 1999] and [Bræk 1999] are points of departure for our work on services and role modelling. They define terms such as *service roles*, *session roles* and *projection roles*, and their relationships to type modelling. [Bræk 1999] clarified important shortcomings of UML1, such as its inability to treat role descriptions (classifier roles) in a general way, independently of type descriptions or classifiers (which UML1 required that they were bound to), and the absence of two-way signal exchange in UML interfaces. The former is addressed by UML2 collaborations, while the latter is as yet still missing, as we shall see.

It is desirable that roles should be described and analysed separately from actors. Actors can then be assigned roles depending on what services they take part in, see Figure 1.4[6].

[Floch 2003] used SDL for formal modelling of behaviour, but had an informal description of structure. We seek to express horizontal and vertical relationships between roles using UML2. We know from [Bræk 1999] that UML1 did not provide support for role behaviour on interfaces. We seek to explore other ways of describing interface behaviour in UML, investigating whether UML2 offers new possibilities.

4. In UML the semantics is defined in prose form, with some additional constraints expressed in the Object Constrain Language [OCL 2.0] and many semantic variation points. It lacks the formal semantics of SDL [SDL Semantics] or MSC [MSC Semantics].
5. TIPHON (Telecommunications and Internet Protocol Harmonisation Over Networks) uses MSC and SDL to define functional entities, but the overall architectural service models are informal diagrams supported by text.
6. This is an informal diagram, meaning it does not use UML. It is adapted from [Floch 2003].

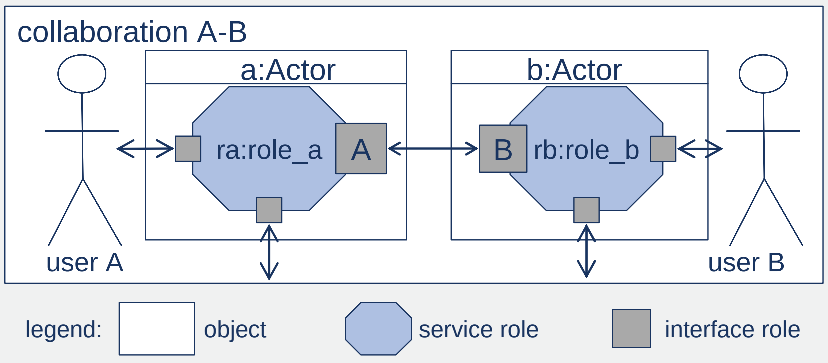


Figure 1.4 : Services as collaborations between service roles played by actors

1.3.3 Service goals and goal sequences

Goals can be used to capture the various objectives a system or component should achieve. Expressing goals is well known in the domain of requirements engineering [Lamsweerde 2001]. However, existing approaches to goal orientation do not seem to be concerned with interaction behaviour between distributed cooperating components. We seek to find ways of expressing *service goals*, as well as methods of validating whether service goals can be met during interactions between cooperating objects.

Service goals identify desirable states or events. They characterize whether a system, a component or an interaction is useful or not. In addition comes the need to express relationships between services goals, recognizing that the achievement of preceding goals can make succeeding goals achievable. This is what we call *goal sequences*.

1.3.4 Validation

Formal validation of communicating systems has a long tradition in the telecommunication domain, and is witnessed by the efforts put into the formal semantics of modelling languages [SDL Semantics] [MSC Semantics] and validation tools like the SDL Validator [Telelogic]. Formal model checking is also a mature field, and has resulted in specific languages and tools like Promela and SPIN [Holzmann 1991, 2003].

A limitation of these approaches is that they are not incremental, but must be repeated for each system composition. In addition validating real systems is often infeasible due to the state explosion problem. The standard solution to this is to build validation models for selected aspects of a complete system in order to simplify the analysis.

[Floch 2003] and [Floch and Bræk 2003a] contributed to improve this, and went into detail on modelling *service roles* in SDL 2000, the projection of service roles into *projection roles*, and a constructive and corrective approach to role validation. While this approach validates basic safety properties, i.e. checking whether errors occur, it does not address basic liveness properties, that is checking whether something useful can happen in a cooperation. Thus there is a need to add the possibility of expressing and validating basic liveness properties in the same framework.

Liveness properties can for instance be expressed in Promela and checked by SPIN. However, in addition to the state explosion problem, such languages and tools need dedicated experts to formulate the models and interpret the results. One of the strong points of [Floch 2003] is that the validation techniques do not require a separate model or modelling language, but can use the design language commonly used, which in her case was SDL. Analysis does not entirely depend on tools, as with traditional model checking, but can be performed by any skilled SDL designer, thanks to the focus on interfaces and projections.

1.3.5 Service discovery

Service discovery is a topic receiving due attention from the ICT community. Frameworks and protocols for service discovery have been defined in well-established technologies, such as CORBA [CORBA 2001], as well as in newer and emerging technologies, such as IETF's Service Location Protocol [SLP 1999, 2002], Bluetooth's Service Discovery Protocol (SDP), Parlay [OSA 2003], [JINI 2004] and Web services with its Universal Description, Discovery and Integration protocol [UDDI 2004], [JAIN 2004].

However, most of the contributions and discussions in this area relate to discovery of client-server services. According to that paradigm, service discovery entails finding which servers are available to perform what services, and a service is typically defined by a remote procedure signature. Through service discovery, clients can find both what services are available (in terms of signatures and service capabilities), and where to direct service requests, since servers may be distributed. The mechanisms involved typically require servers to register their services and capabilities at so-called lookup servers.

In the peer-to-peer paradigm, the challenge is profoundly different. An initiator of a telephone call normally wants to call a specific peer, and is not looking for a server that is capable of performing some general computation service. The initiator wants to know what peers can be reached using which service. A peer is not just a record in a distributed database, like a passenger seat in an aeroplane flight, but also an object with substance and behaviour of its own. It is the peer's potential behaviour that is of interest for service discovery, for instance if it is capable of accepting an incoming call or an instant message, both as a general capability at some time, and as a specific capability at the present time.

How can entities with a desire to communicate with each other determine in advance if they are capable of achieving goals while interacting? Static interfaces can be compared, but we also need to analyse their possible behaviour. Are requested and provided behaviour compatible? Given that service opportunities come and go depending on a peer's context and state, how does this influence service discovery? Are there ways of learning service opportunities on the fly? These are some of the questions we seek answers to.

1.4 Introduction to the approach

In this section we give a brief introduction to the main elements of our approach. Our intention is to give the reader an overview, prior to presenting the details. Hence in the following we do not precisely define any of the terms used.

As shall be seen, the approach consists of a number of techniques:

1. Modelling *semantic connectors* using UML2 *collaborations* with a pair of roles called *semantic interfaces* whose behaviour is described by state machines;
2. Modelling composite services by binding semantic interfaces to *service roles* using UML2 *collaboration uses*;
3. Modelling computational objects (*actors*) by binding service roles to them;
4. Expressing *service goals* for collaborations, service roles and semantic interfaces;
5. Modelling *goal sequences* for collaborations, roles and actors;
6. Validating services, which entails validating the *basic safety and liveness properties* of collaborations, roles and actors; and
7. Service discovery, meaning *discovering compatible actors* and *service opportunities*, as well as *role learning*.

In the following subsections we give a short outline of each technique.

1.4.1 Semantic connectors, semantic interfaces and goals

Semantic connectors and their semantic interfaces form the building blocks for compositional service design, and play a central part in goal sequences, service validation and service discovery. Semantic connectors are used to express the *specified* interface behaviour of actors and/or service roles. In contrast, projection roles express *actual* interface behaviour of actors and/or service roles.

Semantic connectors are modelled in UML2 using elementary collaborations, i.e. UML2 collaborations with two and only two collaboration roles, see Figure 1.5.

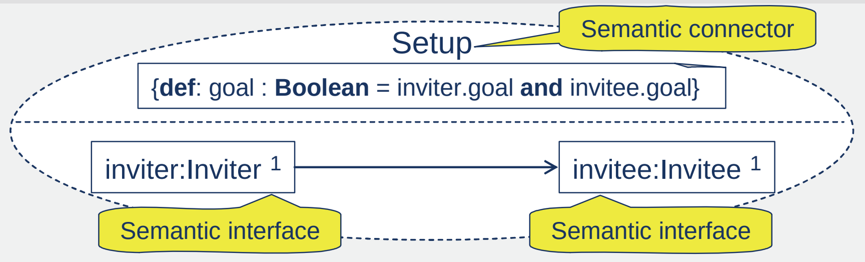


Figure 1.5 : Semantic connector and a pair of semantic interfaces

A semantic connector has a dual pair of semantic interfaces, each defining one end of the semantic connector. A semantic interface comprises a role name, a role type and role goals. A collaboration goal can be defined by a goal expression, i.e. a property condition that characterizes the goal of the semantic connector as a whole. In Figure 1.5, for instance, the collaboration goal is expressed as the conjunction of two role goals.

Role goals are represented by boolean attributes in the scope of the semantic interfaces. The semantic interfaces define their interface behaviour in UML using a form of state machine diagrams that we call *extended protocol state machines*, see Figure 1.6.

The interface behaviour includes assignments of goal values, and can specify event goals and state-like goals. Event goals are marked by progress labels, while state-like goals are expressed by goal assertions, as in Figure 1.6.

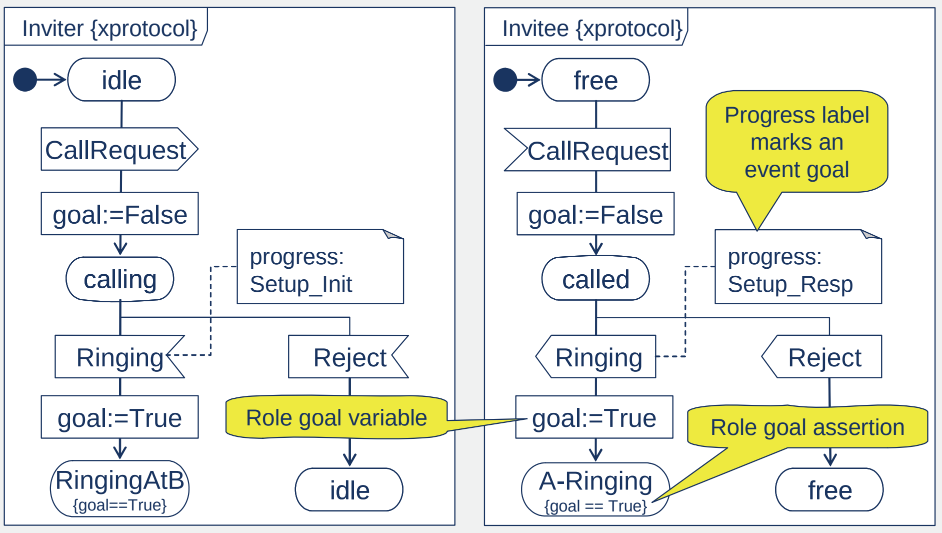


Figure 1.6 : Interface behaviour of a pair of semantic interfaces - with role goals

Note that a semantic interface does not define complete object behaviour; it describes interface behaviour, i.e. the sequence of input and output events over a semantic connector. In particular, a semantic interface does not define *why* events are output; this is instead described by service roles.

More complex services, i.e. with multiple goals and multiple roles, are composed of semantic connectors. This is achieved by binding the roles of the semantic connectors to service roles in UML2 collaboration uses, see Figure 1.7.

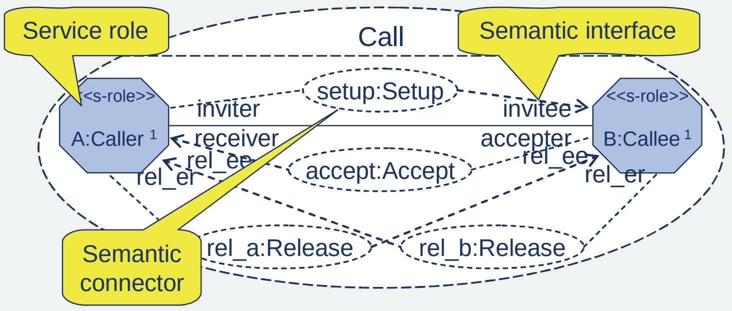


Figure 1.7 : Call *composed of semantic connectors*

A composite collaboration identifies the service roles of the composite service, and which semantic interfaces of the semantic connectors are bound to them. We represent service roles with octagons to differentiate them graphically from interface roles.

In Figure 1.7 we see for instance that the four semantic interfaces *inviter*, *receiver*, *rel_ee* and *rel_er*, which are defined by the three semantic connectors *Setup*, *Accept* and *Release*, are bound to the service role *A*. In fact, *A* can play both the roles defined by *Release*. Similar role bindings are defined for the service role *B*.

UML requires that the role bindings are compatible: in our terms, the classifiers must be *compliant with* the role types bound to them. Compatibility in role binding is a semantic variation point in UML2. In our approach, compliance means:

- An actor must accept all the input specified by the bound role, however it can accept more;
- It can provide less output compared to the bound role, but enough to enable basic liveness;
- It must not provide more output compared to the bound role when playing with an actor that is compliant with the opposite semantic interface.

We say an actor or a service role is *compliant* with a semantic interface if the projection of its behaviour on the semantic connector is a *live subtype* of the interface behaviour of the semantic interface. This entails providing enough output to achieve goals (hence *live*).

We do not require that projected behaviours are identical to the behaviour of the bound role types. For instance, it suffices that the projection of *A* on the connection represented by *setup* must be a live subtype of the role type of *inviter* in Figure 1.7, i.e. a live subtype of the *Inviter* in Figure 1.6.

Additional properties of a service can be modelled in collaboration diagrams, see Figure 1.8, as well as in other diagrams.

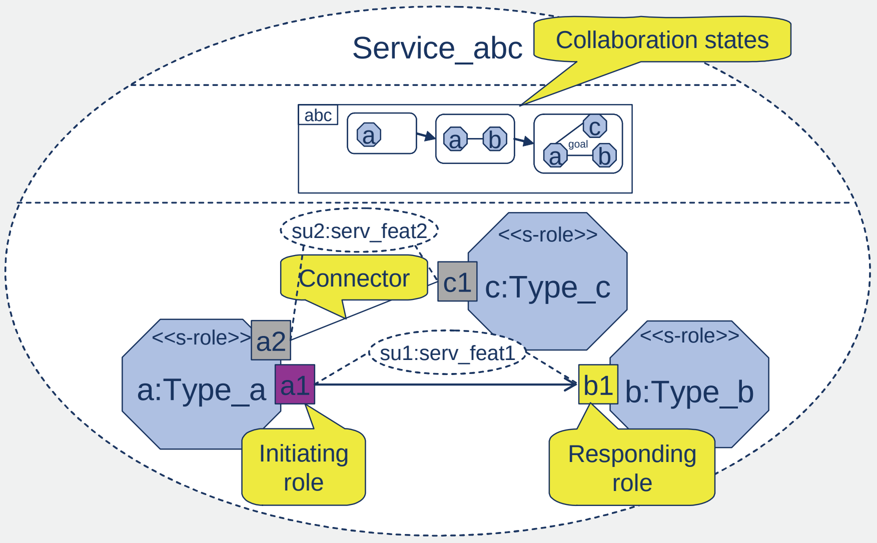


Figure 1.8 : Modelling details of services, roles and goals

Examples of additional properties include:

- modelling the connections between service roles as connectors; on connector ends rectangular icons can optionally be used to highlight the interface role names;
- distinguishing between initiating role and responding role to show which can take a first initiative. This is shown by dark and light colouring of connector end icons respectively, and/or as an arrow head on the connector end or the role binding;

- defining collaboration states in state machine diagrams, possibly referring to the appropriate states of the role types, and optionally defining collaboration goals;
- defining collaboration interactions in sequence diagrams to express collaboration goals and/or role goals; see Figure 1.9. As we shall see, goal assertions in sequence diagrams are closely related to progress labels found in semantic interfaces.

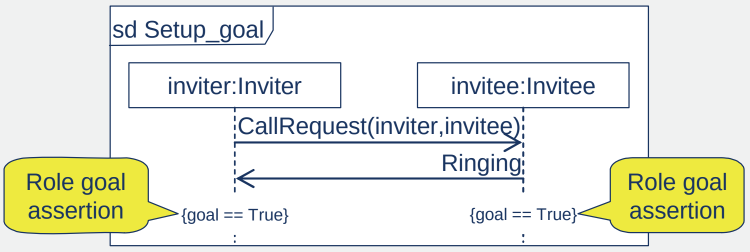


Figure 1.9 : Service interactions and role goals

UML2 collaboration uses are used to bind service roles to actors[7], see Figure 1.10. This is analogous to composing services from semantic connectors by binding semantic interfaces to service roles.

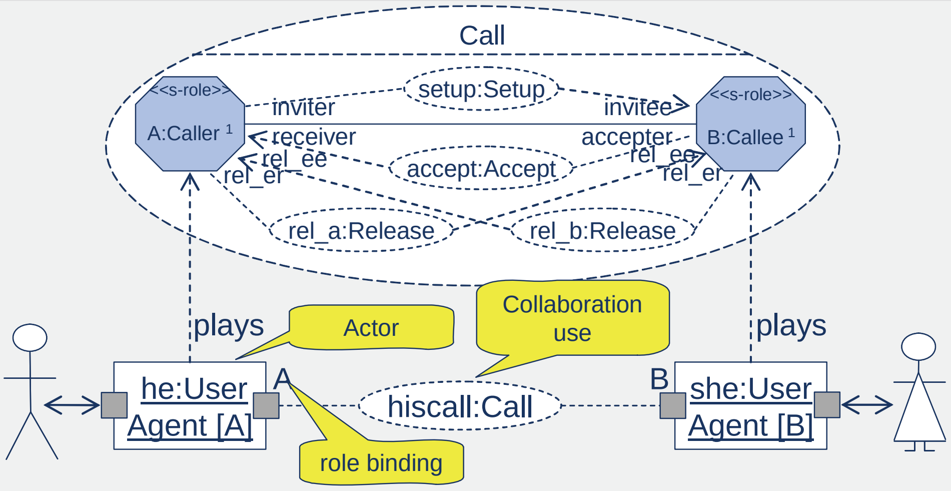


Figure 1.10 : Actors play roles typed by a composite service

The composite service *Call* in Figure 1.10 is composed of four semantic connectors. Actors are typed with the semantic connectors bound to them. This simplifies dynamic validation and enables dynamic service discovery.

UML requires that roles are compatible with the classifiers they are bound to. In Figure 1.10 *UserAgent* must be compliant with both *Caller* and *Caller.* Our approach seeks mechanisms to validate role compliancy at design time and at runtime.

7. Formally actors are active objects with behaviour compliant with the service roles bound to them, i.e. actors are the system components that actually execute the service roles.

Note that the synthesis of service role behaviour from semantic interfaces lies outside the scope of our work. The same applies to the synthesis of actor behaviour from service roles and/or semantic interfaces.

1.4.2 Goal sequences

With composition of services from semantic connectors comes the need to express relationships between the constituent parts. This is where *goal sequences* come in. Goal sequences express vertical and horizontal goal relationships. They express how reaching one goal is an enabling condition for another.

We identify three different types of goal sequences, see Figure 1.11:

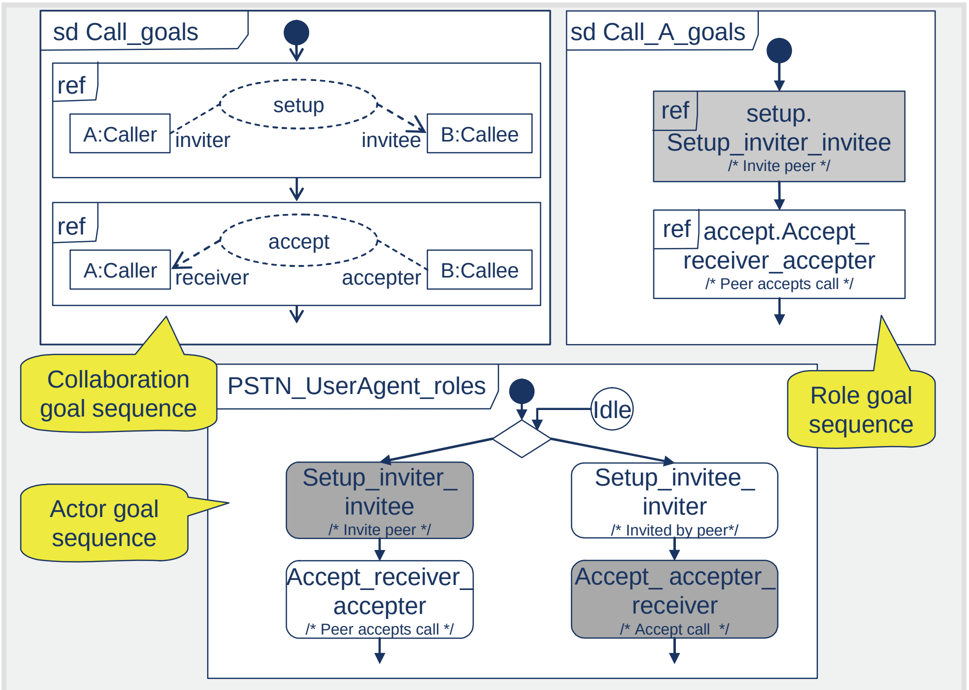

sd Call_goals
ref
setup
A:Caller
inviter
invitee
B:Callee
ref
accept
A:Caller
receiver
accepter
B:Callee
sd Call_A_goals
ref setup. Setup_inviter_invitee /* Invite peer */
ref accept.Accept_ receiver_accepter /* Peer accepts call */
Collaboration goal sequence
PSTN_UserAgent_roles
Idle
Role goal sequence
Actor goal sequence
Setup_inviter_ invitee /* Invite peer */
Setup_invitee_ inviter /* Invited by peer*/
Accept_receiver_ accepter /* Peer accepts call */
Accept_ accepter_ receiver /* Accept call */

Figure 1.11 : Goal sequences

- *collaboration goal sequences* define how the achievement of collaboration goals enables new collaborations, thus defining both horizontal and vertical goal relationships; the sequences refer to goals of semantic connectors that the service is composed of;
- *role goal sequences* are derived from collaboration goal sequences, and define goal sequences of semantic interfaces played by service roles, showing how the achievement of goals enables new semantic interfaces to be played. These are constraints that must be respected by the actors playing the service roles;
- *actor goal sequences* express the sequence of semantic interfaces supported by an actor type, showing how the achievement of goals enables new interface roles to be played. Actor goal sequences are constrained by the collaboration and role goal sequences. Actors with different role-playing capabilities have different actor goal sequences.

Goal sequences distinguish between initiating and responding roles. This is used during service discovery.

1.4.3 Validating services

The validation method exploits the chosen modelling approach:

- Composing services as a collaboration of service roles allows a focused validation of the collaboration *per se*, while disregarding other aspects. One can perform model checking on the collaboration, and ascertain that goals can be reached;
- A special case is the elementary collaborations that define a semantic connector. These can be validated independently. The pair of semantic interfaces of a semantic connector can subsequently be bound to actors and service roles;
- Actors are typed by semantic connectors, and one can validate that actors are compliant with the semantic interfaces bound to them. The same applies to service roles;
- Connections are validated by checking semantic connectors.

The result is a validation method that is modular and compositional. Validation can be performed on types at design time; once performed it does not need to be applied to actor instances or repeated at runtime. Hence the validation approach scales well.

Two sets of techniques are presented: *validating state-like goals* and *connector validation*.

Validation of state-like goals uses model checking techniques to validate whether collaboration goals, role goals and actor goals can be reached.[8] Each technique is performed as a separate step addressing a specific aspect, as illustrated in Figure 1.12.

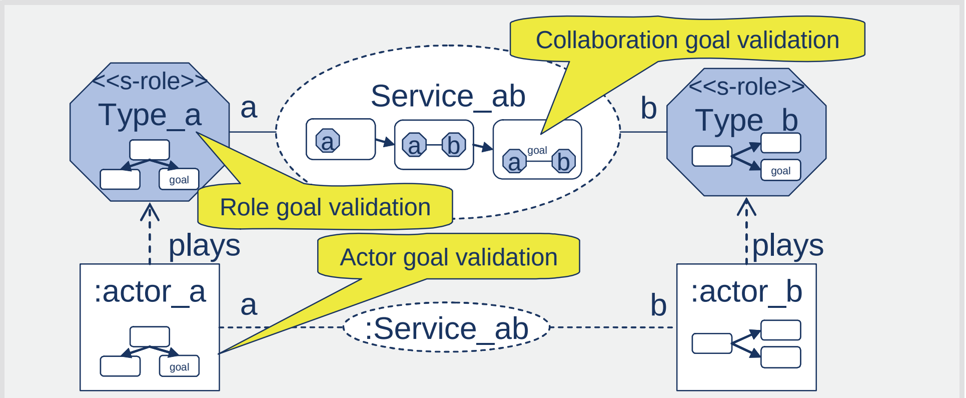


Figure 1.12 : Validating state-like goals

Standard model checking techniques are used to explore the state space of the combined model elements in search of achievable service goals. Validating state-like goals is carried out on service roles and on the actor types that play the service roles. *Collaboration goal validation* checks whether a collaboration can achieve its designated collaboration goals.

8. Note that validation of state-like goals uses traditional model checking techniques, and may run into the problems that follow from limitations of space and time that accompany these techniques.

Role goal validation establishes which goals can be reached by the service roles, given a particular binding of semantic interfaces to service roles. *Actor goal validation* determines whether an actor is compliant with roles bound to it.

The other validation technique is called *connector validation*. It complements validation of state-like goals by validating the basic safety and liveness properties of actual or specified interface behaviour, see Figure 1.13.

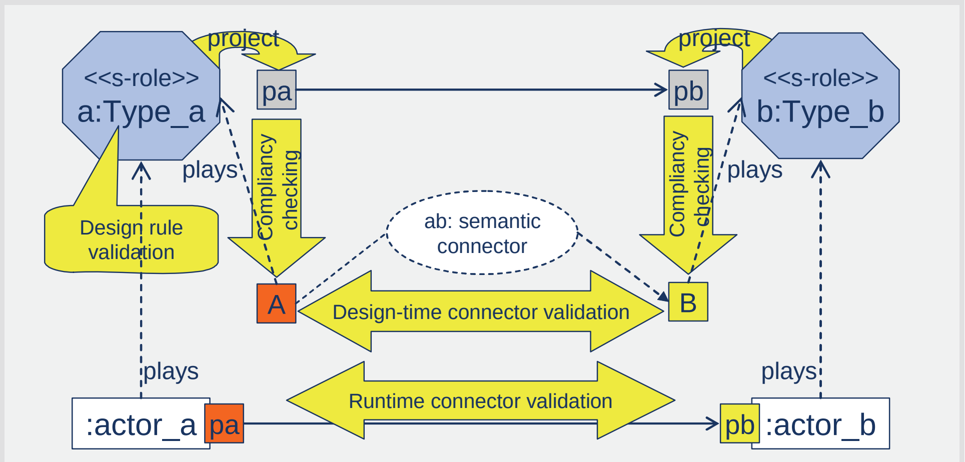


Figure 1.13 : Connector validation

An advantage of connector validation is the simplification obtained by specifying state machines that define the interface behaviour. This results in more moderate demands for computational resources to perform the validation. The computationally demanding parts can be done at design time and are limited to the connector. Hence this technique is used wherever possible.

Improved scaling of runtime validation is enabled, as this can be reduced to compatibility checks on semantic interfaces. However, in practice a combination of the two techniques is needed, as suggested in the validation method.

Validating whether a so-called *projection role* (*p-role*) is compliant with a semantic interface implies checking whether the p-role is a live subtype of the semantic interface. If so, then the actor or service role is compliant with the semantic interface, and can play safely and usefully with actors that are compliant with the dual semantic interface.

A point that should be stressed is the central position of semantic connectors and their pair of semantic interfaces. A prerequisite is that the basic safety and liveness properties of the semantic connector have been validated. Failing these two tests means that the collaboration representing the semantic connector is not well-formed. Both connector validation and validation of state-like goals are used to establish these facts.

1.4.3.1 Assumptions regarding validation of liveness

In our validation approach we introduce mechanisms to ascertain that progress can be achieved. We call this a validation of usefulness or *basic liveness*. However, this form of validation can arguably be viewed as checking a safety aspect, and not ascertaining liveness in the strict meaning of the term, see e.g. [Alpern and Schneider 1985]. This is due to a key assumption made in role validation, which is that *output eventually will be sent*.

Our techniques do not check whether this assumption holds. To do so would constitute a validation of "liveness proper". Checking complete liveness properties is an integral part of the development process, and needs to come in addition to the techniques presented in our approach. However, we argue that the validation mechanisms we suggest are meaningful; without possessing basic liveness properties according to our terms, actors playing the roles will not be able to achieve goals, and validation of "liveness proper" would be a waste of effort.

1.4.4 Service discovery

Semantic interfaces can be exploited to achieve static and dynamic discovery of compatible actors, see Figure 1.14.

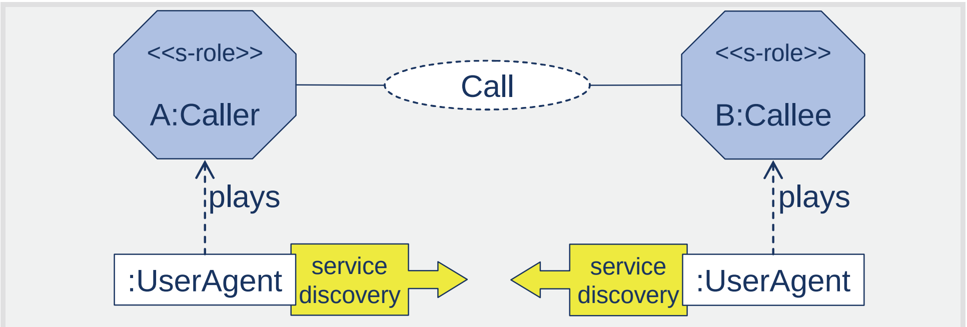


Figure 1.14 : Discovery of compatible actors

Semantic interfaces simplify the mechanisms required to find *compatible actors*. This can be combined with goal sequences to establish an actor's *service opportunities* toward a set of compatible actors. The techniques can also be used to facilitate *role learning*.

1.5 Guide to the thesis

The thesis is structured as follows:

1. This introduction has given the motivation behind the work, and discussed the research problems addressed, as well as giving an introduction to the approach;
2. Chapter 2 outlines the point of departure for our subsequent contributions. It gives an overview of previous work, such as RM-ODP and The Integrated Method - TIMe. We define basic concepts such as services, roles and actors, discuss existing validation approaches and our modelling objectives, including why we use UML;
3. Chapter 3 presents our approach to modelling services using UML2 collaborations and collaboration uses;
4. Chapter 4 presents service goals, and how these are integral parts of semantic connectors and semantic interfaces;
5. Chapter 5 presents goal sequences, where we express relationships between semantic connectors in terms of goal achievements, expressing how the reaching of preceding service goals enables succeeding goals to become achievable;

6. Chapter 6 presents our contributions to validation, i.e. validation of state-like goals and connector validation, including the validation of basic liveness in collaborations by checking for the presence of progress labels in role projections;
7. In chapter 7 we discuss future directions for service discovery, how it can be facilitated by semantic interfaces, outlining an approach to static and dynamic discovery of service opportunities, and indicating how role learning can be achieved;
8. Chapter 8 contains our conclusions, and present plans and suggestions for future work.

Appendix A discusses an alternative service modelling approach using UML association classes. This is included to underscore how UML2 collaborations are superior to this.

Appendix B lists a number of open issues that were identified during the course of the work, and discusses some possible ways forward.

Appendix C lists the definitions of terms in alphabetical order.

A list of references is included at the end.

We have illustrated our approach by a set of toy services. These have been carefully chosen to demonstrate various aspects of the approach. To keep them small, many details have been omitted, e.g. all network issues are abstracted away. Typically, service protocols are shown at a high level, disregarding details such as signal parameters.

We stress that the techniques are not limited to the simple examples; our motivation for including examples is to enable the reader to appreciate some of the principles underpinning the techniques.

2
Point of departure

In this chapter we present parts of previous work upon which we build our approach. We define our use of terms such as *services*, *connectors*, *roles* and *actors*, and present the principles underlying the validation mechanisms that we extend. We list the objectives underpinning the modelling of services, and motivate for our choice of UML2.

2.1 RM-ODP - a framework for distributed processing

It is common practice to distinguish between different levels of abstraction or modelling viewpoints when describing or prescribing ICT systems of any complexity. ICT professionals commonly accept RM-ODP, *Reference Model for Open Distributed Processing*, as a taxonomy of such viewpoints.

RM-ODP defines a range of basic modelling concepts that we base our work on, such as the concept of *objects*. Some of the important passages are quoted from [RM-ODP 1998]:

> *ODP system specifications are expressed in terms of objects. An object is a representation of an entity in the real world.* [...] *A system is composed of interacting objects. An object is characterized by that which makes it distinct from other objects and by encapsulation, abstraction and behaviour.*
>
> *Encapsulation is the property that the information contained in an object is accessible only through interactions at the interfaces supported by the object. Because objects are encapsulated, there are no hidden side effects of interactions. That is, an interaction with one object cannot affect the state of another object without some secondary interaction with that object taking place. Thus, any change in the state of an object can only occur as a result of an internal action of the object or as a result of an interaction of the object with its environment.* [...]
>
> *Objects can only interact at interfaces, where an interface represents a part of the object's behaviour related to a particular subset of its possible interactions. Each interface is identified with a set of interactions in which the object can participate. Note that these interactions do not necessarily occur with other objects: an object can interact with itself. An important characteristic of the concept of object in the RM-ODP is that an object can have a number of interfaces.* [...]
>
> *Object composition yields composition of states and behaviours and it is therefore possible to speak of a composite behaviour and of a composite state.* [...]

> *One object is said to be behaviourally compatible with another object in some environment if the first object can replace the second, without the environment being able to detect any difference. Any particular interpretation of behavioural compatibility will impose constraints on the allowed behaviour of the environment.* [...]
>
> *A contract is an agreement that governs cooperation among a number of objects, and embodies the ideas of obligation, permission, prohibition and expectation associated with cooperating objects.* [...] *A contract can specify the roles of objects and the obligations applying to roles, i.e. the expected cooperative behaviour* [...]
>
> *Binding behaviour establishes a contractual context (a binding) between interfaces and enables object cooperation. A binding can exist at several levels of abstraction.*
>
> *A liaison is the relationship that exists between the objects cooperating under the auspices of a binding. When the liaison is in place, an object knows that the other objects in the liaison obey the contract. An object can be involved in several simultaneous liaisons: for each of these liaisons there is a corresponding contract.*

In this thesis, we primarily address issues pertaining to the *computational viewpoint*:

> *The computational viewpoint: A viewpoint on the system and its environment that enables distribution through functional decomposition of the system into objects which interact at interfaces.* [...]
>
> *The computational viewpoint is directly concerned with the distribution of processing but not with the interaction mechanisms that enable distribution to occur. The computational specification decomposes the system into objects performing individual functions and interacting at well defined interfaces.*

Our focus lies in service modelling, and our definition of service deviates from the informal use of the term in RM-ODP. However, the concepts that we suggest, such as service structures, service goals and goal sequences, are all aimed at the computational viewpoint, as are the mechanisms for validation of liveness properties at interfaces.

Our research also addresses services at the level of domain models, and as such is related to the *enterprise viewpoint*[1] of RM-ODP. Service goals are property descriptions belonging to domain models and application models. Domain models capture issues in the problem domain, and identify domain-given objects and properties. In an ideal world domain models will be common for a large range of systems, each with its own system description. A domain model will be stable as long as the problem area remains stable.

Issues related to the other viewpoints in RM-ODP (informational view, engineering view and technology view) lie outside the scope for our work.

2.2 Systems engineering according to TIMe

An additional pillar upon which we base our work is the understanding of the systems engineering challenge (and a suggested method of conquering it) according to TIMe - The Integrated Methodology [TIMe 1999]. We do not suggest that TIMe is in all ways superior

1. A viewpoint on the system and its environment that focuses on the purpose, scope and policies for the system.

to other system development methodologies, but it does address issues that are important to our work. According to TIMe, the essence of systems engineering is to understand needs and to design systems having properties that satisfy the needs in a cost-effective way.

In this section we present the elements of TIMe that are particularly relevant to our work.

2.2.1 Abstractions in models

Concrete systems in the real world are composed of physical parts and software that provides services to users. To implement such systems we make detailed descriptions of the physical composition and software, so-called implementation descriptions.

However, implementation descriptions are often too detailed to be easily understood by humans. There is a conflict between the needs of machines and the needs of humans. To satisfy humans, we need abstractions that remove technical detail of implementations and allow us to concentrate fully on the aspects that are important for the designer and user.

To bridge the conflicting needs of human interpretation on one hand and physical construction on the other, TIMe defines models at two main levels of abstractions: the abstract world and the concrete world, as illustrated in Figure 2.1

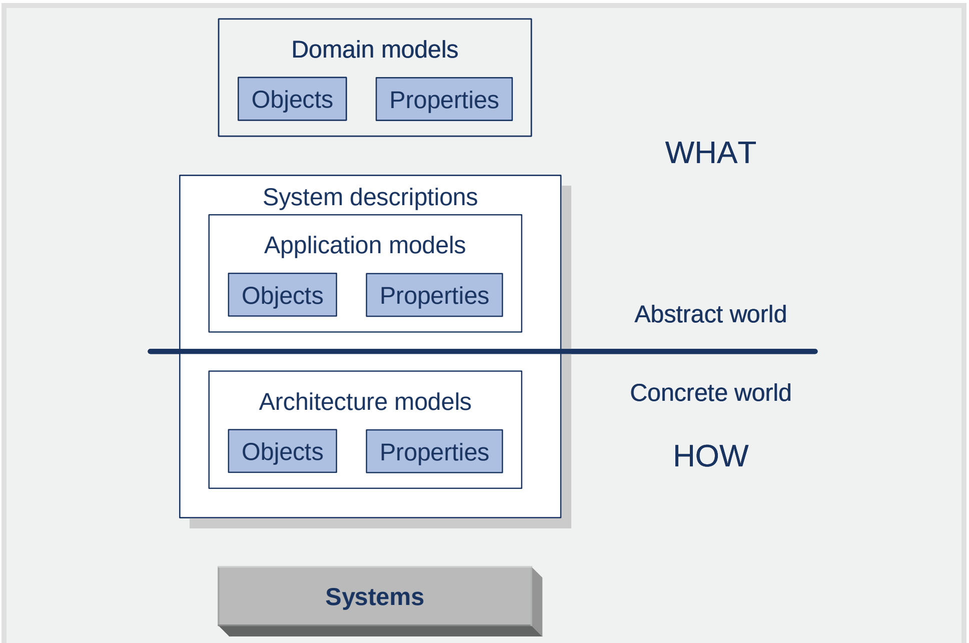


Figure 2.1 : Systems and their abstractions in models [TIMe 1999]

The abstract world emphasizes concepts and behaviour related to user needs. They define system behaviour in an abstract form that can be understood, communicated and analysed without binding the implementation more than necessary.

The concrete world describes the implementation. It includes the architecture models, which are a high level description of the physical implementation. In addition come the implementation descriptions, which are composed of a variety of notations and languages for hardware design and programming.

The main point here is that we use abstractions primarily to improve human understanding and communication and enable reasoning.

2.2.2 Objects and properties

As seen in Figure 2.1, TIMe makes the important distinction between two related model types: object models and property[2] models.

- Object models describe how a system or component are composed of objects, connections and relationships. They are constructive in the sense that they describe how an entity is assembled from parts. This is the perspective of designers;
- Property models describe properties of a system or component without prescribing a particular construction. They are not constructive, but used to characterise an entity from the outside. There are many kinds of properties: behaviour properties, performance properties, maintenance properties, etc. This is the perspective preferred by users and marketing people. It is also the main perspective in specifications.

A central idea in TIMe is that every object is characterised by properties that can be used:

1. to understand what the object does;
2. to check whether it is suitable for the environment where it is used;
3. to synthesize the design;
4. to verify that the design satisfies needs by matching provided and required properties;
5. to retrieve a suitable object from a library, given some required properties.

Property models are not necessarily bound to object models, while object models shall normally be bound to property models. This holds for all object models: domain models, application models, and architecture models.

Concrete world properties are associated with the concrete models and state properties relevant to the implementation. They are often termed *Non-functional properties* and characterize the implementation. But as this is not a topic in this thesis, we do not go into more detail on these types of properties.

Abstract properties are associated with the abstract object models. Since abstract models focus on functionality (behaviour), these properties are often termed *Functional properties*. They characterise the behaviour of objects, and the collaboration between objects.

Functional properties are classified in the following categories:

- General properties, which are properties that can be expressed independently of particular objects, services or interfaces. An important class of general properties is safety properties, which state what should never happen, such as unspecified signal reception, deadlock and improper termination;
- Service properties, which are properties related to specific services. Important aspects

2. We use the term *property* in the sense of "a characteristic trait or quality" of the object in interest.

are the service roles that objects shall play to perform the service and the interaction behaviour between these roles,

- Interface properties, which are properties related to specific interfaces. Services are controlled via interfaces, and interfaces may have properties of their own. These properties (i.e. protocols) must be followed by both sides of the interface. Objects may have several interfaces, and the same interface may apply to several objects;
- Data properties. These express what can be said about the data contained in a system in terms of what they mean for the environment.

Users tend to think in terms of services and interfaces. Therefore it is customary to characterise systems using a service-oriented perspective. This is best explained in contrast to the vertical perspective illustrated in Figure 1.3 on page 4. Many services will naturally involve several objects. A normal call in a telephone system involves at least two objects: the calling subscriber and the called subscriber. There is no point in one without the other. The service perspective allows us to see the two in combination, but only to see fragments of each object. In the object perspective we can see the complete object, but only fragments of each service.

As we said in the introduction, the service perspective provides a horizontal view, taking in all the collaborating parts. Even the human end user can play a role. This is opposed to a vertical view of taking in just one object, and considering its complete behaviour.

TIMe makes two important observations:

1. Service and interface properties span several objects. They are composed of (sub)properties of different objects. An important advantage of the property perspective is the possibility to combine and describe interaction properties of different objects in one place. It encourages us to describe service and interface properties in one place so that they may be used to characterise all object types using the service or interface.
2. Object properties are composed of sub-properties belonging to different services and interfaces. However, the composition of properties into objects is not as simple and well-defined as the composition of objects into systems. The reason is that objects encapsulate behaviour and have interfaces, whereas object properties are likely to be only fragments of behaviour without clearly defined interfaces.

TIMe poses two requirements on functional property models:

1. It should be possible to express property models without referring to specific objects or object types. The reasons for this is that we sometimes need to specify properties without knowing the objects (types) they shall be associated with, and that we may want several different objects to share the same properties, such as a common interface.
2. It should be possible to compose the properties of an object from parts described in different property models.

The *role* concept aims to satisfy both these requirements. Roles represent objects in property models in an anonymous fashion, and one may compose the properties of an object from different roles described in different property models.

It is not obvious that a property description is consistent with an object design. When relating object design descriptions and property descriptions we seek principles that can:

1. perform verification and validation based on described properties;
2. check compliancy between object models and property models;
3. extract and describe properties of objects in a way faithful to the object design;
4. synthesize a consistent object design from property specifications.

The notion of roles is a key to all this. We shall return to roles after having defined what we mean by services.

2.3 Services

Many of the terms and concepts we use to discuss ICT systems have multiple definitions, ranging from their ordinary linguistic use to the range of more or less precise definitions used within certain ICT fields. This is certainly true of the terms that are most central to this thesis, such as *services*, *actors* and *roles*; each has a colloquial meaning, and each conveys a different understanding among various groups of ICT professionals.

The term *services* is probably the most overloaded of them all. In daily use we may think of a service as something an organization or system provides to the public. For instance, a service is *useful labour that does not produce a tangible commodity* [Sassen and Macmillan 2005]. Among ICT professionals, the term has several more or less specific meanings.

In her discussion of services, [Floch 2003] suggests a taxonomy, see Figure 2.2.

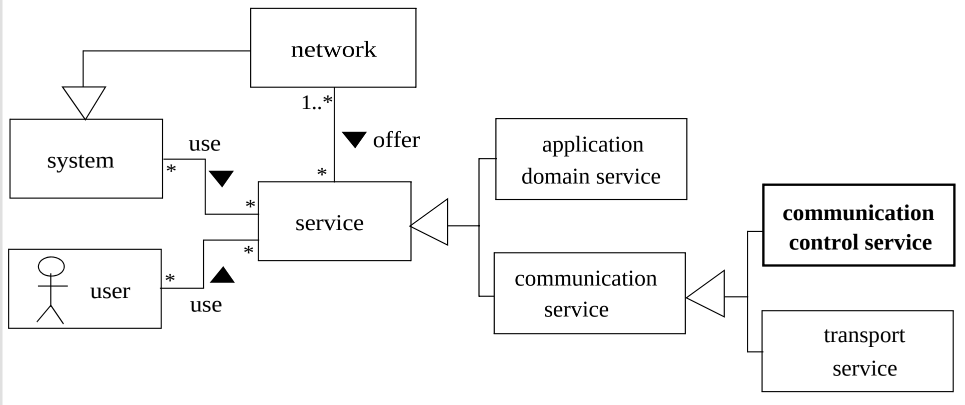


Figure 2.2 : Networks and services [Floch 2003]

Most ICT professionals refer to *application domain services* when they use the term service. As we pointed out in section 1.2, this is the viewpoint of the client-server paradigm. According to this view, services are software modules that are accessed by name via an interface, typically in a request-reply mode, i.e. all initiatives originate on one side.[3] This

3. This is consistent with the term *service* as used in UML, where a service is "*a stateless, functional component that returns a value*" [UML2 Ref] p. 589.

is the perspective of *service-oriented computing*. Service requests only influence each other to the degree that they compete for the same resources, such as seats in a cinema or positions in a queue of printing jobs on a printer.

In contrast to this, for *communication control services*, coordination is part and parcel. Such services take care of what is often called service logic, meaning the “things that happen when users or devices take initiatives towards each other”. Generally such services are symmetrical or peer-to-peer, meaning that initiatives may originate at several computational objects, and potentially cross each other and cause conflicts that must be resolved.

Setting up a telephone call between two or more participants is a typical example of a communication control service, and can involve a whole range of complex decisions for the parties involved. Who is the caller allowed to call? What to do if (some of) the callee(s) are busy? What to do if a callee initiates call back? These are but a few of the issues that typically have to be settled before a call is connected.

In general we can say that services are either single-sided or multi-sided:

- Single-sided services are of the client-server type, where all initiatives originate on one side. Most application domain services are single-sided;
- Multi-sided services are of the symmetric peer-to-peer type where initiatives may originate on several sides and potentially cross each other, causing conflicts. Communication control services are usually multi-sided.

In this thesis the needs of communication control services are addressed, being an important aspect of *convergent services*. When used alone, the term *service* should be understood as meaning a convergent service.

Communication control services cannot exist by themselves. The rely on *transport services* to allocate and set up transport channels in the network, and to allocate network resources in order to secure connectivity and transport capacity. Typical transport services are voice, video and data transfer over various network technologies such as ISDN, GSM or IP. In this thesis we abstract away from all details of the transport network, assuming that it provides a network-independent interface to convergent services.

Before we formulate our definition of service, let us assess a few candidates. One definition is the following:

> *A service is a unit of behaviour which characterizes what a system (or component) provides for the user. A service is normally given a name. Services may be interleaved in time.* [TIMe 1999]

In our view this definition puts too much focus on the user, and does not cater for services between computational objects. A more general definition is as follows:

> *A service is an identified (partial) functionality, provided by a system, component, or facility, to serve a purpose for its environment.* [Bræk 1999]

This definition considers a service to be relative to goals of the environment it serves. It identifies functionality as an asset for the environment. As such, it may have a price and be provided as a commercial offering, i.e. a service as defined by Intelligent Networks [IN 1993]. A drawback with this definition is that it does not include the behaviour of the environment as part of the service, and thus does not fully cover the peer-to-peer aspect. This is true of RM-ODP as well, it informally refers to services as “offered by objects”.

We need the concept of service to cover the joint behaviours of the computational objects, as well as the purpose relative to the environment. We also see the need to distinguish between service types, which we just call *services*, and instances of these, which we call *service invocations*.

Our definitions of the terms service and service invocation are as follows:

Definition: Service
A service is a collaboration between concurrent and potentially distributed service roles played by computational objects in order to provide some identified functionality to the environment.

We defer defining *service roles* to section 2.4.1.

Definition: Service invocation
A service invocation is an instance of a service.

A service invocation exists from the first event occurrence in the role taking the initiative to commence the service, and until the involved service roles have completed the last event occurrence of the service.

In our approach we model service using UML2 collaborations, see Figure 2.3.

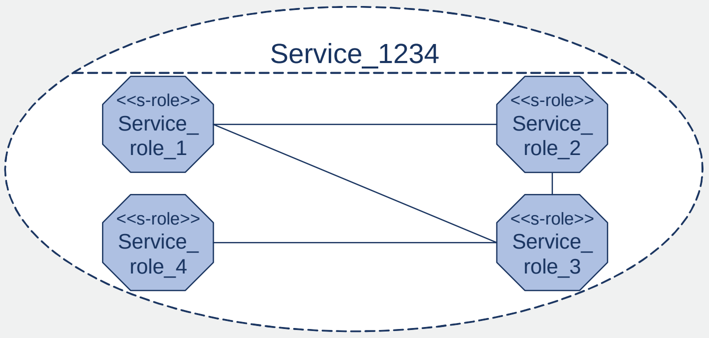


Figure 2.3 : A service consisting of four service roles

In our work, we address convergent services that combine traditional telecom, multimedia, messaging, context-awareness and information services. They are convergent in the sense that they combine elements of the peer-to-peer paradigm with client-server solutions. In other words, the multi-sided services we address can have elements that follow the pattern of single-sided services. This does not pose a problem for our approach; the communication mechanisms needed to support multi-sided services can support single-sided services, while the opposite is not true [Bræk and Floch 2004][4].

4. Synchronous communication can work well for single-sided services, but not for multi-sided. Asynchronous communication works for both kinds of services. In [RM ODP 1998] interactions between computational objects are essentially asynchronous; operations are explained in terms of a combination of asynchronous signals.

2.3.1 Sessions and connectors

In communication control services, a *session* is a fundamental concept. It corresponds to the RM-ODP term *liaison*. Many communication control services involve sessions in one form or another: a two-party call session, a conference session, and a chat session are typical examples of service sessions.

In this thesis we use the term *connector* when we are referring to session types[5].

Definition: Connector
A connector is a binding between two roles that can carry the interactions of a service.

Since our focus is on types and not on instances, we use the terms *session* and *connection* informally, meaning a path[6] between computational objects for the interactions of a service invocation, see Figure 2.4.

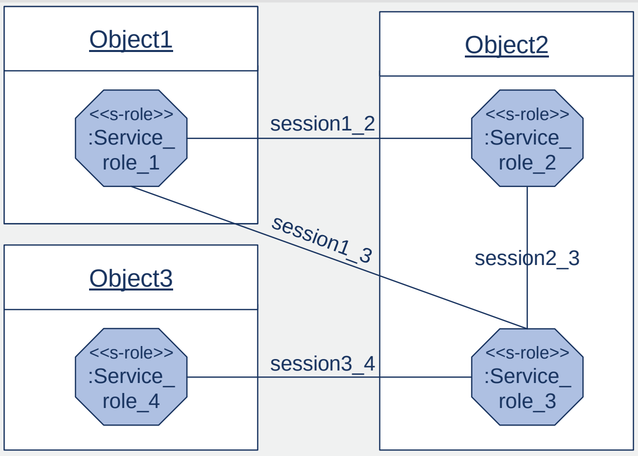


Figure 2.4 : Sessions, objects and service roles of a service invocation

In general, sessions and connectors are not confined to communication control services; for instance [TINA 1999] defines concepts such as access sessions (between user and system), communication sessions (co-ordination of network resources) as well as service sessions (provision of the service itself). However, in this thesis, we do not distinguish between media streams and control streams, and do not go into in-band signalling or any other network related issues, since the focus is on convergent services.

Note that not all of the services exemplified in this thesis are traditionally characterised by sessions. Positioning services, instant messaging or email are all examples of services that involve data exchanges of an instantaneous nature, and one does not normally speak of a session being established. However, we shall treat such “instantaneous” service role relationships in the same way as more long-lasting relationships, in order to present a uniform approach for all types of advanced services.

5. Note that session types are called service associations in [Floch 2003].
6. In UML parlance a session or connection corresponds to a *link* between *objects*.

2.4 Roles

The concept of *roles* is well known from daily life, where we speak of ourselves in terms of functions we perform as members of some organisation, such as employee, conductor etc., or in relation to others, such as parent, child, sibling etc. One can indeed distinguish between *functional roles* and *relational roles*:

- In a play, such as Peer Gynt by Henrik Ibsen, we find roles such as Peer and Mor Aase. During a theatre performance we find actors playing Peer and Mor Aase. The roles, as described by Ibsen, specify required properties of the actors without specifying what other properties they may have. Good actors provide the properties in a way that make us believe that Peer and Mor Aase are real. After the play is over, the actors will do other things and provide other properties. This notion of a role can be formalized as the *properties of an object appearing in the context of a service,* i.e. *functional roles*;
- Another notion of role comes from the relationship between objects. A person has the role of father in relation to his daughter, husband in relation to his wife and owner in relation to his car. This notion of role can be formalized as *properties of an object appearing in relation to another object*, i.e. *relational roles*.

  Relational roles are typically related in pairs. The role of daughter is complemented by the role of father. It is also typical that the role corresponds to properties required in that relationship. One may well play many relational roles, but they should not all be mixed. Reactions are bound to surface if a person mixes the role of “lover” with the role of “parent”, for instance.

  Relational roles are commonly known from entity-relationship modelling, and are for instance used within role-based access control, see e.g. [Ferraiolo et alia 2001].

Both types of role are useful in understanding and describing reality, and have gradually made their way into modelling languages. Both can for instance be modelled in UML[7]:

- Functional roles can be modelled as *roles* in UML2 collaboration diagrams, capturing the role that classifiers play when interacting with other classifiers[8];
- Relational roles can be modelled as *properties* of classifiers related by associations in class diagrams[9].

Roles are a useful concept in systems modelling for a number of reasons:

- So much seems to depend on the point of view. Every object will have a relative view on other objects in its environment. This applies to both types of roles;
- Relational roles are often represented by associations in UML, e.g. the father - daugh-

7. Note that UML2 has tidied up the multiple use of the term *role* in UML1: ClassifierRole, AssociationRole, and AssociationEndRole have been replaced and generalised by ConnectableElement, Connector and ConnectorEnd respectively. Association end is no longer a model element, and has been superseded by Property. The “role” term is referred to in Use Cases, but only in an informal way. Collaboration role has been introduced to reference connectable elements, and represents roles that instances may play within a UML2 collaboration.
8. *A collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality* [UML 2.0] p. 164.
9. Thus capturing “*participation of instances of the classifier connected to the end in links of the association*” [UML 2.0] p. 36.

ter roles. Such roles are important to describe and to understand;

- There is some symmetry among the functional roles played at the two sides of an interface.

In our work, it is the functional roles that are of interest, for exactly the latter reason. As we shall see, a special kind of functional role called an *interface role* can play an important part in designing and analysing behaviour.

The role concept we use has its roots in the SISU project [TIMe 1999] and in the Plug & Play project [Aagesen et alia 1999], with its theatre analogy. We use [Bræk 1999] as our starting point for defining roles; this extends [Aagesen et alia 1999] to define different types of roles that computational objects play in convergent services. Finally we use revised definitions of roles as set out in [Floch 2003][10].

Below we define two types functional roles: *service roles* and *interface roles*.

2.4.1 Service roles and actors

Service roles are the parts that computational objects play in a given service. In a basic telephone call, for instance, it is common to differentiate between an initiating and a terminating subscriber role. These roles must be played by different computational objects in the same call since an object cannot call itself[11]. Service roles can be dynamically assigned, implying that objects take on different service roles over time.

Definition: Service role
A service role is the part a computational object plays in a service[12]. A service role can play several interface roles, both simultaneously and alternately.

In other words a service role is the part of the behaviour of a computational objects that relates to a particular service. In our work we assume that the behaviour of a service role is described by a composite state in UML or SDL. A service role hence cannot act by itself, but is part of a computational object[13], and is entered from and exits to the state machine of that object. We call such computational objects *actors*[14].

Definition: Actor
An actor is a computational object that can play service roles. An actor can play several service roles, both simultaneously and alternately.

An actor can be described and implemented as an extended finite state machine, communicating with other actors via asynchronous signals. Actors coordinate between composite states of service roles. Underpinning our approach is the understanding that interacting

10. Session roles in [Bræk 1999] were called service association roles (a-roles) in [Floch 2003]. We in our turn chose to call them *interface roles*. Both semantic interfaces and projection roles are interface roles.
11. Note, though, that an object may be capable of playing both roles simultaneously in different calls.
12. [Floch 2003] p. 21.
13. If a service role has attributes, an object playing the service role must accommodate them.
14. In UML, the term *actor* has a special use. "*Actors model entities external to the subject*" [UML 2.0] p. 571. The UML term for actor as we use it is Active Object, "*an object that, as a direct consequence of its creation, commences to execute its classifier behaviour, and does not cease until either the complete behaviour is executed or the object is terminated by some external object*" [UML 2.0] p. 424.

actors can belong to different systems, i.e. run on distributed devices communicating over disparate networks, and play a part in services offered by different service providers.

The relationship between services, service roles and actors is summarized in Figure 2.5

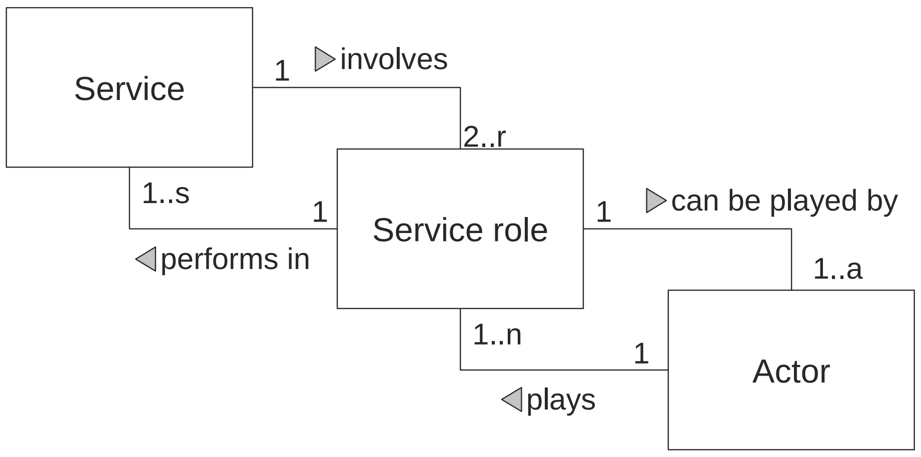


Figure 2.5 : Services, service roles and actors

A service consists of two or more service roles, as illustrated by Figure 2.4. Service roles can be composed of other service roles; service roles that are not decomposed are denoted *elementary service roles*.

[Floch 2003] defines design patterns for the coordination of concurrent service roles; this is not a topic of our work.

2.4.2 Interface roles

RM-ODP defines the term *role* as follows:

> *A role identifies, in a template for a composite object, a behaviour to be associated with one of the component objects.*
>
> *A role may correspond to a subset of the total behaviour of a component object. When an object is viewed in terms of a role, only a named subset of its actions is of interest, and other actions are abstracted away, possibly to other roles. A component object may have several roles at a given time depending upon its interactions, and may take different roles at different times. These roles may be associated with interfaces.*

This corresponds to what we call an *interface role*. In general, every computational object provides some roles at its interfaces, and expects compatible roles to be played by objects in its environment, see Figure 2.6.

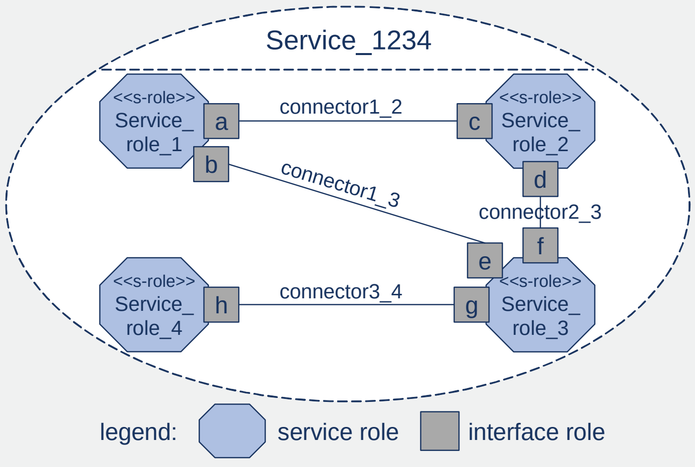

Service_1234
<<s-role>> Service_ role_1
<<s-role>> Service_ role_2
<<s-role>> Service_ role_3
<<s-role>> Service_ role_4
a
b
c
d
e
f
g
h
connector1_2
connector1_3
connector2_3
connector3_4
legend:
service role
interface role

Figure 2.6 : Service roles and interface roles

Figure 2.6 shows a service consisting of four service roles, here represented by octagons.[15] The connections between the service roles are represented by connectors, and are constituent parts of the service. At each endpoint of a connector there is an *interface role*.

Definition: Interface role
An interface role describes the (actual or specified) interface behaviour of an actor or service role at a connector endpoint.

Interface roles are property descriptions that play an important part in designing and analysing actor behaviour:

- It should be possible to characterise internal behaviour in a purely external way, removing all irrelevant internal detail. Such behaviour is factored out in interface roles;
- It should be possible to use such interface roles to simplify the validation of connectors, so that one can routinely ascertain that actors are instantiated only in environments where they will work properly;
- Validation effort, like development effort, should be modular. It should be possible to focus validation and verification effort on types in a manner that simplifies the validation of actors. Validating the interface roles of an actor type should be sufficient to establish its ability to interact safely and usefully with its environment.

Interface roles allow us to describe and study interaction behaviour at a particular interface, and to disregard details of the interactions at the other interfaces. Using interface roles, we may describe and analyse the properties of each interface separately. In that way we obtain an external view of an actor or service role from a particular vantage point.

Interface behaviour works in two ways: it may describe the *actual* behaviour exhibited by the actor or service role, or behaviour that the actor or service role is *specified* to exhibit.

15. The icons for service roles and interface roles are not standard UML. UML stereotypes can be used to define such icons.

Definition: Actual interface behaviour
Actual interface behaviour is the behaviour an actor or service role exhibits at an interface.

Definition: Specified interface behaviour
Specified interface behaviour is the behaviour an actor or service role is specified to exhibit at an interface.

The service roles in the role structure shown in Figure 2.6 represent anonymous actor objects. Roles should not need to specify which actors play them; roles should be defined independently of objects. Actor behaviour can in turn be bound to a set of service roles. An allocation (or binding) of the roles of Figure 2.6 to actors is exemplified in Figure 2.7.

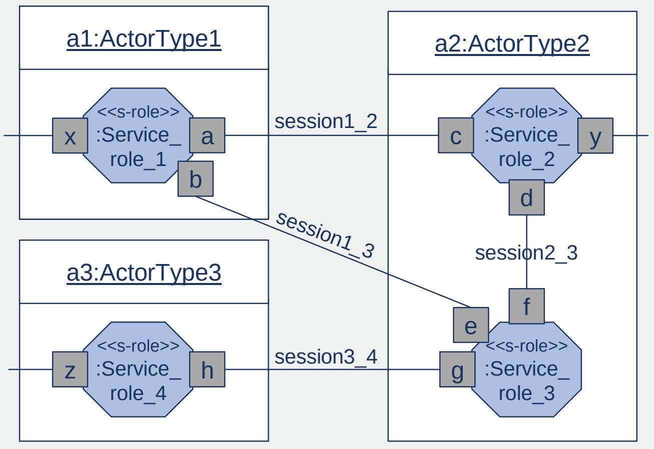

a1:ActorType1
a2:ActorType2
a3:ActorType3
<<s-role>> :Service_ role_1
<<s-role>> :Service_ role_2
<<s-role>> :Service_ role_3
<<s-role>> :Service_ role_4
x
a
b
c
y
d
e
f
g
z
h
session1_2
session1_3
session2_3
session3_4

Figure 2.7 : Service roles and interface roles played by actors

The binding of interface and service roles to actors is the mechanism used to support the two-dimensional composition of services illustrated in Figure 1.3. We discuss compliancy of role binding in section 2.5.4 below.

Note that service roles can be involved in more than one service, as defined by the *performs in* relationship in Figure 2.5. This is illustrated above by the interface roles *x*, *y* and *z*, which belong to some other service(s) not shown in Figure 2.3 or Figure 2.6. For instance the service role *Service_role_1* is involved in the service depicted in Figure 2.4, as well as in some other service of which *x* is an interface role.

UML provides a concept of *interfaces* for structured classifiers, see Figure 2.8.

Several connectors can be combined on the same UML interface. Figure 2.8 shows one way the interface roles of Figure 2.7 can be bound to UML interfaces. In this example, the interface roles of actor *a2* have been combined into a single interface, although they could have been split into several, while the interface roles *d*, *f*, *x*, *y* and *z* of Figure 2.8 have not been bound to interfaces.

An object will often be able to play several service roles, which may be accessible from the same interface, as shown by *a2:ActorType2* in Figure 2.7. Figure 2.8 also illustrates how provided and required interfaces are attached to ports in UML2.

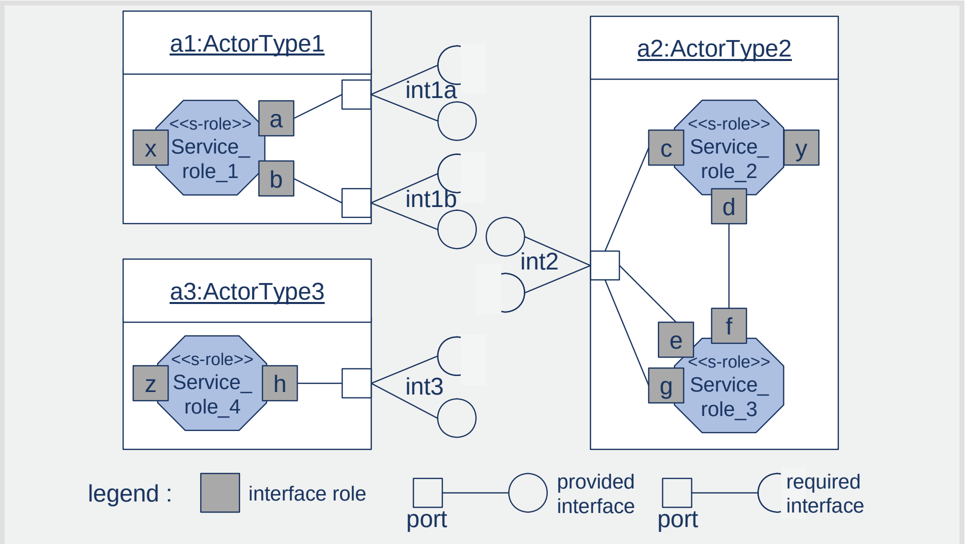


Figure 2.8 : Binding interface roles to UML interfaces

[Bræk 1999] and [TIMe 1999] defined so-called "association roles" describing the visible behaviour of a computational object at a connector end, with the special case "interface role" describing the visible behaviour at an interface. Since synthesis of actor behaviour is not addressed by our work, we do not need to distinguish between observable behaviour at a connector end or at an interface. Specifically we do not consider composite interfaces like *int2* in Figure 2.8.

2.4.3 The role-playing principle

Role-playing is symmetric; validating an interface is to check whether both sides play the roles they mutually require from each other. The notion of roles are closely connected to the notion of validation and thus to system quality. TIMe summaries this with the following method rule:

Method rule: Role behaviour [TIMe 1999]
Define the [interface] behaviour of each role in the system and in the environment. Use the roles as a basis for behaviour synthesis and validation.

Interface roles are fundamental to our approach to validation. They may be used to relate design descriptions and property descriptions:

1. to analyse the properties of an interface role by itself, to see whether it is well formed. This can be done independently of any particular application of the role;
2. to analyse the actual interface behaviour of an actor, to see whether it behaves in a well-formed manner over a connector. This can be done independently of any peer;
3. to verify that the actual behaviour of an actor is compliant with its specification. This means checking actual interface behaviour against specified interface behaviour;
4. to synthesize object behaviours that are correct by construction. Note that this is not part of our present work.

2.4.4 Connected roles

A service involves at least two distinct service roles, as defined in Figure 2.5. For instance a call is initiated by one party, and is directed towards another. In the telecom domain these roles are commonly recognized and defined by service designers, e.g. the former role is referred to as the *Caller* or *A*, while the latter is denoted *Callee* or *B*.

The roles that collaborate to provide a service are called *connected service roles*. *Caller* and *Callee* are examples of connected roles. *Connected interface roles* play a central part in our approach, which is why the term *connected role* refers to interface roles only.

Definition: Connected role
An interface role is called a connected role with respect to an opposite interface role, if it intends to[16] interact with that interface role over a connector, or actually does so.

In Figure 2.7 the interface roles *b* and *e* interact, and are thus connected roles. An interface role is connected to exactly one connected role in a service session. Connected roles are closely related; one represents the environment of the other.

Note that we use the term *connected role* to mean a potentially connected counterpart that one does not necessarily interact compatibly with, e.g. one targeted in a role request (see below). Role compatibility is checked by the validation techniques, see section 2.5.1.

At least one of a pair of connected interface roles must be able to send a first signal over the connector between them. Inspired by initiating and responding actors in RM-ODP, we denote such a role as an *initiating role*, while an initially “passive” role is called a *responding role*[17].

Definition: Initiating role
An initiating role is an interface role that can send a first signal over a connector.

Definition: Responding role
A responding role is an interface role that does not send any first signal over a connector.

We have considered using the terms “required” for an initiating role, and “provided” for a responding role, as these are well established terms for interfaces. However, due to the definition of interfaces in UML, an interface role needs both a provided and a required UML interface in order to support two-way signal exchange, see the discussion in section 3.3.4.

The fact that a role sends a first signal is important in connection with service discovery and validation, which is why we defined the terms responding role and initiating role. Note that both interface roles of a connector can be initiating roles, i.e. both can send a first signal.

---

16. “Intends to” refers to an attempt to bind a pair of actors at runtime. In certain situations a pair of actors may attempt to initiate an interaction without having validated *a priori* that they can interact compatibly.
17. “Responding” in this context means “accepting the first communication”. In traditional two-party telecom services, one is called the initiating or originating role, while another is called the target or terminating role. However, this relates to the initiative of establishing the connection, which may be different from sending a first signal.

2.4.5 Role requests and role arbitration

We have argued that roles are important modelling terms. But they may also be important concepts in implementation frameworks. For instance in the experimental service development and execution framework *ServiceFrame* [ServiceFrame 2002], sessions (instances of connectors) are established as a result of *role requests*, i.e. a requesting actor asks for a certain interface role to be played by a requested actor, see Figure 2.9 below.

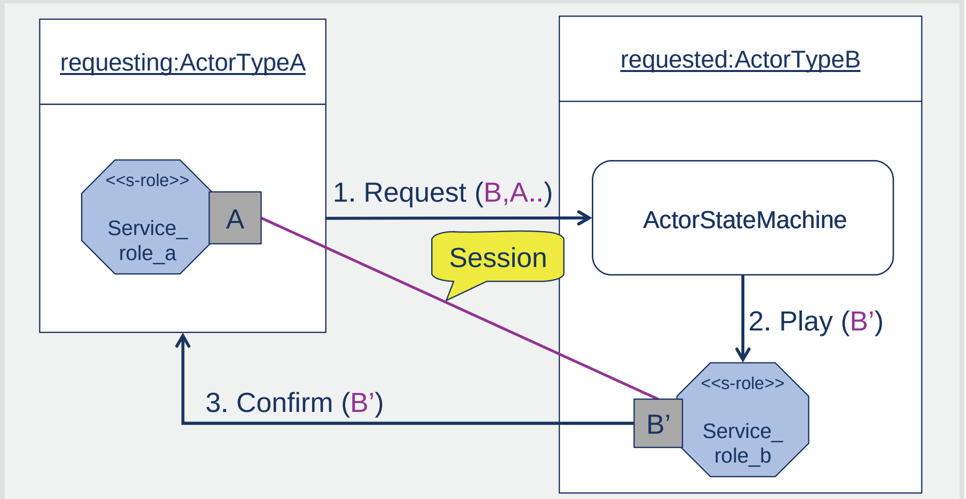


Figure 2.9 : Role request pattern in ServiceFrame

In this interaction the requesting actor supplies an identification of the desired role (*B*) and/or its own interface role behaviour (*A*) in the request, and receives an identification of the connected role's actual interface role behaviour (*B'*) in the confirmation. Role arbitration and validation can be performed by *ActorStateMachine*, based on knowledge of (*A* or *B*) and *B'*.

The role request pattern can be used to achieve dynamic role binding, see [Castejón and Bræk 2005]. In our work we discuss how the role request pattern can support an aspect of service discovery called *role learning*, see section 7.4.

2.5 Service validation

Validation can generally be said to consist of two classes of techniques: static analysis and dynamic analysis:

- Static analysis checks that interfaces between components are compatible, for instance by checking the type compatibility of signals exchanged. Static analysis is a basic technique, and is not treated further. In our approach to validation we take it for granted that static analysis is successfully performed prior to dynamic analysis;
- Dynamic analysis takes into consideration the order of events occurring during system execution; an aim is to evaluate all possible interactions that may occur between cooperating objects. Dynamic analysis demands more computing resources in terms of time

and space, and checking all possible interactions often exceeds available resources due to the state space explosion encountered in reachability analysis. Real systems must often be simplified in order to perform dynamic analysis.

Since our focus is on services, we use the term *service validation* rather than system validation. Service validation targets services that cross system boundaries.

2.5.1 Role projection and validation

Role projections are similar to geometrical projections in that they show everything that is visible from a given angle, and hide the rest. As in geometry, we can use projections to:

- synthesize new objects, like a carpenter builds a house from a set of blueprints;
- make projections of existing objects, in order to document and analyse their properties.

Instead of geometrical views, we are interested in the observable behaviour of objects. Synthesis is not a theme of this thesis, while validation of the behaviour of objects is.

The concept of projection was proposed in [Lam and Shankar 1984] for the analysis of single functions in a protocol. In that work, protocols were decomposed into modules that handle different functions, and each module was defined as a projection of the whole protocol. This projection technique was exploited in [Bræk and Haugen 1993] to sketch a projection transformation and analyse the projected interfaces. [Floch 2003] (see also [Floch and Bræk 2003a]) developed this idea further, and her results are a basis for our present work. She introduced the use of role projection to perform safety checks of the derived roles. The principle is illustrated in Figure 2.10.

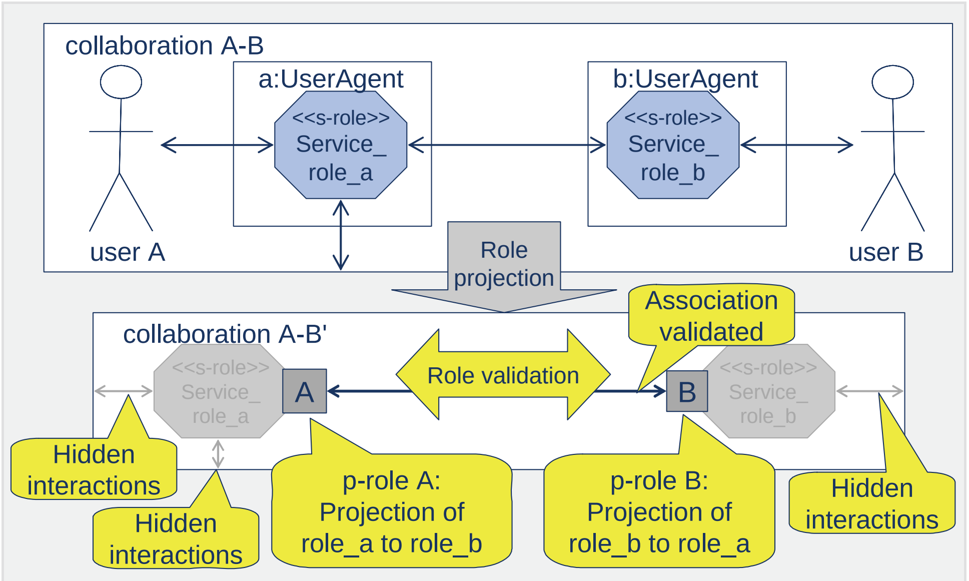


Figure 2.10 : Role projection and role validation

Projection is an abstraction technique that results in a simplified system description emphasising some of the system properties while hiding some others. Rather than analysing an entire system, the analysis is carried out on projections only. In the work of [Floch

2003], the projection only retains the aspects required to validate a connector between two service roles. Projection hides internal actions and external interactions that are not relevant in the validation of a particular connector.

[Floch 2003] defines a projection transformation for the generation of interface roles from service roles. Such interface roles describe the visible (interface) behaviour of service roles over connectors, and hide the behaviour not visible to the connector being analysed. We refer to the interface roles obtain by projection as *projection roles* (p-roles)[18].

Definition: Projection role (p-role)
A projection role (p-role) is an interface role describing the actual interface behaviour of a service role visible at a connector endpoint.

Role validation can subsequently be applied to p-roles to ensure that they interact safely. [Floch 2003] did not include validation of liveness properties, which our work addresses by introducing service goals and related validation mechanisms, see chapter 4 and section 6.1 respectively.

2.5.2 Constructive and corrective methods

[Bræk and Haugen 1993] distinguished between *constructive methods* that aim to generate error-free systems, and *corrective methods* that aim to detect and correct the errors that are nonetheless made. Role validation can be applied in two ways:

- As a constructive method, one use of role validation techniques is to generate compatible roles from particular interface roles;
- As a corrective method, role validation is used to check whether two connected roles interact compatibly. This may be checked at design time or at run time.

Rather than directly checking the compatibility of two service roles, one may first check whether or not the interface roles have the properties required to interact safely [Floch 2003]. This is illustrated in Figure 2.11 (adapted from [Floch 2003]).

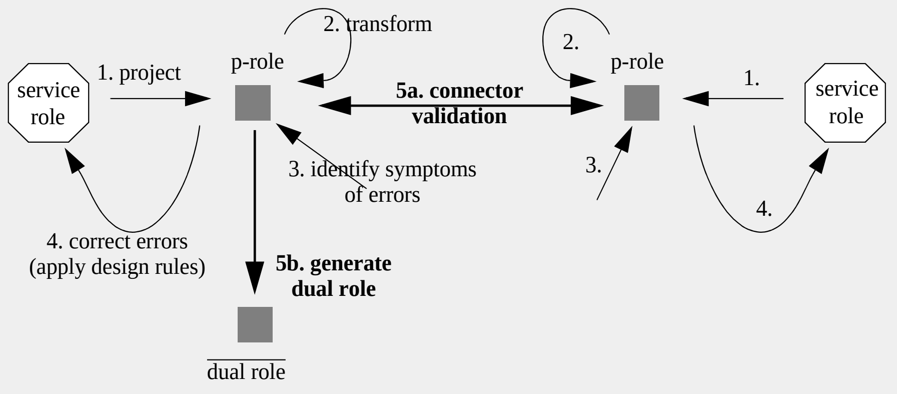


Figure 2.11 : Constructive and corrective methods

18. These are called service association roles (a-roles) in [Floch 2003].

The steps are as follows:

1. Projection is first applied in order to generate p-roles from service roles. P-roles are defined as state machines. Projection can be performed by a tool;
2. The p-role graphs are then transformed in order to simplify further validation operations. This can be performed by a tool;
3. The simple definition of p-roles enables us to detect ambiguous or conflicting behaviours automatically. Results reported by tools seem easy for designers to understand;
4. Design rules have been proposed by [Floch 2003] that support the designer in removing errors and defining well-formed service roles. This step can be assisted by tools;
5. When p-roles obey the design rules, the safety properties of a connection between two roles can be checked (5a), and/or dual interface roles can be generated (5b).

By identifying and removing errors before connector validation we prevent having to analyse poorly designed service roles. The algorithm for connector validation can thus be simplified, and the number of states in the working space used by the algorithm can be kept low. Requiring that each interface role is well-formed does not restrict the useful functionality expressed in the roles, but results in designs that are less likely to cause dynamic errors.

The transformation techniques applied to service roles and p-roles can be performed manually by the designer, or be supported by tools[19]. The validation approach of [Floch 2003], although formal, is arguably easy to understand and use. In addition it is compositional, as dual roles can be used to construct environment roles in a systematic way.

2.5.3 Interaction safety

As interface roles are defined in terms of state machines that communicate through signal exchanges, safety and liveness violations can be characterized in terms of signals and states. We restrict ourselves to the detection and prevention of logical errors. Physical errors such as signal loss, communication channel defect and actor defect lie outside the scope of our work.

Note that parameters of signals are not treated in our work, all issues concerning compatibility of parameter data types are hence not treated.

[Floch 2003] focuses on safety properties, i.e. ensuring that bad things never happen. Unspecified signal receptions[20], deadlocks and improper terminations are classified as violations of safety properties. Unspecified signal reception is a symptom of possible design errors. Therefore strong requirements are enforced on interacting roles: all signals sent by a role should be explicitly consumed by the connected role.

Definition: Unspecified signal reception
An unspecified signal reception occurs when an interface role consumes a signal that is not specified as input of the current role state.

19. The algorithms of [Floch 2003] have been implemented by [Korda 2004], [Alsnes 2004] and [Birkeland 2005].
20. In SDL, signals that are unspecified in an agent's current state are consumed without any transition occurring (i.e. they are discarded from the input queue), and thus unspecified signal reception will not cause any immediate error or failure. In UML2 the handling of unspecified event occurrences is open: "If an event in the [input] pool satisfies no triggers at a wait point, it is a semantic variation point what to do with it." [UML 2.0] p. 420

Definition: Deadlock
A deadlock occurs when two interface roles are unable to proceed because they wait endlessly for signals from each other.

Definition: Improper termination
Improper termination occurs:

- when two interface roles do not terminate in a coordinated manner: no signal should be sent to a role that has terminated;[21]
- when the exit conditions[22] attached to the interface role terminations are not consistent with each other. Two exit conditions are consistent when they represent the same termination cases, or when one of the conditions represents a termination case that covers the termination case represented by the other condition.

Definition: Interaction safety
A pair of interface roles are said to interact safely when their interactions do not lead to any unspecified signal reception, deadlock or improper termination.

2.5.4 Compliancy of role bindings

Service validation also concerns what UML calls *compatibility of role bindings*. We rather say that a classifier is *compliant* with a role that is bound to it.

Definition: Role compliancy
A classifier is compliant with a role bound to it if the interface behaviour of the classifier is a live subtype of the interface behaviour of the bound role.

The concept of *live subtyping* is defined in chapter 4.

2.6 Modelling objectives

In this section we describe our modelling objectives concerning service structures and service goals. In addition we discuss our objectives concerning the use of UML.

The notion of the “horizontal” and “vertical” axes of service composition was introduced in section 1.3.1. Both represent essential aspects of service structures, and we start by discussing what relationships and what behaviour needs to be modelled.

2.6.1 Horizontal relationships between roles

We need to model the “horizontal” relationships between roles in terms of the service they provide. It should be a modular structure where it is possible to factor out roles without premature binding to actors.

Roles interact with each other over connectors. Our primary concern is to model the interaction over a connector using role types. We are not concerned with specifying what service invocations take place during the lifetime of systems (i.e. specifying what are

21. Checking for this is analogous to detecting unspecified signal reception.
22. *State exit conditions are defined by labelled exit points* [Floch 2003] p. 46.

called *binding actions* in RM-ODP). Representing other relationships between actors lies likewise outside the scope of our work, such as the representation of knowledge, including how actors know about each other. Such concerns belong to more detailed design.

2.6.2 Vertical relationships between roles

We need to model the “vertical” relationships between different roles that apply when the roles are played, regardless of what actor plays them. This includes expressing how the achievement of one role goal is a precondition for another role.

A related aspect we want to model is the configuration of roles as they are played by an actor, i.e. to model what roles a given actor type can play, and in what order.

2.6.3 Flexibility when binding roles to actors

Detailing the inner structure of an actor and synthesis of actor behaviour lies outside the scope of our work, we limit our approach to defining how roles may be bound to actors.

Roles should be described in a way that is independent of particular actors, specifying only aspects that are important for the correct functioning of the service.

Binding roles to actors should be flexible, meaning that a role can be bound to any actor capable of playing the role. Roles can then be specifications of behaviour that a potentially wide range of actors can comply with. Actors should be able to take on roles dynamically.

2.6.4 Collaboration behaviour, role behaviour and interface behaviour

A final aspect we want to model is the behaviour of a service. In particular we need to express behaviour that constitutes the successful reaching of service goals, while letting other details remain undefined.

It is desirable that the collaboration behaviour of the service as a whole can be expressed by itself, so that the service can be understood without taking in the detailed behaviour of the roles. This is desirable because role behaviours can in themselves be quite complex; understanding their combined behaviour is not trivial in such cases.

We want to model interface behaviour as a description of its own, separate from the model of role behaviour and collaboration behaviour. Interface behaviour should only focus on the behaviour visible on an interface, and not describe complete behaviour.

Collaboration behaviour, role behaviour and interface behaviour should be modelled in a way that enables us to check whether they are consistent with each other.

2.6.5 UML as modelling language

In this thesis we have used the Unified Modeling Language (UML) version 2.0 [UML 2.0, UML 2.0 Infra, UML2 Ref] as a modelling language to define our approach. In our work we have searched for and used elements in UML as far as they suit our purposes, and have at certain points suggested extensions or different interpretations or semantics. The diagrams that illustrate our approach use UML, at times adding graphical additions of our own. Sometime these additions are discussed, at other times not, especially if the figures are only illustrations, like many in this chapter.

Our objective is not to argue that UML is the most appropriate modelling language for capturing the essence of convergent services. In our opinion, a modelling language is like a tool; if it fits the job then things get done faster. The question is if UML can promote both the human understanding and the mechanical analysis of convergent services. Since UML is widespread, the answer to this question is an interesting research endeavour.

There are shortcomings in UML, the most important being its informal semantics. We have had a long professional experience and preference for the ITU languages, in particular SDL and MSC in their various versions[23], and previous modelling notations like SOM before them [SOM 1981], [Bræk and Emstad 1986]. We have been strong champions of practical methods for system development [Bræk and Haugen 1993], and have co-authored a methodology for their use [TIMe 1999].

Our experience is that all modelling languages leave much to be desired. But since UML is a popular language with much attention from users and tool vendors, our view is that demonstrating our approach in UML increases the chances of its being adopted, given adequate tool support. In addition, we maintain that we are demonstrating new ways of using UML, making use of new opportunities in UML2, such as UML2 collaborations. We believe this can contribute to the improvement of the UML language and encourage it use.

The 2.0 version of the UML modelling language is in our view the first truly promising edition. Through the active participation of key players in the telecom community, UML2 has included a large part of the expressive power of MSC, SDL and TTCN, although lacking the same level of formal foundation.

For goal expressions we use the Object Constraint Language (OCL) version 2 [OCL 2.0]. There are different opinions on the merits of OCL in the ICT community, where some say languages such as Alloy [Alloy 2002] are more suited to fulfilling the objectives of OCL. We do not take sides in this debate, and have chosen to use OCL in our approach[24] and in our examples simply because it is there as part of the UML family.

A note of warning must be expressed about the use of UML2 in the thesis. UML 2.0 has been subjected to considerable changes during the standardisation process. For instance there were substantial changes in the superstructure from [UML 2.0 Adopted] via the revised version [UML 2.0 Revised] to the final version [UML 2.0]. Since we have used many of the modelling elements that are new to UML2, it is likely that these will be subject to more changes in subsequent versions of UML2 than other more stable parts of the language. Hence there may be differences between later versions of UML2 and the one we have based our approach on. References are to the final version [UML 2.0] or to the *de facto* reference book [UML2 Ref] (which in itself deviates somewhat from the various versions of the UML 2.0 standard).

23. For instance [MSC-92, MSC-96, MSC-2000, MSC-2004, SDL-88, SDL-92, SDL-96, SDL-2000]
24. Arguably, our use of OCL can be branded as "misuse", since we do not use it to constrain what constitutes valid UML models, which is the intended use of OCL.

2.6.5.1 SDL profile for UML

Validation of behaviour requires that the dynamic semantics is formally defined. UML deliberately does not define this, identifying instead a number of semantic variation points that can be bound by UML profiles.

Our approach to validation is based on and extends the work of [Floch 2003], which in turn is based on SDL. Therefore we assume that SDL semantics applies when we validate interface behaviour modelled in UML. This implies that we assume the existence of an SDL profile for UML, specifying a number of key assumptions:

1. a FIFO queuing mechanism on ports (signals are consumed in the order in which they are received);
2. message overtaking does not occur on connections (signals are received in the order they are sent);
3. unspecified signal receptions are discarded (signals are not persistent), and similar details.

No such profile currently exists. However, work is in progress at the ITU-T to define an SDL profile for UML. It is scheduled to be completed early in 2007 as a revision of [Z.109 1999]. We assume that this will provide the formal basis needed to support our approach.

3 Modelling services in UML

In this chapter we discuss how services can be modelled in *service structures*. We show how to model *elementary collaborations*, the building blocks from which more sophisticated services can be constructed.

In our work we investigated two modelling approaches in UML, one based on associations or association classes, and one based on UML2 collaborations. The former approach turned out to be too restricted, which is why UML2 collaborations were investigated. The limitations of associations and association classes are discussed in chapter 9.

The structure of the chapter is as follows: we first introduce UML2 collaborations, then we present the modelling approach using UML2 collaborations, and discuss the modelling of service behaviour. A discussion of related work and summaries of method guidelines are provided towards the end.

3.1 Collaborations and collaboration uses in UML2

UML2 collaborations and *collaboration uses*[1] are new to UML, and deserve an introduction[2]. *Collaborations* in UML2 are considerably different from their predecessors in UML1[3]. This is one of the UML language changes aimed at obtaining more flexible encapsulation, and enabling the composition of internal structures from smaller parts.

A UML2 collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality. The behaviour of a collaboration will eventually be exhibited by a set of cooperating objects (in our terms: actors specified by actor types) that interact with each other by sending signals or invoking operations. In other words, the roles of a collaboration are played by actors when the actors interact with each other.

The connectors defined by a collaboration represent sessions between the actors when they perform the interaction behaviour specified in the collaboration. A collaboration specifies a view (or projection) of a set of interacting actors. It describes the required con-

1. Collaboration uses were called *collaboration occurrences* in [UML 2.0 Adopted].
2. Parts of this introduction are adapted from [UML 2.0] and [UML 2.0 Ref].
3. UML2 makes a clear separation between the static structure (the collaboration) and the dynamic structure (the interaction). As a result UML1 collaboration diagrams have become *communication diagrams* in UML2, and are a form of interaction diagram, while UML2 collaborations are expressed in UML2 *composite structure diagrams*.

nectors between actors playing the roles of the collaboration. Several collaborations may be used to describe different views of the same set of actors.

A given actor may simultaneously play several roles in different collaborations, but each collaboration would only represent those aspects of the actor that are relevant to the service described by the collaboration. An actor may simultaneously play several roles of one collaboration, unless specific constraints hinder it[4].

A UML2 collaboration is not directly instantiable[5]. Instead, the interaction defined by the collaboration comes about when the actual interaction takes place between the actors that play the roles defined by the collaboration.

A UML2 collaboration use relates features[6] of the given collaboration type to connectable elements in the classifier owning the collaboration use. Any behaviour defined by the collaboration type applies to the set of roles and connectors bound by a given collaboration use. In this way an interaction among parts of a collaboration applies to the classifier parts bound by a collaboration use.

3.2 Services modelled as UML2 collaborations

We seek ways of modelling service structures in UML that meet the objectives laid out in section 2.6. Recall that in the previous chapter we defined a service as a collaboration between actors playing service roles, using the term *collaboration* in the general sense.

Adopting the modelling views discussed in section 2.6, we start with the horizontal interrelationships, and explore the modelling opportunities offered by UML2 collaborations.

3.2.1 The suitability of UML2 to our service modelling requirements

The introduction to UML2 collaborations in the UML2 reference book [UML2 Ref] does not mention the term “service”, but talks about “an arrangement of objects and links that work together to accomplish a purpose”. However, that UML2 meets our requirements listed in section 2.6 becomes apparent when analysing quotations from [UML2 Ref], starting with UML2 connectors:

> *A connector is a relationship between two roles that exists only within the collaboration. Connectors may have types, which must be associations that constrain the connector to be implemented as a link of the given association.*

UML2 connectors[7] thus correspond to our definition of the term *connector.*

4. An example of such a constrain is that *Caller* and *Callee* cannot simultaneously be played by one and the same actor in the same collaboration use - i.e. an actor cannot through-connect to itself.
5. When we speak about a collaboration instance, we are referring to collaboration roles being played by actors.
6. The term *feature* used here is the UML term, meaning a property, such as an operation or attribute, that characterizes the instances of a classifier.
7. According to the UML metamodel, a connector is not limited to linking just two roles, but may be n-ary. In our approach we restrict them to connecting two and only two roles.

At runtime, objects and links are bound to the roles and connectors of the collaboration. A collaboration instance does not own the instances bound to its roles (unlike an instance of a structured class, which does own its parts). It merely references them and establishes a contextual relationship among the objects bound to its roles for the duration of the collaboration instance. The objects playing the roles must exist previously. They must be compatible with the declared types of the roles to which they are bound.

An object can be bound to one or more roles. If an object is bound to multiple roles, then it represents an "accidental" interaction between the roles - that is, an interaction that is not inherent in the roles themselves, but only a side effect of their use in a wider context. Often, one object plays roles in more than one collaboration as part of a larger collaboration. This is an "inherent" interaction between the roles. Such overlap between collaborations provides an implicit flow of control and information between them.

A collaboration can be used by binding the roles to classifiers within a particular context, such as the internal structure of a class or the definition of a larger collaboration. Such a bound collaboration is called a collaboration use.

A collaboration use may be used to attach a collaboration to a classifier for which it shows an aspect of behaviour.

Collaboration uses therefore enable actors to play more than one role, and supports defining roles and role types independently from the actors playing them. This is a distinguishing feature that makes UML2 collaborations superior to UML associations, as we discuss in the appendix.

Collaborations may be nested. A collaboration may be implemented in terms of subordinate collaborations, each of which implements part of the overall functionality. The subordinate collaborations are indirectly connected by their participation in the outer collaboration.

A collaboration use may appear within the definition of a larger collaboration. In this context, its roles are bound to roles of the larger collaboration, rather than classifiers. The types of the roles of the larger collaboration must be compatible with the types of the roles of the collaboration use.

Consequently, UML2 collaboration uses support the construction of composite services from elementary services.

Note that a collaboration use can only bind aspects within a single classifier. This may be a limiting factor for synthesis of behaviour, as pointed out by [ARTS 2003]. In our present work this does not pose a problem, since we do not address synthesis of actor behaviour.

We conclude that UML2 collaborations and collaboration uses seem to offer what is needed. In the following we see how they can be put to use to fulfil the objectives of our approach.

3.2.2 Definition of terms

We start by defining a number of terms. Firstly, a service[8] is modelled in what we call a *service structure*.

Definition: Service structure
A service structure defines a collaboration by name, and identifies (names) the roles that collaborate to provide the service or service feature. The service structure also defines the multiplicity and type of the roles.

A service structure with two interface roles is called an elementary collaboration.

Definition: Elementary collaboration
An elementary collaboration is a service structure defining the roles and collaboration behaviour for a cooperation between two and only two interface roles.

3.2.3 Service structures

A generic two-party service modelled by a UML2 collaboration is shown in Figure 3.1.

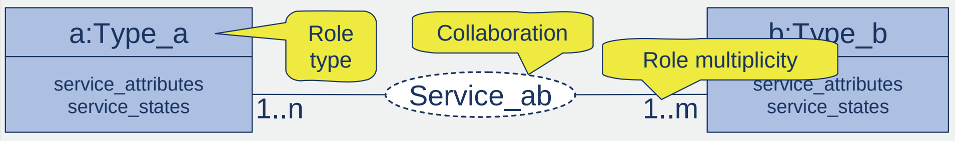


Figure 3.1 : Two-party service modelled using UML2 collaboration

Figure 3.1 depicts a service structure with two roles, of which there can be many instances, as defined by role multiplicities (1..n) and (1..m). It does not specify whether it models interface roles or service roles.

Figure 3.1 uses the standard UML notation for classifiers and the presentation option where a collaboration named *Service_ab* identifies the associated roles *a* and *b* without specifying the connectors between them. Omitting connectors can be useful when there are more than two roles in a service structure, as in Figure 3.2.

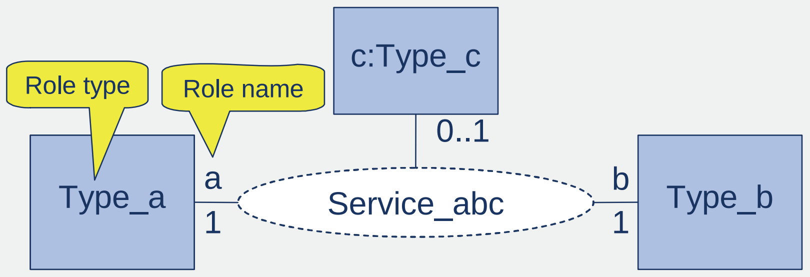


Figure 3.2 : Three-party service modelled using UML2 collaboration

Note that Figure 3.2 exemplifies the two ways of naming roles (see roles *a* and *c*).

8. Or a sub-service, what is called a *service feature*, as defined in [IN 1993].

For two-party services such as the one in Figure 3.1 nothing is gained by using this presentation option, since the two roles must be connected, or else no interaction is possible.

3.2.4 Modelling service roles in UML2 collaborations

UML2 collaborations can be used to model service roles, see Figure 3.3

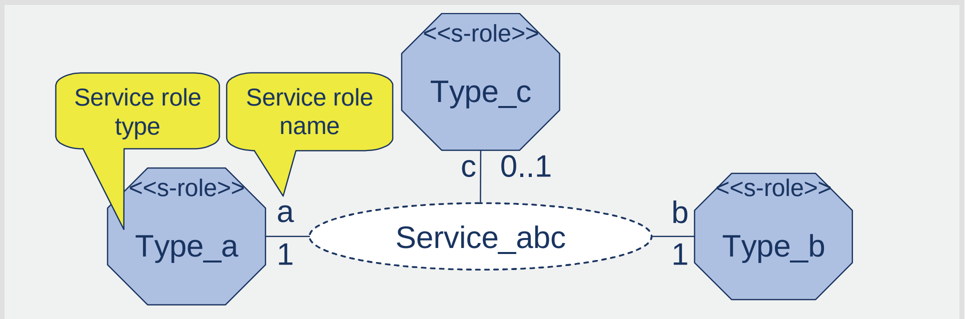


Figure 3.3 : Service roles modelled using UML2 collaboration

In Figure 3.3 we introduce a stereotyped presentation of the role classifier, using an octagonal icon instead of the rectangular class symbol. This is to emphasise that the roles in this service structure diagrams are service roles, adhering to the notation used by [Floch 2003]. We can thus distinguish service roles from interface roles by their different icons.

Connectors are shown in the normal presentation form of UML2 collaborations, see Figure 3.4. Connectors define the interaction paths between the roles.

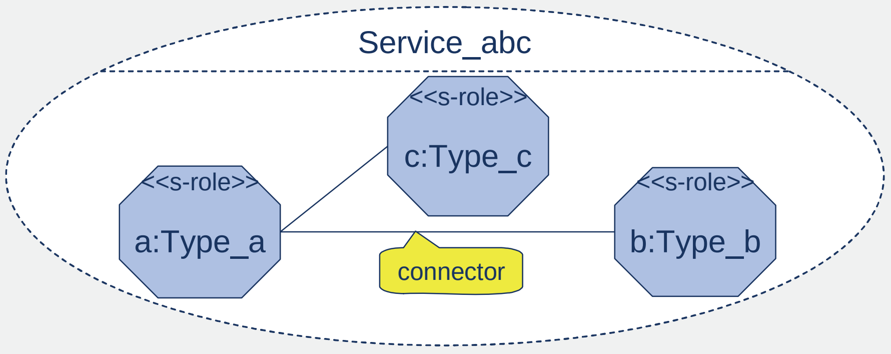


Figure 3.4 : Modelling connectors using UML2 collaboration

Service roles and interface roles are closely related: the behaviour of interface roles characterises the interactions on the communication paths (i.e. on the connectors) between the service roles. Hence, when modelling service roles with UML2 collaborations, we let *connectors* represent connectors, and let *connector ends* represent interface roles. This is illustrated in Figure 3.5.

In UML2, connector ends do not have any graphical representation apart from the multiplicity elements, as Figure 3.3 shows. To achieve a graphical rendition of interface roles, we introduce an icon for connector ends: rectangular icons that can contain the role name.

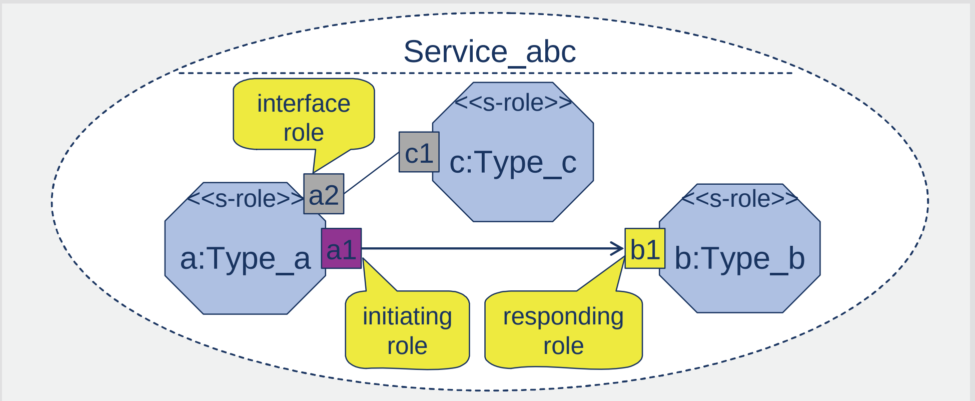


Figure 3.5 : Modelling service roles with interface roles

Using an icon enables us to highlight interface roles in the service structure, as in Figure 3.5, where *a1*, *b1* etc. are the names of interface roles.

A collaboration like the one in Figure 3.5 may refer to many interface roles. The number of interface roles depends on the interaction patterns of the service roles:

```
2(No_s_roles - 1) <= No_i_roles
```

Here `No_i_roles` is the number of interface roles and `No_s_roles` the number of service roles in a given collaboration. The lower limit is due to the requirement that all service roles must be associated with at least one other service role in the collaboration. There is no upper limit, since there is no limit to the number of collaboration uses that can be bound in a composite collaboration. By definition a service role may not have a connector with itself.[9]

The classification of interface roles into "initiating" and "responding" is important for service discovery, as will be discussed in chapter 7. We suggest using a dark colour for initiating roles, and a light colour for responding roles, or adorning the connector with an arrow pointing to the responding role[10]; see the connector between the roles *a* and *b* in Figure 3.5.

The proposed additions can easily be integrated into UML via a profile introducing the stereotypes, which may also use their own graphical notations.

3.2.5 Approach to the modelling of service structures

The suggested approach to modelling service structures in UML2 is as follows:

- We model *elementary collaborations* as UML2 collaborations with two roles of multiplicity 1. Each role of an elementary collaboration defines an interface role name[11], whose role type[12] is used to define the interface behaviour of the interface role;

9. An actor can have sessions with itself if and only if it plays several different roles of the same service.
10. A stereotype of a connector can be defined that uses an arrow notation. Note that there is no placeholder in the standard UML definition to capture the direction of a connector.
11. The role name is the name of the connectable element of the collaboration.

- *Composite service structures* are defined by employing collaboration uses to bind interface roles to service roles. Connector ends define cardinality constraints on service sessions, i.e. the number of service role instances involved in a service invocation;
- We use *classes* to represent service roles that play interface roles, and *collaboration uses* to bind interface roles to service roles. Such collaboration uses represent connectors between service roles. The service roles must be *compliant* with the interface role(s) to which they are bound;
- We also use classes to represent actor types that play service roles, and collaboration uses to bind service roles and/or interface roles to actor types. Such collaboration uses represent service invocations, i.e. the links represent sessions between actors. Actor types must be compliant with their bound roles.

Following this approach, arbitrarily complex composite services may be composed of elementary collaborations, meaning that any number of interface roles can be bound to roles or classifiers by collaboration uses. This implies that a potentially large set of interface roles are ultimately bound to an actor, i.e. the set of interface roles defining or characterising an actor's service repertoire.

There can be an ordering imposed on the sequence of roles played by an actor; this will be discussed in connection with goal sequences in chapter 5. Validating compliancy in role bindings will be discussed in chapter 6.

In the following sections we illustrate the use of this approach by way of examples, starting with elementary collaborations, and then proceeding with the composition of larger services from elementary collaborations.

3.2.5.1 Elementary collaborations and interface roles

An elementary collaboration consists of a pair of interface roles of single role multiplicity. As we shall see in chapter 4, a semantic connector is an elementary collaboration. An elementary collaboration defining a pair of interface roles is shown in Figure 3.6.

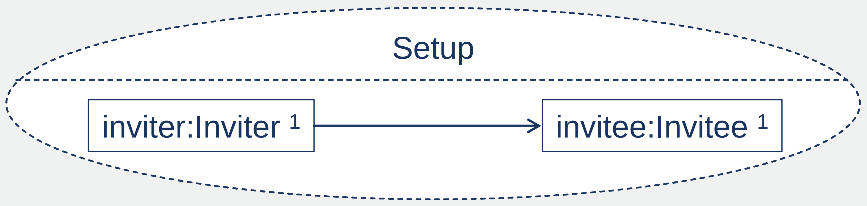


Figure 3.6 : Elementary collaboration and a pair of interface roles

In Figure 3.6 the collaboration *Setup* defines two interface roles, *inviter* and *invitee* of respective role types *Inviter* and *Invitee*. An arrowhead[13] on the connector end attached to *invitee* indicates that *inviter* is the initiating role; *invitee* is a responding role.

12. The role type of the collaboration role is the (optional) *type* of the *typed element* that the *connectable element* of the collaboration inherits from. The initial version of UML2 [UML 2.0 Adopted] did not support this, as the connectable element was a *named element*. This was partially corrected in [UML 2.0 Revised] (Figure 95 was wrong), and fully corrected in the final version [UML 2.0].
13. Arrowheads should be placed at both ends if both roles start with mixed initiatives.

To complete the definition of an elementary collaboration we need to define the behaviour of the interface roles. We defer this to section 3.3.

3.2.5.2 Composite two-party services

Composite services can be composed of elementary collaborations by using UML2 collaboration uses. A Call composed of elementary collaborations is shown in Figure 3.7.

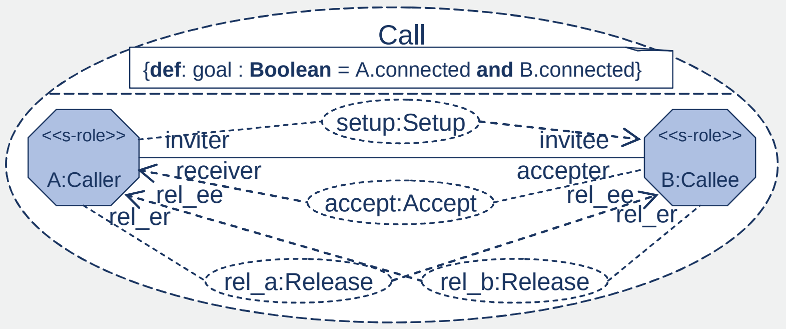


Figure 3.7 : Composite service composed of elementary collaborations

In Figure 3.7 the classic *Call* service is composed of the elementary collaborations *Setup*, *Accept* and *Release*[14]. The *Setup* roles *inviter* and *invitee* are bound to the *A* and *B* roles of *Call* respectively[15], as are the *Accept* roles *receiver* and *accepter*. *Release* has the interface roles *rel_ee* and *rel_er*; in Figure 3.7 there are two instances of *Release*, where the role bindings of *rel_ee* and *rel_er* to the *A* and *B* roles are swapped. This exemplifies how collaboration uses allow elementary collaborations to be reused in a flexible way, in this case to support that each party can take the initiative to release the call.

When binding interface roles to service roles or actors we can differentiate between the initiating and responding roles of the elementary collaborations by using an arrow at the role binding of the responding role. In Figure 3.7 the interface role *rel_ee* is the responding role, and its binding to *A* or *B* is adorned with an arrow. This adornment makes it easy to see that collaboration uses *rel_a* and *rel_b* are initiatives in different directions between *A* and *B*, a so-called mixed initiative, a fact that the experienced service designer knows must be treated with care.

In Figure 3.7 the octagonal icons indicate that the roles of *Call* are service roles, implying that *Caller* and *Callee* are composite states that can be used for the composition of actor behaviour, according to [Floch 2003].

Interface roles can also be composed of elementary collaboration roles, see Figure 3.8.

Figure 3.8 defines a collaboration *Call_Init*, with interface roles *a* and *b* that are composed of the elementary collaboration roles *inviter*/*receiver* and *invitee*/*accepter* respectively. In Figure 3.7 we can replace *Setup* and *Accept* with *Call_Init*.

14. The Basic Call service consists of a number of “normal procedures”, including Call Setup, Call Acceptance and Call Release, according to the standard [Basic Call 1988].
15. The ITU-T standards for services typically describe actor roles such as User A or Originating User, and User B or Terminating customer, which are the service roles of the Basic Call service [Basic Call 1988].

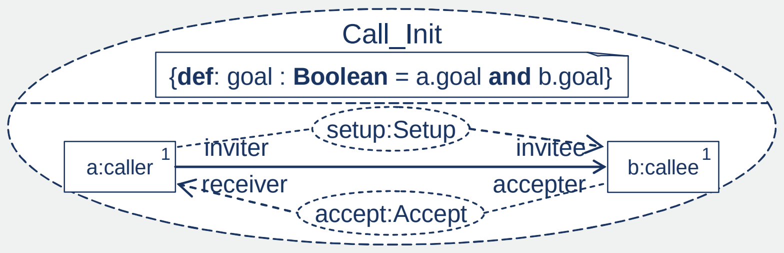


Figure 3.8 : Interface roles composed of elementary collaboration roles

Binding roles to actors with the help of collaboration uses is shown in Figure 3.9.

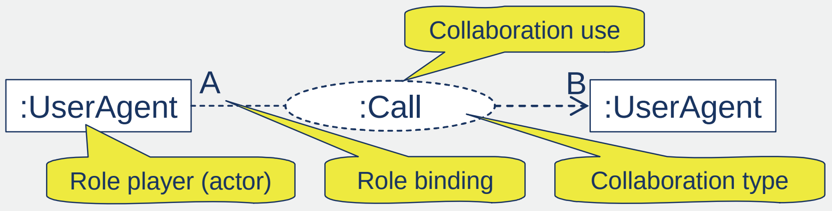


Figure 3.9 : Assigning service roles to actors

Figure 3.9 illustrates how a collaboration use binds collaboration roles[16] to specific actors (instances or types); here both the service roles *A* and *B* of the composite service *Call* are bound to the actor type *UserAgent*. The composition of *Call* in Figure 3.7 implies that the *UserAgent* must be compliant with six interface roles (the union of the interface roles of *A:Caller* and *B:Callee*), i.e. *inviter*, *invitee*, *receiver*, *accepter*, *rel_ee* and *rel_er.*

Note that nothing is said about the number of roles played by an actor at a given point in time, e.g. Figure 3.9 does not specify whether *UserAgents* can partake in several simultaneous originating and/or terminating *Calls* or not. We discuss how an actor's role-playing capabilities can be characterised in section 5.3.

3.2.5.3 Composite multi-party services

Multi-party services (i.e. services with more than two roles) can be composed of subordinate elementary collaborations. An example is shown in Figure 3.10.

Figure 3.10 shows how the roles of three elementary collaborations are bound by collaboration uses to four service roles. The example shows how a service role can play several interface roles; e.g. the service role *CallerAgent* plays the interface role *iu* of the elementary collaboration *initCall*, and the role *a* of the elementary collaboration *Call_Init*. The connectors between the service roles are not shown in Figure 3.10.

A more complex example is provided by the Meeting Place Conference, a service defined in [AMIGOS 2004].

16. In this case service roles are bound to actors, though in principle they could be interface roles bound to actors. The composition of actor behaviour from service roles is a more direct constructive approach than composition from interface roles, see the composition of actor behaviour from composite state machines in [Floch 2003]. Note that synthesis of actor behaviour from service roles or semantic interfaces is not treated in our work.

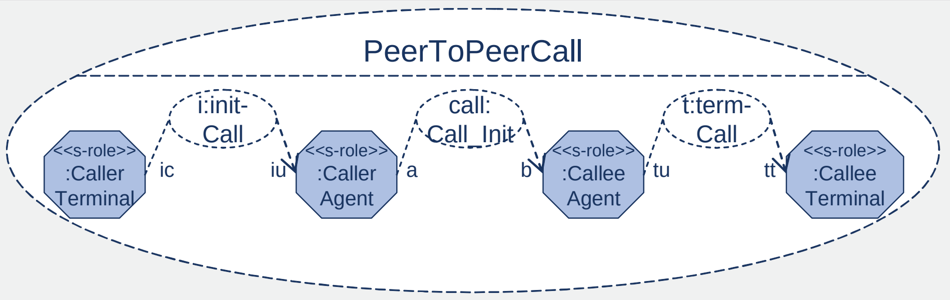


Figure 3.10 : Multi-role service composed of elementary collaborations

As an introduction, consider the sequence diagram[17] in Figure 3.11, which shows a scenario combining traditional telecom services such as conferencing [CONF 1988] with services such as chat rooms, called *Meeting Places* in [AMIGOS 2004].

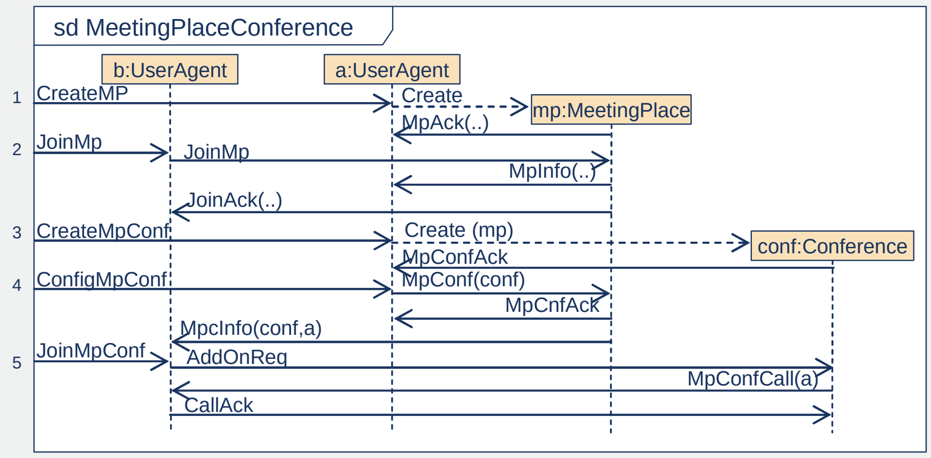


Figure 3.11 : Meeting Place Conference scenario

We choose to consider that the interaction consists of several stages, which are indicated by the numbered initiating signals from the environment in Figure 3.11:

1. User *a* creates and configures a MeetingPlace, becoming the *MeetingPlaceController*;
2. A second user *b* joins a MeetingPlace, and thus becomes a *MeetingPlaceParticipant*;
3. User *a*, the MeetingPlaceController, creates a Conference for the MeetingPlace, thus becoming the *ConferenceController*;
4. User *a*, the ConferenceController, configures the MeetingPlace with the Conference;
5. The MeetingPlaceParticipant *b* joins the Conference, thereby becoming a *Conferee* (in the scenario the only other conference[18] member is the MeetingPlaceController *a*).

From the interactions in Figure 3.11 we see that the Meeting Place Conference service involves four actors of three different types. However, as we pointed out in chapter 2, we

17. Sequence diagrams are rather too detailed at this level of abstraction, but can help to explain the manner of operation. Sequence diagrams typically only describe certain cases, not all behaviour between the entities.
18. A two-party connection can be viewed as a special kind of conference.

consider actors to be playing *roles* in the services. In service modelling it is the service roles and interface roles that interest us. Actors are only interesting in their function as role players and by the domain entities they represent.

We identify four service roles in the *MpConf* service: *conferee*, *controller*, *mp* and *conf*. The service structure of *MpConf* is shown in Figure 3.12 below, using the presentation option for collaborations where only the roles are identified. Figure 3.13 uses the normal presentation form to define the communication paths between the roles of *MpConf*.

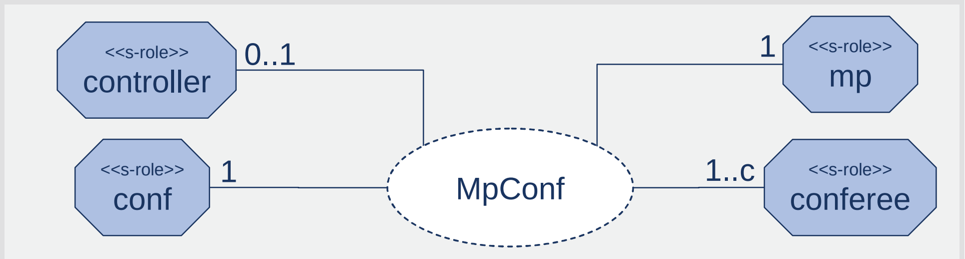


Figure 3.12 : N-party service modelled as a collaboration - without connectors

In accordance with our approach, multiparty services are composed of a set of elementary collaborations. The elementary collaborations of *MpConf* are:

1. *Mp*: a Meeting Place is created and configured;
2. *MpSession*: someone joins a Meeting Place;
3. *MpCnf*: a Conference is created for the Meeting Place;
4. *Mpc*: the Meeting Place is configured with information about the Conference;
5. *MpcInfo*: participants of the Meeting Place are informed of the Conference;
6. *MpcAddOn*: a Meeting Place participant joins the Conference.

MpConf composed of these elementary collaborations is shown in Figure 3.13.

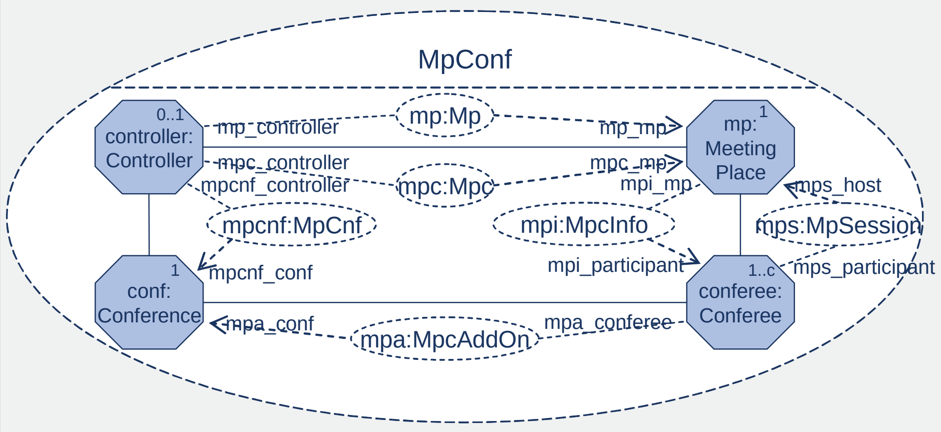


Figure 3.13 : N-party service composed of elementary collaborations

This is an example of a bottom-up approach to service specification, arriving at a domain level model based on an existing service design. During service specification one can iterate between a top-down and bottom-up approach. However, the development process employed lies outside the scope of our work.

3.3 Service behaviour

Since UML2 collaborations inherit properties from both structured classifiers and behavioured classifiers, they have a large range of expression forms at their disposal. In addition to expressing structural relationships, it is possible to express all forms of behavioural aspects of collaborations, such as interactions, activities and state machines. The standard [UML 2.0] and reference book [UML2 Ref] focus mainly on the structural features of collaborations, and provide few guidelines on how the behaviour of a collaboration is related to the behaviour of its constituent parts, i.e. the role types[19].

The collaborative behaviour of services can therefore be defined in a variety of places:

- in the context of the collaboration, as interaction diagrams;
- in the collaboration itself, in the form of state machine, interaction or activity diagrams;
- in the role types, as state machines or interaction diagrams.

Defining behaviour in several places offers opportunities for validation between different behaviour descriptions; this is treated in chapter 6.

3.3.1 Activity diagrams

Activity diagrams highlight the coordination of execution of subordinate units, with actions connected by flows from outputs from one node to inputs of another. They focus on the process of computation rather than the objects performing the computation or the data involved. Activities concern the states of a computation, possibly across many objects, and explicitly model the flow of control and information among nodes.

Service behaviour can conceivably be modelled using activities, where each collaboration or role is an action. This perspective is useful in expressing the coordination between services and between roles of different services.

[Kraemer and Herrmann 2006] use activity diagrams to specify behavioural aspects of collaborations, as this aligns well with their use of compositional Temporal Logic of Actions (cTLA) as an underlying formalism. Our work leans on the formalism of SDL, and we prefer to use state machines and interactions to specify the behaviour of collaborations. We use activity diagrams to model actor goal sequences in chapter 5.

3.3.2 Interaction diagrams

Two kinds of interaction diagrams are found in UML2: communication diagrams[20] and sequence diagrams[21]. Both typically define partial behaviour, but their focus is different:

- Sequence diagrams show object interactions in a time dimension. They do not show object relationships.

19. [UML2 Ref] p. 227 states that behaviour in collaborations may be described by interactions. Clearly this is not the complete story.
20. Communication diagrams in UML2 were called *collaboration diagrams* in UML1.
21. Sequence diagrams have undergone a major revision in UML2 compared to UML1.

- Communication diagrams show interactions organized around parts of a composite structure or the roles of a collaboration, and explicitly show the relationships among the elements. Communication diagrams do not show time as a dimension, so sequence numbers must be used to determine sequences of signals and concurrent threads;

Communication diagrams have a cumbersome sequencing technique, while the relationships between objects that they support are adequately covered by service structures.

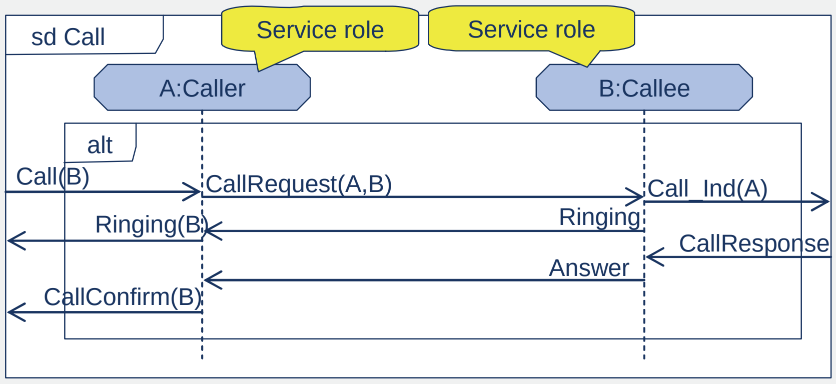


Figure 3.14 : Sequence diagram for composite Call service

In the sequence diagram presented in Figure 3.14 the lifelines use a stereotyped icon with an octagonal shape for the name compartment instead of the standard UML rectangle. This is something we suggest in order to highlight that these lifelines represent service roles and not actors or interface roles.

Note that service roles exchange signals with the environment, not just with each other. Interface roles on the other hand exchange signals only with each other, see Figure 3.15.

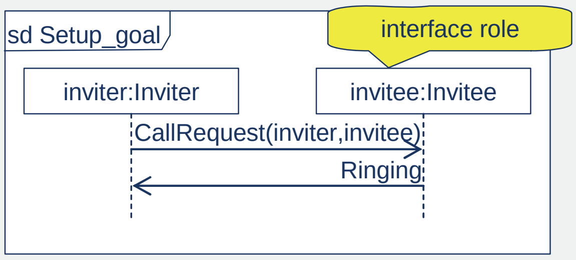


Figure 3.15 : Sequence diagram for the elementary collaboration Call Setup

3.3.3 State machines

State machines are well suited for describing the complete behaviour of classifiers, as opposed to the partial behaviour normally defined in interactions. In the following sections we will discuss:

- The limitations of current UML in terms of defining interface behaviour
- Our suggested extension of UML to define interface behaviour

- The modelling of role behaviour and actor behaviour, exemplified by SDL
- The modelling of collaboration behaviour in UML

3.3.4 Interface state machines

Defining interface roles is supported by UML2 collaboration role types, and the definition of behaviour of classifiers such as role types is possible using UML state machines. It would seem natural to connect or bind interface roles to UML interfaces.

Interfaces and thereby ports can be associated to UML *protocol state machines*, but as we shall see, their semantics is not suited to the general description of interface roles, or indeed of general two-way signal exchange between actors defined by state machines. Protocol state machines are used to specify the legal sequence of operation calls and signals received (or consumed) by an object, and thus only tell part of the story, since output is not specified.

If protocol state machines were used to type an interface, the signals sent and the signals received would have to be split into two protocol state machines, one for signals received, and one for signals sent, see Figure 3.16.

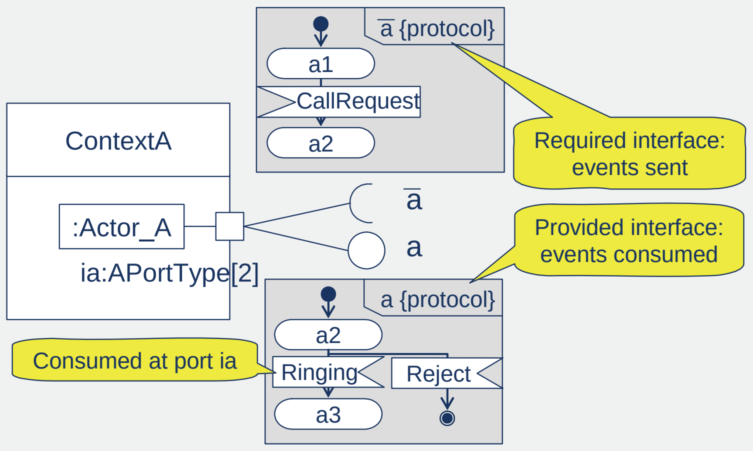


Figure 3.16 : Interfaces and UML protocol state machines: not recommended

In Figure 3.16 an anonymous (unnamed) instance of the actor type *Actor_A* is attached to a port *ia* of type *APortType*, which has two associated interfaces: one for signals received (provided interface *a*) and one for signals sent (required interface $\overline{a}$). Both interfaces are described by protocol state machines, fragments of which are shown. The protocol state machine for signals sent, $\overline{a}$, must be expressed as the sequence of signals that would be received by its connected role. This is confusing for the reader, since output is modelled as input, and causes alignment problems during validation, as we shall see.

Validation of connected roles amounts to the situation depicted in Figure 3.17.

When interfaces are described in this way, validation consists of comparing two pairs of protocol state machines: $\overline{a}$ with *b*, and *a* with $\overline{b}$. In addition, the protocol state machines that type an interface (such as *a* and $\overline{a}$) must be aligned. This alignment can be done

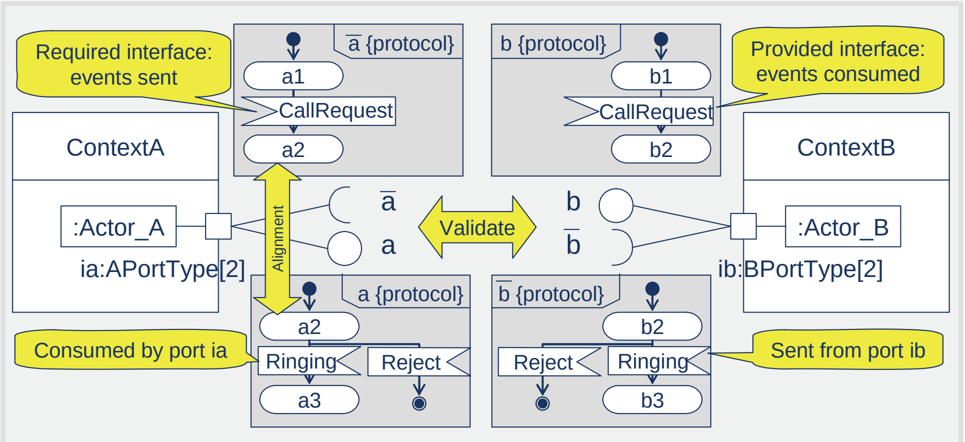


Figure 3.17 : Validation of connected protocol state machines

through conditions (i.e. state names), for instance the postcondition *a2* of *a* is a precondition *a2* of *a*.

Special consideration must be given to final states; if a final output is followed by an input, the final output of the output state machine must not be followed by a final state, since a final state cannot be aligned to a state in the corresponding input protocol state machine. Instead, a dummy final state must be named, plus a conditional transition to a final state based on the input protocol state machine having finished. This is not the case in Figure 3.17.

This approach seems to be rather cumbersome. It is also counter-intuitive, since output is modelled as input. Hence we do not suggest pursuing this further.

The current limitations of UML interfaces would be overcome, if UML supported the definition of interface behaviour in terms of interface state machines that model the combined input/output behaviour of a component at the interface. The solution we propose is to extend protocol state machines to include output, see Figure 3.18.

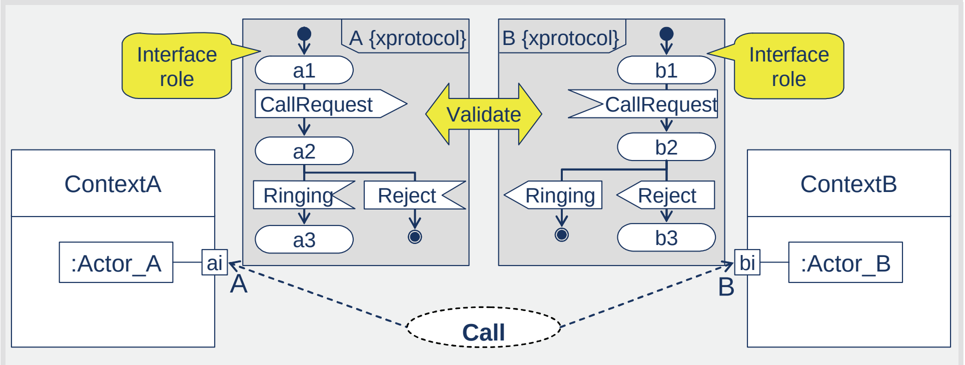


Figure 3.18 : Binding interface roles to ports: extended protocol state machines

Figure 3.18 shows the “extended” protocol state machine of a interface role being bound to actor ports. This is similar to the Port State Machines (PoSM) proposed by [Mencl 2004], and also to the port state machines originally part of [ROOM 1994].

The transitions defined by the extended protocol state machines are what we need to describe interface behaviour. They differ from normal state machines in that they contain spontaneous output and spontaneous transitions (τ-transitions as used in [Floch 2003]).

The lack of support for asynchronous signals in interfaces has been pointed out previously [Bræk 1999]. One can hope that UML interfaces in the future will be able to support interface roles[22]. We believe there is a strong argument for the extension of protocol state machines in UML to include two-way interface behaviour.

3.3.5 Interface behaviour

Expressing the interface behaviour of interface roles is of central importance in our approach. These role behaviours are not complete, as they only describe the collaboration behaviour over a connection to an opposite role. But they are complete in the sense that they accurately specify the total behaviour of the interface role.

We suggest that interface behaviours are defined in what we call *extended protocol state machines*, see Figure 3.19. We motivated the extensions in section 3.3.4 above.

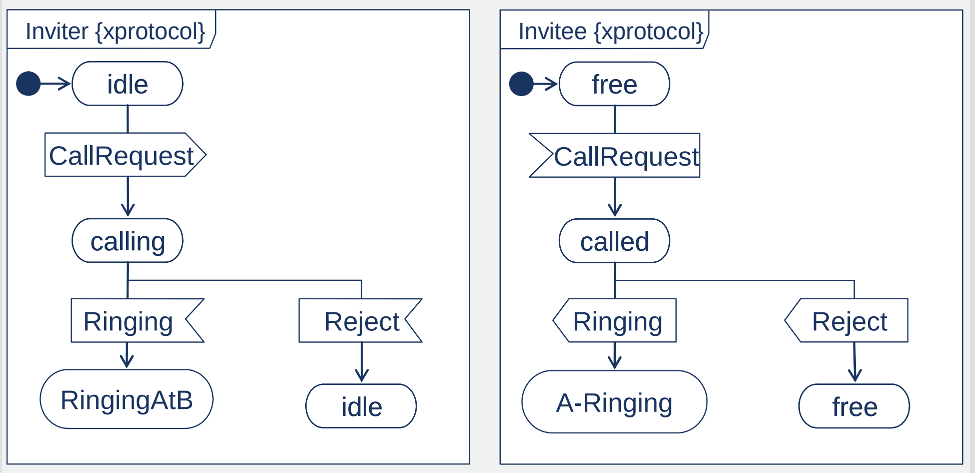


Figure 3.19 : Role behaviour for interface roles

To express interface behaviour we need only model the sending and consumption of signals, the changing of states visible on the connection and any event goals that apply. Behaviour not visible at the interface is removed, such as the calling of internal actions, the manipulation of internal data, and the setting and resetting of timers. While service roles may make use of all the expressive power of UML state machines, few of these constructs[23] are needed for defining interface behaviour.

22. Bran Selic has suggested in private communication that OCL postconditions to signal input events can be exploited to express output, and OCL being part of UML implies that UML thus supports two-way interfaces. In our view this solution is not satisfactory, and we would suggest changes to UML *per se*.
23. Including constructs such as state redefinition, composite states, submachine states, certain types of pseudostates (deep and shallow history), state activities (do-activities), entry and exit behaviour, time events, change events, guarded transitions, history states and conditions, and multiple transitions triggered by an event.

On the other hand, we need to express state changes due to interactions on other connections or interfaces not visible to the interface in question. For this reason the modelling of spontaneous state changes is a necessary part of extended protocol state machines.

Note that when we define input behaviour, we are in fact expressing signal consumption, not signal reception. This is due to the SDL semantics that underlie our approach, inherited from [Floch 2003]. SDL attaches FIFO input queues to agents, and the state machines describe the consumption of signals from the input queue, not the actual input of the signals into the queue. UML does not make any distinction between reception and consumption, since event pools and input queues are a semantic variation point in UML.

The run-to-completion semantics of UML state machines suits our modelling needs. According to this semantics, events are detected, dispatched and then processed by state machines, one at a time. The order of dequeuing is not defined in UML. Events that are not enabled and not deferred are discarded, i.e. incoming signal events are not persistent.

The following elements of UML state machines are used to model interface behaviour[24]:

- simple states with a single region;
- deferred triggers (needed for projection of p-roles if the service role has deferred triggers);
- transitions with triggers (consumption of incoming signal events), guard constraints (for spontaneous transitions) and activity expressions (for outgoing signal events);
- pseudostates for entry points (for initial transition), exit points, terminate, choice and junction (the latter if elegant graphical rendering is needed);
- final states.

Figure 3.19 shows the interface behaviour of the interface roles *Inviter* and *Invitee*, which are part of the elementary collaboration *Setup* described in Figure 3.6 on page 49.

Interface behaviour specifies one event per transition. Interface behaviours allow outputs directly following a state, i.e. spontaneous output. An example is the *CallRequest* output in the *idle* state of the *Inviter* role in Figure 3.19. In a service role bound to play such an interface role, one would find this output in a transition caused by input on another connection, e.g. from *init-call* in Figure 3.10 on page 52.

3.3.6 Service role behaviour

The behaviour of a service role is defined by a composite state, see Figure 3.20.

Figure 3.20 is an example of a state machine for a service role, and is designed according to the SDL techniques suggested by [Floch 2003]. The composite state *csCaller* can be used in the composition of actor behaviour, as shown in Figure 3.26 on page 65. The composite state *csCaller* is entered by a *UserAgent* actor upon consumption of the *Call* signal. This signal is received from the environment in Figure 3.14 on page 55.

24. Parameters of triggering events and assignment specifications for the handling of signal parameters is not included. Taking this into consideration involves the treatment of attribute typing, and is left for further work.

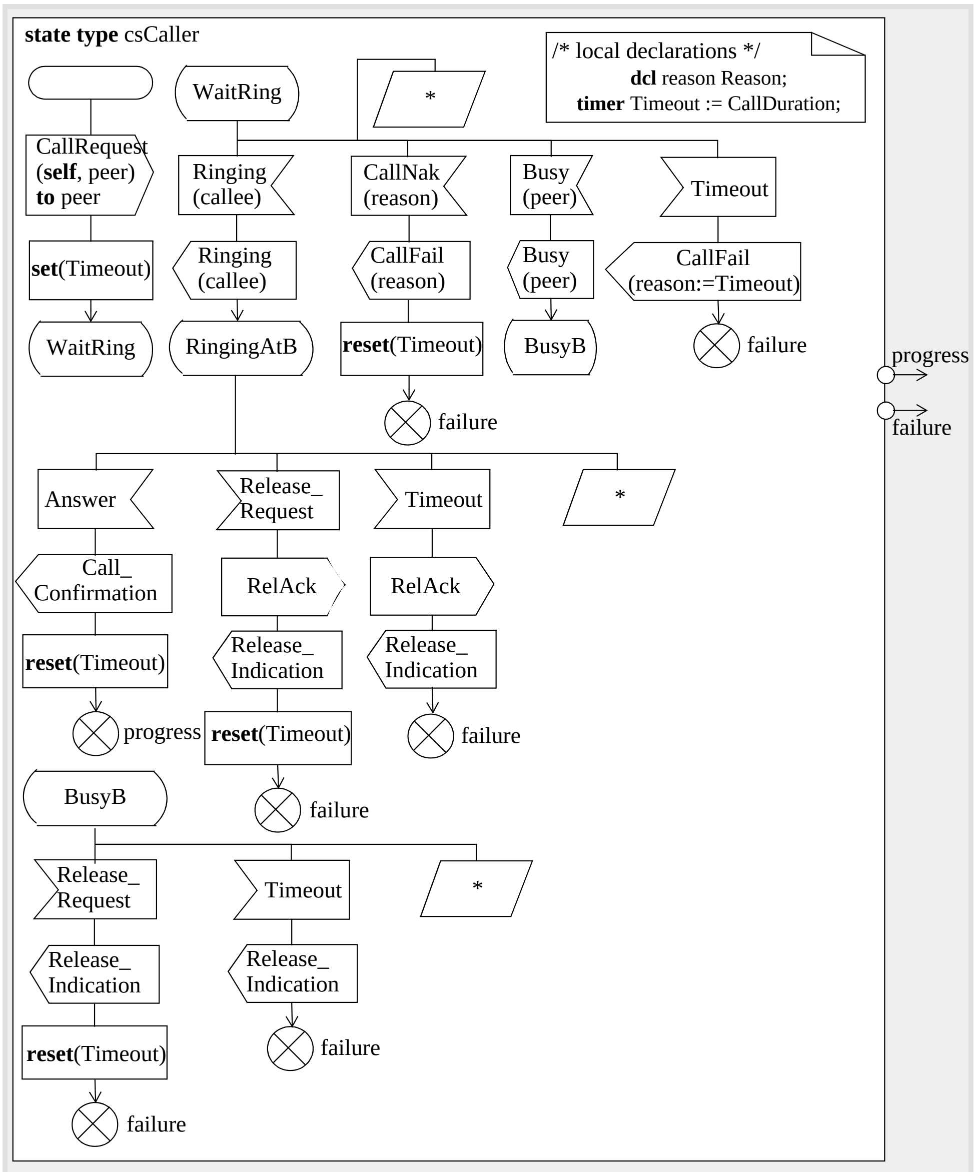


Figure 3.20 : SDL state machine diagram for service role Caller

The exit label[25] *progress* is a *progress label*; these will be discussed in chapter 4.

Using projection one can analyse the signals that flow on the various connectors, and validate that *csCaller* behaves like the interface role *Inviter* in Figure 3.19 on the connection to an *Invitee*. I.e. *csCaller* is *compliant* with *Inviter*, as will be discussed in chapter 6. Note that unlike the interface role *Inviter*, the service role *csCaller* has no spontaneous output[26].

25. *Exit label* is an SDL term; in UML they are called *exit points*.
26. The output of *CallRequest* is part of an initial transition that fires when the service role is initiated by the actor, and is not a spontaneous output.

3.3.6.1 Overview of service role behaviour

In order to focus on high-level service descriptions it can be desirable to make simplified sketches instead of specifying complete behaviours. Sketches of the role behaviour of the role types *Caller* and *Callee* are found in Figure 3.21.

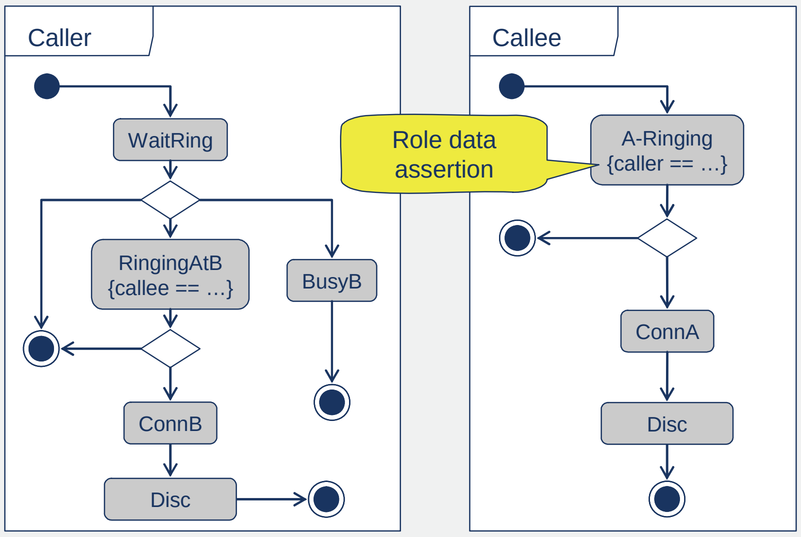


Figure 3.21 : Sketch of service roles of the Call service

The state machine diagrams in Figure 3.21 show an overview of some of the states of the role types of *A* (*Caller*) and *B* (*Callee*) of the *Call* service, without detailing the transitions between the states or conditions in choices. It may also be useful to include invariants (assertions) that role attributes have certain values in certain states, such as the *callee* attribute having a valid value in the *Caller*'s state *RingingAtB*. *Goal assertions* are discussed in chapter 4 (these were not included in Figure 3.20).

3.3.7 Collaboration behaviour

A collaboration does not represent a separate entity with its own capability to perform behaviour. On the contrary, all collaboration behaviour is a result of the behaviour of the constituent parts, and a description of collaboration behaviour only serves to characterize the role behaviour of the actors that the roles eventually are bound to.

In Figure 3.14 above we provided an example of an interaction. This interaction can be used to characterize a collaboration, and be part of the package defining the *Call* collaboration.

UML also allows us to define collaboration behaviour in a state machine diagram. Figure 3.22 provides an example of a collaboration state machine.

In Figure 3.22 an overview of the states of the collaboration *Call* is described in a state machine diagram. In this case details of the transitions between the states are not defined.

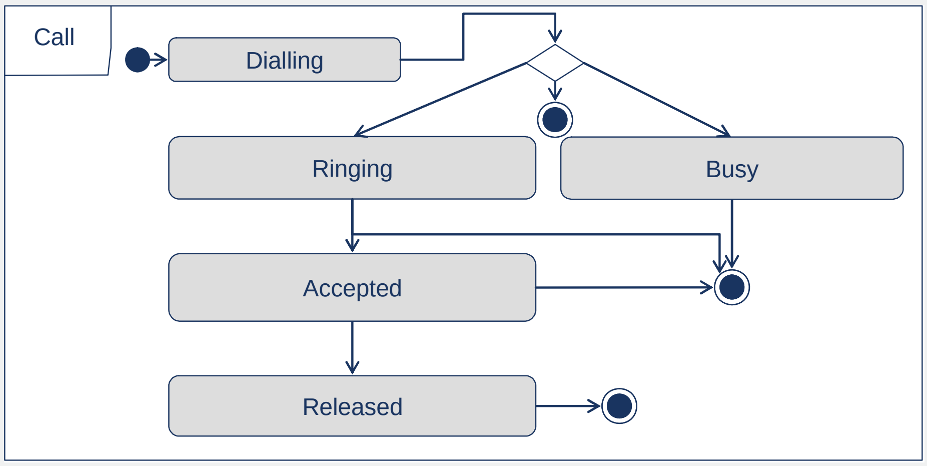


Figure 3.22 : Overview of collaboration states for the Call service

But what does a collaboration state really represent? How can one consider a collaboration state as being apart from the states of the roles? Here there is no answer to be gained from the UML specifications, and the result is open to interpretation.

We note that the collaboration does not represent an object type, i.e. no single object is ever in a collaboration state. Rather it is so that collaboration states represent sets of state tuples representing the joint states of the participating roles. Role states, on the contrary, represent states that must ultimately be found in actors playing the roles. Hence, we consider a collaboration state to be a conceptual state that can be used to express conditions on the combined states of the roles[27] during the collaboration.We adopt a *state-oriented* view, where each state of the collaboration is described in terms of appropriate states of the collaborating roles, see Figure 3.23.

In Figure 3.23 the states of the roles that are part of the collaboration are inserted into the collaboration states of Figure 3.22. Here the roles are service roles, represented by octagons. For each collaboration state the relevant role icons contain a rounded rectangle representing a role state assertion. This is a notational extension to UML. Technically they are icons inserted into the state compartments of the collaboration states.

For instance in the collaboration state *Dialling*, it is stated that role *A* should be in state *WaitRing*, while nothing is stated about the *B* role; it can implicitly be in any state. In the collaboration state *Busy*, role *A* must be in state *BusyB*, while role *B* can be in any state[28]. In the collaboration states *Ringing* and *Accepted* there are assertions about actor attributes: the value of *A*'s attribute *callee* should be equal to the identity of the player of the connected role *B*, and vice versa for *B*'s attribute *caller*[29].

27. In the UML metamodel behind this example (see Figure 3.7), the collaboration roles *A* and *B* are the names of connectable elements, while *Caller* and *Callee* are the type of the connectable element. It is the latter that are in states, thus when we say that a role is in a state, we imply that an instance of the role type is in this state.
28. An example of a detailed design decision is whether or not the *B* role is actually played at this point.
29. The scope of the collaboration states and the invariants are within the particular connector.

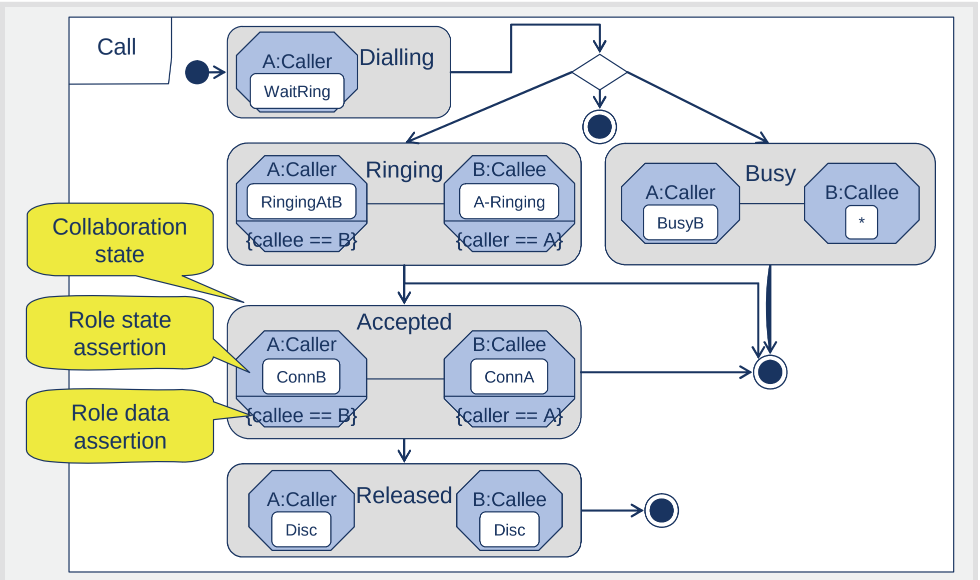


Figure 3.23 : Collaboration states for the Call service with state orientation

Validating that the role behaviour and the collaboration behaviour descriptions are consistent is a validation opportunity that can be exploited, and will be discussed in chapter 6.

3.3.7.1 Role states with collaboration state assertions

As stated earlier, there is a relationship between role behaviour and collaboration behaviour. It may be desirable that this relationship is made explicit, so that designers gain a better understanding of a design, and so that tools can validate the consistency of the relationships.

This may be achieved using some graphical enhancements to UML state diagrams to express relationships from the role behaviour to the collaboration behaviour. The states of the roles in Figure 3.21 can be defined in a state-oriented fashion, where the state of the collaboration is asserted in each role state, see Figure 3.24 and Figure 3.25.

The state symbols in Figure 3.24 and Figure 3.25 detail object-internal issues such as the values of certain attributes and what timers are running. We have included a graphical rendering of the service role(s) in each state, indicating the appropriate collaboration state corresponding to each role state, in accordance with the collaboration states defined for the *Call* service in Figure 3.22.

The graphical representation of the situation of an actor state, as exemplified in Figure 3.24 and Figure 3.25, is inspired by what was called *state orientation* in [SOM 1981]. Including such a graphical description of the state situation is a notational addition to UML. In addition to the opportunity for improved quality assurance (see discussion in section 6.3.3.2), we believe they improve readability by being goal oriented rather than activity oriented.

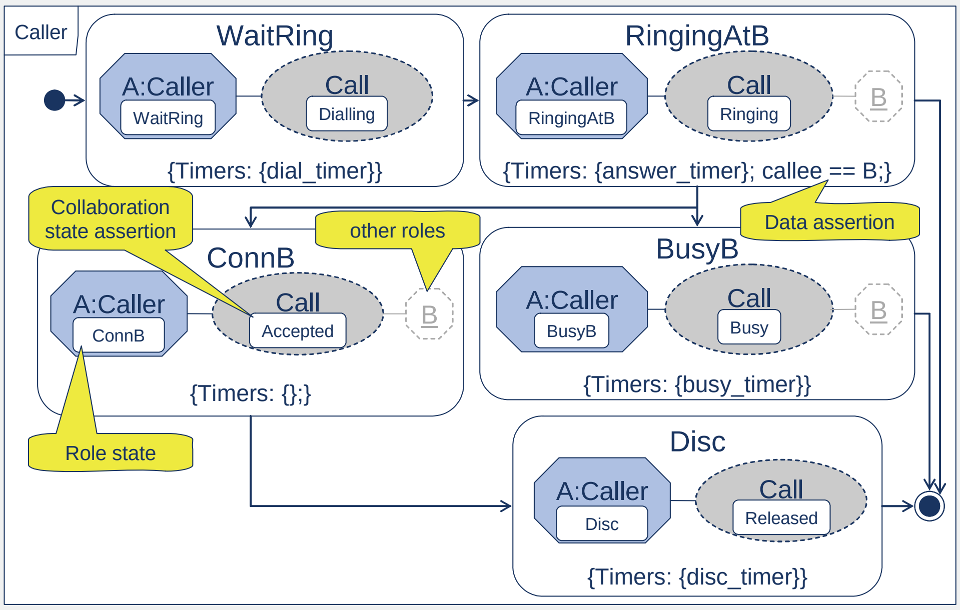


Figure 3.24 : Role states for Caller role with state orientation

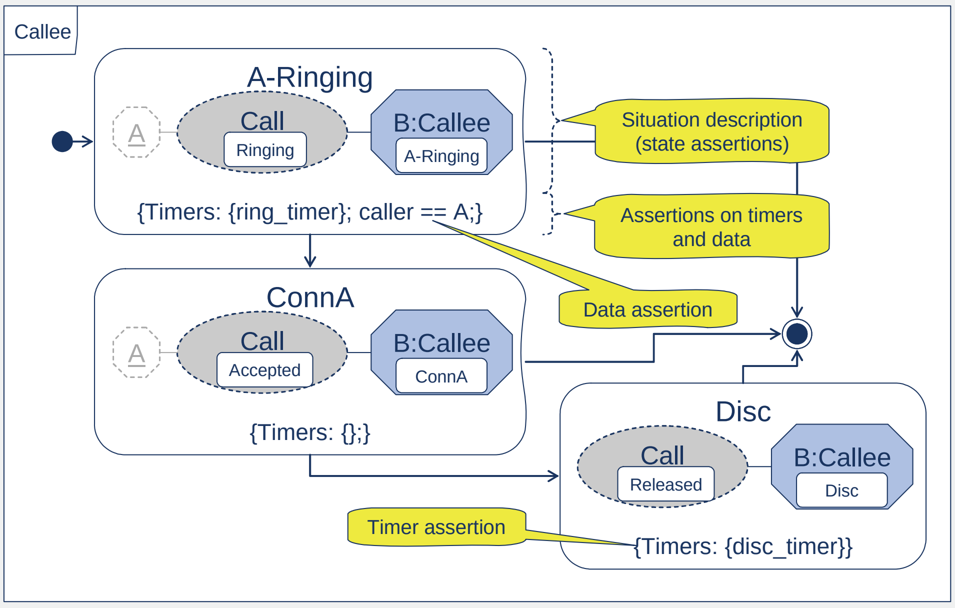


Figure 3.25 : Role states for Callee role with state orientation

We also suggest giving room in the state descriptions for optionally referring to opposite roles by rendering them in grey-toned text and lines, and within dashed octagons. This is to emphasise that the actor has a reference to them; for instance in the state *RingingAtB of* Figure 3.24 the role attribute *callee* should be equal to the identity of the actor playing *B*.

3.3.8 Actor behaviour

Ultimately it is actors that play service roles and that exhibit interface behaviour. Actors are active objects whose behaviour is defined by state machines. In this thesis we do not address synthesis of actor behaviour, and limit our discussion to an identification of some of the issues involved.

[Floch and Bræk 2003b] discuss actor composition, and provide rules for synthesizing actor behaviour from service roles using SDL. They define service roles using SDL composite states, while actors use these composite states in their behaviour graphs.

An example showing this approach is shown in Figure 3.26; here we see part of the behaviour of a *UserAgent* actor (the *Caller* behaviour) using composite states in SDL.

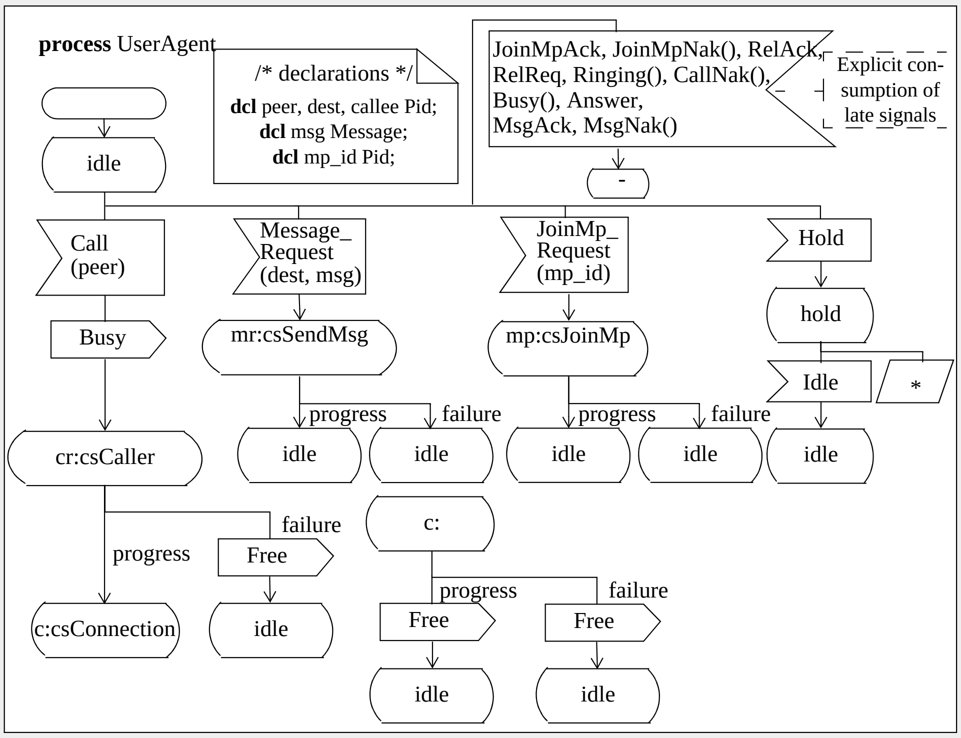


Figure 3.26 : SDL description of a UserAgent actor type (excerpt)

Figure 3.26 is an example of actor behaviour. Again no spontaneous output is present[30]. Note that the composite state *csCaller* (defined in Figure 3.20 on page 60) is called after the output of *Busy*. The other composite states that likewise represent service roles, i.e. *csConnection*, *csSendMsg* and *csJoinMp*, are not detailed.

30. Note that the output of *Free* is not spontaneous, but follows an exit from the composite state machines.

3.3.8.1 Overview of actor behaviour

Rather than specifying full actor behaviour including all the necessary detail, it may be desirable to specify an overview of actor behaviour, i.e. a simpler, less complete description. This is consistent with an approach to service modelling at a high level of abstraction.

An example of a overview of an actor's behaviour is given in Figure 3.27.

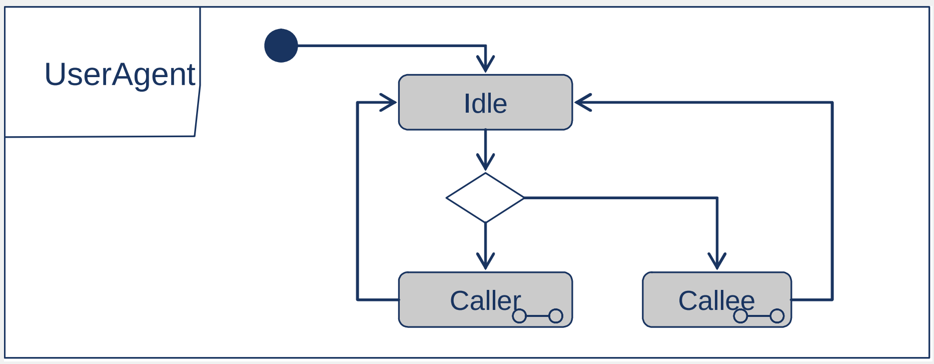


Figure 3.27 : Overview of Actor behaviour emphasising service role selection

The state machine diagram in Figure 3.27 gives an overview of (some of) the states of a *UserAgent* actor type, without detailing the transitions between the states. The figure shows an actor that can alternately play both the role types *Caller* and *Callee* of Figure 3.21. According to Figure 3.9 on page 51 the *UserAgent* must be able to play both roles, though not necessarily simultaneously.

3.4 Related work

The understanding that services involve collaboration between distributed components is not new; indeed, this was recognized since the early days of telecommunications. In terms of modelling, the interaction of collaborations, various dialects of interaction diagrams existed prior to the first standardization of MSC [MSC-92]. A slightly different approach was taken in the use cases of OOSE [Jacobsen et alia 1992], where interactions were described textually. However, interactions alone do not really cover the structural aspects of the roles and the flexible binding of roles to classifiers.

Collaborative designs such as protocols have traditionally been specified by state diagrams, using combinations of informal descriptions and formal models, e.g. using SDL ([SDL-88], [SDL-92], [SDL-96] and/or [SDL 2000]) or similar (e.g. [Harel 1987], [Estelle 1989], [ROOM 1994]). But while state diagrams describe complete object behaviour, the overall goals and the joint behaviour tend to be blurred.

The concept of role was already introduced in the end of the 70s in the context of data modelling [Bachman and Daya 1977] and emerged again in the object-oriented literature. Using roles for functional modelling of collaborations was of primary concern in the OOram methodology [OOram 1995], and was one of the inputs influencing the UML work on collaborations in OMG. Within teleservice engineering it has been a long-standing convention to describe telephone services using role names like A and B. [Bræk 1999] classified different uses of the role concept, and pointed out that UML1 was too restrictive, since a *ClassifierRole* could bind to only one class, so they were not independent concepts that could be reused in different classes.

The work of [Rössler et alia 2001], [Rössler 2002], [Rössler et alia 2003] suggested collaboration-based design with a tighter integration between interaction and state diagram models, and created a specific language, *CoSDL*, to define collaborations. CoSDL was aligned to SDL-96. [Floch 2003] also proposed a notation for collaboration structure diagrams, where components were designed in SDL-2000.

Modelling collaborating services with UML2 collaborations has earlier been suggested by Haugen and Møller-Pedersen [ARTS 2003], and has been an important input to our work.

In the FUJABA approach described in [Burmester et alia 2004], so-called coordination patterns are used for similar purposes as our semantic interfaces. They use a model checker to provide incremental verification based on the coordination patterns.

Illustrating situations has been suggested by [Diethelm et alia 2002]; they use communication diagrams to illustrate use cases and to illustrate do-actions in states. This is similar to the state-oriented view of role states in collaboration behaviour (and vice-versa).

Our work is also related to ongoing standardization work for harmonization between networks. The so-called "meta-protocols" in [TIPHON 2003] constitute an attempt to abstract services from particular networks. TIPHON, as with Intelligent Networks [IN 1992- 2001], has a network-centric viewpoint; we advocate that a service-centric approach should be adopted, as discussed in [Sanders 2002].

Service engineering that pays attention to the horizontal dimension has been suggested within the automotive domain [Krüger 2003][Krüger et alia 2004][Deubler et alia 2005]. Services are defined in a combination of an extended MSC language and UML Use Cases (using *extends* and *uses* relationships). Their approach includes role mapping to system architectures, and they describe tools to support the generation of state machines from service descriptions. Their extensions to MSC include succinct expressions of *broadcast* signals (including responses) and *preemption* ("exceptional" signals). They express progress or liveness properties (i.e. service goals) in what they call *triggers*, and compose services using a number of operators, e.g. *join* and *preemption*. Trigger composition, entailing the joining of liveness expressions with functional scenarios, enables compact expressions of progress that can be validated (proof obligations).

Compared to our approach, their focus on the automotive domain means that broadcast and preemption signalling is important, while for the convergent services we address, it suffices to support general signalling mechanisms. They have not exploited the opportunities provided by UML2 collaborations; this is a distinguishing feature of our approach.

3.5 Method guidelines

We sum up our modelling approach in a number of method rules.

Method rule: Identify elementary collaborations
Elementary collaborations are identified by a unique collaboration name <*feature*>.

Method rule: Define interface role behaviour of the interface roles
Define the interface behaviour of each interface role in a state machine diagram (an extended protocol state machine).

Method rule: Identify composite services
Composite services are identified by a unique name <*service*>. Use this name as the name of the composite collaboration that defines the service.

Method rule: Identify roles (interface roles or service roles) of composite services
Service roles or interface roles are identified by role names. These must be unique within the collaboration defining the composite service.

Method rule: Identify service role multiplicity of composite services
For service roles in a composite service, state the multiplicity of the role, according to the characteristics that apply to the service invocations. Multiplicities can be optional (0..1), unary (1..1) or n-ary, i.e. (0..n) or (1..n).

Method rule: Compose services from elementary collaborations
Bind the interface roles of elementary collaborations to the (s-)roles of the composite service (or to other composite interface roles composed in the same way). Use a unique name for the collaboration use within the collaboration defining the composite service.

Method rule: Compose composite role behaviour from interface role behaviour
Compose full service role behaviour (or interface behaviour in the case of composite interface roles) from the role behaviour of the bound interface behaviours, taking goal sequence diagrams as input, and using tool support if available.

Method rule: Model composite service structure in a collaboration, wo/connectors (optional)
Define a collaboration called <*service*>*_structure*, using the presentation option showing the identified roles names and the role classifier with associated multiplicity, linked by a collaboration icon naming the service. Connectors are not shown. If tool support is available, use octagonal icons for the service roles.

Method rule: Model composite service structure in a collaboration, w/connectors
For each service, define a collaboration called <*service*>, using the presentation option where the identified roles names and the role classifier with associated multiplicity, linked by two-way connectors. If tool support is available, use octagonal icons for the service roles and square icons for connector ends. If initiating and responding roles have been

identified, colour initiating roles dark and responding roles light, and/or use a connector stereotype with an arrow pointing towards the responding role(s).

Method rule: Identify actor types
Actor types that play service roles are identified as active classes. Inheritance relationships can be used.

Method rule: Bind service roles to actors
Bind the roles of composite services or interface roles to actors in a collaboration use. Use a unique name for the collaboration use within the collaboration defining the actor binding. Any role data defined for the bound roles must be supported by the actor.

Method rule: Compose actor behaviour from interface role behaviour
Compose the actor behaviour from the behaviour of the interface roles bound to it, taking role goal sequence diagrams and role goal interactions as input, using tool support if available. (Method not defined in our work)

4
Service goals

In this chapter we present the concept of service goals, how they can be used to express basic liveness properties, and how they can be used to model semantic connectors and semantic interfaces. We show how live subtypes of semantic interfaces can be used to guarantee basic liveness properties when objects interact.

The structure of the chapter is as follows: first we motivate for the use of service goals by discussing role projection and basic safety properties, which is the work of [Floch 2003]. Then we define service goals and related terms, elaborating on event goals, progress labels and semantic interfaces. Next we present the concept of live subtyping, discussing how this relationship can be exploited. Finally we discuss the modelling of service goals in UML. A discussion of related work, and summaries of goal expression forms and method guidelines are provided at the end.

4.1 Role projection and safety properties

Role projection is an abstraction technique that results in a simplified description emphasising some properties while hiding others. Rather than analysing a complete system, the analysis is limited to projections, thus simplifying the validation process.

The projection of service roles to projection roles (p-roles) is due to [Floch 2003]. A p-role is an interface role that retains only the aspects significant for the purpose of validation of a connection between two service roles. A p-role hides internal actions and interactions of the service role that are not visible to a connected role.

We briefly introduced this in section 2.5.1, which Figure 4.1 recapitulates.

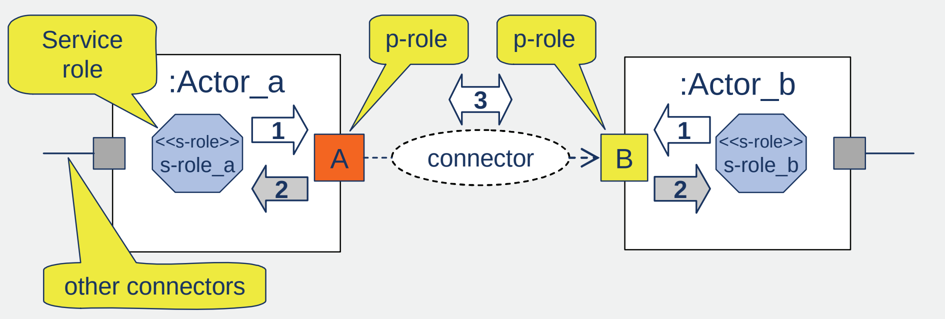


Figure 4.1 : Projection of service roles to p-roles

The principle is to derive interface roles (p-roles) from service roles by projection, and subsequently validate them according to the following steps:

- Step (1a): *Projection*: hide internal actions and interactions on other interfaces not visible on the connection; hidden state transitions are represented by τ-transitions;
- Step (1b): *Distillation*: perform gathering, minimization and merging to obtain a minimal visible interface behaviour;
  - *Gathering* is a transformation that merges interface role states linked by τ-transitions into a single state, if the linked states have the same input behaviour.[1]
  - *Minimisation* is a transformation that replaces equivalent states by a single state.[2]
  - *Merging* is a transformation that is applied on distinct states reachable from a state triggered by equivocal transitions through the same sequence of events.[3] Two or more transitions are equivoque when they are defined for the same state and the same event (i.e input, output or τ-event), and lead to distinct non-equivalent states.[4]
- Step (2): Detect *breaches of the design rules*[5]; if any rules are breached, correct the service role manually and repeat from (1);
- Step (3): *Compatibility checking* between pairs of interface roles - see section 4.1.3.

Algorithms to validate that safety properties are fulfilled by connected roles have been defined by [Floch 2003]. The design rules and validation techniques of [Floch 2003] constitute what we call the *validation of safety properties*, and the algorithms are denoted *safety checking algorithms*. *Design rule validation* comprises steps (1) and (2).

4.1.1 Safe service roles

[Floch 2003] defines design rules that, when followed, ensure that service roles and their projections into interface roles (p-roles) are safe, meaning that they do not violate safety properties. A service role that breaks the design rules is considered a safety liability, as there is no environment that can explore all features of its state machine without running into safety problems. We refer to the design rules as *safety rules*.

We define *safe service roles* and *safe interface roles* as follows:

Definition: Safe service role
A safe service role is one where all projections to p-roles result in safe interface roles.

Definition: Safe interface role
A safe interface role is an interface role that satisfies the safety rules.

Interface roles that pass Step (2) above are safe interface roles. That a service role is safe does not imply it is compatible with a connected role. Compatibility checking between pairs of interface roles, Step (3) above, is performed to establish this.

1. [Floch 2003] p. 108.
2. Ibid. p. 124.
3. Ibid. p. 164.
4. Ibid. p. 126.
5. Floch defines more than half a score of design rules that are checked, see [Floch 2003] pp. 190 and 207.

4.1.2 Dual role

Safe interface roles can be used constructively; [Floch 2003] shows that for any given safe interface role (say, *A*), it is possible to automatically generate a *dual role* (say, *B*) that is guaranteed to play safely with *A*, and that can cover the full behaviour of *A*. An attractive characteristic of dual roles is that they by definition are compatible with each other:

Definition: Dual role[6]
A dual role is an interface role that interacts compatibly with a given interface role. The full behaviour of the given interface role can be reached interacting with the dual role.

4.1.3 Containment and obligation

Interface roles are said to interact consistently when their interactions do not lead to any unspecified signal reception, deadlock or improper termination.[7] This definition of *interaction consistency* does not require that every transition in an interface role has to be reachable when a pair of roles interact. Rather, the roles will interact safely if and only if they are related by both *containment* and *obligation*:

- A containment relation exists between two interacting interface roles when, in each state reached during the interaction, any interface role is able to at least consume any of the signals received from its complementary role. We also say that the input behaviour of each interface role contains the output behaviour of the other interface role.[8]
- An obligation relation exists between two interface roles when, at each interaction step where the input ports of the interface role machines are empty, at least one of the interacting interface roles can send a signal.[9]

Containment and obligation are complementary techniques; containment focuses on preventing unspecified signals and improper termination, while obligation deals with deadlock prevention. This is exemplified in Figure 4.2 (adapted from [Floch 2003]).

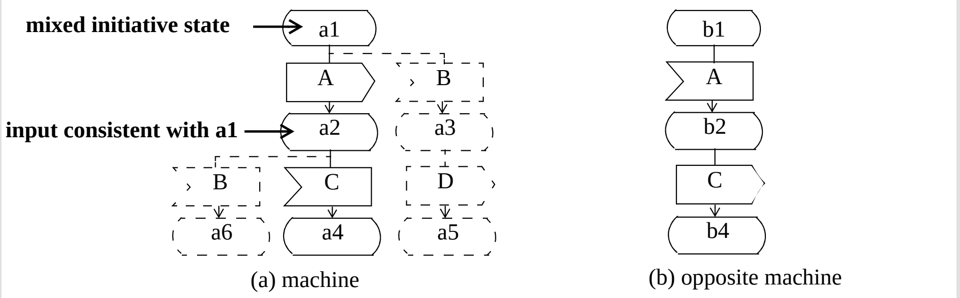


Figure 4.2 : Containment and obligation

Figure 4.2 presents fragments of two state machines that illustrate the containment and obligation relations. In this case only one of the machines, (a), plays an initiating role, i.e. can take the first initiative to send, which it does in state `a1`.

6. Adapted from [Floch 2003] p. 150. Note that several interface roles may be dual roles to a given interface role.
7. [Floch 2003] p. 78.
8. Ibid. p. 212.
9. Ibid. p. 123.

Note that machine (a) is specified to handle a mixed initiative in the mixed initiative state *a1*[10], and is input consistent[11]. Here state *a2* is input consistent with state *a1*.

In Figure 4.2 role (a) assumes in state *a1* that role (b) may send the signal *B*. Safety checking will reveal that (b) doesn't, and that the dashed portions of (a) will thus not be reachable when playing with (b).

(a) and (b) are related by containment, since each role is able to consume all signals produced by its connected role. And they are related by obligation, since at least one of the roles can always send a signal when the input ports are empty.

4.1.4 Discussion

[Floch 2003] has shown that validating interface roles requires less state space than a full reachability analysis of the behaviour of two interacting objects. In addition, the results are easy to understand for designer, and separate validation models are not needed.

The validation technique is an improvement to the validation of static interfaces, which would falsely conclude that (a) and (b) in Figure 4.2 are incompatible. Validation of static interfaces would claim that (a) requires that (b) should be able to consume "*D*", which it in fact won't, since "*B*" is never sent by (b), and "*D*" will thus never be sent by (a). Safety checking on the other hand correctly concludes that since the transition sending "*D*" is not reachable, their combined behaviour is safe.

To motivate our liveness extensions of [Floch 2003], we present a toy example of interface behaviour of a composite service. Figure 4.3 shows the role structure of the service.

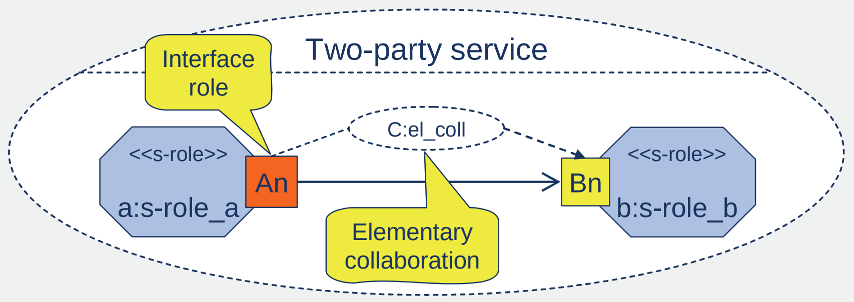


Figure 4.3 : Two-party service with interface roles An and Bn

The focus is on the interface roles *An* and *Bn*, i.e. the behaviour of *s-role_a* and *s-role_b* in Figure 4.3 are not detailed. In the examples we provide alternative versions of *An* and *Bn*, naming them *A1*, *A2*,... and *B0*, *B1*,... to tell them apart.

The first example of interface behaviour of *s-role_a* is *A1*, as shown in Figure 4.4.

Note in Figure 4.4 that the signal *discon* being sent by *A1* and by its connected role more or less simultaneously is an example of conflicting initiatives[12].

10. *A mixed initiative state is a state where both signal consumption and sending can occur.* ([Floch 2003]. p.143)
11. *A state is input consistent with another state if the set [of] inputs enabled in this state contains the set of inputs enabled in the other state. Two states are input consistent if they accept the same set of inputs.* (ibid. p. 178).
12. This is a classical example from telecom: both peers may at any time disconnect a two-party connection.

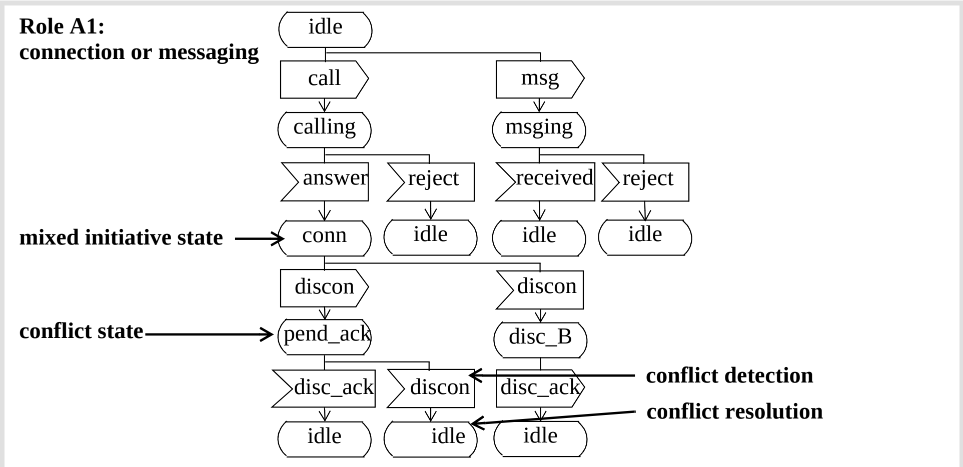


Figure 4.4 : Role A1: connection or messaging

Note also that *A1* includes conflict resolution: *discon* can cross without causing an error (unspecified signal reception or deadlock) in *A1*; *A1* has chosen to treat *discon* in the conflict state *pend_ack* in much the same way as a *disc_ack*, proceeding to the *idle* state[13].

We can generate an opposite role for *A1* by mirroring, using the function *mirror()*[14]:

B0 = mirror (A1)

B0, the mirrored role of *A1*, is depicted in Figure 4.5 below.

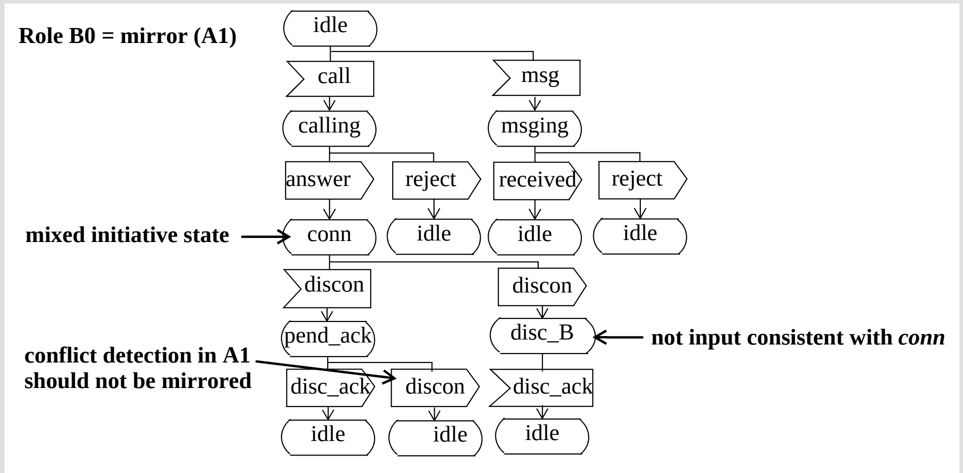


Figure 4.5 : Role B0 = mirror (A1), the mirrored role of A1, is not safe

While mirroring of simpler interface roles[15] than *A1* can result in a dual role, the mirrored role *B0* is not safe, since the resulting state *disc_B* is not input consistent with the preced-

13. This is a simple conflict resolution without involving any signalling.
14. mirror() is a function that creates an interface role from another interface role by changing inputs to outputs and vice versa. State names are copied unchanged, as are entry and exit points. Deferred triggers (called *save* in SDL) are not maintained in the mirrored graph. For details see [Floch 2003] pp. 150-154.

ing mixed initiative state *conn*. To achieve a dual role according to [Floch 2003], the conflict detection in *A1* (input of *discon*) should not be mirrored in *B*, and the necessary addition of conflict detection in *B0* (adding input of *discon* in state *disc_B*) should mirror the conflict resolution in *A1* (in this case: going to *idle*).

A safe opposite role of *A1* can be generated by the function *dual()*[16]:

B1 = dual (A1)

B1, the dual role of *A1*, is depicted in Figure 4.6 below[17].

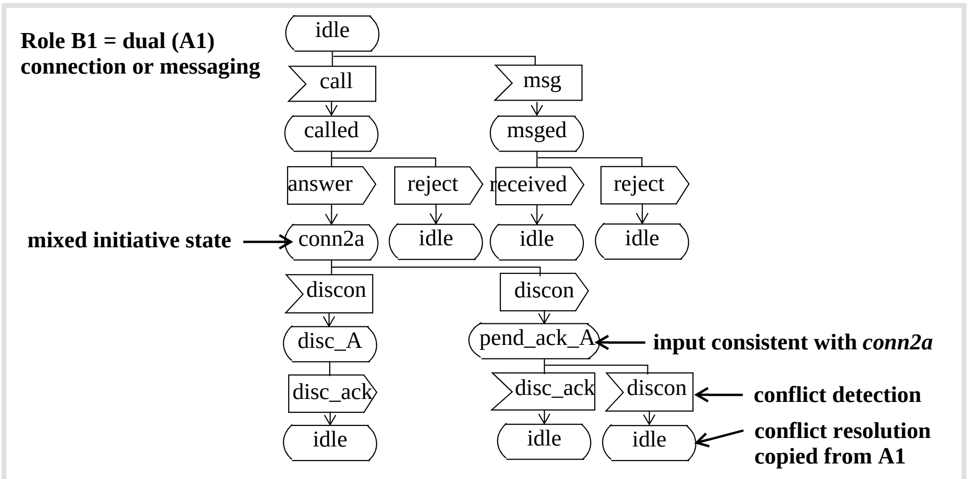


Figure 4.6 : Role B1 = dual (A1), the dual role of A1, is safe

The dual role *B1* can by definition play safely[18] with *A1*, as it is capable of exercising the full behaviour of *A1*, meaning that all behaviour of *A1* is reachable when validated against *B1*. We introduce the notation of truncation:

Definition: Truncated role
A truncated role is the sub-tree of an interface role that is reachable when playing with some specific connected interface role. Role *A* truncated by role *B* is written *A with B*.

Returning to the example above, we see that *A1* truncated by the dual role *B1* results in *A1*, meaning nothing in *A1* is lost when interacting with *B1* (all of *A1* is reachable):

A1 with B1 = A1 with (dual (A1)) = A1

However, two roles, say, *A* and *B*, do not need to be dual for the pair to interact safely; *B* may have either greater or lesser capabilities compared to the dual role of *A* (e.g. *B* accepts more input and/or produces less output than *dual(A)*), and still play safely with *A*.

While a dual role can be generated so that the full behaviour of an interface role can be reached, safety checking does not require full behavioural reachability of a pair of con-

15. *Safe, dual roles can be created by mirroring if the source role has no equivoque transitions, no acute τ-transitions, and does not contain any mixed initiative states* [Floch 2003] p. 153. *Acute τ-transitions are τ-transitions that cannot be removed [..] by gathering and minimisation* (ibid. p.136).
16. For details see [Floch 2003] p. 191.
17. State names in the dual role *B1* reflect a possible B-role, rather than copying the *A1* state names.
18. Provided they start consistently, and that any spontaneous sending can occur, see [Floch 2003] page 192.

nected roles. According to the safety rules it suffices that safety properties are not violated, i.e. nothing bad should happen. “Bad” in this context means interactions resulting in deadlock, improper termination or unspecified signal reception (see definitions page 39).

4.1.5 Motivation for expressing basic liveness properties

As noted above, the dual role *B1* in Figure 4.6 need not be the only role that can play safely with *A1*. Figure 4.7 shows other interface roles that are safe alternatives to *B1*.

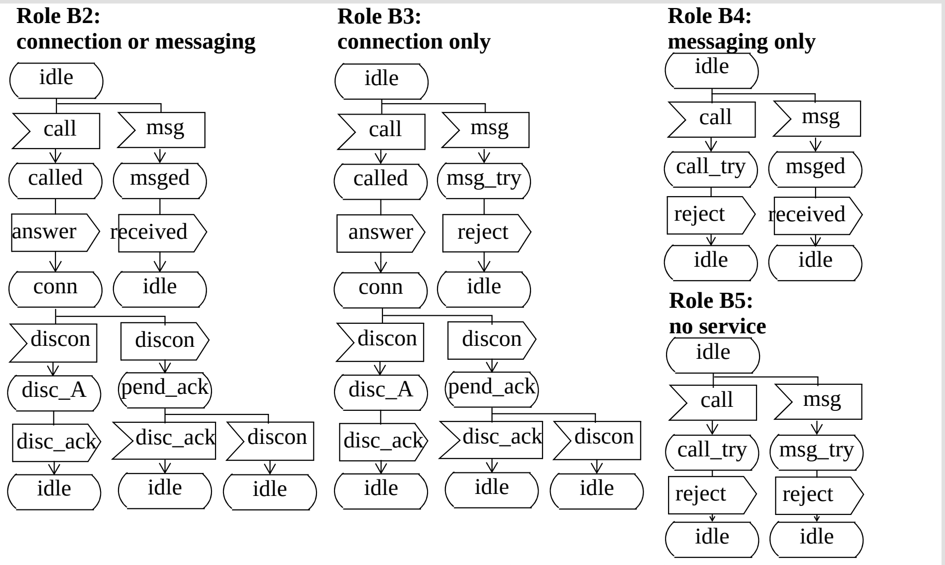


Figure 4.7 : Roles B2, B3, B4 and B5: safe but less useful alternatives to B1

The safety checking algorithms of [Floch 2003 section 7.2.4] show that all the roles *B2*, *B3*, *B4* and *B5* in Figure 4.7 maintain the containment and obligation relationships with *A1*. *B1* through *B5* are all safe as connected roles of *A1*.

However, it is clear that the truncated roles *A1 with B2* through *A1 with B5* will be smaller than the full behaviour *A1 with B1*. *A1 with B5* will result in a very limited play: all initiatives are rejected; nothing useful happens.

Figure 4.8 illustrates *A1 with B5* in more detail: here the transitions of *A1* that will never happen are dashed in the diagram to the left, and the truncated role “*A1 with B5*” (with dashed elements removed) is shown to the right.

An interesting property of truncated roles can be observed: A pair of connected roles will each have its truncated role; if they interact compatibly, and neither contain deferred signals[19], then the pair of truncated roles will always be dual roles:

A with B = dual (B with A)

19. Deferred triggers (saved signals in SDL) are not maintained by dual().

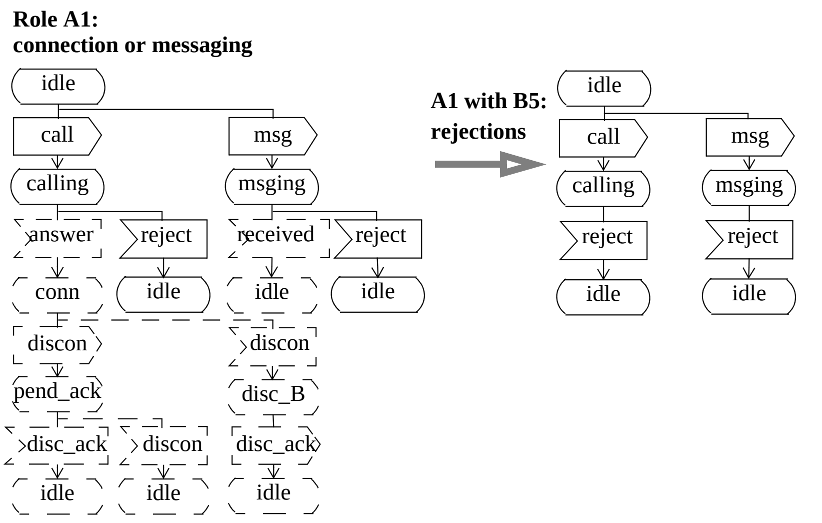


Figure 4.8 : Role A1 combined with B5: a safe but useless alternative to B1

Justification: A and B will truncate each other so that an output from one is matched by a corresponding input in the other. Any superfluous events and states are pruned. All inputs and outputs will be matched, otherwise they would not interact compatibly.

This motivating introduction has shown that we need means to compare connected roles with respect to their achievement of useful behaviour, and we need to find criteria to select the best among a set of safe alternative role behaviours.

4.2 Expressing basic liveness properties with service goals

All interactions should be safe, as discussed in the previous section. But in addition, some useful interactions should be possible. Liveness properties, i.e. desirable things that should eventually happen, are not addressed by [Floch 2003]. We suggest expressing usefulness by the systematic use of *service goals*.

The service goal concept arises from the recognition that behind each service invocation lies some primary goal. Such goals can be identified and discussed at a high level, independently of the implementation details of terminals, networks and service protocols.

Our main contribution to the expression of basic liveness properties involves labelling events with what we call *progress labels*. Progress labels are a way of expressing *event goals*, and mark events that constitute something useful to the service. They are inspired by the marking of so-called “progress states” in Promela [Holzmann1991, 2003][20].

The structure of this section is as follows: First we define the concepts related to service goals, such as *state-like goals* and *event goals*, placing particular emphasis on event goals expressed using *progress labels*. We exemplify progress labelling, and discuss different types of progress labels and how they can be used. We show how progress labels can be

20. A progress-state label in Promela marks a state that must be executed for the protocol to make progress.

derived from goal states. We also discuss aspects of service goals in general, and distinguish between specified interface behaviour and actual interface behaviour.

4.2.1 Definition of terms

We use the type elements of a service structure to express *service goals*.

Definition: Service goal
A service goal is a property that characterizes a point in the behaviour of a collaboration between roles as having achieved something useful.

Semantic connectors and the *semantic interfaces* are the basic elements of our approach. By definition they must contain reachable service goals:

Definition: Semantic connector
A semantic connector is an elementary collaboration with a consistently defined pair of semantic interfaces and service goals, i.e. where:

- the semantic interfaces are dual roles[21], so by definition they are safe interface roles;
- goals are defined consistently: a goal in one role is matched by a goal in its opposite role;
- it optionally defines *collaboration goal(s)* that are reachable when the roles interact.

Definition: Semantic interface
A semantic interface is an interface role describing specified interface behaviour. A semantic interface has at least one *event goal*.

Definition: Role compatibility
A pair of roles is compatible if and only if the roles interact safely, and are capable of achieving service goals when doing so.

Definition: Collaboration goal
A collaboration goal is a predicate expressing when a goal is achieved seen from the perspective of the collaboration as a whole.

We differentiate between two ways of expressing goals: *state-like goals* and *event goals*. The former relates to stable situations, while the latter relates to transient events.

Definition: State-like goal
A state-like goal is a predicate that can be evaluated at any time when a system is stable, i.e. in between the handling of events and outside the execution of operations.

A state-like goal is a *goal expression* or a *goal assertion*.

Definition: Goal expression
A goal expression is a predicate defined within the scope of a service structure. If evaluated to *True*, the goal of the element is currently achieved.

21. By *dual* we imply that all signals sent by one role are consumed by the other role, with no unreachable transitions.

Definition: Goal assertion
Goal assertions are expressed by goal expressions in structure descriptions, or by state invariants in behaviour descriptions.

Definition: Event goal
An event goal is related to the occurrence of an event: the event occurring implies that something useful is achieved at that point in the behaviour. An event is a signal sent or consumed.

Definition: Goal state
A goal state is a state or exit point of a state machine where a goal is achieved. Goal states can be expressed by goal assertions and/or by progress labels.[22]

Definition: Progress label
A progress label marks an event goal or goal state in a service role or interface role.

To distinguish between roles with and without progress labels, we introduce the terms *basic roles* and *live roles*:

Definition: Basic interface role
A basic interface role is an interface role without progress labels.

Definition: Live interface role
A live interface role is an interface role with progress label(s).

Goals are ultimately achieved through the successful playing of service roles by the actors participating in a service invocation. We can thus talk about actors achieving goals.

Definition: Role goal
A role goal is a service goal defined for a collaboration role.

Definition: Actor goal
An actor goal is a service goal defined for an actor. An actor goal is usually related to the role goal of a service role or a semantic interface an actor can play.

4.2.2 Examples of progress labelling

We introduce the technique of progress labelling by way of an example, showing how signals sent or consumed are marked by progress labels. Here we only focus on progress labels in the interface roles, and do not detail the service roles to which the interface roles are bound or from which they are projected.

Figure 4.9 shows a reworking of the basic interface role *A1* we introduced in Figure 4.4 on page 75, into the live interface role *A2*, where notes containing the strings *"progress:2"* and *"progress:8"* are progress labels. *A2* is a semantic interface, as we shall see later.

If *A2* was served by an actor playing *B5* introduced in Figure 4.7 on page 77, the truncated behaviour of *A2* would be as depicted in Figure 4.10 below.

In Figure 4.10 the transitions that can never occur are dashed in the diagram to the left, and the truncated role graph is shown to the right.

22. In section 4.2.5.2 we show how goal states marked by assertions can be used to derive progress labels.

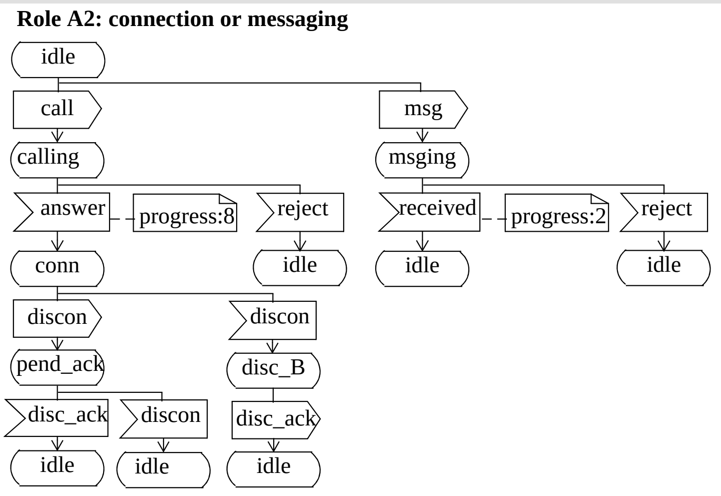


Figure 4.9 : Role A2: Graded progress labels added to A1

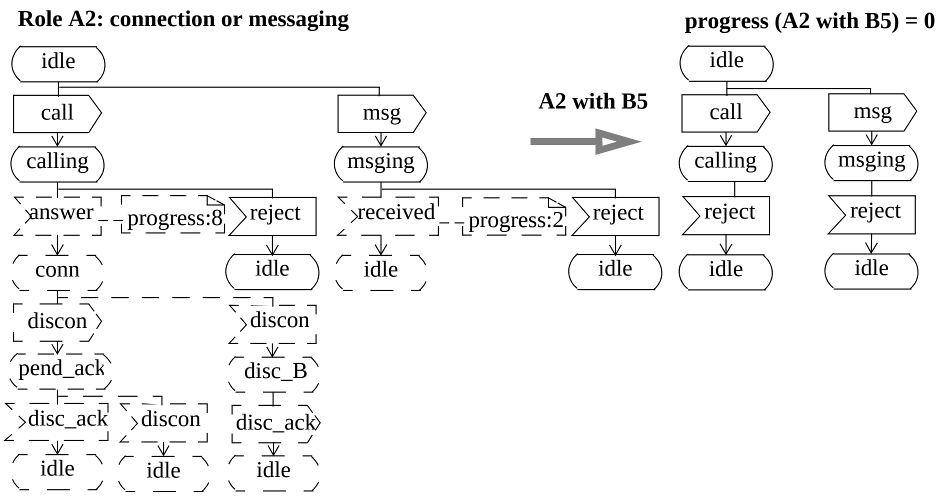


Figure 4.10 : Role A2 combined with B5: no progress achievable

The validation of safety properties first determines whether the interface roles are safe, which they are in this case. The truncated role *A2 with B5* is then be analysed for the presence of progress labels. In this case there are none; all the progress labels are contained in the portions of *A2* that are not reachable when playing with *B5*. Progress checking determines that no progress can be made in this case, i.e.:

progress (A2 with B5) = 0

leading to the conclusion that *A2* will not reach any progress when playing with *B5*, although the pair are compatible in terms of safety properties.

Figure 4.11 shows examples of interface role combinations capable of yielding progress.

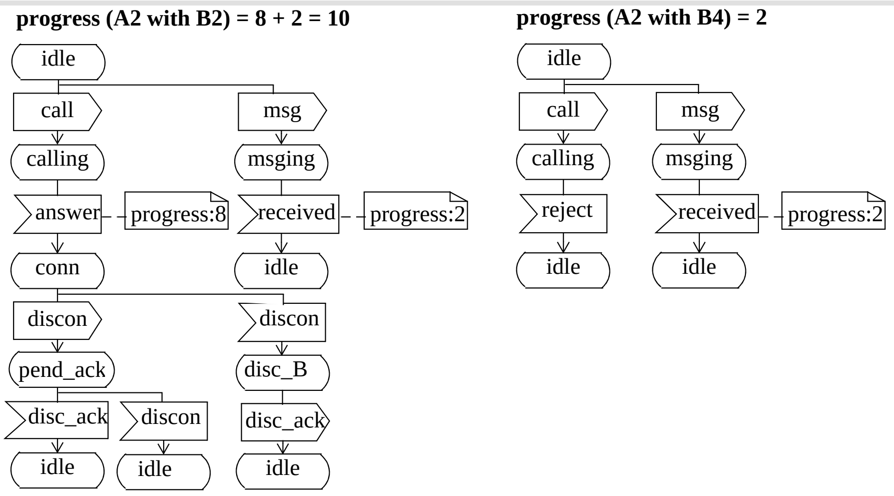


Figure 4.11 : Role A2 combined with B2 or B4: progress achievable

In Figure 4.11 the truncated graphs of *A2* are shown in each case. Safety rules are respected in both combinations. Progress checking concludes that *A2* will reach progress when combined with both *B2* and *B4*:

progress (A2 with B2) = 8 + 2 = 10
progress (A2 with B4) = 2

4.2.3 Types of progress labels

Two types of progress labels are defined in the following:

- Graded progress labels;
- Service-specific progress labels.

We discuss each in the subsections below.

4.2.3.1 Graded progress labels

The examples in Figure 4.10 and Figure 4.11 show examples of progress labels in the form of "*progress:<n>*", where the designation *<n>* quantifies the progress achieved. We call these *graded progress labels*.

Definition: Graded progress label
A graded progress label is a progress label with a numerical designation of the relative amount of progress; the higher the number the greater the relative progress.

If a graded progress label is found in the truncated role, then that interface role is said to have at least that *level of progress* with respect to its connected role.

Definition: Level of progress
The level of progress of an interface role truncated by another interface role is the sum of the graded progress labels in its truncated role.

A progress label “*progress*” without a numerical value implies a progress level of “1”. A basic interface role has *progress () = 0*, since no graded progress labels are present.

As we have discussed above, there is a need for a mechanism to determine a best choice among a set of alternative roles. Given that a set of alternative roles satisfies the safety rules, criteria to distinguish relative role-playing capabilities are required. If several alternative role behaviours are possible, e.g. *B1-B4* in the example, it is then possible to compare them with respect to their levels of progress. The alternative interface role with the highest level of progress is the best choice for a role arbitration mechanism[23]. Alternative roles with the same level of progress are considered equally good alternatives.

In Figure 4.10 the level of progress is zero, i.e. no progress; despite satisfying safety requirements, *A2 with B5* gives no progress. In Figure 4.11 the level of progress of *A2 with B2* is 8 + 2 = 10, while the level of progress of *A2 with B4* is 2. Progress checking concludes that *B2* is the best choice of alternative role relative to *A2*.[24]

4.2.3.2 Service-specific progress labels

The second type of progress label is what we call *service-specific progress labels*. It enables one easily to identify which semantic connectors has achieved a goal. This is useful when several semantic connectors are bound to a service role or an actor.

Definition: Service-specific progress label
A service-specific progress label is a progress label that identifies a role goal.

If a service-specific progress label is found in a truncated role, then the interface role that is truncated is said to include that *service progress* with respect to its connected role.

Definition: Service progress
The service progress of a role truncated by another role is the set of service-specific progress labels in its truncated role. An empty set implies no progress.

We insert service-specific progress labels that identify role goals, e.g. *progress: Call_Init* or *progress: Msg_Init*. An example is shown in Figure 4.12.

Progress checking of *A3 with B5* results in an empty set of service-specific progress labels, i.e. no progress. Validating *A3 with B4* results in a set of one progress label, while *A3 with B2* results in the set of four progress labels:

```
progress (A2 with B2) = {Call_Init, Disc_Init, Disc_Resp, Msg_Init}
progress (A2 with B4) = {Msg_Init}
```

Several applications of service-specific progress label are possible:

- In terms of choosing between alternative roles, the role yielding the largest set of service-specific progress labels is the better choice, e.g. *B2* is the best choice for *A2*;[25]
- A role request could state progress requirements in terms of service-specific progress labels, e.g. *Request (“B”, “A3”, Atleast Label(“Call_Init”))*.

23. This is used for runtime connector validation as an element of role requests, see section 6.5.1.
24. This can be applied to role requests introduced in section 2.4.5: the requesting role could state a minimum level of progress during a role request, e.g. *Request (“B”, “A2”, AtleastLevel(<n>),...)*.
25. Graded progress labels can likewise be used to select the best among several alternative roles.

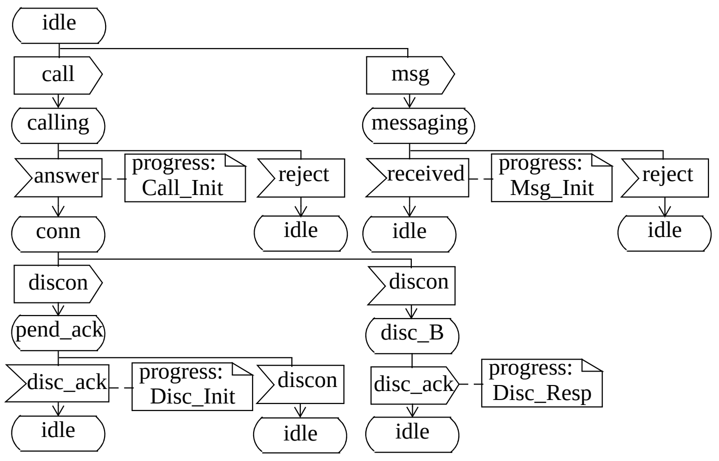


Figure 4.12 : Role A3: A1 with service-specific progress labels added

Note that service-specific progress labels are linked to goal states, see section 4.2.5.

4.2.4 Using progress labels

4.2.4.1 Employment of progress labels

We have defined two types of progress labels. The label types provide different benefits at design time and runtime. We suggest two approaches to the use of progress labels:

- *Role arbitration* which calculates the achieved progress between two roles, and can compare the progress level of several alternative roles. In the role arbitration scheme, graded progress labels can be used. The preferred role is the alternative role that returns the highest value when challenged with a particular role;
- *Compatibility check*, which checks whether or not a specified role is able to provide one or more specific service. In the compatibility check the service specific progress labels can be used to check whether or not a specified service is accomplished during interaction with some specified role.

Algorithms for both approaches have been designed and implemented, see section 6.1.3.

4.2.4.2 Attaching progress labels to exit points

Progress labels may be attached to exit points; an example was first seen using SDL in Figure 3.20 on page 60. The principle is to use the exit points to indicate what progress has been achieved. An example using UML2 is shown in Figure 4.13 below.

In Figure 4.13, the interface role *connection_or_messaging* is specified using a composite state. The name of the exit point indicates whether progress is achieved or not. If the name of the exit point begins with “*progress*”, then it constitutes a progress label. Either graded progress labels or service-specific progress labels can be used for such labelling.

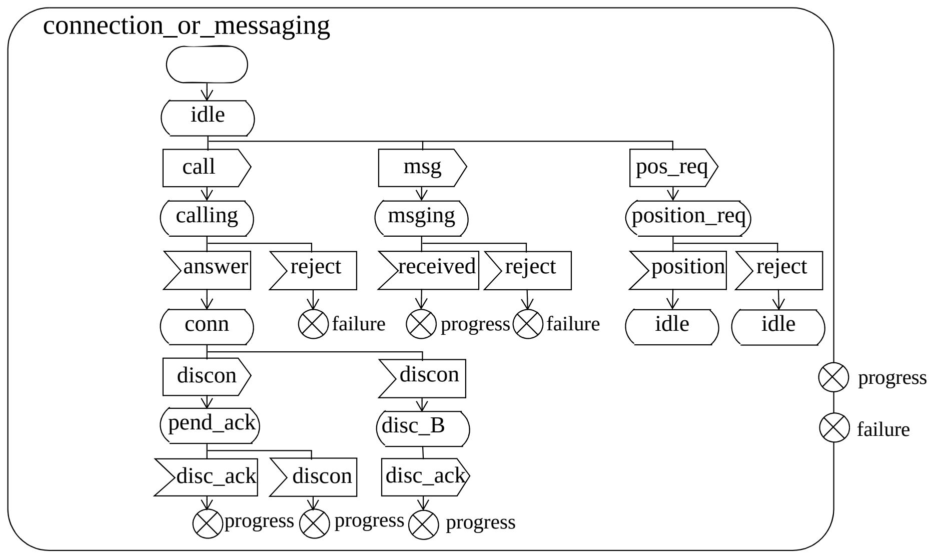


Figure 4.13 : Role with progress labels on exit points

Labelling exit points is useful for composite states that terminate after reaching a goal. If exit points and role goals correspond directly, then attaching labels to exit points is feasible.

However, exit points cannot be used for service logic containing behaviour cycles, i.e. behaviours where service features are invoked in cyclic fashion, and where a scheme of “one composite state per service invocation” will not work. In this setting exit points would not be reached during normal usage.

For instance the composite state in Figure 4.13 must be invoked repetitively if the service logic is used several times in succession, e.g. to send more than one signal to the associated *B*-role. This may not be a desirable way to design service logic as it imposes restrictions on the progress labels. Were we only to rely on labelling exit points to express progress, we would not be able to attribute progress to behaviour other than that which leads to an exit point. For instance, achieving the goals of the positioning logic of Figure 4.13 could not contribute to the calculation of progress.

4.2.4.3 Insertion of progress labels

Progress labels are manually inserted into service roles or semantic interfaces by a designer, or they are automatically inserted into p-roles by the projection tool.

The manual insertion of progress labels is a process aimed at specifying what events or exit points in a state machine constitute progress. As with all manual processes, it can be creative but also error-prone. Validation algorithms and tools should take this into account, and support the designer in an iterative process during the design activities.

Use graded progress labels in service roles, and service-specific progress labels in semantic interfaces. Note that the placement of progress labels in service roles must take projection into consideration, since projection removes events and event goals that are not visible from the viewpoint of the connection; see section 6.1.3.

Method rule: Insert graded progress labels in service roles
Insert graded progress labels in service role behaviour so that for each role projection of the service role there is a least one event that has a progress label. Mark all useful alternative paths with a progress label. Use graded progress labels to distinguish between alternative functional service levels, rating the most useful highest.

Method rule: Insert service-specific labels in semantic interfaces
Insert service-specific progress labels in semantic interfaces so that there is a least one event that has a progress label. Mark all useful alternative paths with a progress label. Use service-specific progress labels to identify which semantic connector has progress.

In either case one must take care to attach progress labels to unique events, see the method rule "Attach progress labels to unique events" on page 137.

4.2.5 Deriving progress labels

Since manually inserting progress labels is subject to human inaccuracy, an attractive alternative is to let tools derive the progress labels from other descriptions of the service.

4.2.5.1 Deriving progress labels from role goal interactions

Goal assertions expressed by UML2 state invariants in role goal interactions can be used to automate the insertion of service-specific progress labels in interface roles. Figure 4.14 illustrates the principle: the lower right half is a role goal interaction, with a goal assertion defined by a state invariant. To the left is the definition of the behaviour of the semantic interface with a service-specific progress label attached to an exit point.

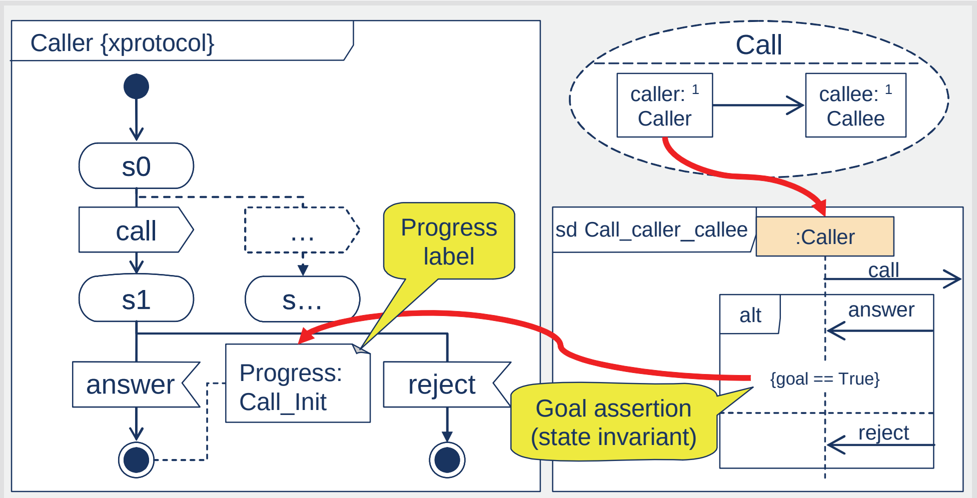


Figure 4.14 : Deriving progress labels from role goal interactions

The tool needs to identify the event immediately preceding the state invariant, and mark this event in the interface role with an appropriate progress label. It is possible to automatically insert service-specific progress labels in this way.

An appropriate naming scheme for progress labels that lends itself to automation is a concatenation of the name of the semantic connector (e.g. Call) and "_Resp" for responding role and "_Init" for initiating role. Which role is responding or initiating is defined by the semantic connector, see the top right of Figure 4.14.

4.2.5.2 Deriving progress labels from goal states

Goal states can be identified by goal assertions. Since achieving a goal state by definition constitutes progress, a tool could be used to derive appropriate progress labels from goal states that are expressed by goal assertions. Figure 4.15 illustrates the principle.

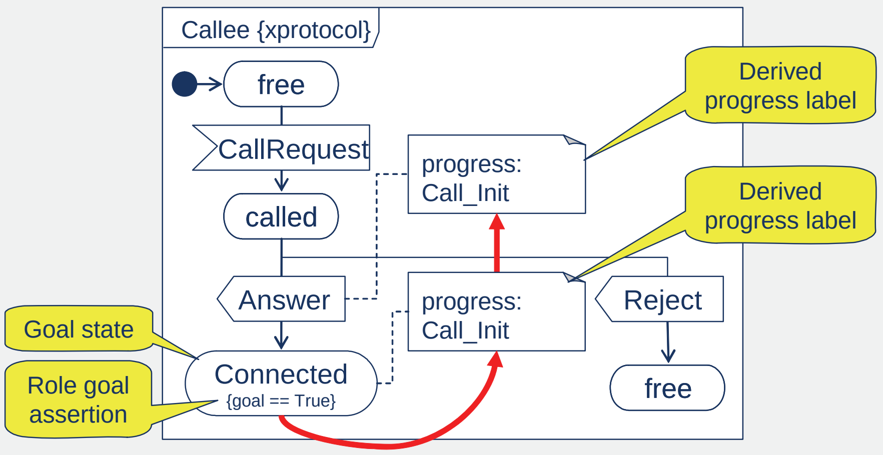


Figure 4.15 : Deriving progress labels from goal states

Such a tool needs to identify the goal states of the role, and to mark the states or events that lead to these states with appropriate progress labels. Note that if several events lead to a goal state, each event will be marked by a derived progress label.

Method rule: Derive service-specific progress labels
In interface roles, insert service-specific progress labels on goal states (or events that lead to goal states), such that they correspond one-to-one with goal assertions defined in role goal interactions. If possible use tools to derive these progress labels.

4.2.6 General aspects of service goals

Different goal expression forms can be used to express the various goal types, depending on what type element is used, and can vary in complexity. For instance, the simplest form of collaboration goal is the conjunction of a set of goal assertions.

A service element may have several service goals. For instance a service role may have more than one role goal, e.g. at least one for each semantic interface it supports. Likewise, a composite service structure many have more than one collaboration goal.

Service goals are property descriptions that ultimately characterise actors in terms of the services they partake in. Service goals are instantiated when the roles they relate to are instantiated, which occurs when the service roles are played by actors. Goal assertions can be evaluated during the playing of roles. The value of goals prior to service invocation is undefined. The result of a goal assertion when the role is no longer played is undefined.

Service goals define successful service invocations. Exceptional cases, error handling and other forms of failure are not expressed by service goals. The purpose of service goals is to highlight essential service intentions. Nonetheless, the failure to reach service goals is important, since subsequent services may depend on preceding service goals being achieved. The discussion of goal sequences is the subject of chapter 5.

4.2.7 Service goal types

In the sections above we have presented different types of service goals. We summarize the various service goal types and their scope in Table 4.1:

Table 4.1: Service goal types

Service goal type	Purpose	Scope
Collaboration goal		
Collaboration goal assertion	Describes the goal of the collaboration as a whole in the form of a goal expression (predicate)	Within the collaboration itself, following the scoping rules of collaborations
Role goal		
Role goal assertion	Describes the goal of the role in the form of a goal expression (predicate)	Within the classifier (actor) playing the role, following the scoping rules of classifiers
Event goal	Characterizes an event as achieving a role goal. A goal event can be a signal sent or consumed, or a state or an exit point reached.	Within the classifier (actor) playing the role, following the scoping rules of classifiers

4.2.8 Semantic interfaces versus p-roles: specified versus actual

As stated above, p-roles and semantic interfaces are different types of interface roles. The distinction between them lies in how they are derived and used. Semantic interfaces define *specified* interface behaviour, while p-roles describe *actual* interface behaviour.

Following our approach, semantic interfaces are interface roles of semantic connectors, and defining interface behaviour with role goals is mandatory. Differently, a p-role is obtained projecting a service role over a connector, and will only contain role goals if the service role has goals that are visible to that particular connector; see section 6.1.3.

Projection to p-roles and binding to semantic interfaces provides validation opportunities, such as checking whether p-roles are *live subtypes* of semantic interfaces, as introduced below.

4.3 Live subtyping

Here we discuss relationships between alternative roles, investigate their effect on the achievement of progress, and formulate rules for what we call *live subtyping*. As we shall see in chapter 6, the live subtyping relationship is useful for efficient validation of compatibility and compliancy at both design time and runtime.

Below we analyse the effects of redefining interface roles through subtyping, i.e. adding and removing input and output from a pair of semantic interfaces. We also compare live subtyping with the subtyping capabilities of SDL and UML[26].

In Figure 4.16 we recall the role *A3* (*A1* with progress labels added) and its dual role *B1*. *A3* and *B1* constitute a pair of semantic interfaces[27] of a semantic connector *C*.

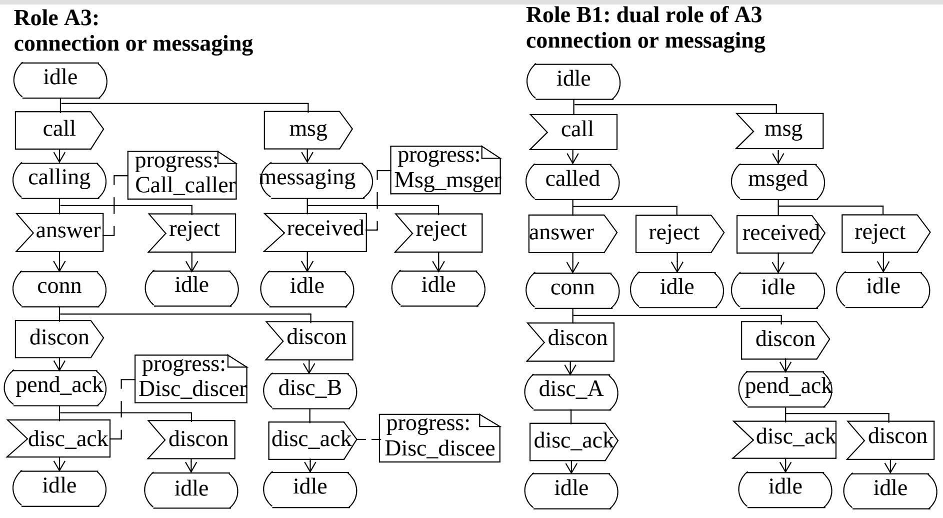


Figure 4.16 : Role A3 and dual role B1: a pair of semantic interfaces

Dual roles play safely by definition, and are capable of reaching the full behaviour of each other. Role validation of a dual pair will ascertain whether basic safety properties and liveness properties are fulfilled: all the progress labels of *A3* are reachable when playing with its dual role *B1*.

4.3.1 Removing behaviour

4.3.1.1 Removing output

We start our analysis with removal of output relative to a well-formed semantic interface. Figure 4.17 below recalls the roles *B2*, *B3*, *B4* and *B5* from Figure 4.7 on page 77. We see that *B2* through *B5* are obtained by removing different amounts of output from *B1*.

26. Compatibility constraints in connection with inheritance of behaviour in UML is a semantic variation point, see [UML 2.0] p 126.
27. Note that *B1* should contain progress labels corresponding to those in *A3*. These are omitted to save space.

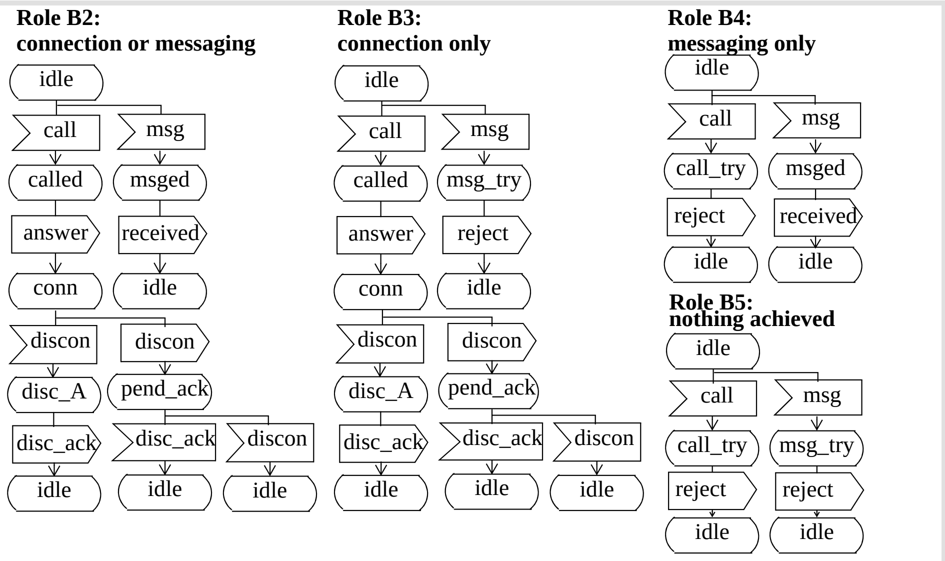


Figure 4.17 : Roles B2, B3, B4 and B5: safely removing output from B1

We have previously analysed the roles *B2*, *B3*, *B4* and *B5*, and determined that they are safe alternatives with respect to playing with *A1*, and hence they are with *A3* too.

Removing output from a role is safe up to a point: If too much output is removed, the obligation relationship will be violated. This is exemplified by the role in Figure 4.18: Whatever signal *A3* sends to this interface role, no response is returned, and *A3* will forever remain in state *calling* or *messaging*.

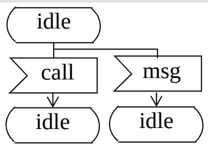


Figure 4.18 : Interface role with too little output: obligation breached

In addition, even if removing output from a role is safe, it comes at the inherent risk of losing progress. The resulting truncated role will be smaller, and the progress achieved towards service goals reduced accordingly. This has been demonstrated for roles *B3* through *B5*, which all result in less progress than the dual role *B1*. On the other hand, *B2* results in the same progress as *B1*, since it removes output that does not affect any progress labels in *A3*. So *B2* is a safe and useful alternative to *B1*.

We conclude that removing output to create a new role from a dual role is possible, but that it is not generally a safe or useful subtyping relationship. Removing output is possible by redefining the transitions of a supertype in a subtype[28], so here we must be stricter than SDL and UML to prevent losing progress. A subtype can have less output, but we must ensure that progress is not lost as a consequence, and that obligation is not breached.

28. A redefinable transition in SDL and UML cannot remove the input of the transition, but any amount of output in the transition of the supertype can be removed by the subtype.

4.3.1.2 Removing input

Input events cannot be removed from any of the roles without safety problems arising. A premise for the dual relationship is that any spontaneous sending can occur, and without a corresponding input in the connected role, unexpected input will occur.

We conclude that removing input to create a new role from a dual role is a breach of the containment rule, see section 4.1.3. This is consistent with subtyping in SDL and UML, which likewise does not allow input to be removed (see footnote 28).

4.3.2 Adding behaviour

Let us analyse the effects of adding behaviour. Consider the interface role in Figure 4.19:

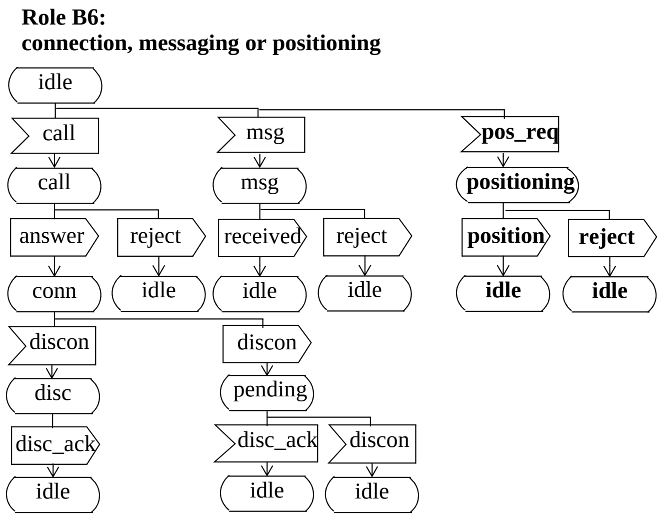


Figure 4.19 : Role B6: adding input-output behaviour is safe

Role B6 of Figure 4.19 is capable of performing large portions of B1's behaviour, the only essential difference[29] being that B6 is capable of handling a positioning request, highlighted in boldface in Figure 4.19. Compared to *B1*, *B6* adds an input *pos_req*, a state *positioning*, and an output *position*.

We can say that *B6* contains *B1*. I.e., *B1* could be a supertype, and *B6* could be a redefinition (subtype) inheriting from *B1* and adding the additional behaviour marked in boldface.

Performing compatibility checking with *A3* would conclude that both *B1* and *B6* would work equally well, since the additional behaviour of *B6* will never be invoked by *A3*. In fact, *A3* would not be able to tell *B1* and *B6* apart.

That input can be added to a subtype without impairing safety and liveness properties is as can be expected. What is more interesting, is that we can safely add output, provided it

29. A few differences in states names are intentionally chosen, to stress the point that state names do not carry any behavioural semantics; only the sequence of events do.

is “guarded” by added input, i.e. output added to states that are hidden when viewed from the perspective of the dual role of the supertype (i.e. states that will not be reached when they interact). An example is the output of *position* or *reject* in state *positioning* in Figure 4.19. Since the state *positioning* will not be reached when *A3* and *B6* interact, A3 will never receive *position.* I.e. the interaction between *A3* and *B6* is safe.

However, adding output is not safe if it is added to a state that is visible to the connected role. Consider for instance the role *A4* in Figure 4.20 below.

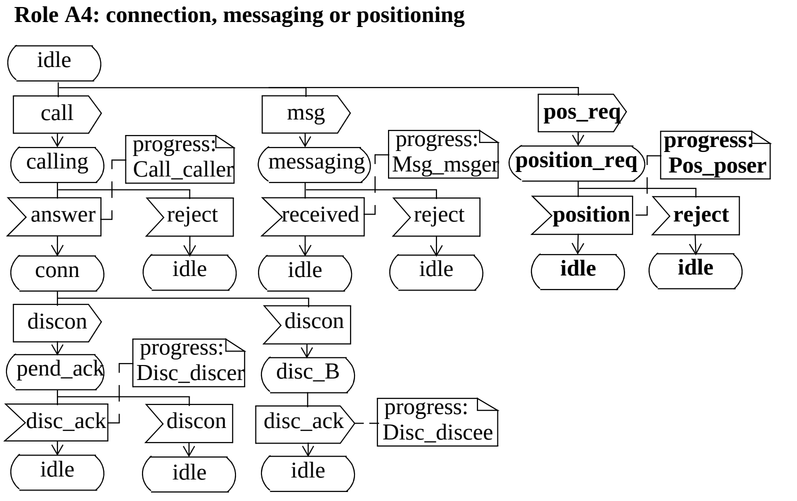


Figure 4.20 : Role A4: adding output-input behaviour is unsafe

A4 relates to *A3* in much the same way as *B6* relates to *B1*: large portions of behaviour are identical, the addition being the positioning functionality shown in boldface. *A4* could be a redefinition (subtype) of *A3*, since it contains all of *A3* and adds the parts in boldface.

Safety checking between pairs of interface roles shows that *A4* cannot interact safely with *B1* (nor with *B2-B5* of Figure 4.17). The reason is that *A4* has the output *pos_req* that these roles do not accept as input, thus breaching the containment rule, see section 4.1.3.

However, *A4* can interact safely with *B6* in Figure 4.19, because this role accepts the extra signal. In fact, *A4* and *B6* are dual roles; one of them can be synthesised from the other.

We conclude that input can safely (and usefully) be added by a subtype. Furthermore that output can be safely added to invisible states (states added by the subtype), while output cannot safely be added to visible states of the supertype. We note that safely added output does not affect progress compared to the supertype.

4.3.3 Live subtyping: compatibility between interfaces roles

We are now in a position to define *live subtyping* and to discuss role compatibility.

Definition: Live subtype
A live subtype is a safe and useful redefinition of an interface role. By useful it is implied that no progress is lost by the redefinition.

Live subtyping is limited to adding input events and states to the supertype, and restricting the addition of output events only to the states added to the supertype. No input of the supertype may be removed. The subtype may have less output than the supertype, provided obligation is not breached, and that no event goals are lost.

A’ being a live subtype of A we write as:

A’ ~> A

In UML diagrams we suggest that live subtyping is indicated by writing “*live*” in the inheritance icon. See Figure 4.21 on page 94 for an example.

For the sake of argument, we define a less strict form of subtyping that is nonetheless safe:

Definition: Safe subtype
A safe subtype is a safe redefinition of an interface role. The redefinition may have less progress then the supertype.

Safe subtyping is limited to adding input events and states to the supertype, and restricting the addition of output events to the states added relative to the supertype. No input of the supertype may be removed. Output may be removed, provided obligation is not breached.

UML2 allows all output to be removed by subtyping, which in general is not safe or useful. With safe subtyping output can be removed as long as obligation is not breached. Safe subtyping can remove progress, and is therefore less restrictive than live subtyping.

Roles related by either live or safe subtyping to a supertype fulfil different interpretations of behavioural compatibility as defined by RM-ODP, see section 2.1. Both can be *alternative roles* of their supertypes with respect to a dual role of the supertype.

Definition: Alternative role
Two roles are *alternatives* with respect to a particular connected role if they both can play with the connected role without violating the safety rules.

If a role breaches safety rules, its environment (i.e. the connected role) will notice it. And with safe subtypes, useful progress can be missing in the truncated role of the connected role; over time this is also noticeable for its environment.

We therefore define our interpretation of behavioural compatibility as follows:

Definition: Compatible connected roles
A role is behaviourally *compatible* with another role if and only if the roles interact safely and their truncated roles contain all their respective progress labels.

Dual roles are by definition compatible. But compatible roles do not have to be dual; both roles may be truncated by each other, but they must not loose progress as a consequence, and they must interact safely.

From the discussions above we see that live subtypes are also compatible with the dual roles of their supertype. Furthermore, two live subtypes where each of their respective supertypes are compatible, are compatible with each other (and with their complements respective supertype). This will be further discussed in the next section.

Note that live subtyping is also central to the definition of compliancy between service roles and the interface roles (semantic interfaces) that are bound to them:

Definition: Compliancy between a service role and a semantic interface
A service role is compliant with a semantic interface if its p-role projected over the connection represented by a semantic connector is a live subtype of the semantic interface.

4.3.4 Live subtyping and collaborations

Here we analyse the effects on collaborations between roles that inherit from semantic interfaces based on live subtyping.

We take it as given that *C* is a semantic connector with interface roles *A* and *B*, and that *A'* is a live subtype of *A*. See Figure 4.21.

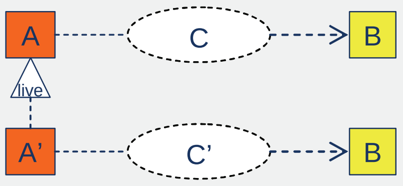


Figure 4.21 : Live subtyping: extending one role is safe and useful

Since *A'* is a live subtype of *A*, *A'* and *B* will by definition be compatible, i.e. the truncated role *A' with B* will by definition have all the progress of *A with B*. The progress of *C* will be identical to that of *C'*, since all that *A'* adds to *A* will be truncated by *B*.

Thus we conclude that progress is neither lost nor gained by live subtyping:

A' ~> A => {progress (A' with B) == progress (A)} => progress (C') == progress (C)

Note that *C'* and *C* will be not necessarily be identical, since *A'* can remove some output.

What if live subtypes of both semantic interfaces collaborate? Observe the case in Figure 4.22.

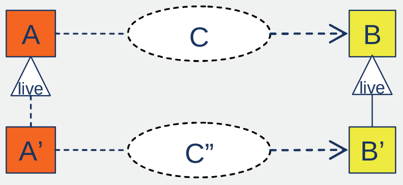


Figure 4.22 : Live subtyping: extending both roles is safe and useful

In Figure 4.22 *A'* is a live subtype of *A*, and *B'* is a live subtype of *B*. *C"* is the collaboration between *A'* and *B'*; the question is what relationship is there between *C"* and *C*.

Since *A'* is a live subtype of *A*, *A'* and *B* will by definition be compatible. Likewise, since *B'* is a live subtype of *B*, *B'* and *A* will also be compatible. The truncated role *A' with B'* will have all the progress of *A with B*. No progress will be added to *C"* due to *A'*, since all that *A'* adds relative to *A* will be pruned by *B'*, as *B'* does not add any output that can use the additional parts of *A'*. The same argument holds for progress added to *C"* due to *B'*.

Thus we can conclude that also *C"* and *C* have the same progress; progress is neither lost nor gained:

{ A' ~> A and B' ~> B } => progress (C") == progress (C)

C" and *C* will not necessarily be identical, since both *A'* and *B'* can remove some output.

We see from the above relationships that live subtyping never results in a loss of progress, and that all live subtypes of semantic interfaces are compatible with both the opposite semantic interface and all its live subtypes. For live subtypes it is thus unnecessary to repeat the validation performed for their supertypes; we known they are compatible.[30]

We have previously discussed how it is possible to create dual roles from safe interface roles. We could use these techniques to create "live" extensions of semantic connectors, i.e. adding new features without losing previously defined progress. However, the interface role thus created would not be a live subtype. See Figure 4.23.

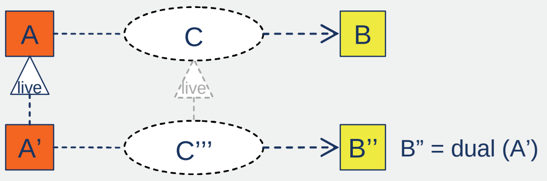


Figure 4.23 : Live subtyping: extending one role, creating dual role from it

In Figure 4.23 *A'* is a live subtype of *A*, and *B"* is created as dual role of *A'*. Since *A* is a safe role, and *A'* is a live subtype of *A*, we know that *A'* is a safe role, and that a dual role *B"* can indeed be created from it. *C'''* is the collaboration between *A'* and *B'*; the question is what relationship there is between *C'''* and *C*.

Since *A'* is a live subtype of *A*, and *B"* is the dual role of *A'*, all progress in *C* is present in *C'''*, since no progress of *A* is removed by *A'*, and *B"* covers all of *A'*.

However, whatever *A'* has added to *A* will be reachable in *C'''*, so *C'''* can have more progress than *C*:

A' ~> A and *B'''* = dual(A) => progress (C''') > progress (C)

We can say that *C'''* is a "live collaboration subtype" of *C*; no progress of *C* is lost, but some additional goals may be achieved. *B"* on the other hand is *not* a safe subtype of *B*, since it adds output that cannot be accepted by *A*.

A consequence of this is that *dual()* cannot be used to create live subtypes.

4.4 Service goals expressed in UML2

In this section we show how service goals can be expressed using UML2, extending the examples presented in chapter 3. We will show that goals can be expressed:

30. If more lax inheritance relationships other that live subtyping are used, e.g. safe subtyping that removes output related to progress, the result will always risk losing basic liveness and/or safety properties; in such a case validation must be performed to determine progress. This was demonstrated in the examples: roles B2 through B5 are safe subtypes of B1, but each had to be analysed against A1 (or A3) to ascertain the resulting progress.

i. in composite structure diagrams as goal expressions in OCL;

ii. in sequence diagrams as goal assertions in continuation labels or state invariants;

iii. in state machines as goal assertions in OCL.

4.4.1 Composite structures

A two-party service structure enriched with *goal expressions* is shown in Figure 4.24.

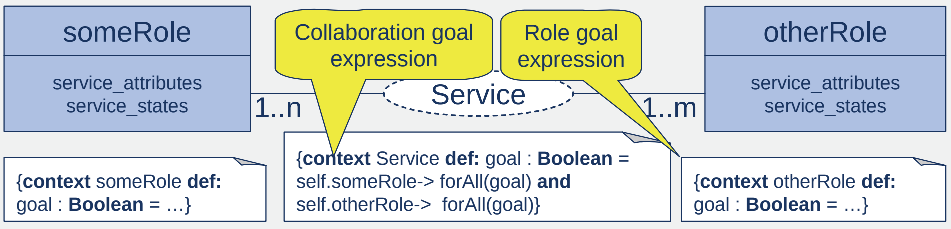

someRole
service_attributes
service_states
Collaboration goal expression
Role goal expression
Service
1..n
1..m
otherRole
service_attributes
service_states
{context someRole def: goal : Boolean = …}
{context Service def: goal : Boolean = self.someRole-> forAll(goal) and self.otherRole-> forAll(goal)}
{context otherRole def: goal : Boolean = …}

Figure 4.24 : Two-party service structure with goal expressions

A three-party service structure enriched with goals is shown in Figure 4.25.

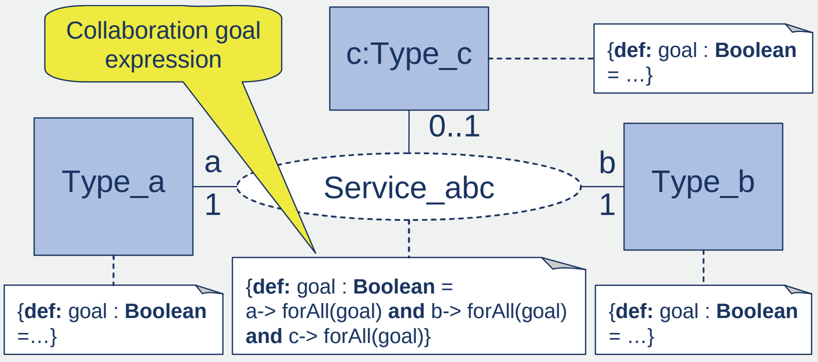

Collaboration goal expression
c:Type_c
{def: goal : Boolean = …}
0..1
Type_a
a
1
Service_abc
b
1
Type_b
{def: goal : Boolean =…}
{def: goal : Boolean = a-> forAll(goal) and b-> forAll(goal) and c-> forAll(goal)}
{def: goal : Boolean = …}

Figure 4.25 : Three-party service structure with goal expressions

4.4.1.1 Semantic connectors and semantic interfaces

A semantic connector with a goal expression is shown in Figure 4.26.

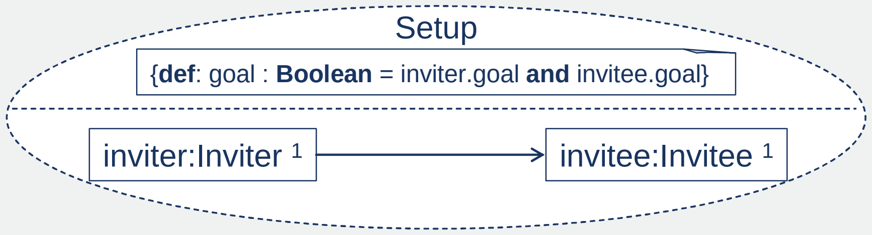


Figure 4.26 : Semantic connector defines a pair of semantic interfaces

In Figure 4.26 the collaboration *Setup* defines two semantic interfaces, *inviter* and *invitee* of respective role types *Inviter* and *Invitee*.

In this example a collaboration goal is expressed as the conjunction of the goals of the semantic interfaces. Expressing a collaboration goal for a semantic connector is optional according to the method guidelines given in section 4.7.

4.4.2 Interactions

Sequence diagrams are well suited for determining when service goals are true, since they define cross-cutting behaviour. Sequence diagrams can be enriched with service goals.

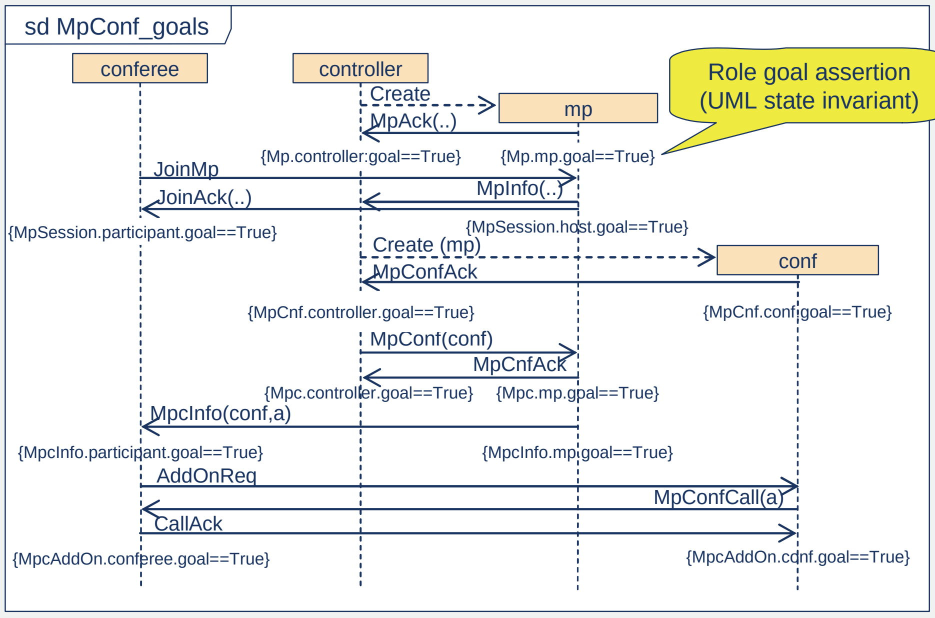

sd MpConf_goals
conferee
controller
mp
Role goal assertion (UML state invariant)
Create
MpAck(..)
{Mp.controller:goal==True}
{Mp.mp.goal==True}
JoinMp
MpInfo(..)
JoinAck(..)
{MpSession.participant.goal==True}
{MpSession.host.goal==True}
Create (mp)
conf
MpConfAck
{MpCnf.controller.goal==True}
{MpCnf.conf.goal==True}
MpConf(conf)
MpCnfAck
{Mpc.controller.goal==True}
{Mpc.mp.goal==True}
MpcInfo(conf,a)
{MpcInfo.participant.goal==True}
{MpcInfo.mp.goal==True}
AddOnReq
MpConfCall(a)
CallAck
{MpcAddOn.conferee.goal==True}
{MpcAddOn.conf.goal==True}

Figure 4.27 : Meeting Place Conference scenario with role goals

In Figure 4.27 semantic connectors have been factored out by grouping the interactions of Figure 3.11 and inserting pairs of state invariants[31] in Figure 4.27 to indicate pairs of *role goal assertions* for each semantic connector.[32]

Figure 4.28 shows an interaction that results in the achievement of the service goals of the composite *Call* service:

- The achievement of the collaboration goal is expressed by a *continuation label*[33] covering the lifelines representing the service roles;
- The achievement of role goals is expressed by *state invariants*, here using curly brackets. These are what we called *goal assertions*, here expressed in OCL.

31. State invariants in UML can be written in state symbols, or, like here, between curly brackets.
32. Note that event occurrences from different semantic connectors interleave in Figure 4.27, like the *MpInfo* signal belonging to the *Mp* collaboration appearing in between *MpSession* events.
33. Continuation labels were introduced in UML2 to allow conditional fragments to be semantically combined to continuations in referencing interaction fragments, and normally appear as the first or last element of a conditional operand. To use this modelling element, one must place the interaction in an *alt* operand.

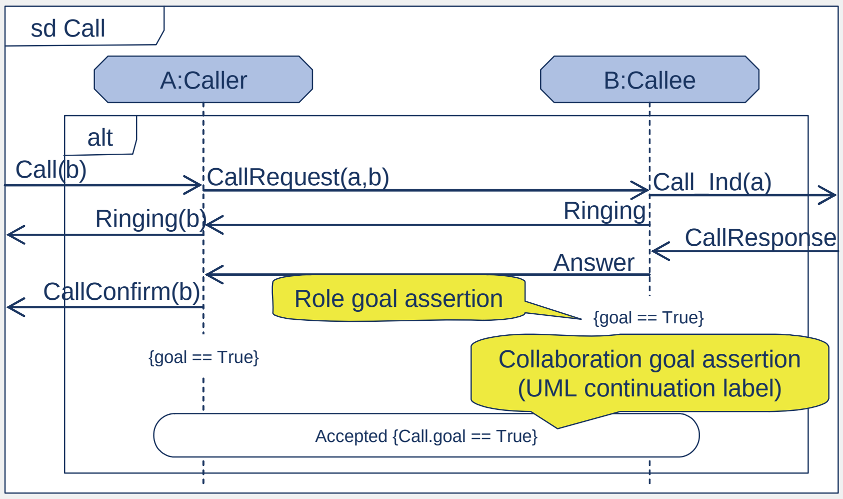


Figure 4.28 : Sequence diagram for composite Call service with service goals

Note that UML *continuations* are just labels, and not real states. However, they can be informally used to represent collaboration states. We use the continuation as a placeholder for collaboration goal assertions. The label of the continuation in Figure 4.28 is *Accepted.* This is simply a naming convention indicating that it marks a goal achievement of the semantic connector of the same name. In addition to the label we insert a collaboration goal assertion. It is expressed in OCL, and is placed within curly brackets.

Note that goal expression in the state invariants on a lifeline refers to the role goal of the lifeline.

E.g. in Figure 4.28 the OCL expression *goal == True* on the lifeline *B* is equivalent to the OCL expression *B.goal == True*. The navigation capabilities of OCL can be used to evaluate more complex expressions.

Sequence diagrams are well suited for expressing the role goals of semantic interfaces, as can be seen in Figure 4.29.

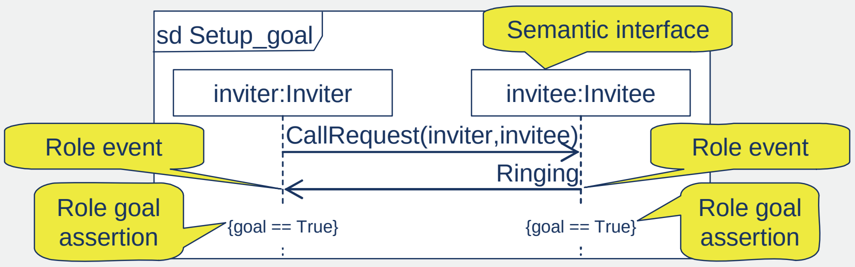


Figure 4.29 : Sequence diagram for semantic connector with goals

Figure 4.29 depicts a sequence diagram for the semantic connector *Setup* with goal assertions. Note that interactions with the environment are not included as they were in Figure 4.28, and that state invariants are used to express role goal assertions.

State invariants on lifelines of sequence diagrams express goals. In Figure 4.29 the last event on each lifeline causes a goal to be reached, according to the goal assertions. The state invariants can be used to derive event goals; this was discussed in section 4.2.5.1.

4.4.2.1 State invariants in interaction diagrams

In interaction diagrams we exploit state invariants to express the achievement of role goals. As stated earlier, a state invariant is a constraint evaluated when the next event occurs on the lifeline to which it is attached. The constraint may require that the object is in a named state, or require that expressions on values reachable from the object evaluate to *True*. It is the latter we use to express assertions on role goals. Note that the assignment of appropriate values (*True*/*False*) to goal variables is not depicted in interaction diagrams; this is deferred to the state machine diagrams defining role behaviour.

The semantics of state invariants in UML models is as follows: If the constraint is violated (i.e. does not evaluate to *True*), then the model is ill-formed. In the trace semantics that underlie UML interaction diagrams, a trace that contains an event on an instance line directly following a state invariant that is violated becomes a member of the set of invalid traces. I.e. all traces that have a state invariant with a false constraint are invalid.

This semantics suits our purpose well, as it helps to enforce the consistent use of role goals. Tools can help service designers validate whether the models are valid and consistent. This will be discussed in chapter 6.

Collaboration goals modelled by continuation labels do not have the same properties. We use continuation labels as placeholders for collaboration goal assertions, although they lack the capability of marking traces as being valid or invalid.

4.4.3 State machines

State machines are used for describing the behaviour of actors, service roles and interface roles. How to express service goals in state machines is discussed below.

4.4.3.1 Semantic interface behaviour

The behaviour of semantic interfaces is defined in extended protocol state machines. By definition, semantic interfaces contain goals. A pair of semantic interfaces has consistently labelled event goals: an event marked with a progress label in one semantic interface is marked with a corresponding progress label in its opposite role, and vice-versa.

Figure 4.30 shows an example of consistently defined event goals.

Figure 4.30 exemplifies different ways of expressing role goals:

1. Goals may be expressed by role goal assertions, as indicated by callouts. Assignment of values to the goal variable used in the assertion is also shown in the example.
2. Goals may be expressed by event goals, i.e. events marked by progress labels. Goal events can be signals sent or consumed, or states or exit points reached.

Note that the pair of semantic interfaces described in Figure 4.30 have consistently defined event goals, although the model elements to which they are attached are different.

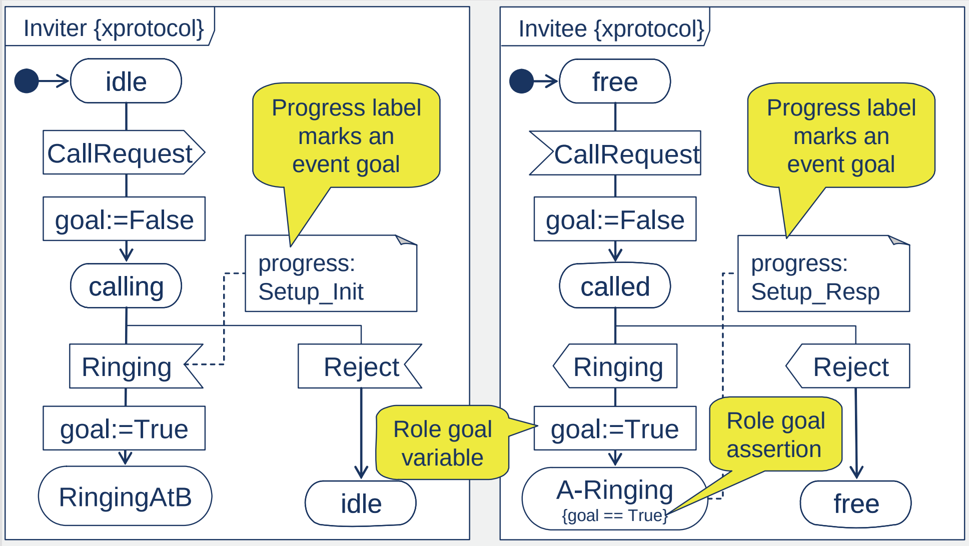


Figure 4.30 : Role behaviour for interface roles with goals

Note also that there is a close relationship between role goal assertions and event goals, in that event goals can as a rule be attached to goal states marked by goal assertions. This is a mechanical relationship that can be supported by tools, see section 4.2.5.2.

4.4.3.2 Service role behaviour

Sketches of the behaviour of the service roles *Caller* and *Callee* are found in Figure 4.31.

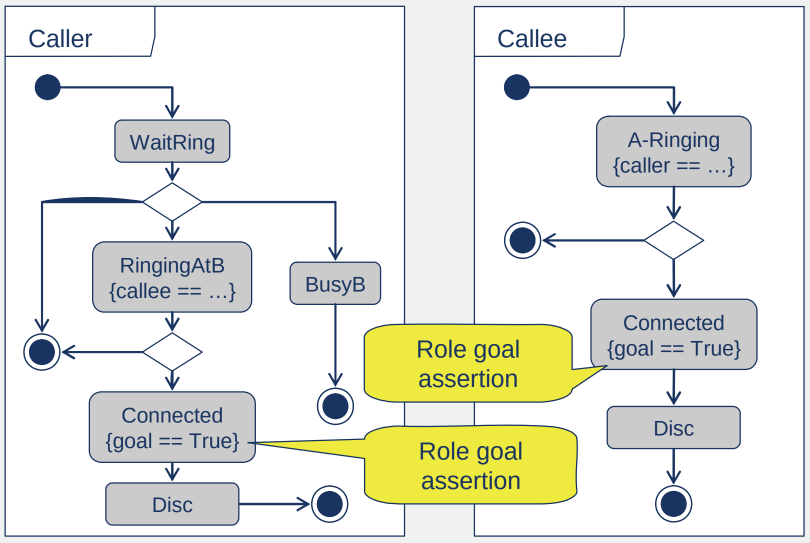


Figure 4.31 : Sketch of service roles of the Call service with service goals

The state machines in Figure 4.31 express service goals in the form of state assertions. Two service goals are defined, one in each service role, as indicated by callouts.

4.4.3.3 Collaboration behaviour

UML2 allows us to define collaboration behaviour in a state machine diagram, though nothing is said in the standard or reference book about what can be achieved by this.

In our approach, we exploit this modelling opportunity to highlight the expression of collaboration goals. Figure 4.32 shows a collaboration state machine with collaboration goals.

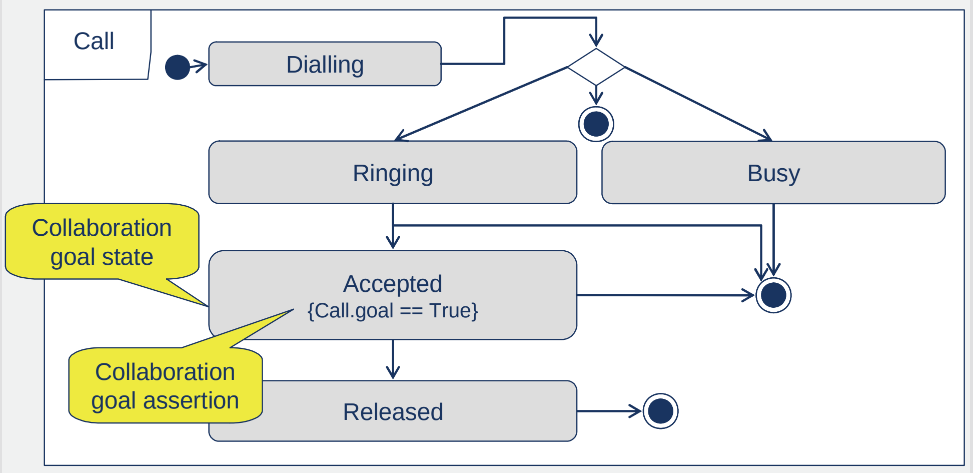


Figure 4.32 : Overview of collaboration states - goal added

In Figure 4.32 an overview of the states of the collaboration *Call* are described by a state machine. Here a collaboration goal state is identified by a collaboration goal assertion.

In the collaboration state *Accepted*, the collaboration goal expression is asserted to be *True*. This indicates that the collaboration goal is achieved in this state. We call this a *collaboration goal state*. We contend that Figure 4.32 shows in a succinct way that the purpose of the *Call* collaboration is that the combined behaviour of the roles should result in the call being accepted.

However, it is not clear from Figure 4.32 what the states of the involved roles are. As stated in section 3.3.7, we can adopt a state-oriented view to express this. Collaboration goals can be inserted in collaboration state machines when using the state-oriented view, see Figure 4.33.

By expressing collaboration goals as state invariants one may express properties that shall hold for some joint states of the roles. This ultimately places requirements on actors that play the collaborating roles.

4.4.3.3.1 Role states with collaboration state assertions

As stated in section 3.3.7, we can include references to collaboration states when we describe the state machines of the service roles, by adopting the state-oriented view. In terms of expressing goals, referencing to collaboration goal states in the role states is of paramount importance.

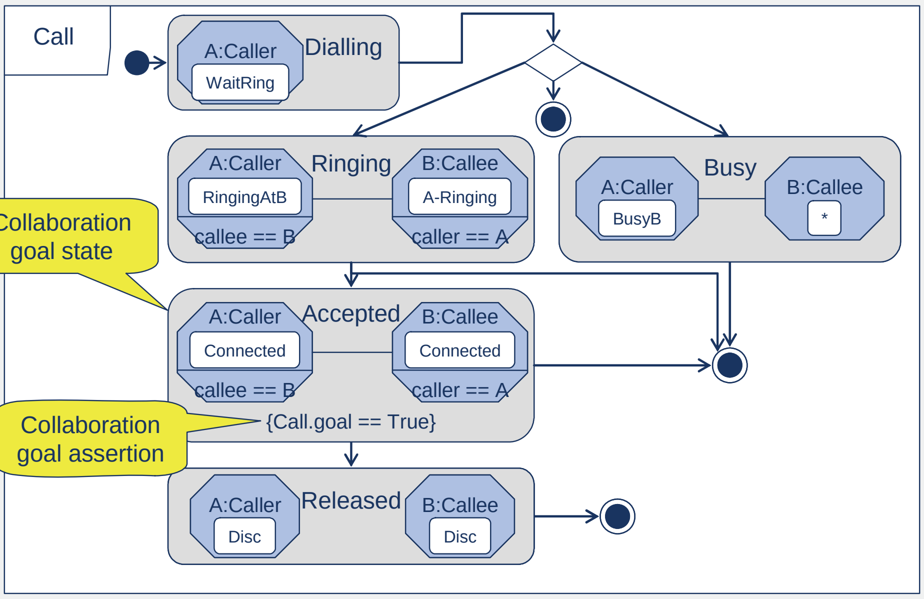


Figure 4.33 : Collaboration states using state orientation - goal added

The states of the role *Caller* is shown in Figure 4.34, here defined in a state-oriented fashion, i.e. where the state of the collaboration is asserted in each role state.

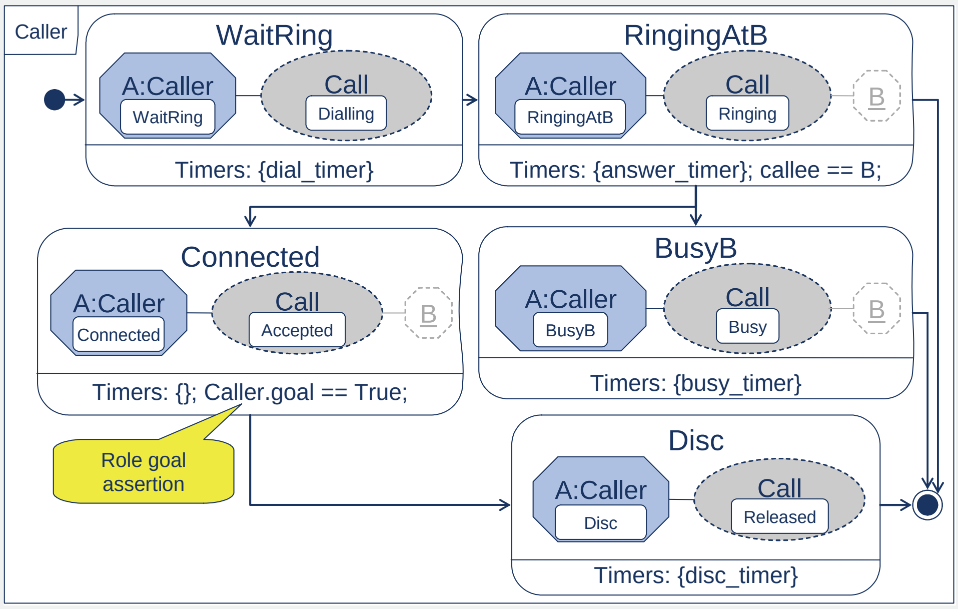


Figure 4.34 : Role states for Caller role with state orientation

In Figure 4.34 we highlight role goal assertions; these are states where the role goal expression is asserted to be *True*, as indicated by a callout in Figure 4.34. As we can see, this role goal state refers to the collaboration state *Accepted*, where the collaboration goal is fulfilled.

State orientation opens up for validation opportunities, and we argue that the state-oriented viewpoint is particularly suited for human analysis, since the consistency between goals defined in roles and in collaborations can easily be discerned by the experienced service designer. This applies to an even greater extent when these state machine diagrams are compared with goal sequence diagrams, to be introduced in the next chapter.

4.5 Related work

Goal-driven software engineering is a direction within requirements engineering.

The concept of goal has been used by the artificial intelligence community for over a decade, e.g. [Cohen, Levesque 1990]. Methodologies [Myklopoulos et alia 1999] and languages [GRL 2003] identifying actors and goals have been developed, but they mainly address non-functional aspects.

There has been some work on functional goals. The [KAOS] project defines a goal as "*a prescriptive statement of intent about some system whose satisfaction in general requires the cooperation of some of the agents forming that system.*" Goals prescribe intended behaviours, and are formalized in a real-time linear temporal logic (LTL). They define *Agents* to be active components such as humans, devices or software components that play some role towards goal satisfaction. *Functional goals* refer to services the system is expected to provide.

In KAOS, an *operation* is an input-output relation over objects; operation applications define state transitions. When specifying an operation a distinction is made between domain pre/postconditions and additional pre-, post- and trigger conditions required for achieving some underlying goal. A pair (domain precondition, domain postcondition) captures the elementary state transitions defined by operation applications in the domain. A *required precondition* for some goal captures a permission to perform the operation when the condition is *True*. A *required trigger condition* for some goal captures an obligation to perform the operation when the condition becomes *True* provided the domain precondition is *True*. A *required postcondition* defines some additional condition that any application of the operation must establish to achieve the corresponding goal.

In KAOS terms, our service goals are "functional goals" that define "required preconditions". However, there is no obligation to perform anything due to a goal condition in our approach, so we do not express "required triggered conditions". We use UML and OCL rather than LTL, and provide a compositional and modular approach that also enables validation and discovery mechanisms.

To the best of our knowledge, no one has addressed goals of state machines in the way suggested herein.

4.6 Summary: Modelling of service goals

In the sections above we have presented different ways of expressing the various types of service goals. We summarize the service goal types and what model elements they are attached to in Table 4.2:

Table 4.2: Modelling of service goals in UML

Service goal type	Goal expression form	UML model element	UML diagram
Collaboration goal			
State-like goal	Goal expression in OCL	Collaboration	Composite structure diagram
	Goal / state assertion	Collaboration state	State machine diagram
	Continuation label	Role lifeline	Sequence diagram
Role goal			
State-like goal	Goal expression in OCL	Classifier	Composite structure diagram
	Goal / state assertion	Classifier state	State machine diagram
	State invariant	Role lifeline	Sequence diagram
Event goal	Progress label	Trigger, state, exit point	State machine diagram

4.7 Method guidelines

We summarise our goal modelling approach in a number of method rules.

Method rule: Identify semantic connectors
Semantic connectors are identified by a unique collaboration name <*feature*>.

Method rule: Identify semantic interfaces
Semantic interfaces are identified by collaboration roles within a semantic connector. Semantic interfaces have a role multiplicity of exactly 1.

Method rule: Identify goal expressions of semantic connectors (optional)
Define collaboration goal expressions in OCL as a conjunction of the role goal expressions of the semantic interfaces (if defined). Place OCL expressions in notes, either attached to the collaboration, or by identifying the target element in the context of the OCL expression.

Method rule: Identify role goal expressions in semantic interfaces (optional)
Define role goal expressions of semantic interfaces in OCL. The expressions can only use data visible to the role, including role states. Place OCL expressions in notes, either

attached to the roles, or by identifying the target element in the context of the OCL expression.

Method rule: Identify role goal variables
Identify the role data needed to determine goal fulfilment. Use the naming convention *goal* for the goal variable, or *<service>_<role>_goal* if several role goals apply due to the service role taking part in multiple collaborations with disjunct goals.

Method rule: Identify or derive event goals of semantic interface
Define goals of a semantic interface by inserting progress labels on events that uniquely mark progress. If goal states are identified by goal assertions, then define progress labels on these states or on the unique events leading to the goal states. Use tool support for this derivation process, if available.

Method rule: Compose services from semantic connectors
Bind the semantic interfaces of semantic connectors to the service roles of the composite service (or to other composite interface roles composed in the same way). Use a unique name for the collaboration use within the collaboration defining the composite service.

Method rule: Identify collaboration goals of composite services (optional)
Define collaboration goal expressions for composite services in OCL, e.g. as the conjunction of goal expressions defined for the service roles or semantic interfaces. If a role is optional or n-ary, use the OCL expression *forAll(...)* to evaluate the goals of role instances.

Method rule: Insert goal assertions in service roles (optional)
For each service role, place assertions on the goal expressions in states where the goal is achieved. The assertions can use data visible to the role, including goal variables and states. Goal variables must ultimately be found in actors that play the roles.

5 Goal sequences

In this chapter we discuss ways of expressing how services and service goals depend on each other, and to what extent their dependencies can be described as sequences of collaboration uses in what we call *collaboration goal sequences*, how these can be used to derive *role goal sequences*, and how sequences of roles played by actors can be described in *actor goal sequences.* The latter can be used in service discovery, the topic of chapter 7.

Collaboration goal sequences are unique in that they express a combination of horizontal and vertical dimensions introduced in section 1.3.1: the horizontal role relationships defined by collaborations are combined with the vertical structure (sequence of goals) of roles bound to classifiers. We shall see that the composition of services from semantic connectors enables us to express goal sequences in a structured manner.

5.1 Collaboration goal sequences

As we have seen in the previous chapter, UML2 collaborations and collaboration uses enable services to be composed of semantic connectors, as in Figure 5.1.

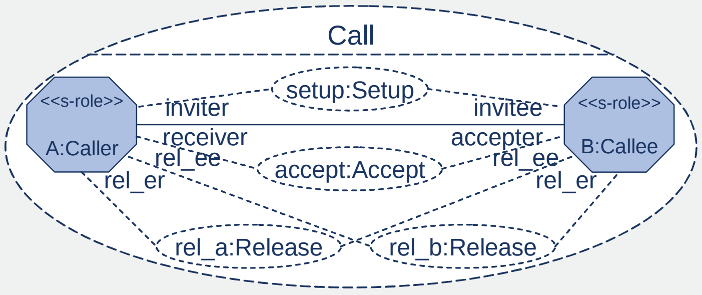


Figure 5.1 : Call composed of semantic connectors

In Figure 5.1 the *Call* service is composed of the semantic connectors *Setup*, *Accept* and *Release*, as referenced by collaboration uses. We argue that in using this approach:

- we achieve a structure that provides a good overview;
- each collaboration use defines a reusable unit of behaviour - the semantic connectors.

As pointed out in section 2.6.2, when we reuse the semantic connectors in new compositions we need to express how their goals related to goals of other semantic connectors. For instance, it is not clear from Figure 5.1 whether the goals of the semantic connectors are ordered in any particular way, i.e. if a certain order of goal achievement of *setup*, *accept*, *rel_a* and *rel_b* is required or implied by the composite service *Call*. It is not clear whether the events of the semantic interfaces are fully interleaved, strictly sequential, or a combination. More generally the ordering operators are *sequence*, *parallel*, *choice* and *call*.

Call indeed assumes a particular sequencing of the events and goals of the semantic connectors *Setup* and *Accept*. This can be gleaned from the sequence diagram in Figure 5.2.

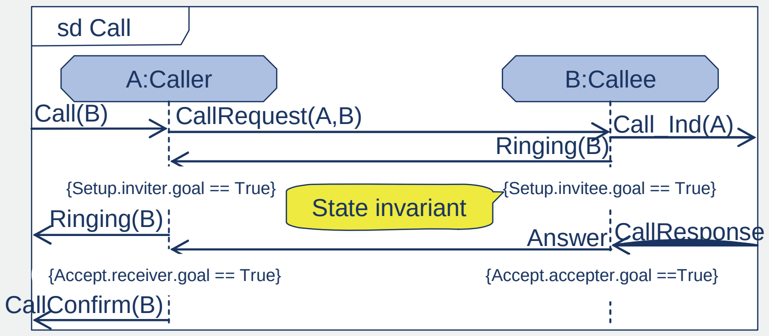


Figure 5.2 : Interaction for successful Call

Figure 5.2 shows an interaction for *Call* consisting of call setup followed by call acceptance. Note that the lifelines are instances of the service roles *A* and *B* in Figure 5.1, while the state invariants identify goal achievement for the semantic connectors *Setup* and *Accept*.

Here goal achievement is expressed in terms of goals of the interface roles (*inviter/invitee* and *receiver/accepter*, respectively). A state invariant attached to a lifeline representing a role expresses:

i. the fulfilment of a role goal due to preceding events;

ii. that the role goal shall be fulfilled prior to subsequent events on the lifeline.

For instance in Figure 5.2, the state invariant *Setup.invitee.goal* expresses that this goal is achieved after the signal *Ringing* has been sent from *B*. It also implies that this expression must be valid (evaluate to *True*) when the *CallResponse* signal is received by *B*.

To express relationships between the goals of semantic connectors that are bound to service roles, we use what we call *collaboration goal sequences*.

Definition: Collaboration goal sequence
A collaboration goal sequence describes the global ordering of goals of semantic connectors in the context of a composite service structure.

The semantics of goal sequences is discussed in section 5.4.

Collaboration goal sequences may be expressed:

1. as dependencies between the collaboration uses in a collaboration;

2. using interaction overview diagrams.

We discuss both alternatives[1] in the following sections.

5.1.1 Collaboration goal sequences modelled by dependencies

Dependencies in UML indicate semantic relationships between two or more model elements, and indicate a situation in which a change to the supplier element may require a change to the client element of the dependency[2]. UML includes predefined dependencies, some with particular semantics, such as associations and generalizations. None of the predefined types suits our purpose, so a special *goal dependency* is suggested.

Definition: Goal dependency
A goal dependency is a dependency between two collaboration uses expressing that goal achievement for the supplier element is a precondition for goal achievement of the client element. In UML a stereotyped *<<goal>>* dependency is used.

The stereotype can be used to impose a particular semantics on the collaboration goal sequence relationship, imposing a strict order on goal achievement and a partial order on the collaboration events. The semantics of goal dependencies is discussed in section 5.4.

5.1.1.1 Example 1: Call service

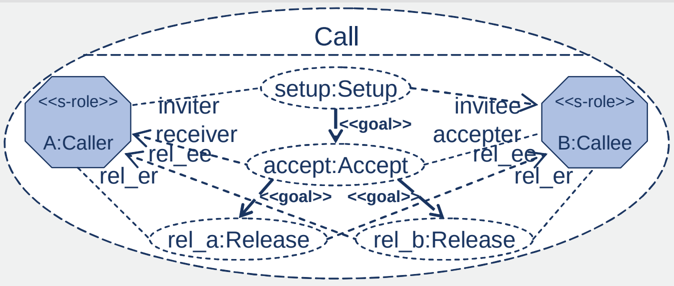


Figure 5.3 : Collaboration goal dependency for Call

Figure 5.3 exemplifies goal dependencies. The goal dependency from *setup* to *accept* expresses that *setup* reaching its goal is a precondition for *accept* to be enabled in the context of the *Call* service. The goal dependencies from *accept* to *rel_a* and *rel_b* express that reaching the goal of *accept* enables two succeeding semantic connectors, i.e. call release initiated either by the *Caller* (*rel_a*) or the *Callee* (*rel_b*)[3]. Note that goal dependencies do not define causality, so the reason that one or the other is invoked remains undetermined.

5.1.1.2 Example 2: MpConf service

A second example is the Meeting Place Conference service *MpConf*, see Figure 5.4.

Figure 5.4 shows *MpConf* as a composition of six semantic connectors. Here too the semantic connectors must achieve their collaboration goals in a certain order.

1. Activity diagrams are also an option. An additional possibility, which we do not elaborate, is to express goal relationships using pre- and post-conditions. OCL can only express preconditions on operations. Since services are not modelled by operations but by collaborations, some mechanism or language other than OCL must be used.
2. See [UML 2.0] p. 58.
3. The *Caller* can abort the call setup before *setup* or before *accept* reaches its goal. The *Callee* can likewise cancel an incoming call (e.g. in GSM). In this example we assume that the signals related to cancelling call setup belong to the semantic connectors *Setup* and *Accept*; and that *Release* specifies the case where an established call is release. How best to specify aborting initiatives (e.g. from users or due to timeouts) is a topic for further work.

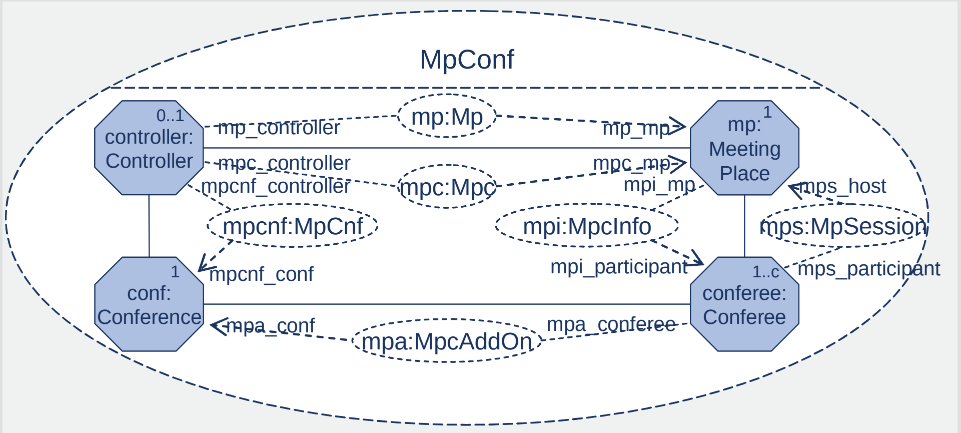


Figure 5.4 : MpConf composed of semantic connectors

For instance in the context of *MpConf* it is not possible to join a Meeting Place with *mps* if the *mp* service that configures the Meeting Place fails, so the success of *mp* is a precondition for *mps*. Such goal relationships are not evident from Figure 5.4. We may state such dependencies explicitly using dependencies, see Figure 5.5.

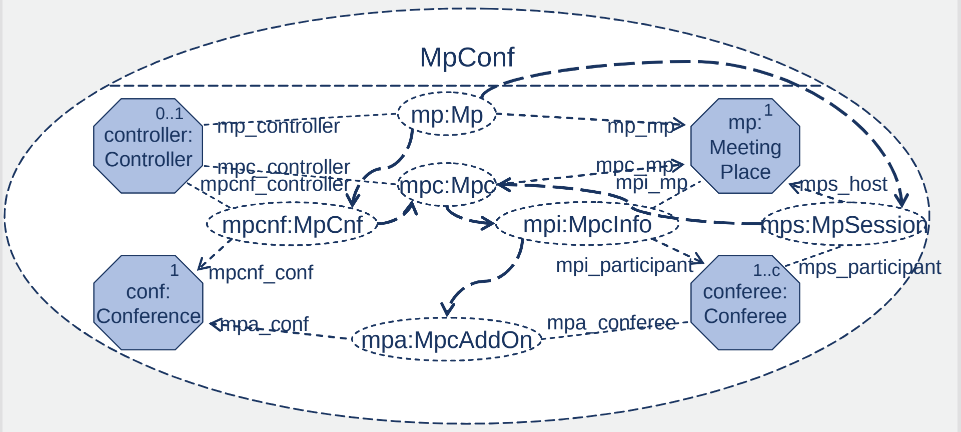


Figure 5.5 : Collaboration goal dependencies for MpConf

In Figure 5.5 we have included a number of goal dependencies, such as the requirement that *mps* and *mpcnf* achieve their goals before *mpc* and subsequently *mpi* and *mpa* can achieve theirs. To prevent cluttering up the diagrams, the stereotype name *<<goal>>* is omitted.

However, adding such dependencies clutters up the diagram. We therefore seek an alternative way of expressing goal dependencies in the next section.

5.1.2 Collaboration goal sequences modelled by interaction overviews

Interaction overview diagrams are new to UML2, and can be usefully employed in composite service structures. A promising approach is the following:

1. for each semantic connector, define a *connector goal interaction*, i.e. a sequence diagram with interactions that lead to the achievement of a goal of the semantic connector;
2. for each composite service structure, define a *collaboration goal sequence* that references the connector goal interactions as interaction uses[4], using the sequencing mechanisms available in interaction overview diagrams[5] to represent the sequential relationships between the goals of the composed semantic connectors.

Definition: Connector goal interaction
A connector goal interaction is an interaction defining the goal achievement of a semantic connector.

It is desirable to compose services from semantic connectors, and in such a way that the semantic connectors are designed independently of their use in compositions. Therefore a connector goal interaction should be defined without knowledge of its use in a composition. This is supported by UML2 collaborations, since role binding makes it easy to reuse interactions in new settings, without having to parameterize or rename interaction instances (i.e. the lifelines of the interaction diagrams).

We introduce a diagram type designed to describe collaboration goal sequences, which we call a *goal sequence diagram*.

Definition: Goal sequence diagram
A goal sequence diagram is a graphical adaptation of the interaction overview diagram showing how goal achievements in semantic connectors of a composite service structure are ordered.

We introduce the principles and diagrams by way of two examples.

5.1.2.1 Example 1: Call service

Relationships between a semantic connector, a composite service structure, an interaction overview diagram and a pair of goal interactions are exemplified in Figure 5.6.

Considering Figure 5.6, note that:

- The connector goal interactions *Setup_goal* and *Accept_goal* are defined within the semantic connectors *Setup* and *Accept*, respectively;
- The lifelines of *Setup_goal* and *Accept_goal* end with state invariants;
- The lifelines of *Setup_goal* and *Accept_goal* use their own collaboration role names, and not the role names of the composite service structure;
- The interaction *Accept_goal* does not have any initial state invariants referring to *Setup_goal*, i.e. *Accept* is in itself not restricted to be a successor to *Setup*;

4. *An interaction use is a parameterized reference to an interaction within the body of another interaction. When an interaction use is executed, the effect is the same as executing the referenced interaction with the substitution of the arguments supplied as part of the interaction use*. [UML2 Ref] p. 412.
5. Interaction overview diagrams combine the control flow mechanisms from activity diagrams with the sequencing of events from sequence diagrams, but follow the interleaving trace semantics of interactions. They are an adaptation of the high-level message sequence charts introduced in [MSC-96].

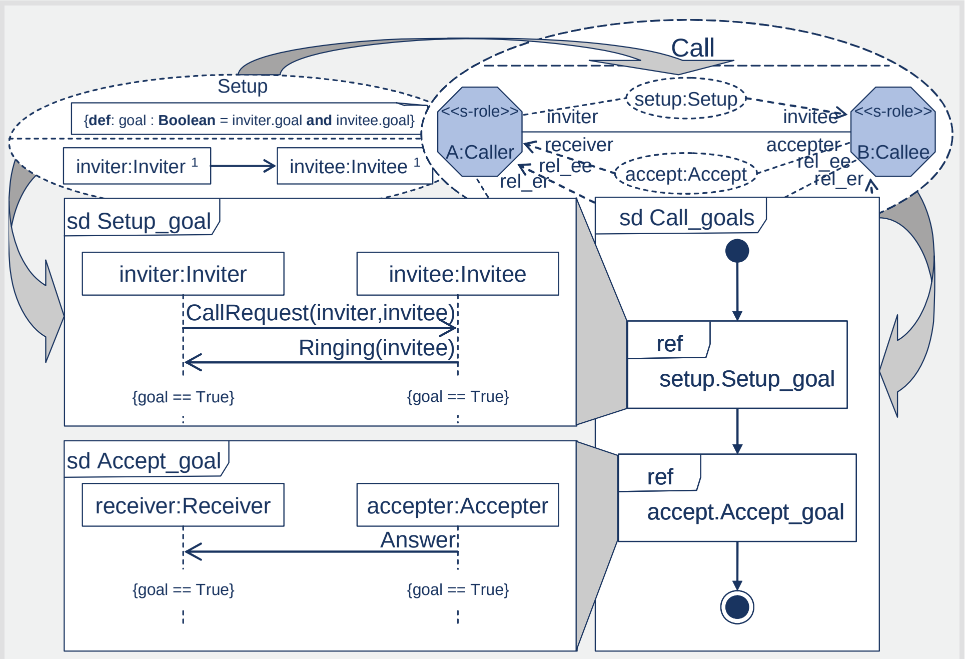


Figure 5.6 : Connector goal interactions and interaction overview for Call

- The interaction overview diagram *Call_goals* is defined in the composite service structure *Call*. *Call_goals* references the interaction uses defined by the collaboration uses, e.g. the interaction use *setup.Setup_goal* refers to the interaction *Setup_goal* defined by the collaboration use *setup* (of collaboration type *Setup*);
- The collaboration *Call* binds the roles (i.e. the lifelines) of the interactions to roles of the collaboration (e.g. *inviter* and *receiver* are bound to the collaboration role *A*), making it clear that it is *A:Caller* that outputs the *CallRequest* signal of the interaction *Setup_goal*, and not *B:Callee*;
- The resulting interaction seen from the point of view of the collaboration *Call* is identical to the interaction in Figure 5.2 on page 108.

Interaction overview diagrams combined with goal interactions defined in subordinate collaborations provide a powerful and flexible means of expressing composite interactions. To highlight the aspect of service goals we propose an adaptation of interaction overview diagrams, which we call *goal sequence diagrams*.

The interaction overview diagram for *Call* and its corresponding goal sequence diagram are both shown in Figure 5.7.

The goal sequence diagram for *Call* is to the right in Figure 5.7. The conversion from interaction overview diagram to goal sequence diagram is straightforward: in goal sequence diagrams we replace the names of interaction uses of the interaction overview diagrams with illustrations of the semantic connector involved, showing the collaboration uses with their role bindings according to the composite service structure.

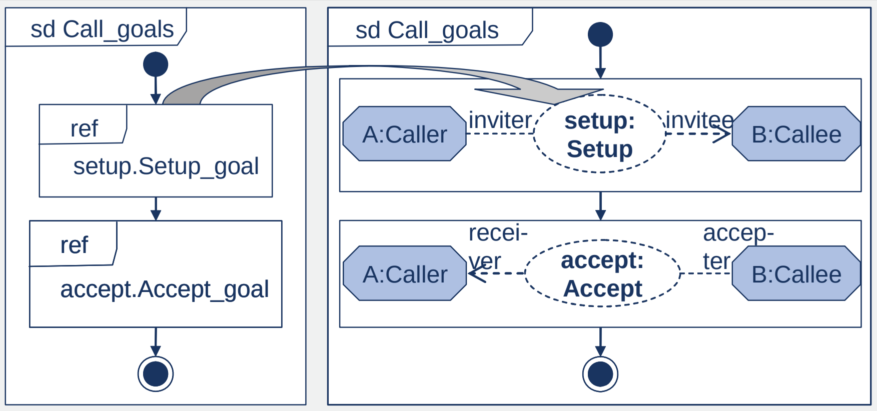


Figure 5.7 : Interaction overview and goal sequence diagrams for Call

Each element of a goal sequence diagram has a corresponding connector goal interaction, e.g. *setup:Setup* corresponds to *setup.Setup_goal* (as indicated by the block arrow).

In Figure 5.7 the lower element of the goal sequence diagram refers to the interaction use *accept.Accept_goal*. It illustrates that the goal of the preceding collaboration use *setup:Setup* is a precondition of the goal of *accept:Accept* when used in the *Call* service. See Figure 5.8 for the related collaboration goal expressions.

{**context** Setup
def: goal : **Boolean** =
invitee.goal **and** inviter.goal}

{**context** Accept
def: goal : **Boolean** =
receiver.goal **and** accepter.goal}

{**context** Call
def: goal : **Boolean** =
Accept::goal}

Figure 5.8 : Goal expressions relevant for Call

Note that the goal of the composite service *Call* is not fully understood without the goal sequence diagram: the goal expression for *Call* in Figure 5.8 only refers to *accept* being reached and says nothing of *setup*; the goal sequence diagram states that *setup* must reach its goal prior to *accept*, thus completing the picture.

We have suggested ways of referring to preceding service goals in goal sequences by various graphical adornments; see the discussion on this in section 10.2.2. Alternative forms of goal sequence diagrams are discussed in section 10.2.8.

5.1.2.2 Example 2: MpConf service

A more comprehensive example is provided by the *MpConf* service, whose composition was given in Figure 5.4. Connector goal interactions for the semantic connectors that *MpConf* is composed of are shown in Figure 5.9 and Figure 5.10.

The goal dependencies for *MpConf* are as follows:

- a Meeting Place cannot be joined in an *MpSession* before it is configured by *Mp*, nor can a Conference be configured for use in a Meeting Place before *Mp* succeeds;
- the Meeting Place cannot be configured with a conference by *Mpc* before the conference is configured by *MpCnf*, and at least one user has joined with *MpSession*;

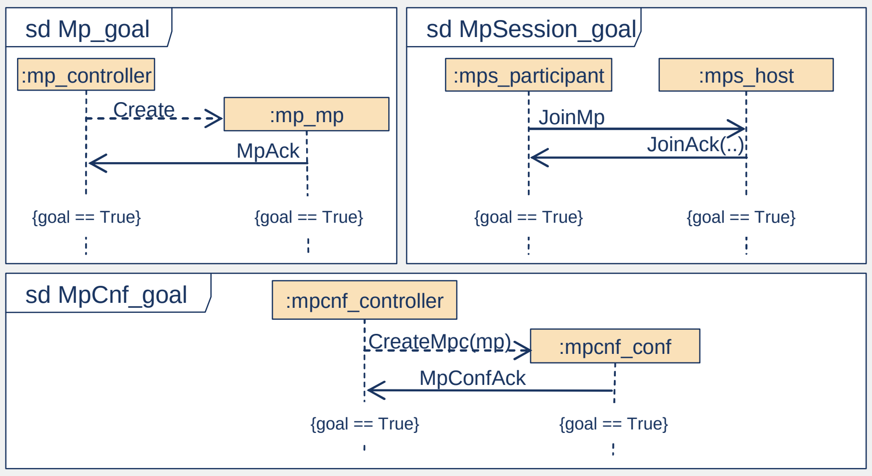


Figure 5.9 : Connector goal interactions for Mp, MpSession and MpCnf

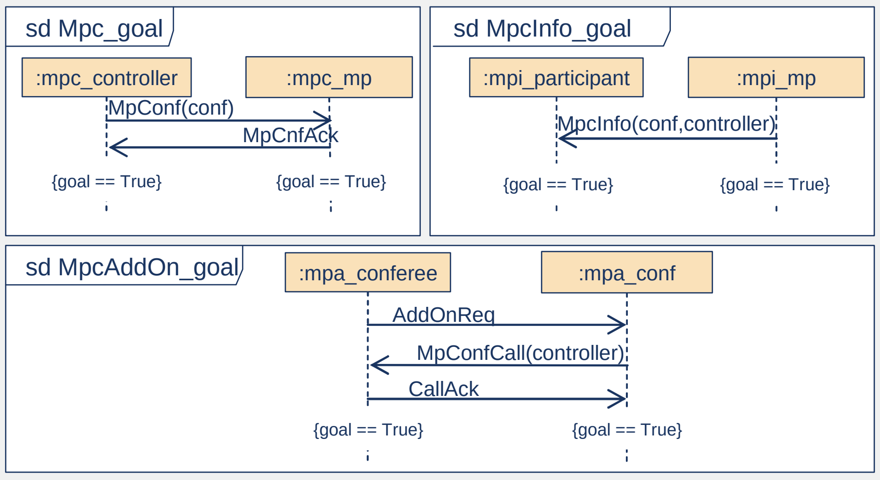


Figure 5.10 : Connector goal interactions for Mpc, MpcInfo and MpcAddOn

- information about the Meeting Place Conference cannot be sent by *Mpi* before the Conference service *Mpc* is running;
- A participant cannot choose to join the Meeting Place conference before it is informed of the conference's existence by *Mpi.*

These goal relationships are shown using an interaction overview diagram in Figure 5.11.

The parallel merge construction of the fork and join nodes in Figure 5.11 states that the achievement of the goals of *mpcnf* on the left side, and *mps* on the right side, are independent of each other. Conversely, the join node prior to *mpc* expresses a dependency between the *Mpc* collaboration and the goals of the joined paths: the goals of both *mpcnf* and *mps* must be achieved prior to *mpc*.

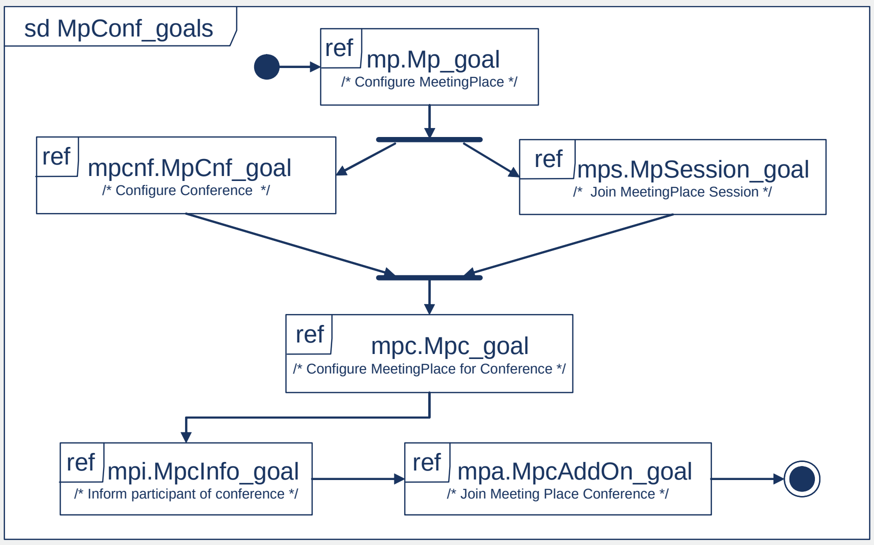

sd MpConf_goals
ref mp.Mp_goal
/* Configure MeetingPlace */
ref mpcnf.MpCnf_goal
/* Configure Conference */
ref mps.MpSession_goal
/* Join MeetingPlace Session */
ref mpc.Mpc_goal
/* Configure MeetingPlace for Conference */
ref mpi.MpcInfo_goal
/* Inform participant of conference */
ref mpa.MpcAddOn_goal
/* Join Meeting Place Conference */

Figure 5.11 : Interaction overview diagram for MpConf

There are no loops in Figure 5.11; although loops can be expressed in UML, e.g. to describe several participants joining a *MpConf*, we do not suggest using loops in collaboration goal sequences. One reason is that showing complete system behaviour is not the aim of collaboration goal sequences. It is only to show the relationships between the goals of the constituent semantic connectors. Another reason is that loops can cause problems for the service discovery mechanisms defined in chapter 7. Thirdly, we can express loops in actor goal sequences, as we shall see in section 5.3, and thus suggest doing without them here. See section 10.1.2 for a discussion on loops in goal sequence diagrams.

The goal sequence expressed using a goal sequence diagram is shown in Figure 5.12.

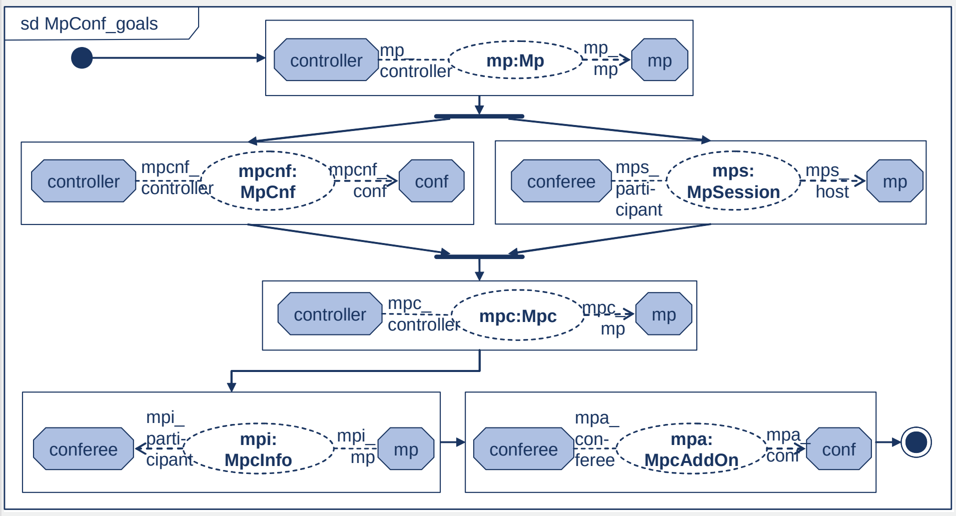


Figure 5.12 : Goal sequence diagram for MpConf

5.1.2.3 Discussion

Goal sequence diagrams visualise the role bindings between related semantic connectors, which interaction overview diagrams do not. For instance Figure 5.12 highlights the pivotal role that *mp* has in *MpConf*, and shows that the *conferee* must be a Meeting Place *participant*. These relationships are not easy to understand from sequence diagrams like the one in Figure 3.11 on page 52.

In themselves goal sequence diagrams do not represent any change to the UML semantics; we consider them to be a presentation option. In Figure 5.7 we for instance illustrate that the *receiver* role of the *Accept* collaboration is bound to the *A:Caller* role of the *Call* service; the *A:Caller* role is likewise bound to play the *inviter* role in the preceding semantic connector *Setup*. Formally the role bindings are defined in composite collaborations as in Figure 5.1, while goal sequence diagrams describe how the goals of semantic connectors are ordered.

Goal sequence diagrams can be formalised beyond what we have done; this will be discussed in section 8.2.2.1.

5.1.3 Method guidelines

The method guidelines for collaboration goal sequences are summarized as follows:

Method rule: Define connector goal interactions in sequence diagrams
For each semantic connector, define a sequence diagram describing event occurrences leading to its collaboration goal. Name the interaction *<feature>_goal*. Only lifelines representing the semantic interfaces of the semantic connector should be included. Lifelines should bear the names of the interface roles. Signals to and from the environment should be suppressed. Only events related to the collaboration should be included.

Method rule: Include only successful interactions in connector goal interactions
Focus on defining (a set of) "normal" interaction traces that lead to the achievement of a service goal. Do not strive for complete behaviour coverage, and prevent including interactions that do not lead to achievement of the service goal, i.e. abnormal or erroneous sequences.

Method rule: Define terminal state invariants in connector goal interactions
Insert terminal state invariants after the last event on each lifeline.

Method rule: Omit event occurrences after goal achievement
Event occurrences that can occur after goal achievement should not be included in connector goal interactions.[6]

Method rule: Define collaboration goal sequences in interaction overviews
For each composite service structure *<service>*, define the relationships between the goals of the constituent semantic connectors in an interaction overview diagram. Name the dia-

6. This is to ensure that interaction uses referenced in goal sequences have well- defined terminal state invariants.

gram *<service>_goals*. The interactions referenced are the connector goal interactions defined for each semantic connector. Do not show exceptions that do not lead to goal achievement of the composite service structure.

- If tool support for goal sequence diagrams is available, display the role binding of the semantic connector roles to the service roles of the composite service structure as graphics in the body of the interaction use (see e.g. Figure 5.12);
- Without such tool support, refer to the interaction uses by name (see e.g. Figure 5.11).
- Connect elements using arcs and/or pairs of fork and join nodes, as appropriate, ensuring that branching and joining of branches is properly nested.

5.2 Role goal sequences

According to the requirements in section 2.6.2, we need to express the goal relationships between roles. Here we shall discuss how collaboration goal sequences can be used to derive what we call *role goal sequences*.

Definition: Role goal sequence
A role goal sequence describes the inter-relationships between preceding and succeeding role goals for an interface role.

Role goal sequences express goal dependencies at the interface role level. While collaboration goal sequences combine both horizontal and vertical dependencies, role goal sequences define only vertical goal relationships between interface roles. In particular, role goal sequences hence express goal relationships for semantic interfaces.

Role goal sequences describe properties that must be respected by actors playing the interface roles, in that role goal sequences constrain the order of interface roles played by actors. However, they do not define an actor’s capability of playing interface roles simultaneously; this is the subject of *actor goal sequences* presented in section 5.3.

The observant reader may ask whether *goal sequences for service roles* are useful. We maintain that they are not. The reason is that they do not add anything new compared to collaboration goal sequences. Nor do they contribute to the resolution of role conflicts; see the discussion in section 10.2.1. Hence we conclude that service role sequences are an unnecessary step on the way from collaboration goal sequences to actor goal sequences

5.2.1 Role dependencies derived from collaboration dependencies

Deriving role goal sequences is a mechanical process that can be automated. It entails analysing the collaboration goal dependencies, and determining the sequence of goals of the interfaces roles. An example applied to the *Call* service is shown in Figure 5.13.

Figure 5.13 indicates how the goal dependencies between the interface roles of the semantic connectors (i.e. between *inviter* and *receiver* on one hand, and *invitee* and *accepter* on the other) are derived from the collaboration goal dependency between *setup* and *accept*[7].

7. The example here does not include the two collaboration uses for disconnection, to prevent cluttering up the figure.

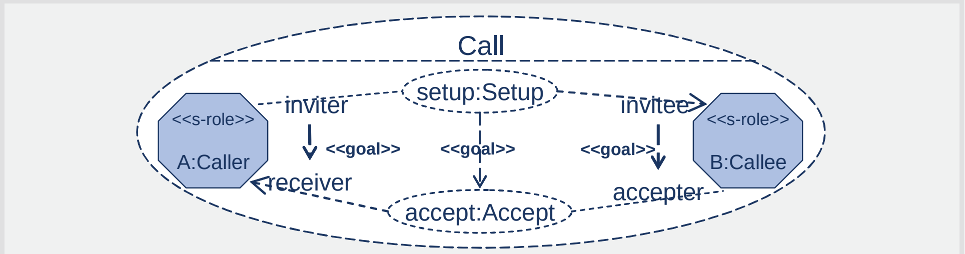


Figure 5.13 : Role dependencies derived from collaboration dependencies

Compared to the collaboration goal dependencies of Figure 5.3 on page 109, Figure 5.13 does not add anything new.

The usefulness of role goal dependencies is more apparent in cases where a number of semantic connector roles are bound to more than two service roles. An example is the *MpConf* service, see Figure 5.14.

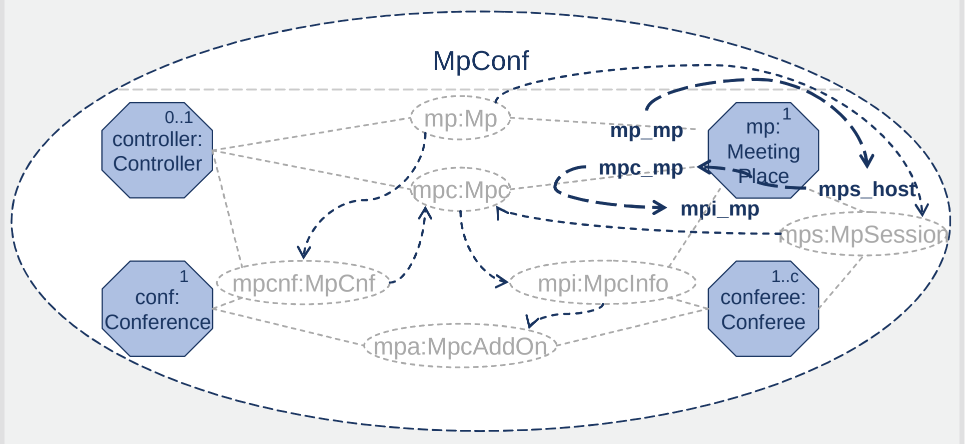


Figure 5.14 : Role goal dependencies for MpConf

In Figure 5.14 the dependencies between the interface roles bound to the service role *mp* are shown using thick dotted lines[8]. The dependencies express that the goals of the interface roles must be achieved in a certain order in the context of *MpConf*. Role goal dependencies can be derived automatically from collaboration goal dependencies. The collaboration goal dependencies are shown using thin dotted lines in Figure 5.14.

Since role goal dependencies can be derived from collaboration goal dependencies, there is no need to model them explicitly; Figure 5.14 and the diagrams in the next subsection are included for sake of argument only. Adding all the role goal dependencies to Figure 5.14 would result in a very dense picture.

8. Role goal dependencies for the three other service roles are suppressed to prevent cluttering up the figure.

5.2.2 Role goal sequences in interaction overviews

Like collaboration goal sequences, we could consider describing the goal relationships between interface roles using interaction overview diagrams. This would comprise:

1. for each interface role, defining a sequence diagram with interactions that lead to the achievement of its role goal(s); we call this a *role goal interaction*.
2. for each role of a composite service structure, defining an interaction overview diagram that references the role goal interactions as interaction uses, using the mechanisms available in interaction overview diagram to represent the role goal relationships. We call these interaction overview diagrams *role goal sequences*.

Definition: Role goal interaction
A role goal interaction is an interaction defining the goal achievement of an interface role.

Role goal interactions can be derived from the connector goal interactions of the semantic connectors, by factoring out each role and replacing signals exchanged between connected roles by signals exchanged with the environment. This is a simple mechanical process that can be supported by tools. Each connector goal interaction can be used to derive two role goal interactions, one for each semantic interface.

As with collaboration goal sequences, *role goal sequences* express a sequence of goals. They express how goals of preceding and succeeding interface roles are ordered. Role goal sequences can also be derived by tools from collaboration goal sequences. Role goal sequences can distinguish between initiating and responding roles, e.g. by colour: initiating roles in a dark colour, responding roles in a light colour, as shown in Figure 5.16 below. The distinction between initiating and responding roles is useful in connection with service discovery, as will be discussed in chapter 7.

We introduce the principles and diagrams by way of two examples.

5.2.2.1 Example 1: Call service

Figure 5.15 shows role goal interactions for the *Call* service. These can be derived from the connector goal interactions in Figure 5.6.

The role goal sequences for the *Call* service are shown in Figure 5.16.

In Figure 5.16 we have shown the sequence of role goals of the *Call* service in an interaction overview diagram that references the role goal interactions of Figure 5.15. Two sequences are shown, one for each role of the composite service. Initiating and responding roles are indicated by dark and light colouring, respectively.

5.2.2.2 Alternative roles

Interaction overview diagrams are based on trace semantics, although the token passing semantics of activities is useful when interpreting them. Our requirements to service discovery imply a certain interpretation of some of the semantics.

Decision nodes as in Figure 5.16 model a choice of behavioural flow. Our interpretation is as follows: A token arriving at a decision “enables” the roles represented by the subsequent interactions. Which path is chosen depends on what event occurrences take place.

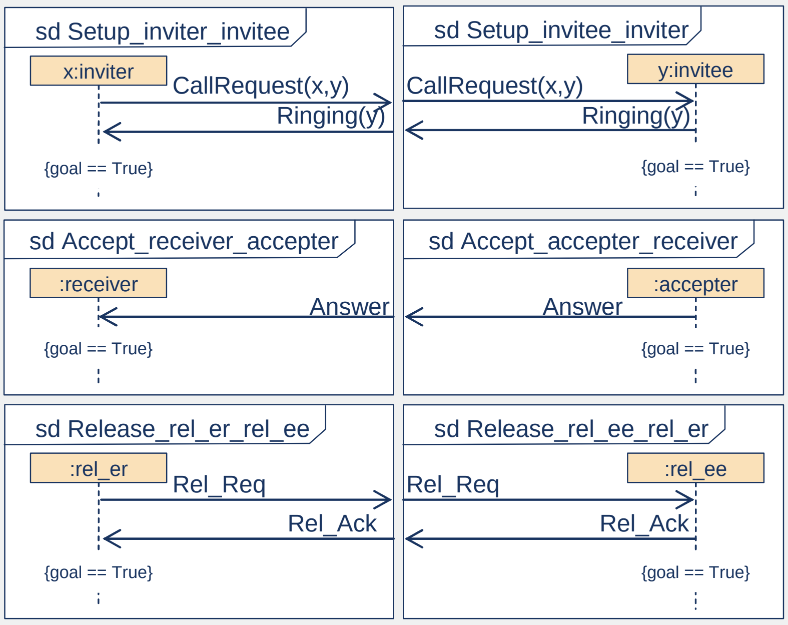


Figure 5.15 : Role goal interactions for setup, accept and release

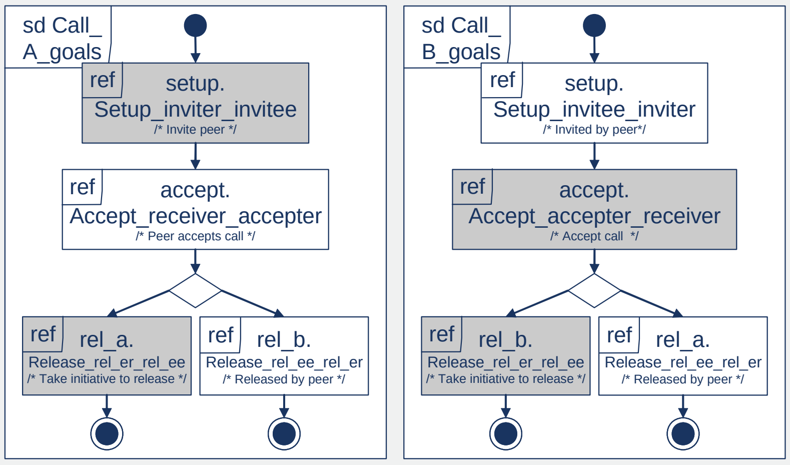

sd Call_ A_goals
ref setup. Setup_inviter_invitee /* Invite peer */
ref accept. Accept_receiver_accepter /* Peer accepts call */
ref rel_a. Release_rel_er_rel_ee /* Take initiative to release */
ref rel_b. Release_rel_ee_rel_er /* Released by peer */
sd Call_ B_goals
ref setup. Setup_invitee_inviter /* Invited by peer*/
ref accept. Accept_accepter_receiver /* Accept call */
ref rel_b. Release_rel_er_rel_ee /* Take initiative to release */
ref rel_a. Release_rel_ee_rel_er /* Released by peer */

Figure 5.16 : Role goal sequences for Call

Enabling means that role playing is possible, though not mandatory. Normally in UML, guards on the outgoing edges of decisions are evaluated to determine which path is traversed. We suggest not specifying the guards, intentionally leaving causality undefined. We return to the issue of semantics in section 5.4.1.

In Figure 5.16 the decision nodes choose between taking the initiative to release the call, or having the call be released by the peer; which one happens depends on the actors at runtime. This is an example of mixed initiatives; both roles can initiate call release simultaneously. This calls for conflict resolution in the service role, and indicates that service roles are non-trivial in such cases.

Note however that the resolution of mixed initiatives is not formally expressed in role goal sequences. One reason is that expressing concurrent flows combining fork and merge nodes is not possible in collaboration and role goal sequences; only properly nested pairs of fork and join or decision and merge nodes.[9] Combining fork and merge nodes is possible in actor goal sequences, as these are based on activity diagrams, see section 5.3.1.2.

5.2.2.3 Event goals

Event goals are an important element of our approach. They are not expressed in OCL, but are instead related to event occurrences, such as a signal being sent or consumed.[10] Event goals can be derived from the role goal interactions in a mechanical process supported by tools: The last event occurrence prior to a state invariant representing the role goal is by definition an event goal (e.g. the *MpAck* input event in the role goal interaction *Mp_controller_mp* in Figure 5.17 below).

Each role goal interaction defines at least one event goal.[11] The event goals can subsequently be used to generate progress labels used in connector validation.

When an interface role reaches its goal, the control token is passed over the outgoing edge of the role goal sequence.[12]

5.2.2.4 Example 2: MpConf service

Role goal interactions derived from the connector goal interactions of Figure 5.9 and Figure 5.10 are shown in Figure 5.17 and Figure 5.18.

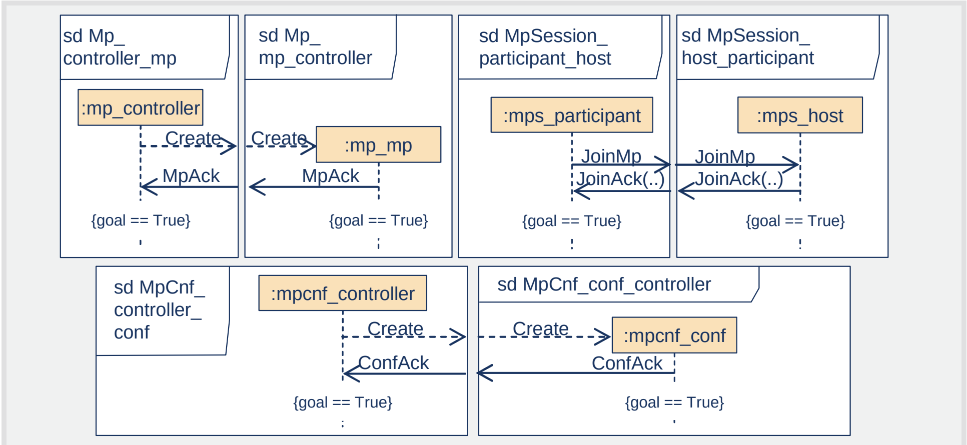


Figure 5.17 : Role goal interactions for MpConf (1 of 2)

9. *Branching and joining of branches must in Interaction Overview Diagrams be properly nested.* [UML 2.0] p. 499.
10. A time event occurrence can also be used to indicate a goal success; this can be considered to be an input event.
11. If there are several alternative event sequences that lead to the goal, the role goal interactions can define several event goals using the “alt” operator.
12. Note that the roles represented by interactions might not necessarily terminate after reaching their goals. They may still send or consume signals. But the control token can only be passed once, i.e. when the goal is achieved.

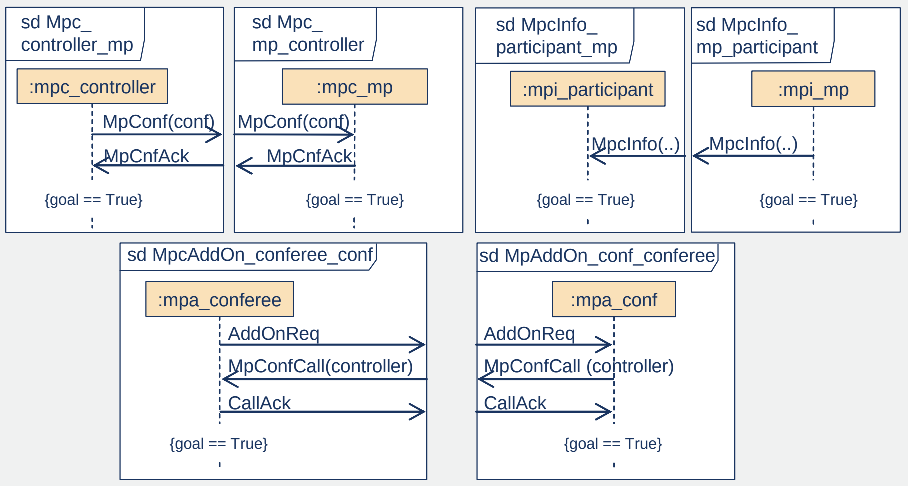


Figure 5.18 : Role goal interactions for MpConf (2 of 2)

Figure 5.17 illustrates how multi-party services result in a set of role goal interactions, the number of which depends on what interface roles exist. In the *MpConf* service, *conferee* has three interface roles, while *conf* has two interface roles.

In Figure 5.19 we have depicted the derived role goal sequences for the *MpConf* service.

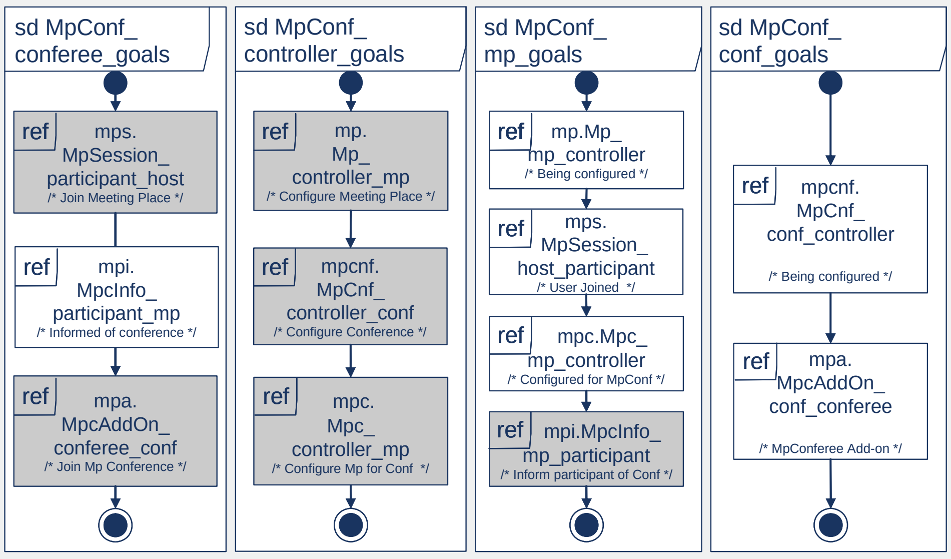

sd MpConf_ conferee_goals
ref mps. MpSession_ participant_host /* Join Meeting Place */
ref mpi. MpcInfo_ participant_mp /* Informed of conference */
ref mpa. MpcAddOn_ conferee_conf /* Join Mp Conference */
sd MpConf_ controller_goals
ref mp. Mp_ controller_mp /* Configure Meeting Place */
ref mpcnf. MpCnf_ controller_conf /* Configure Conference */
ref mpc. Mpc_ controller_mp /* Configure Mp for Conf */
sd MpConf_ mp_goals
ref mp.Mp_ mp_controller /* Being configured */
ref mps. MpSession_ host_participant /* User Joined */
ref mpc.Mpc_ mp_controller /* Configured for MpConf */
ref mpi.MpcInfo_ mp_participant /* Inform participant of Conf */
sd MpConf_ conf_goals
ref mpcnf. MpCnf_ conf_controller /* Being configured */
ref mpa. MpcAddOn_ conf_conferee /* MpConferee Add-on */

Figure 5.19 : Role goal sequences for MpConf

Note that the role goal sequences of Figure 5.19 are defined in the scope of the composite service structure *MpConf*, while the role goal interactions of Figure 5.17 and Figure 5.18 are defined in the scope of the semantic connectors (i.e. in *Mp*, *MpSession*, *MpConf*, *Mpc*, *MpcInfo* and *MpcAddOn* respectively).

5.2.3 Method guidelines

The method guidelines for role goal sequences are summarized as follows:

Method rule: Derive role goal interactions
For each semantic interface of a semantic connector *<feature>*, derive role goal interactions from the connector goal interactions by following the appropriate role lifeline, replacing signals exchanged with its connected role by signals exchanged with the environment. Name the role goal interactions *<feature>_<(from)role>_<(to)role>* to achieve unique names.

It is preferable to use a tool to derive role goal interactions rather than design them by hand.

Method rule: Derive role goal sequences
Use a tool to derive role goal sequences; do not design them by hand. For each service role *<role>* of a composite service structure *<service>*, use the corresponding collaboration goal sequence to derive a role goal sequence in the form of an interaction overview diagram. Each element of the collaboration goal sequence that refers to the service role results in an element in the role goal sequence. Name the diagrams *<service>_<role>_goals*. Compose the role goal sequence diagram from references to the derived role goal interactions. Use a dark colour for initiating roles and a light colour for responding roles. Express sequential role goal relationships by edges, alternative role goal relationships by a preceding decision node (diamond), and alternative preceding roles by a merge node (diamond). OR relationships between several alternative preceding and subsequent roles are expressed with a combined merge and decision node (diamond).

The causality of role choice should intentionally remain undefined in role goal sequences:

Method rule: Causality remains undefined in role goal sequences
Do not define conditions on decision nodes in role goal sequences.

Method rule: Do not describe parallel behaviour in role goal sequences
Definition of parallel roles by a preceding fork node are not expressed in role goal sequences. Parallel goal sequences are deferred to actor goal sequences.

5.3 Actor goal sequences

According to the requirements in section 2.6, we need to express the goal relationships between roles played by actors. An actor is typically capable of playing several roles, both alternately or simultaneously. We describe this in *actor goal sequences*.

Definition: Actor goal sequence
An actor goal sequence describes the relationships between preceding and succeeding goals of the interface roles played by an actor type.

While role goal sequences can and should be derived from collaboration goal sequences by tools, the modelling of actor goal sequences requires the involvement of designers. This is because actor goal sequences express alternative, sequence and parallelism relationships of interface roles that the actor is capable of playing. Actor goal sequences do not describe the inner structure of the actor, only its role-playing capabilities.

Actor goal sequences are a property description of an actor type, and depend on two factors:

- the role goal sequences for the service roles that the actor can play;
- the capabilities of the actor, i.e. the capacity it has to play roles simultaneously or alternately.

Note that actors play both service roles and interface roles. In actor goal sequences, as in role goal sequences, we focus on goal sequences of interface roles, and not the goal sequences of service roles. We refer to the discussion in section 10.2.1.

5.3.1 Actor goal sequences modelled by activity diagrams

A promising approach seems to be to define actor goal sequences using activity diagrams, where roles are represented by actions. This is because the token passing semantics of activity diagrams is well suited to expressing alternatives, sequences and parallelism in role playing, and cannot be expressed by interaction overviews. The semantics of actor goal sequences is detailed in section 5.4.2; for now it suffices to state that a token arriving at the input pin of an action means that playing of the corresponding role is *enabled*. That a role is enabled means that role playing is possible, but not mandatory.

To represent roles we use the standard UML graphical form for actions: round-cornered rectangles. The actions in actor role sequences use the names of the derived role goal interactions of the role; i.e. actions of actor goal sequences represent the role behaviour that leads to the achievement of the goals of the interface role they represent. Note that no action behaviour is actually defined - the actions are simply placeholders.

5.3.1.1 Example 1: Call service

An example of an actor goal sequence is shown in Figure 5.20.

Figure 5.20 defines the sequence of roles that a *PSTN_UserAgent* can play, namely the roles of the *Call* service. In Figure 5.20 the first decision node chooses between receiving a call or making a call, and the second (which is a combined merge and decision node) chooses between taking the initiative to release the call, or having the call released by the peer. Both are classical examples of mixed initiatives. How the actor performs conflict resolution, if at all, cannot be read out of the diagram.

The actor goal sequence in this case is a combination of the two role goal sequences for *Call* in Figure 5.16 on page 120. This actor is capable of alternately playing the *A* and *B* role of *Call*. In the case of *Call*, both roles use the semantic connector *Release*, which has been taken into account by the combined merge and decision node in Figure 5.20.

Note the implications of using forks and decisions in activity diagrams: the former expresses parallelism, the latter mutual exclusion. Figure 5.20 uses a decision, and therefore rules out simultaneous incoming and outgoing calls; the flows imply that the roles are mutually exclusive.

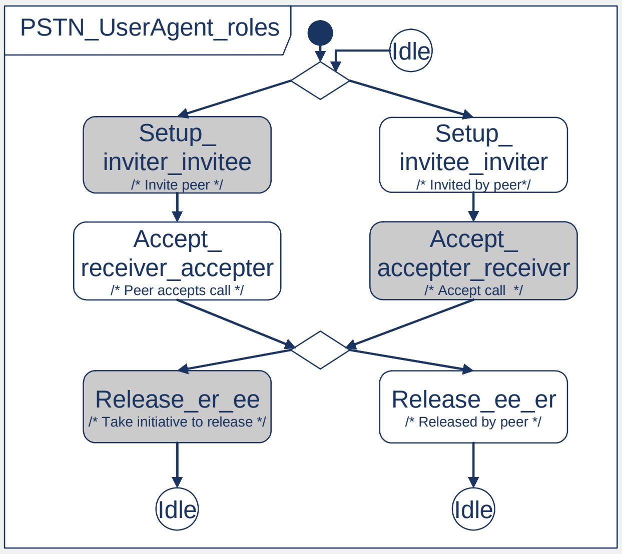


Figure 5.20 : Actor goal sequence for PSTN UserAgent supporting Call

If the decision node had been replaced by a fork node, the roles could have been played simultaneously. For PSTN, only one *Call* role can be played at a time.

5.3.1.2 Example 2: Call and MpConf services

In Figure 5.21 we have depicted a *UserAgent*'s actor goal sequence.

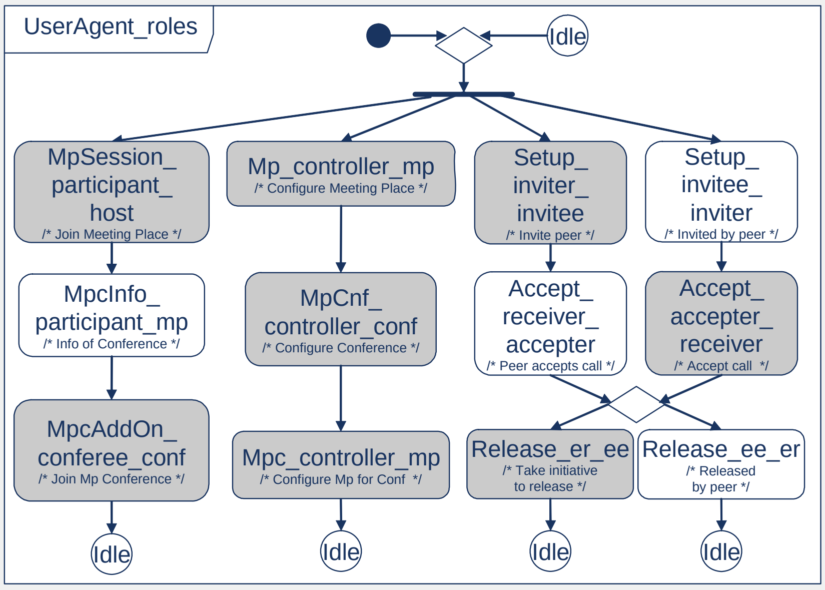


Figure 5.21 : Actor goal sequence for UserAgent supporting multiple services

As opposed to the *PSTN_UserAgent* in Figure 5.20, the *UserAgent* in Figure 5.21 can simultaneously perform incoming and outgoing calls, as well as play the *conferee* and *controller* roles of *MpConf*, the role goal sequences of which were defined in Figure 5.19. The combined merge and decision node in Figure 5.21 expresses that whether it is (simultaneously) called or is itself calling, the call can be released by itself or by its peer.

Figure 5.22 shows actor goal sequences for the actors *MeetingPlace* and *Conference*.

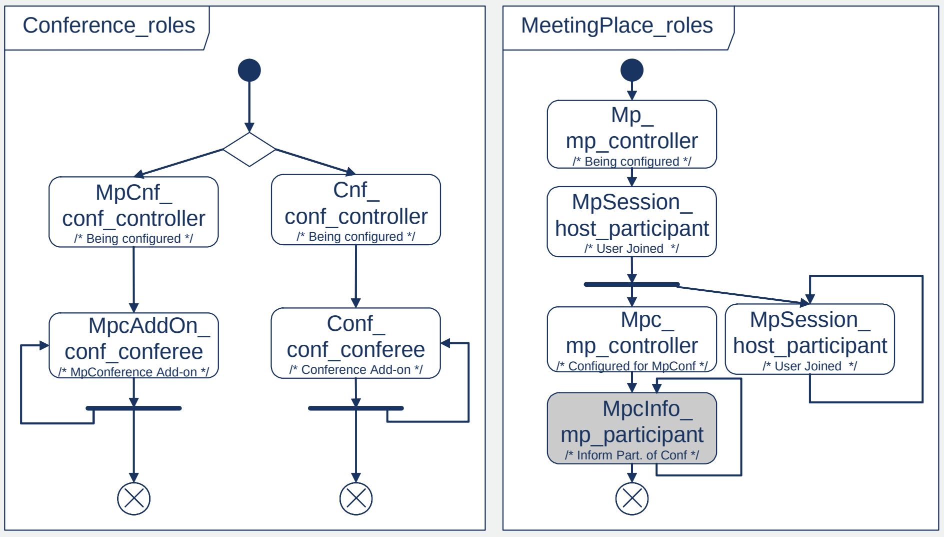


Figure 5.22 : Actor goal sequences for Conference and Meeting Place

Note the support for multiple conference call sessions and meeting place sessions are made explicit in Figure 5.22; this was not specified in the role goal sequences:

- The *MeetingPlace* actor's goal sequence has loops that model its support for multiple participants joining a meeting place (i.e. multiple sessions) and being informed of the conference;
- The *Conference* actor's goal sequence contains loops that model its support for multiple conferees joining a conference.

Note how the first *MpSession* reaching its goal enables the *MeetingPlace* actor to play the *mpc_mp* role of *Mpc*. In addition it can play *MpSession* with new participants.

Note also that the flows in Figure 5.22 end with the *flow final node*, implying that other actions (roles) are not terminated (as opposed to all activities of the actor terminating due to an *activity final node*).[13]

5.3.2 Method guidelines

A method rule for actor goal sequences is suggested as follows:

13. *A flow final destroys all tokens that arrive at it. It has no effect on other flows in the activity.* [UML 2.0] p. 362. *A token reaching an activity final node terminates the activity.* [UML 2.0] p. 320.

Method rule: Model actor goal sequences
Express actor goal sequences in activity diagrams for each actor type, named *<actor_type>_roles*. Sequences of roles are represented by actions. To achieve unique names, use *<service>_<my role>_<opposite role>* as role names. Use *fork nodes* to express parallel role playing, and *decision nodes* to express mutually exclusive paths. Do not express conditions on the outgoing edges of the decision nodes. To gather alternatives, use *merge nodes* (“OR” relationships to preceding roles)[14]. Use loops to express multiple instances of the same roles (multiple sessions). Do not use *activity final nodes* to terminate flows, but return tokens to an appropriate start node or use a *flow final node*.

5.4 Semantics

Here we discuss the semantics of goal sequences.

Goal sequences intend to express that there is some way of reaching the goals of the referenced services, and in that order. The focus of goal sequences is on the relationships between the service goals, and not on the complete specification of the event occurrences.

The scope of our approach is to enable basic liveness validation and the discovery of services. The semantics of goal sequences is therefore defined with this in mind:

- For service validation the objective is to establish that service goals can be achieved by some sequence of events; it suffices to express a sequence of event occurrences that leads to the achievement of a goal and to validate that the sequence is present in the behaviour tree of the role types and actor types;
- For service discovery the objective is to express service opportunities, where the achievement of a role goal enables other role goals to be achieved. Service discovery is needed only by actors, and thus the requirements for service discovery focus on actor goal sequences.

An approach that takes actor synthesis into account will most likely need a different semantics, since the correct handling of all events is required. We return to this is in section 5.4.3.

We express sequences of goals in collaboration goal sequences, role goal sequences and actor goal sequences. The former two exploit interaction overview diagrams with trace semantics, while actor goal sequences are based on activity diagrams with token passing semantics. In the following we discuss semantics along these two lines.

5.4.1 Semantics of collaboration goal sequences and role goal sequences

The semantics of both *<<goal>>* dependencies and collaboration goal sequence diagrams is as follows: if two collaboration uses *C1* and *C2* with respective collaboration goals *G1* and *G2* are related so that the achievement of *G1* is a precondition for the achievement of *G2*, then *C1* and *C2* can be related by a *<<goal>>* dependency or a collaboration goal

14. Do not use *join* nodes to model “AND” relationships to preceding role goals, as these are not supported by service discovery mechanisms. Use *merge* nodes instead.

sequence, where *C1* is a preceding semantic connector, and *C2* a succeeding semantic connector. This is illustrated in Figure 5.23, where semantic connectors are represented by actions in an activity diagram.

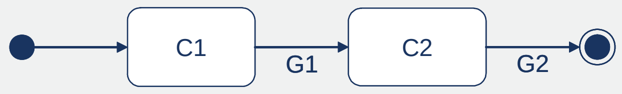


Figure 5.23 : Collaboration goal sequence

Collaboration and role goal sequences in UML rely on the trace semantics, the basis of interactions and interaction overviews. Our requirement is to express a partial order between the goals, and only indirectly express requirements on the event occurrences involved in reaching the goals. Below we discuss how the UML semantics fits our needs.

5.4.1.1 Preconditions and postconditions in UML

In UML and OCL, preconditions and postconditions are evaluated at the invocation of an operation and at its completion, and can be attached to an action, an activity, an operation, or a transition in a protocol state machine. It is not possible to attach them to interactions, which implies that we cannot use UML preconditions and postconditions in collaboration and role goal sequences.

5.4.1.2 State invariants and event traces

Connector goal interactions and role goal interactions use UML state invariants to indicate role goal achievement. See for instance Figure 5.2 on page 108.

Collaboration and role goal sequences define valid traces that achieve goals. The trace events are described in interaction uses, and the relationships between them in interaction overview diagrams.

Note that interaction overview diagrams can include parallel paths, enabled by pairs of fork and join nodes, as in Figure 5.11 on page 115. The interleaving trace semantics of UML interactions implies that event occurrences in each parallel path can be interleaved, provided the partial event occurrence orders within each interaction use are obeyed. In addition the order of goal achievement in parallel paths can be freely interleaved.

The interleaving semantics implies that the goals of a goal sequence can be achieved by different sequences of events. All event orders that obey the partial ordering described by a goal sequence are valid event orders in terms of reaching the goals specified.

Note that it is not clear from the UML2 semantics what the interpretation of state invariants is between interaction uses referenced in interaction overview diagrams. See the discussion in section 10.2.7.

5.4.1.3 Interaction constraints

We have considered using interaction constraints as guards before interaction uses in interaction overview diagrams, see Figure 5.24.

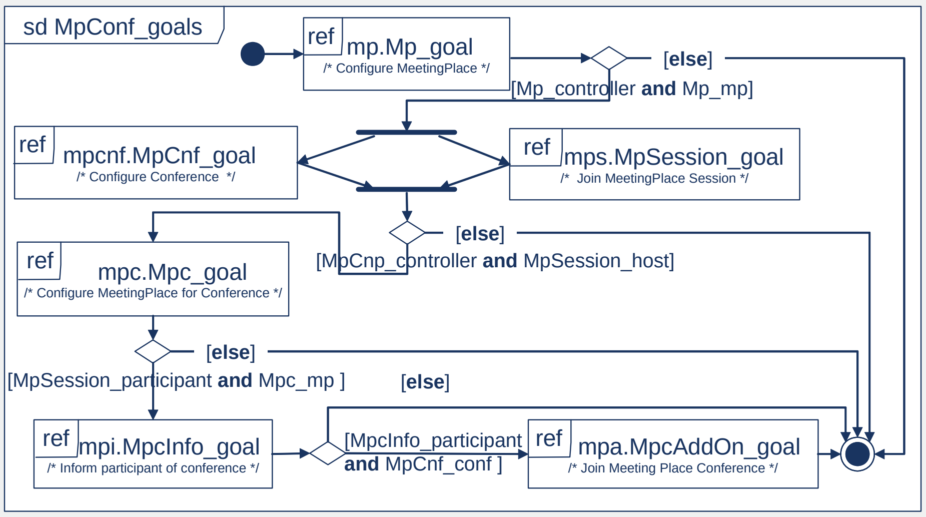

sd MpConf_goals
ref
mp.Mp_goal
/* Configure MeetingPlace */
[else]
[Mp_controller and Mp_mp]
ref
mpcnf.MpCnf_goal
/* Configure Conference */
ref
mps.MpSession_goal
/* Join MeetingPlace Session */
[else]
[MpCnp_controller and MpSession_host]
ref
mpc.Mpc_goal
/* Configure MeetingPlace for Conference */
[else]
[MpSession_participant and Mpc_mp]
[else]
ref
mpi.MpcInfo_goal
/* Inform participant of conference */
[MpcInfo_participant and MpCnf_conf]
ref
mpa.MpcAddOn_goal
/* Join Meeting Place Conference */

Figure 5.24 : Collaboration goal sequence with interaction constraints

However, the interaction constraints in Figure 5.24 are not legal in UML, as they must refer to a single lifeline.[15] The reason for this restriction is that it must be possible to evaluate them “atomically”, that is immediately and without side effects.[16]

Interaction constraints can only be used to guard an operand in a combined fragment, and do not distinguish between valid and invalid traces in the way that state invariants do.

5.4.2 Semantics of actor goal sequences

The activity diagrams in UML, which we use to capture actor goal sequences, are based on token passing semantics. Token passing is attractive to our approach, as it fits in well with our desire to express and determine role-playing capabilities of actors. Here we define what token passing semantics means in terms of role playing.

5.4.2.1 Tokens represent service opportunities

When an action representing a role has a token on (all) its incoming control edges, we interpret this as meaning that <u>the role may be played</u> - we say that the service role is *enabled*.

Definition: Enabled role
A role is enabled for an actor if there exists a token on all incoming edges of the action representing the role in the actor goal sequence.

Whether an action executes (i.e. a role is played) or not depends on what event occurrences take place, e.g. what initiatives the actor or its environment take. The token on the

15. *The dynamic variables that take part in the [InteractionConstraint] must be owned by the ConnectableElement corresponding to the covered Lifeline* [UML 2.0] p. 470.
16. *Evaluating the value specification for a constraint must not have side effects* (ibid. p. 55).

incoming control edges means that role playing is possible, not mandatory. In chapter 7 we shall see how this is interpreted in terms of discovering service opportunities.

In other words the role represented by the action will first start executing when certain event occurrences happen in the actor or in the environment of the actor; which events these are is not detailed. The behaviour of UML actions can be defined in terms of communication actions, but we suggest that the actions are treated only as placeholders for the role goal interactions bearing the same name as the actions.

In our approach, the token remains in the incoming edge until the action starts. Once the action starts, the token is consumed, and the service opportunity ceases to exist, unless there are additional tokens on the incoming edge, see section 5.4.2.5.

5.4.2.2 Decision nodes represent multiple service opportunities

One deviation from the standard regards the interpretation of decision nodes. A decision node is a control node that chooses between outgoing flows, so that each token arriving at a decision node can traverse only one outgoing edge. While we do not change the UML semantics, we interpret the situation such that when a token arrives at a decision node, all the roles of the outgoing edges are enabled, meaning that a service opportunity exists on each, though only one can be chosen by subsequent events. See Figure 5.25.

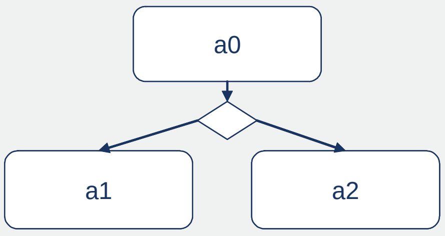


Figure 5.25 : Decision nodes and service opportunities

A token traversing an edge means that this particular path has been chosen. In UML this choice is commonly modelled by guards; in our approach we omit guards. Semantically this can be considered to represent ANY edge being chosen.[17] If we had used guards, they would pose no semantic problems as long as they only use attributes or events visible to the actor. When an edge has been chosen, there is no longer a token in the decision node, implying that there no longer exist any service opportunities on the other edges.

Following this approach, the interpretation of Figure 5.25 is as follows: after *a0* outputs a token by reaching its goal, a service opportunity arises for both *a1* and *a2*. The token remains in the decision node until some event determines that a1 or *a2* is chosen. After the choice is taken, the service opportunity of each of *a1* and *a2* ceases.

This interpretation of decision nodes means that decision and fork nodes work identically in terms of enabling all the service opportunities represented by the outgoing edges. The difference lies in what happens when the roles are played:

- with decision nodes, the service opportunities cease to exist, thus resulting in mutual

17. This resembles the ANY clause used to model indeterminism in SDL.

exclusion of the alternative roles;

- with fork nodes, each path lives its own life, and roles can start and either reach their goals or fail, independently of the other roles.

5.4.2.3 Token passing models goal achievement, not action (role) completion

We impose the interpretation that actions pass the token when the role goal is achieved, and not necessarily when the action (role) terminates. In other words, in as far as the graphs are concerned, the action terminates when the token is passed on to the outgoing edges. It may be that a role goes on being played after the goal is reached, i.e. that it consumes and sends signals.

This interpretation is due to the needs of service discovery in relation to goal achievement: at certain points in a role behaviour, some useful progress is made, and that progress enables subsequent roles. Whether more events take place in a role or not after a goal has been reached is not important to the succeeding roles.

However, such events may be valid for the service. But rather than modelling “post-goal” behaviour explicitly, with all the complexity and diagram clutter involved, we simply assume that services and roles can go on playing after their goal reaching has been conveyed in the form of a token being passed.

Typically such additional events can be related to output or consumption of signals such as the *MpInfo* and *JoinInfo* signals in Figure 3.11 on page 52, or it can be events related to the termination of the connection, status events, error events, and the like. However, recall the method rule "Omit event occurrences after goal achievement" on page 116, which enforces that such post-goal events should not be included in connector goal interactions, even though they may be part of service.

5.4.2.4 Persistence in the event of failure

An implied but not expressed aspect of goal sequences is that the actions may not reach their goals, and may “prematurely terminate”. UML2 declares a semantic variation point for CompleteActivities.[18]

In goal sequence diagrams we do not explicitly show the control flow when a role fails to reach its goal. It is possible to describe goal failure in activity diagrams, but it clutters up the diagrams.

Instead we suggest that goal failure causes the token to be consumed, and not passed on to any succeeding actions. How the activity representing the actor handles such situations lies outside the scope of our work; it could be that the actor restarts the flow from the beginning, or that it rolls back to the situation immediately before the failing action commences; this would amount to inserting a token into the control node preceding the failing action. As long as it is possible to track what actor roles are enabled, the service discovery mechanisms we suggest will work.

18. See [UML 2.0] p. 303.

5.4.2.5 Token queueing

In actor goal sequence diagrams, loops can cause a token to be “returned”. If they return to the same activity, this expresses that the action can be duplicated or repeated. An example is presented in Figure 5.21 on page 125.

A loop can also cause a token to be multiplied anew by a fork node, causing more than one token to wait at the input of an action (role) on one of the parallel paths caused by the fork. This is allowed according to the token semantics, and in our approach implies that an active role “is still enabled” (as it already was, due to the presence of a token on its input).

5.4.3 Formalized semantics for behavioural composition

The semantics we have chosen are aimed at fulfilling the needs of liveness validation and service discovery. If goal sequences are to be used to express complete behaviour and be used for composition of service role and actor behaviour supported by tools, we presume that a stricter semantics is needed. In particular it must clarify issues such as:

- Given a goal dependency defined in a collaboration goal sequence, does failing to reach the goal of a preceding semantic connector mean that the succeeding semantic connector cannot reach its goals?
- Can the succeeding semantic connector exist prior to the achievement of the goal of the preceding semantic connector? I.e. can it have any interactions prior to the goals of the preceding semantic connector being reached?
- Can the preceding semantic connector continue to exist after goal achievement? I.e. are additional interactions possible in a preceding semantic connector after its goal is reached?

The formalisation of these issues falls outside the scope of our work, and we refer to the discussion in section 8.2.2.1.

6 Service validation

We discussed the validation of basic safety properties in chapter 4. In this chapter we address the validation of basic liveness properties, as well as validating compatibility of collaborations and their constituent parts.

Compatibility in the binding of roles to classifiers in collaboration uses is a semantic variation point in UML; this chapter presents our definition of role-binding compatibility.

The structure of the sections is as follows:

- the validation of basic liveness properties of interfaces is presented in section 6.1;
- validating compliancy with bound semantic interfaces is presented in section 6.2;
- validating state-like goals is discussed in section 6.3;
- consistency between state diagrams and goal sequences is discussed in section 6.4;
- runtime connector validation is presented in section 6.5;
- section 6.6 summarizes service validation techniques and the validation method.

6.1 Progress checking: validating basic liveness properties

Below we present a method for validating liveness when goals are expressed by progress labels. It entails checking the interface behaviour of actors and service roles, i.e. their actual behaviour expressed by p-roles, and/or their specified behaviour expressed by semantic interfaces that are bound to them.

The validation of basic liveness, or *progress checking* as we call it, can be used to check a pair of connected roles. Progress checking always follows safety checking, meaning that progress checking will only be performed if both the checked roles are found to be safe.

6.1.1 Validation of interface roles

Given a connector between two interface roles, the first step is to check whether the roles are live, i.e. whether they contain progress labels:

```
Boolean live(<interface_role>)
```

The predicate *live()* checks whether progress labels are present in an interface role. If *False* is returned then no progress labels were found, and the validation terminates.

If both interface roles of the connector are live, the next step is to check whether progress can be achieved when the roles collaborate. Two algorithms for evaluating liveness based on progress labels are defined in section 6.1.3 below. The algorithms construct the truncated role of the pair of interface roles and search for the presence of progress labels in the truncated role. The connector is said to exhibit the sum of the progress labels found.

If the roles are dual and have consistently defined progress labels then they constitute a pair of semantic interfaces of a semantic connector.

6.1.2 Validation of service roles

An issue central to our approach is to validate whether service roles and actors are compliant with their specified interfaces, i.e. the semantic interfaces bound to them. According to UML2:

> *A classifier bound to a role must be compatible with the type of the role, if any. It must also obey any constraints on the role.* [UML2 Ref] p. 232.
>
> *It is a semantic variation when client and supplier elements in role bindings are compatible"* [UML 2.0] p. 167.

Validating role bindings means one must define what is meant by compatibility between semantic interfaces (the supplier elements of the collaboration uses) and the service roles or actors (the client elements), or *compliancy* as we call it.

To define compliancy we use the notion of live subtyping introduced in chapter 4.

Definition: Compliancy with a semantic interface
A service role or actor is compliant with a semantic interface if its p-role projected over the connection represented by the semantic connector is a live subtype of the semantic interface.

Compliancy checking is discussed in section 6.2 below.

6.1.2.1 Progress labels and projections

Note that the projection of event goals from service roles to p-roles depends on the visibility of the event goals on the connection over which they are projected. Progress labels survive role projection, meaning that p-roles contain the progress labels that the service role has on the connection over which they are projected.

A service role that has progress labels will project at least one live interface role; however, there may be connections towards which the service role does not have any progress labels, thus resulting in the projection of a basic interface role over that connection. A service role without any progress labels will only project basic interface roles.

The algorithms do not support the projection of goal assertions. However, since event goals can be derived from goal assertions, see section 4.2.5.2, projecting them can be considered superfluous.

6.1.3 Algorithms and tools for validation of progress labels

The liveness validation technique is based on the checking of the presence of progress labels in truncated interface roles. This can be used for a pair of semantic interfaces, or for a pair of connected service roles, the latter approach involving projection to a pair of p-roles and subsequent validation of their basic liveness properties.

A validation tool supporting the latter approach was designed and implemented in a prototype version by [Alsnes 2004].

The following is an account of the joint work done by [Alsnes 2004] and ourselves. First we give an overview of the implementation, then discuss some of the issues that arose, before we provide the pseudocode of the algorithms and the data structure.

6.1.3.1 Implementation

Figure 6.1 gives an overview of the implementation of the safety checking algorithms[1] of [Floch 2003] and our progress validation algorithms.

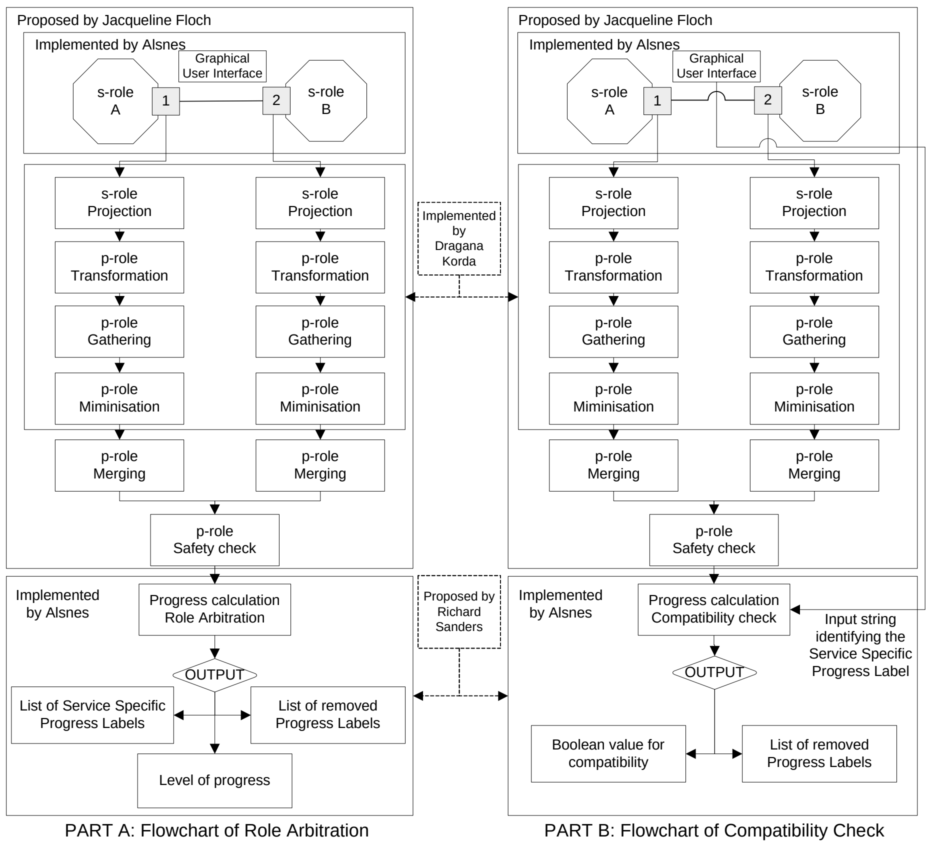

Proposed by Jacqueline Floch
Implemented by Alsnes
Graphical User Interface
s-role A
s-role B
1
2
s-role Projection
p-role Transformation
p-role Gathering
p-role Miminisation
p-role Merging
p-role Safety check
Implemented by Dragana Korda
Implemented by Alsnes
Progress calculation Role Arbitration
OUTPUT
List of Service Specific Progress Labels
List of removed Progress Labels
Level of progress
Proposed by Richard Sanders
Progress calculation Compatibility check
Input string identifying the Service Specific Progress Label
Boolean value for compatibility
PART A: Flowchart of Role Arbitration
PART B: Flowchart of Compatibility Check

Figure 6.1 : Flowchart of validation algorithms

1. The safety checking algorithms of [Floch 2003] were partially implemented by [Korda 2004].

Two progress calculation algorithms were implemented by [Alsnes 2004]:

i. *Role arbitration*: Calculates graded progress labels, and is depicted on the left side of Figure 6.1. It is denoted *Role arbitration* in Figure 6.1, since it is meant to be used in a future automated role arbitration as an element of role request pattern of [ServiceFrame 2002], see section 2.4.5.

ii. *Goal compatibility check*: Calculates service-specific progress labels, and is depicted in the flow to the right in Figure 6.1 (part B).

In both flows safety checking is performed prior to progress checking.

6.1.3.2 The loss of progress labels inserted in service roles

The safety checking algorithms of [Floch 2003] were taken as a starting point, and augmented to incorporate the progress calculation schemes outlined above. A general requirement was that the safety checking algorithms should not be affected.

The algorithms of [Floch 2003] posed a number of challenges concerning the projection of progress labels from service roles to p-roles, since the algorithms remove states and events to reduce the p-role description in order to validate the safety properties:

- *Service role to p-role projection* removes events that are hidden; these events may contain progress labels that should be associated with the service validated;
- *Gathering* can remove transitions; progress labels may be present in the removed transitions but not in the gathered transition;
- *Minimisation* of p-roles removes equivalent states and their transitions, and may thus remove progress labels;
- *Merging* combines identical paths, thus removing states and possibly labels.

Several alternative approaches to the handling of removed labels were investigated:

- Not moving labels, but simply providing a list of removed labels which could be analysed by the designer;
- Moving labels to the retained transitions. The tool could assume that the removed progress labels are meant to be present, and could issue warnings and provide a list of moved and removed labels. Merging of labels becomes an issue in this case, since there might already be a label on the transition that the label is moved to.

The former approach was chosen, and was deemed satisfactory provided the tool is used interactively and not as an automated tool without a manual feedback loop.

Transitions involving progress labels are only removed by merging if they have equivalent transitions elsewhere in the p-role; if the p-role doesn’t contain a removed label, or if it contains a different one, then the designer needs to rethink the placement (and/or the value) of labels. Rethinking is well supported by an iterative use of the tool. See the method rule "Attach progress labels to unique events" below.

6.1.3.3 Challenges of the set-based notation

The set-based notation[2] used to define transition charts in [Floch 2003] posed a challenge due to the fact that it stores event *types* and not event *occurrences* in its data structure. The loss of progress labels due to simplification was thought to be a problem. Several solutions were attempted that would ensure that all occurrences of an event that has a progress label would be retained:

- One solution was to extend the transition relation *T(s,e)*[3] not only to contain the set of target states for the state *s* triggered by *e*, but also to contain the progress labels. However, this would require a redesign of the safety checking algorithms;
- An alternative solution was to augment the set-based notation with a new relation *P(s,e)* that contains progress labels for event *e* in state *s*. Adding such a relation would not affect the safety checking algorithms. After running the algorithms progress labels could be moved and/or removed freely according to a policy chosen by the progress calculation algorithm.

Instead of adopting any of these, the implemented algorithms demand that progress labels are consistently attached to events. In other words, the set-based notation cannot be used for progress calculation unless all (or none) transitions for a certain event have the same progress[4].

This is considered to be a sound principle. If progress labels are not consistently attached to events, the result is what we call *progress ambiguity.*

Definition: Progress ambiguity
Progress ambiguity occurs in an interface role when an event does not have consistent progress labelling for all occurrences of the event.

Progress ambiguity must be removed by redesign. The following rule enforces that:

Method rule: Attach progress labels to unique events
Progress labels should be attached to only transitions that consistently designate progress.

This rule does not place unreasonable restrictions on service role design. Typically the events that designate progress are unique, such as output or consumption of service-specific signals while more “generic” signal names like *cancel*, *error* or *NAK* typically do not constitute progress. Requiring unique events is not a problem in such cases.

If “generic” signal names such as *ACK* designate progress they will only pose a problem if they occur on the same connector due to the nature of projection. If this were the case it would be reasonable to state that the *ACK* is an under-specified event liable to create confusion to designers and errors in an implementation, and should thus be prevented.

A case that can not easily be handled is when a certain number of repetitions of an event designate progress, e.g. the n^{th} occurrence of *ACK*.

2. For a simplified version of the set-based notation, see section 7.5.1.2.
3. T(s,e) is the transition relationship, and returns “next state” for state s and event e.
4. For instance, if an acknowledgement signal marks progress in one place, it must mark the same progress in all other places where it occurs in the semantic interface.

6.1.3.4 Representation of progress labels in models

In the examples progress labels are represented by notes or comments connected to events. An alternative is to create stereotypes of the events that mark progress, and to include the progress labels in the definition of the stereotype.

The selection of preferred alternative is a tool issue. Stereotypes were used in the prototype tools implemented by [Alsnes 2004].

The algorithms do not search for progress labels attached to states. As was discussed in section 4.2.5.2, goal states marked by goal assertions or progress labels can be used by a tool to derive progress labels on the signal events leading to the goal state. However, no such tool support has been made. Only minor extensions of the algorithms and metamodel are required to support progress labels on states.

6.1.3.5 Pseudocode for algorithms for validation of liveness

The pseudocode for the progress checking algorithms is included below. Both algorithms compare two roles; these are denoted "1" and "2", respectively. Variables, sets and relationships marked with superscript "1" refer to *role 1*, while those marked with superscript "2" refer to the connected role, called *role 2*.

The algorithms construct the truncated role of a pair of interface roles, and search for the presence of progress labels. Role arbitration checks for both kinds of progress labels; goal compatibility checks only for the presence of a given service-specific progress label.

If a set of alternative roles is to be compared, e.g. for role arbitration, the algorithms must be called several times with role 1, but with a different connected role 2, and the results compared. This is not shown in Figure 6.1, being outside of the scope of the algorithms.

Some of the relationships are reused from the set-based notation of [Floch 2003]:

- *T(s,e)* is the transition relationship, returning "next state" for state s and event e;
- *Enable(s)* returns the set of events that trigger transitions from state s;
- S_0 is a finite set of initial states;
- S_E is a finite set of exit points;
- $\overline{e}$ is the complementary of event e. The complementary of an input event is an output event, and vice versa.

The progress calculation algorithms define an additional relationship Target(), which maps from a tuple {state, event} in role 1 to a state in role 2. The relationship is initially empty, and is assigned values during a run.

The algorithms start by collecting all states that are successor states of the initial state of role 2, and use this collection as a starting point when comparing with role 1. They traverse through role 1 and check with what events role 1 is able to interact with role 2, thus building the truncated role.

The algorithms were suggested by us, and initially designed and implemented by [Alsnes 2004]. The pseudocode includes extensions suggested by ourselves which search for progress labels in both roles, not just in role 1. The first algorithm has also been extended to return a string containing the names of all service-specific progress labels found.

The algorithms do not validate whether progress labels are consistently present in the two roles. Rather, the union of progress labels in the pair of roles is evaluated.

6.1.3.5.1 Role arbitration

The algorithm builds the truncated role of two roles and searches for progress labels.

For each event of the truncated role, any progress labels present in role 1 and role 2 are retrieved. The algorithm works depth first, recording visited states as it progresses.

Finally the progress level is calculated by adding up the progress labels found, and service specific progress labels are concatenated.

Algorithm 6.1: Role arbitration

```
main ()
{
/* Assumption: the roles are found to be safe by the safety checking algorithms */
    $C^1$ = {};          /* Set of checked states, initialise to empty */
    $N^2$ = {};          /* Set of the next states of the initial states in connected role, initially empty */
    nextIter = {};   /* Set of the next states of s in each iteration, initially empty */
    L = {};              /* Set of found progress labels, initially empty */
    int level = 0;    /* Total level of progress, initially zero*/
    string goals = ““; /* String of service-specific progress labels, initially empty */

for each $s_0^1$ in $S_0^1$ /* $S_0^1$ is the set of initial states in role “1” */
    if $s_0^1$ is not in $C^1$
        progressCheck($s_0^1$);

for each ProgressLabel in L
    level = level + ProgressLabel.progressLevel; /* progressLevel is the numerical value of the label */
    goals = goals + ProgressLabel.name; /* goals is a text string containing all service-specific labels */
return (level, goals);
}

progressCheck($s^1$)
{   add $s^1$ to $C^1$;
    for each $e^1$ in $Enable^1(s^1)$ /* Enable(s) is the set of enabled events of state s */
        if $s^1$ is an Initial state
            initTarget = $T^1(s^1,e^1)$ /* The state transition relation T(s,e) gives the successor state */
            if first iteration          /* Initialise $N^2$ */
                for each $s_0^2$ in $S_0^2$ /* $S_0^2$ is the set of all initial states in connected role */
                    for each $e^2$ in $Enable^2(s_0^2)$
                        add $T^2(s_0^2, e^2)$ to $N^2$;
            for each $x^1$ in $Enable^1$(initTarget)
                for each $n^2$ in $N^2$
                    for each $e^2$ in $Enable^2(n^2)$
                        if $x^1$ equals $\overline{e}^2$ /* $\overline{e}$ is the complementary event of e */
                            set Target($T^1$(initTarget, $x^1$)) to $T^2(n^2,e^2)$ /* Build target relationship */
                            add $T^1$(initTarget, $x^1$) to nextIter
                            if $x^1$ has ProgressLabel add(ProgressLabel) to L
                            if $e^2$ has ProgressLabel add(ProgressLabel) to L
        else /* $s^1$ is not an initial state */
            for each $s^2$ in Target(s)
                for each $e^2$ in $Enable^2(s^2)$
                    if $e^1$ equals $\overline{e}^2$
                        set Target($T^1(s^1,e^1)$) to $s^2$
                        add $T^1(s^1,e^1)$ to nextIter
                        if $e^1$ has ProgressLabel add(ProgressLabel) to L
                        if $e^2$ has ProgressLabel add(ProgressLabel) to L
        for each y in nextIter
            if y is included in $S_E^1$
                if y has ProgressLabel          /* Exit point has a progress label */
                    add(ProgressLabel) to L
            else if y not in $C^1$
                progressCheck(y)                /* Check successor state */
}
```

6.1.3.5.2 Goal compatibility check

This algorithm searches for the presence of a given service-specific progress label in a truncated role. I.e. it checks that the roles are able to reach the label, not merely that it is present in the challenged role. The algorithm is identical to the previous one except that it looks for a specific progress label name, and stops as soon as a progress label is found in the truncated role. If more than one service-specific progress label is sought, the algorithm must be called repeatedly.

Algorithm 6.2: Goal compatibility check

```
main ()
{
/* Assumption: the roles are found to be safe by the safety checking algorithms */
    $C^1$ = {};          /* Set of checked states, initialise to empty */
    $N^2$ = {};          /* Set of the next states of the initial states in connected role, initially empty */
    nextIter = {};   /* Set of the next states of s in each iteration, initially empty */
    boolean match = false; /* Enabled service-specific progress label in connected role, initially false */

for each $s_0^1$ in $S_0^1$ /* $S_0^1$ is the set of initial states in role “1” */
    if $s_0^1$ is not in $C^1$
        progressCheck($s_0^1$, string ServiceName);
return (match);
}

progressCheck($s^1$, string ServiceName)
{   add $s^1$ to $C^1$;
    for each $e^1$ in $Enable^1(s^1)$ /* Enable(s) is the set of enabled events of state s */
        if $s^1$ is an Initial state
            initTarget = $T^1(s^1,e^1)$ /* The state transition relation T(s,e) gives the successor state */
            if first iteration          /* Initialise $N^2$ */
                for each $s_0^2$ in $S_0^2$ /* $S_0^2$ is the set of all initial states in connected role */
                    for each $e^2$ in $Enable^2(s_0^2)$
                        add $T^2(s_0^2, e^2)$ to $N^2$;
            for each $x^1$ in $Enable^1$(initTarget)
                for each $n^2$ in $N^2$
                    for each $e^2$ in $Enable^2(n^2)$
                        if $x^1$ equals $\bar{e}^2$ /* $\bar{e}$ is the complementary event of e */
                            set Target($T^1$(initTarget, $x^1$)) to $T^2(n^2,e^2)$ /* Build target relationship */
                            add $T^1$(initTarget, $x^1$) to nextIter
                            if $x^1$ or $e^2$ has ProgressLabel
                                if ProgressLabel.name equals ServiceName
                                    match = true
                                    return;
        else /* Not an initial state */
            for each $s^2$ in Target(s)
                for each $e^2$ in $Enable^2(s^2)$
                    if $e^1$ equals $\bar{e}^2$
                        set Target($T^1(s^1,e^1)$) to $s^2$
                        add $T^1(s^1,e^1)$ to nextIter
                        if $e^1$ or $e^2$ has ProgressLabel
                            if ProgressLabel.name equals ServiceName
                                match = true
                                return;
        for each y in nextIter
            if y is included in $S_E^1$
                if y has ProgressLabel          /* Exit point has a progress label */
                    if ProgressLabel.name equals ServiceName
                        match = true
                        return;
            else if y not in $C^1$
                progressCheck(y, ServiceName) /* Check successor state */
}
```

6.1.3.6 Metamodel for representing state machines

The implementation of [Floch 2003]’s safety checking algorithms by [Korda 2004], and the implementation of the progress calculation algorithms by [Alsnes 2004] both use a metamodel to represent UML2 or SDL state machines. The metamodel resembles to a certain degree the UML2 metamodel to define state machines, but is specifically designed to represent service roles and interface roles efficiently. The SDL concept *save* (*deferred triggers* in UML) is included, as defined by [Floch 2003].

The metamodel is shown in Figure 6.2 below.

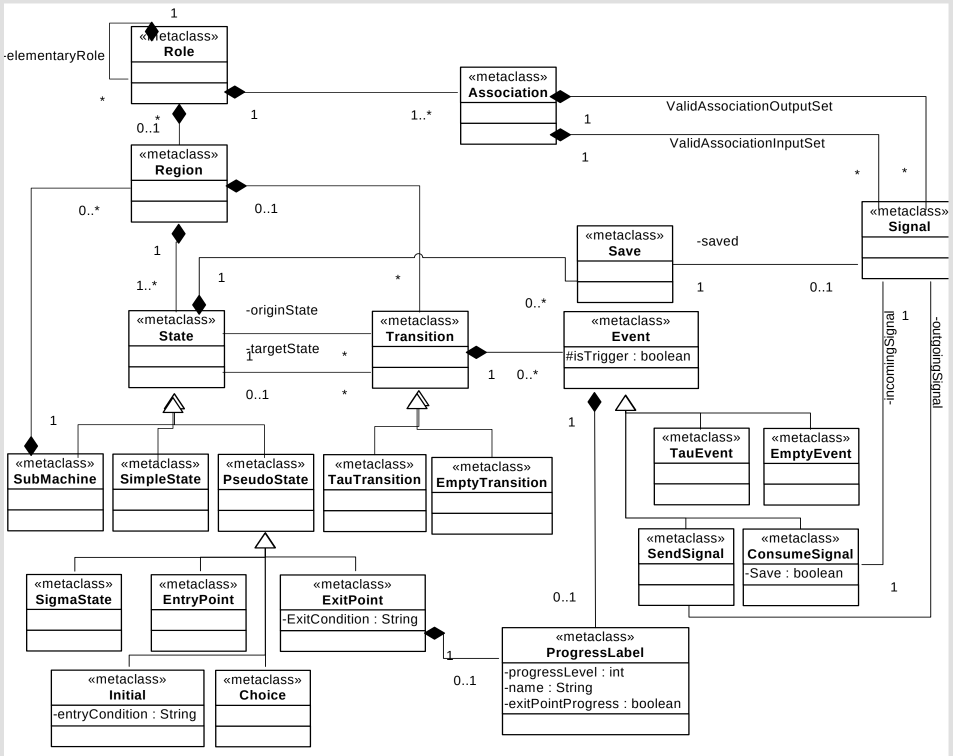


Figure 6.2 : Metamodel for validation algorithms [Alsnes 2004]

Progress labels are modelled by the class *ProgressLabel*. Instances of this class can be contained by signal events and exit points.

SimpleStates must also contain progress labels if goal states are to be supported.

6.2 Validating compliancy with bound semantic interfaces

Assume that the semantic interfaces *A* and *B* are defined by a semantic connector *C*, and that *A* and *B* are respectively bound to service roles *s-role_Ai* and *s-role_Bj* played by respective actor classes *Actor_Ai* and *Actor_Bj*, see Figure 6.3.

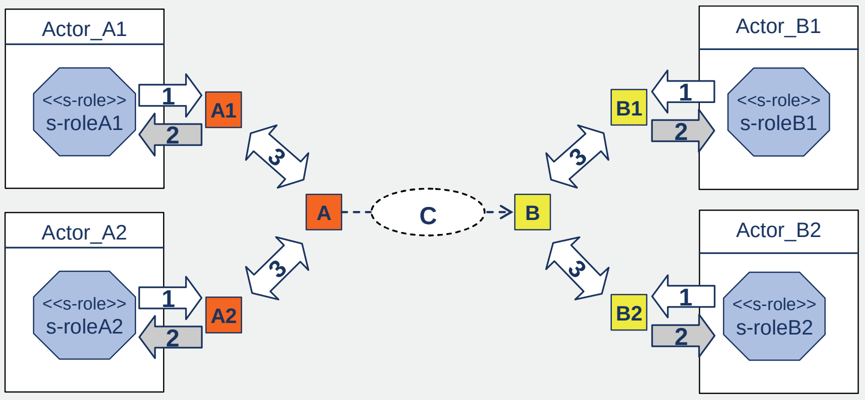


Figure 6.3 : Validating compliancy between actors and semantic interfaces

The role bindings imply that the service roles and actors are specified to be compliant with *A* and *B*.

The following steps can be followed to check the compliancy of the service roles and actors with the semantic interfaces bound to them:

For each actor class *Actor_Ai* and *Actor_Bj* playing service roles *s-role_Ai* and *s-role_Bj*:

1. Derive their p-roles *Ai* and *Bj* by projection.
2. Check whether the service roles service roles are well-formed; correct them if not.
3. Compare the p-roles with the semantic interfaces specified:
   - If (*Ai ~> A*) then *Actor_Ai* is compliant with the semantic interface *A* in *C*;

   Similarly for *B* actor candidates:

   - If (*Bj ~> B*) then *Actor_Bj* is compliant with the semantic interface *B* in *C*;

If *Ai ~> A* and *Bj ~> B*, then the actors are compatible with *A* and *B*, and the goals of *C* can be achieved when they interact. This can be checked for any pair of actors, or for an actor on its own.

Such checks need only be performed once for each actor type, and need not be repeated for each actor instance. They can be performed at design time, and be used to characterize compliancy with semantic interfaces in a library of semantic connectors. Such a library can assist during runtime connector validation between actors, see section 6.5.

6.3 Validating state-like goals

Validating state-like goals is a validation technique that supplements the validation of event goals. State-like goals are either goal expressions or goal assertions, and can be related to collaborations, roles or actors. *Validating goal expressions* means to check whether goals can be reached, i.e. that goal expressions can evaluate to *True* at some point. *Validating goal assertions* is to check whether goal assertions are not falsified anywhere.

That there are multiple ways of expressing state-like goals introduces redundancy. We consider this to be an advantage, as more validation opportunities arise.

In the following sections we first discuss validation of goal expressions, then move on to validating goal assertions, and lastly discuss the validation of actor goals.

6.3.1 Validating goal expressions

Collaboration goal and role goal expressions are defined in the context of the service structure, e.g. as in Figure 6.4.

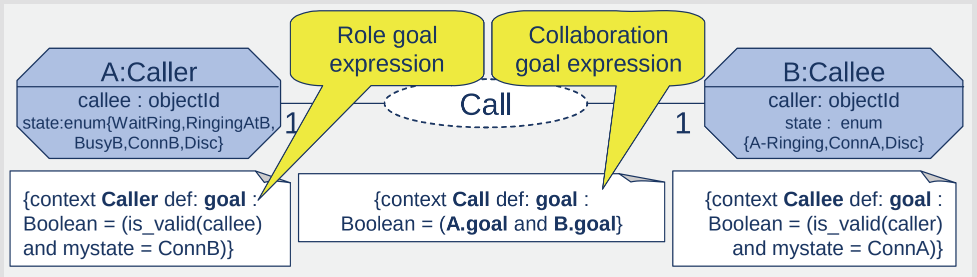

Role goal expression
Collaboration goal expression
A:Caller
callee : objectId
state:enum{WaitRing,RingingAtB, BusyB,ConnB,Disc}
1
Call
1
B:Callee
caller: objectId
state : enum {A-Ringing,ConnA,Disc}
{context Caller def: goal : Boolean = (is_valid(callee) and mystate = ConnB)}
{context Call def: goal : Boolean = (A.goal and B.goal)}
{context Callee def: goal : Boolean = (is_valid(caller) and mystate = ConnA)}

Figure 6.4 : Service goal expressions

Figure 6.4 illustrates the following:

- The *Call* service is described by a collaboration, and service roles *A* and *B* are defined to be of a given role type (i.e. *Caller* or *Callee*). Role types define attributes (such as *callee* and *caller*); here the role state is represented by the attribute *mystate*[5];
- Collaboration goal expressions and role goal expressions use attributes defined in the role types. The collaboration goal can be a conjunction of the role goals.

Given collaboration goal expressions, model checking can be used to validate whether collaboration goals are achievable:

> Given an N-party collaboration *C* with collaboration goal expression *cg* and role state machines *A*, *B*, *... N* with states *a1, a2...; b1,b2…; n1,n2…*: perform a reachability analysis, and for each global state *(ai, bj, ... nk)*:
>
> - Evaluate the collaboration goal expression *cg*;
>
> Validation stops at the first occasion at which *cg* evaluates to *True*.

This form of validation is subject to the state-space explosion problem commonly associated with model checking.

For a semantic connector where the semantic interfaces contain role goal expressions, model checking can be used to validate whether goals are achievable without making use of any collaboration goal expression:

5. States are defined by an enumerated data type; this is an alternative to defining them in state machine diagrams.

Given a 2-party collaboration *C* and role state machines *A* and *B* with states *a1, a2…; b1,b2…* and goal expressions *as* and *bs*; perform a reachability analysis, and for each global state *(ai, bj)*:

- Evaluate the role goal expressions *as* and *bs*;
- Mark the goal of *A* as satisfied if *as* evaluates to *True*;
- Mark the goal of *B* as satisfied if *bs* evaluates to *True*.

Validation stops as soon as both *A* and *B* are marked as satisfied.

This form of validation generally requires modest amounts of resources, provided the goals are achievable, since the algorithm stops as soon as a way to reach them is found.

6.3.2 Validating goal assertions in collaborations

An additional form of model checking is to validate that goal assertions on collaboration goals and role goals are not falsified in any reachable state. Collaboration state machines can contain such goal assertions; indeed, in addition to providing an overview, the specific purpose of collaboration state machines is to enable the validation opportunities that arise. Figure 6.5 shows an example of collaboration states and goal assertions.

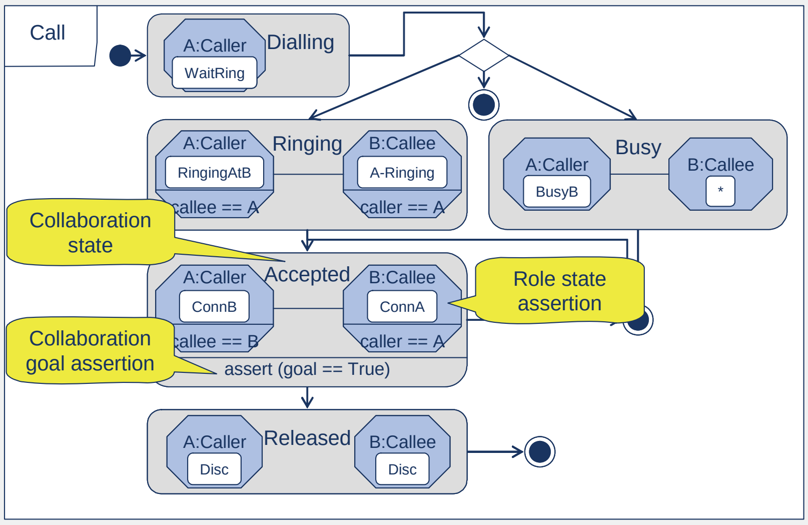


Figure 6.5 : Collaboration states and goal assertions

In Figure 6.5 the collaboration goal is asserted to be achieved in the collaboration state *Accepted*. This state also expresses assertions on states of the roles *A* and *B*.

Using model checking, goal assertions are used to validate the joint collaboration behaviour:

Given a collaboration state machine with goal assertions, and role state machines with their own goal assertions, perform a reachability analysis, and for each global state:

- For each role, evaluate their role state expressions (see section 6.3.3.1 below);

- Mark the global state with the role goal assertions that evaluate to *True*;
- Ensure that collaboration goal assertions are not falsified.

Validation stops if a collaboration goal assertion is falsified; if validation runs through all states of the collaboration without any falsification of collaboration goal assertions, we conclude that the joint behaviour is consistent with the collaboration state machine.

That the collaboration goal is *True* in at least one global state is established by validating the collaboration goal expression (as discussed in section 6.3.1 above).

Note that collaboration states have a scope that takes in all the collaborating roles, and is therefore a "horizontal" state closely related to the service provided by the collaboration.

6.3.3 Validating goal assertions in service roles

As was discussed in section 4.4.3.2, a state machine defining the behaviour of a service role can contain goal assertions.

Generally speaking role goal assertions may include internal and external aspects:

- References: What external entities are known? (E.g. *Callee* knows *Caller*);
- Attributes: What values are held by internal attributes? (E.g. *goal* is *True*);
- Timers: What timers are active? (E.g. *dial_timer* is active in state *WaitRing*);
- Collaborations: What collaborations are active and what are their states? (E.g. in role state *ConnB* the *Call* is active and should be in collaboration state *Accepted*);

The first three cover aspects internal to an actor and directly controlled by it, while the last assesses joint states of collaboration; the latter are partly outside its control, i.e. external, as they also depend on the states of the other collaborating roles.

We may thus distinguish between two sets of opportunities: internal checks and external checks.

6.3.3.1 Internal checks: validating assertions on role states

These checks aim to ensure that assertions expressed for a given state (or a set of states) of an actor will hold for all possible executions. It entails checking all event sequences leading to the given state combined with all possible values of attributes and timers.

In Figure 6.6 assertions are specified in the states. These assert that attributes have certain values in certain states, such as the *callee* attribute in the *Caller*'s state *RingingAtB*, and assert in which states role goals are achieved, i.e. in states *ConnB* and *ConnA*.

If an actor design does not violate the role goal assertions, then one has validated that the design is consistent with the specified role goals.

Using model checking, goal assertions can be used to validate the role behaviour:

Given a role state machine *A* with states *a1, a2...* and a role goal assertion *as*: Perform a reachability analysis for *A*, and for each state *(ai)*:

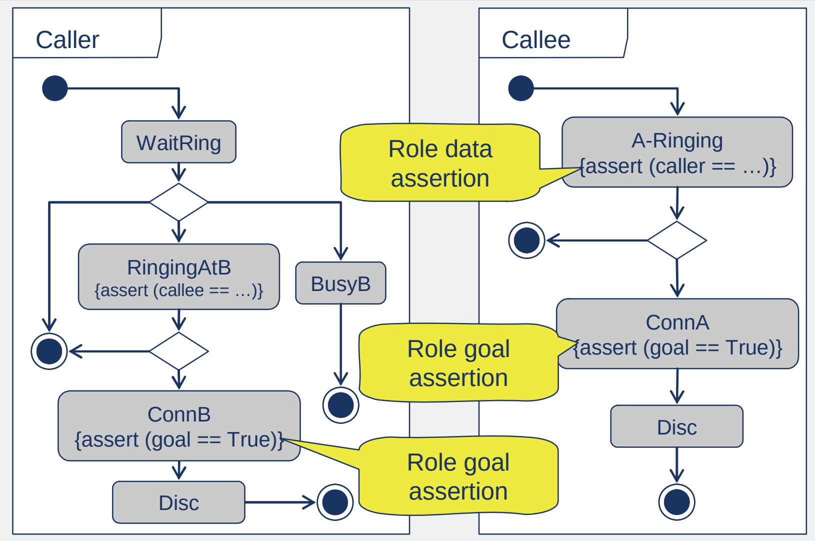


Figure 6.6 : Service role state machines with role goal assertions

- Evaluate the role goal assertion *as* (if present in state *ai*);
- Mark the goal of *A* as satisfied if *as* evaluates to *True*.

Validation stops if the role goal assertion is falsified; if validation runs through all states of *A* without any falsification, we conclude that the assertion *as* is satisfied.

In the popular model checker tool SPIN [Holzmann 2003], role goal assertions can be expressed as *assertions* in Promela.

6.3.3.2 External checks: validating assertions on collaboration states

An additional model checking opportunity is to validate collaboration state assertions expressed for a role state machine. Collaboration state assertions express properties that shall hold across actors and thus provide external information that can be used to make even more comprehensive checks on progress towards goals.

Without any assertions on collaboration states, external checks are limited to checking general safety properties, see section 4.1.3.

As was discussed in section 4.4.3.3.1, roles can be defined in a *state-oriented* fashion, where the collaboration state is asserted in each role state, see Figure 6.7.

The state symbols in Figure 6.7 indicate the appropriate role states and corresponding collaboration states, in accordance with what is defined for the *Call* service. This provides the opportunity for checking the consistency between role and collaboration behaviour.

Using model checking, roles states with collaboration state assertions can be used to validate whether the collaboration goal is achievable seen from the perspective of the role:

Perform a reachability analysis for a role; for each global state:

- Evaluate the role goal assertions (as for internal checks);

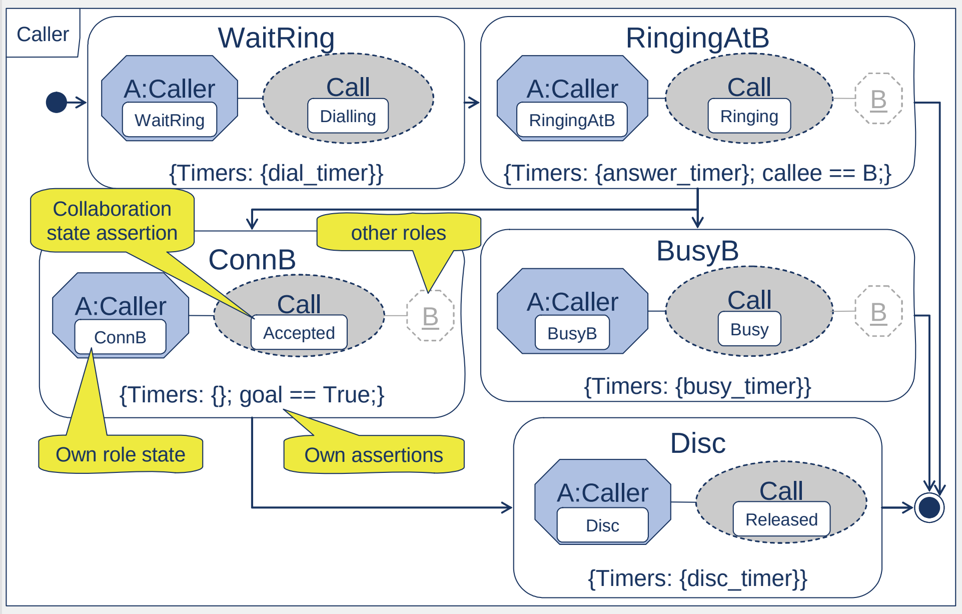


Figure 6.7 : Role states with state orientation

- Check whether the collaboration state assertions are satisfied for each role state in the global state. If not, mark the role state as inconsistent with the collaboration state.

Validation stops if a collaboration state assertion is falsified; if validation runs through all states of the role without any falsification of the collaboration assertion, we conclude that the collaboration state assertions are satisfied, implying that the role behaviour is consistent with the collaboration behaviour.

6.4 Consistency with interaction sequences

While state diagrams are used to specify the behaviour of service roles and interface roles, additional modelling elements are used to capture the cross-cutting behaviour of services, as was discussed in section 3.3. In our approach we put particular emphasis on the systematic use of interaction diagrams and goal sequences.

Sequence diagrams and goal sequences normally specify partial behaviour, as opposed to the complete behaviour defined in state diagrams. This implies that one needs to ascertain that sequence diagrams and goal sequences are consistent with the state diagrams.

Below we discuss the validation techniques that are appropriate to use in this setting.

6.4.1 Consistency with role goal interactions

A basic validation problem is to check whether role goal interactions and role state diagrams are consistent with each other, see Figure 6.8.

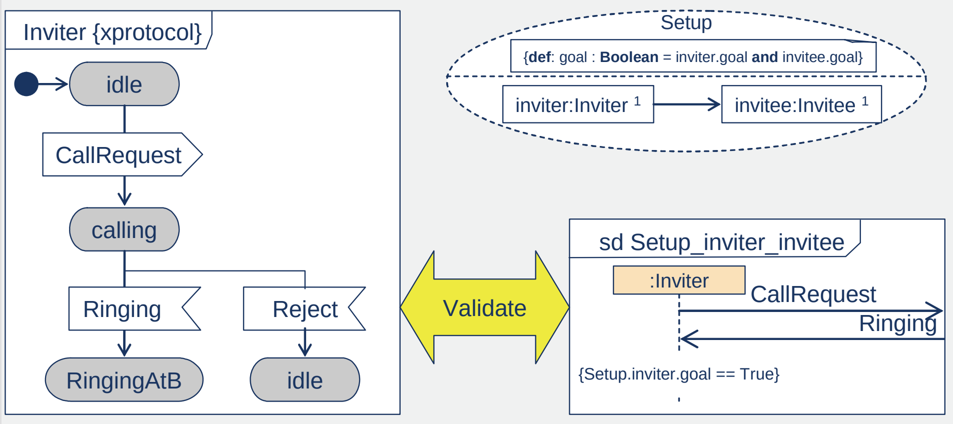


Figure 6.8 : Validating consistency with role goal interactions

Role goal interactions are sequence diagrams that describe a sequence of signals that leads to a role goal being achieved. The object of the validation is to ascertain that there are event paths in the state diagram of the interface role that correspond to the sequences of events specified in the interaction, i.e. that the role state machine can perform at least the sequences specified. Such basic validation functionality is commonplace in software engineering tools such as [Telelogic].

Note that validation of event sequences depends on what assumptions can be made on the connectors between roles, see Figure 6.9.

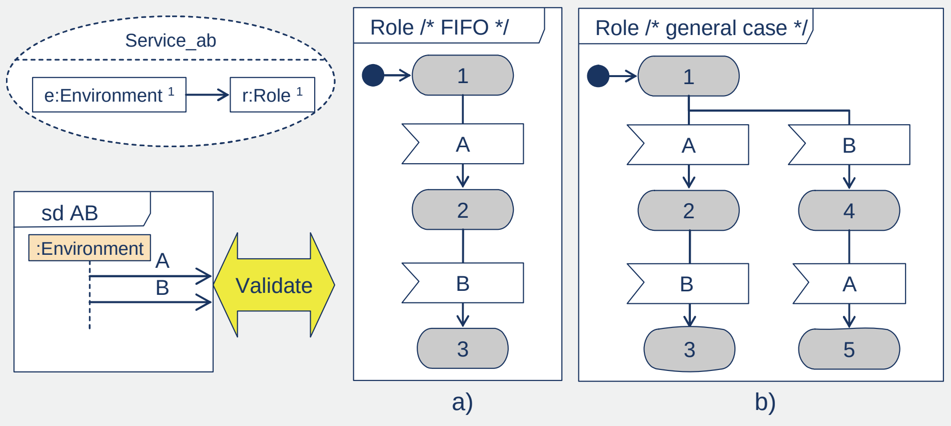


Figure 6.9 : Validating consistency with interactions: implied scenarios

Assuming FIFO properties of connectors, signals will be received in the same order as they are sent, and the role behaviour in Figure 6.9 a) is compatible with an environment specified in the interaction diagram. However, if FIFO properties cannot be assumed, the role behaviour in Figure 6.9 b) is necessary, since *B* can arrive before *A*. The latter case is called an implied scenario.

Note that the definitions of containment and obligation, see section 4.1.3, assume FIFO properties, according to SDL semantics. In SDL message overtaking does not happen on connectors, meaning that the order of signals received over a connector is the same as the order sent.

6.4.2 Consistency with collaboration goal sequences

Collaboration goal sequences are interaction overview diagrams describing sequences of events leading to collaboration goals being achieved. Validation techniques should verify that service roles and collaboration goal sequences are consistent, see Figure 6.10.

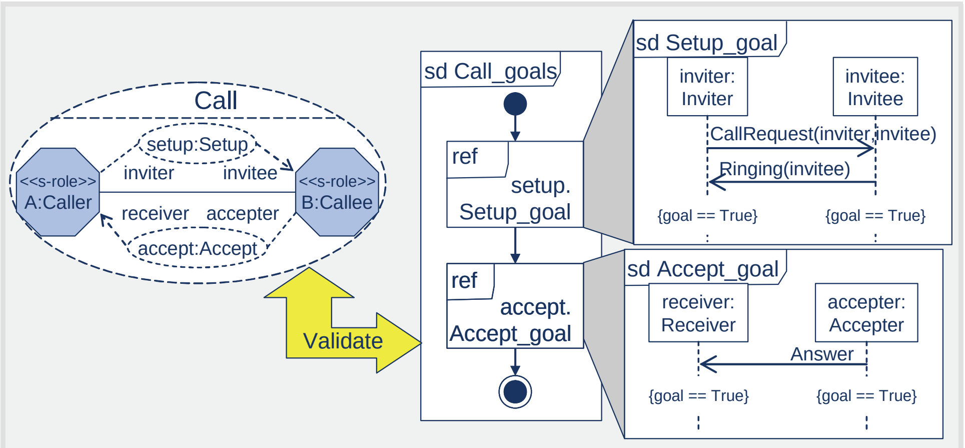


Figure 6.10 : Validating consistency with a collaboration goal sequence

Recall that collaboration goal sequences are used to derive role goal interactions; validating service role behaviour should be performed for each service role based on role goal interactions using standard validation tools as described in section 6.4.1. In the example in Figure 6.10, tools should be used check whether there is an event path in the state diagram of *Caller* that corresponds to the events on the lifelines of *inviter* followed by *receiver*.

The interaction overview diagrams that underlie goal sequence diagrams adopt week sequencing semantics. This means that message overtaking can occur, depending on the nature of the connectors. E.g. *Answer* can in some cases be received before *Ringing*.

6.4.3 Consistency with actor goal sequences

Actor goal sequences are activity diagrams that in effect describe sequences of events leading to actor goals being achieved. Validation techniques are needed to verify that actor designs and actor goal sequences are consistent, see Figure 6.11.

As for role goals, standard validation tools can be used to validate whether there is an event path in the state diagram of the actor that corresponds to the sequences of events specified in the actor goal sequences. The validation should include checking that loops of behaviour expressed in the actor goal sequences are also present in the state diagram of the actor.

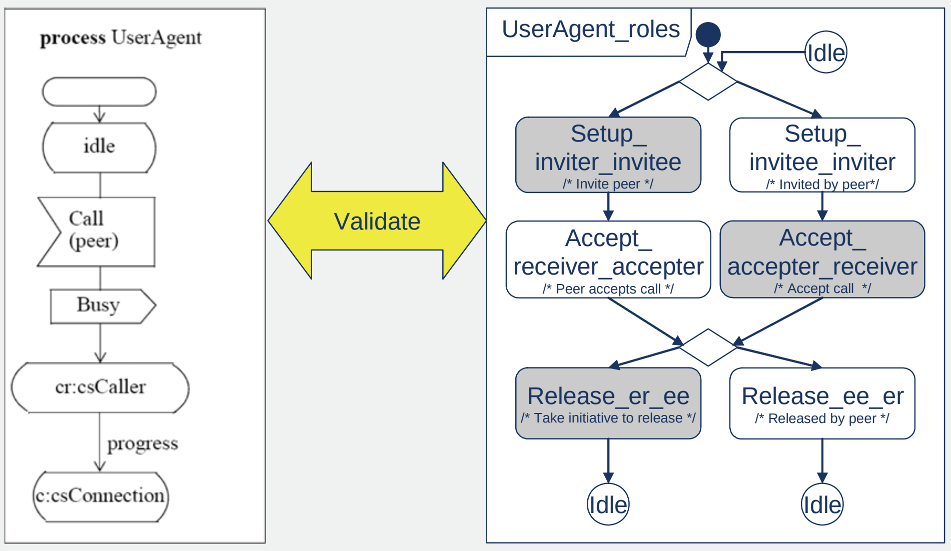


Figure 6.11 : Validating consistency with an actor goal sequence

6.5 Runtime connector validation

Model checking takes place at design time as a quality assurance measure aimed at removing flaws in systems before they are deployed. When design-time connector validation has shown that roles are designed safely and usefully, then the roles can be trusted to perform correctly in all situations where the property model is valid.

However, in a setting where actors are distributed across heterogeneous networks, and services evolve over time, situations may arise where interacting system components have been validated against different versions of semantic connectors. This means that the validation originally performed at design time is not sufficient; validation must be performed for the actual connection.

One solution to this challenge is to validate at runtime, immediately prior to interaction between actors validated against disparate property models. The need for such a capability is increased by the introduction of dynamic service discovery and role learning, which we discuss in the chapter on Service Discovery. Another solution is to calculate substitution relationships at design time, and then use this information at runtime.

The mechanism needed should validate the connections between actors. Runtime connector validation should determine whether actors that are dynamically connected can play well with each other, and reach service goals while doing so.

Runtime connector validation consists of validating the semantic interfaces of connected roles, and is an adoption of the progress checking algorithms presented in section 6.1.3. The only difference is that the algorithms are initiated by the actors or on their behalf as part of their normal operation, rather than at design time and at the behest of a service or systems designer.

Once the semantic interfaces of dynamically connected actors are validated, the result can be recorded for future use, so that subsequent runtime connector validation is reduced to a simple look-up function.

A practical approach to runtime connector validation is to execute the algorithms during role requests. Such an approach is discussed below.

6.5.1 Runtime connector validation as part of role requests

As stated in section 2.4.5, a session can be established as a result of role requests, i.e. an actor asks for a certain role to be played by another actor. This is the approach used in [ServiceFrame 2002]. An example of the role request pattern is shown in Figure 6.12.

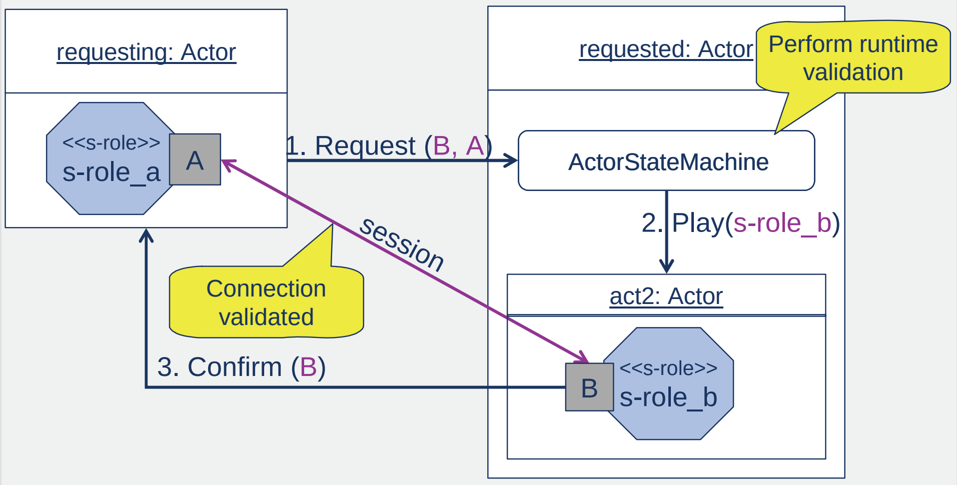


Figure 6.12 : Role request pattern

In Figure 6.12 a request for role playing comes from a requesting actor to the state machine *ActorStateMachine* of a requested actor.

A role request can perform more than just a role allocation; it can also involve runtime connector validation, i.e. a validation between the interface roles on each side of the connection. This is what [Bræk 1999] calls *role alignment*: it includes validation and learning.

In addition to identifying the desired service and other necessary information, the *Request* signal can supply or in other ways identify a description of the interface role of the requesting actor[6]. This information can be used by a validation mechanism in the requested actor to determine whether it is able to play a service role whose semantic interfaces pass a role validation against the semantic interfaces of the requesting actor.

For sake of argument, let us presume that the runtime connector validation mechanism resides in *ActorStateMachine* in Figure 6.12. This means that *ActorStateMachine* performs validation of the semantic interface supplied in the *Request* signal against the requested actors' portfolio of semantic interfaces. It should validate both safety and liveness properties. The validation result should determine the response or confirmation

6. For instance, an XML representation of the interface role can be included in the request.

signal in Figure 6.12; if no semantic interface at the requested side can be successfully validated for safety and usefulness, the role request response should be negative.

6.5.2 Challenging and challenged roles

We have previously introduced the concepts of initiating and responding role, and have discussed requesting and requested actors. Although the actor initiating the role request is the most likely candidate to play the initiating role, role request should not be restricted only to supporting this configuration.

We suggest the term *challenging role* to determine the requesting side of a connection, and *challenged role* for the connected role, regardless of which plays the initiating and responding role, see Figure 6.13.

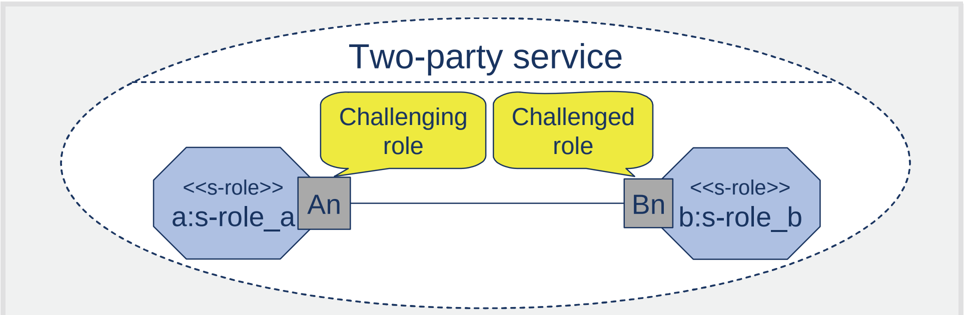


Figure 6.13 : Challenged role associated with challenging role

In the context of role requests, the challenged role is the role that might have substitution alternatives, while challenging role is regarded as fixed.

The progress checking algorithms evaluate progress from the vantage point of the challenging role. However, recall from section 4.2.1 that semantic interfaces must have consistently marked event goals. Hence the result of the progress checking will yield the same result, regardless of in which “direction” the validation is performed.

6.6 Summary of service validation techniques

The validation techniques presented fall neatly into two groups:

i. *connector validation*, which validates safety properties and validates basic liveness properties by checking for the presence of event goals in their interactions;

ii. *validating state-like goals*, which validates liveness properties by checking that goal expressions are reached and that goal assertions are not falsified.

In this section we summarize the techniques and give an overview of the validation method. We also discuss the assumptions regarding the validation of liveness.

6.6.1 Connector validation

Connector validation focuses on the exchange of signals over interfaces. It considers interface behaviour of actors and service roles, i.e. their actual behaviour expressed by p-roles and/or their specified behaviour expressed by bound semantic interfaces.

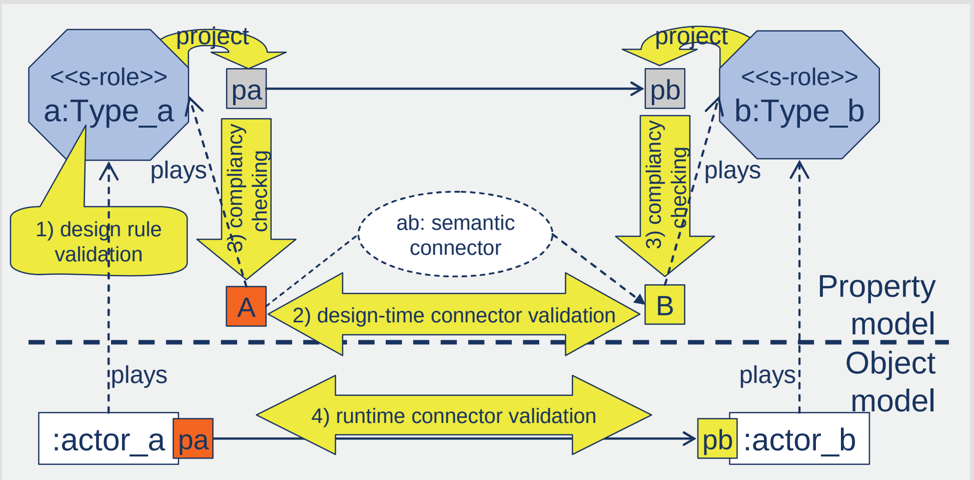


Figure 6.14 : Connector validation

Connector validation has the following elements, see Figure 6.14:

1. Performing *design rule validation* to check whether service roles follow design rules, ensuring that *role projection* from service roles results in safe p-roles;
2. Performing *design-time connector validation* to:
   i. check basic safety properties, which we call *safety checking*;
   ii. check basic liveness properties, which we call *progress checking*;
3. *compliancy checking* between p-roles obtained by projection from a service role, and the semantic interfaces that are bound to a service role or actor, i.e. validating whether the actual interface behaviour is compliant with specified interface behaviour;
4. If necessary: performing *runtime connector validation* on connected actors to validate the basic safety and liveness properties of the connection between them.

The purpose of connector validation is to ensure that interactions are safe *and* useful. Given that all interface roles of a composite service are validated for all its constituent service roles, then a basic level of quality has been established: nothing bad can happen when the service roles collaborate, and at least pairs of service roles can achieve something good. What is not validated are goal assertions, which we discuss below.

Steps 1 through 2i above, i.e. validating that the design rules for service roles, performing role projection from service roles to p-roles, and safety checking between pairs of interface roles, are all due to the work of [Floch 2003]. This was introduced in section 4.1.3.

We have added a basic validation of liveness properties. Our main contribution lies in step 2ii, *progress checking*, and was presented in section 6.1. Algorithms were presented and discussed, and the design and implementation of a tool were described.

Step 3 is also unique to our approach, and was presented in section 6.2.

Steps 1 though 3 can be used at design time. The optional step 4, *runtime connector validation*, is an application of the techniques at runtime. It only needs to be performed for distributed actors in cases where design-time validation (steps 1 though 3) has not been performed using the same service description; in this case *actor_a* and *actor_b* of Figure 6.14 relate to different property models. The technique was outlined in section 6.5.

6.6.2 Validating state-like goals

Validation of state-like goals uses state-space exploration techniques to check whether goal expressions can be true, and that goal assertions are never falsified.

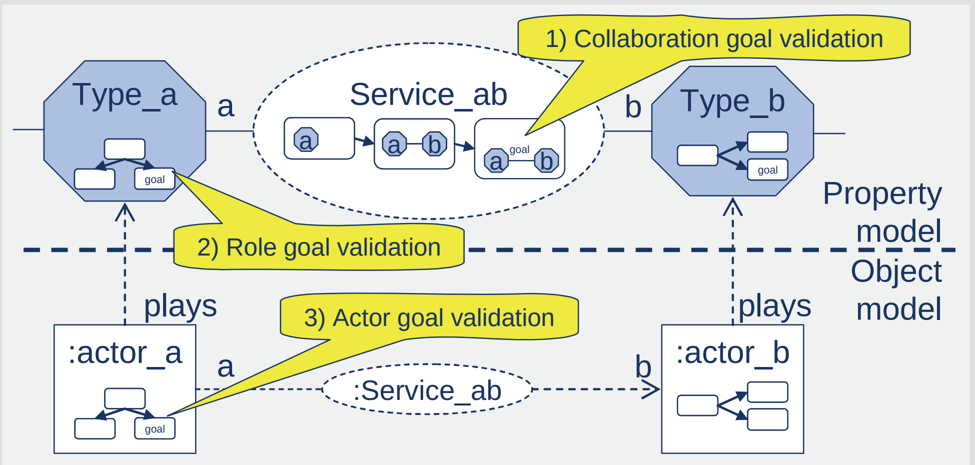


Figure 6.15 : Validating state-like goals

Validating state-like goals is applied to all service structure elements, see Figure 6.15:

1. *Collaboration goal validation*, validating that a collaboration can reach its collaboration goals, and that collaboration goal assertions are never falsified;
2. *Role goal validation*, validating that service roles can achieve their role goals, and that collaboration goal assertions are never falsified;
3. *Actor goal validation*, validating that actors can reach their actor goals.

Validation of state-like goals was described in section 6.3. Validation of collaborations, service roles and actors can be done at design time, and need not be repeated at runtime.

6.6.3 Comparison

While the aim of connector validation is to examine the interface behaviour between pairs of roles, validating state-like goals entails a full reachability analysis of collaboration and role behaviours.

The main benefit of connector validation compared to validating state-like goals is that the former requires fewer resources in terms of time and/or space to reach a conclusion, and that it can be performed at runtime if necessary. The downside is that it only deals with interface behaviour, and hence does not cover all aspects: it does not address goal assertions, so the technique is incapable of validating goals related to a set of connections.

6.6.4 Validation method overview

The following methodical approach to validation is suggested:

1. Validation of a semantic connector and its pair of semantic interfaces:
   i. check whether the interface roles defined by the elementary collaboration are safe; if not then they must be redesigned - see section 4.1.3;
   ii. check whether both interface roles are live (contain role goals); if not then they must be redesigned in order to qualify as semantic interfaces - see section 6.1.1;
   iii. check whether event goals can be achieved for the pair of interface roles when they collaborate, if not the collaboration is not well-formed - see section 6.1.3;
   iv. check whether the goal assertion(s) of the roles, as well as any collaboration goals defined for the semantic connector, are achievable - see section 6.3.3;
2. Validation of collaborations composed of semantic connectors:
   i. check whether collaboration goals (if defined) are achievable; if not the service role (or the collaboration goals) must be redesigned - see section 6.3.2;
   ii. check whether the collaboration is consistent with its collaboration goal sequences (if any) - see section 6.4.2;
3. Validation of service roles and the semantic interfaces bound to them:
   i. check whether service roles can be projected to p-roles; if not, the service roles must be redesigned - see section 4.1.3;
   ii. check whether the p-roles are compliant with a semantic interface bound to it; if not then redesign the service role - see section 4.3.3;
   iii. check whether goal assertions (if any) of the service role are achievable; if not the service role (or the goals) must be redesigned - see section 6.3.3;
4. Validation of actors composed of service roles with semantic interfaces:
   i. check compliancy with semantic interfaces bound to the actor (directly and/or through service roles) - see section 6.2;

ii. check whether the actor is consistent with its actor goal sequence (if any) - see section 6.4.3;

iii. only when necessary is runtime validation performed - see section 6.5.

The validation method uses a combination of techniques presented:

- steps 1.i-iii, 3.i-ii and 4.i and iii use connector validation techniques;
- steps 1.iv, 2.i, and 3.iii uses techniques to validate state-like goals.

The remaining steps 2.ii and 4.ii entail checking consistency with goal sequences.

7

Service discovery

In this chapter we show how service discovery can take advantage of semantic interfaces and goal sequences. We present different forms of service discovery: *discovering compatible actors*, *discovering service opportunities*, and *role learning*, and outline mechanisms to support them. Lastly we discuss scalability issues.

7.1 Introduction

As stated in section 1.3.5, the need for service discovery of telecom services is not addressed by traditional discovery mechanisms supported on client-server platforms:

1. An actor can know of a set of actors, such as contacts in an address book, and would like to know what goals can be achieved with each. For instance, which contacts can be called, which can be sent a message to, which can be included in a multimedia conference. In traditional service discovery, clients discover what interfaces exist, and which servers provide them; which server provides a service is often unimportant;

   Service discovery can also be viewed from the viewpoint of services: given that an actor can initiate a set of services, what other actors (i.e. contacts) can be reached using each service? There is no equivalent to this for client-server systems.

2. Furthermore, since actors have independent behaviour and change state depending on their interactions, the goals that can be achieved between a particular set of actors at a specific point in time depends on the situation. It would be desirable for actors to know what service opportunities are available in the current context;

   While traditional service discovery might be able to determine whether services are available or not, in the sense that servers are on-line and up-and-running or not, there is no tradition for supporting discovery about e.g. instances of information.

With these needs in mind, we propose mechanisms to support different forms of service discovery, with increasing levels of ambition:

1. Determining whether an actor is capable of achieving service goals when interacting with a set of other actors. We call this *discovery of compatible actors*;
2. Determining what semantic interfaces are offered by actors in the environment of an actor at a particular point of time. We denote this *discovery of service opportunities*. This service discovery mechanism takes the current states of actors into consideration;

3. Determining new service opportunities, i.e. finding out whether an actor can perform new or enhanced services by learning new service roles. This we call *role learning*.

In the following sections we discuss each of these mechanisms in turn.

7.2 Discovery of compatible actors

We define services as collaborations between roles played by actors. One form of service discovery is for actors to find a set of actors with whom they can successfully achieve service goals. This means that the collaborating actors play compatible semantic interfaces. We call this mechanism the *discovery of compatible actors*.

Definition: Discovery of compatible actors
Discovery of compatible actors is a service discovery mechanism by which an actor can determine which actors are capable of playing compatible roles.

Compatibility can be calculated once and for all, provided the role-playing capabilities of actors do not change. Supplied with this knowledge, an actor can differentiate over or search for instances of compatible actor types in its environment, and be satisfied that they have the potential of reaching service goals when interacting.

Discovery of compatible actors is somewhat similar to traditional service discovery in IT parlance, as exemplified by service-oriented computing, which talks about a discovery layer "for services to advertise their capabilities and for clients that need such capabilities to locate and use the services" [Singh and Huhns 2005]. Determining which actors support compatible roles is equivalent to "locating" them, assuming as we do that actors can be addressed by communication layers below the service control layer.

Discovery of compatible actors entails a "static" comparison of the semantic interfaces that actor types can play. By *static* we imply that it does not take the current state of actors into consideration, only the role-playing capabilities that are due to the role composition of the actor type in question. If role-playing capabilities change, e.g. due to new role compositions, then the discovery procedure must be repeated.

The following mechanism for discovery of compatible actors is suggested:

- For all initiating roles played by an actor type, determine which actor types can play compatible roles. Instances of such actor types are by definition capable of reaching role goals when interacting;
- Look for actors playing live subtypes of the opposite role. Safe subtypes imply that the interaction between the actor pairs satisfies basic safety and liveness properties.

We outline an algorithm for discovering compatible actors in Algorithm 7.1 on page 172. In the following sections we present the technique through two examples.

7.2.1 Example 1: two-party service

In the previous chapter we discussed how to validate whether actors are compliant with the semantic interfaces that are bound to them. This validation is performed at design time, exploiting knowledge of what semantic interfaces actors are compliant with.

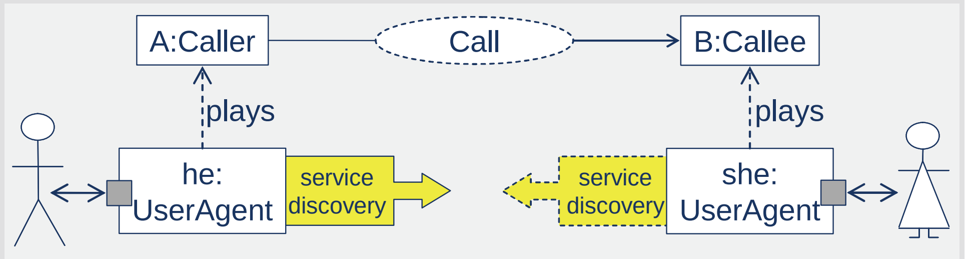


Figure 7.1 : Discovering compatible actors: two-party service

In the example in Figure 7.1 we see two actors compliant with the roles *Caller* and *Callee* of *Call*. In this classical example we presume that *he* has *she* among his contacts, and *he* would like to know whether it is possible to place a call to *she* (or initiate some other service to get in contact). Or conversely, that the *Call* service that *he* has can be used to call *she*. Discovery of compatible actors provides *he* with this knowledge.[1]

It is not obvious that *she* wants to discover which actors can call her; for this reason the block arrow to the right is dashed in Figure 7.1.[2]

7.2.2 Example 2: multi-role service

Multi-role services can be composed of semantic connectors. For instance, the composition of *MpConf* is shown in Figure 7.2.

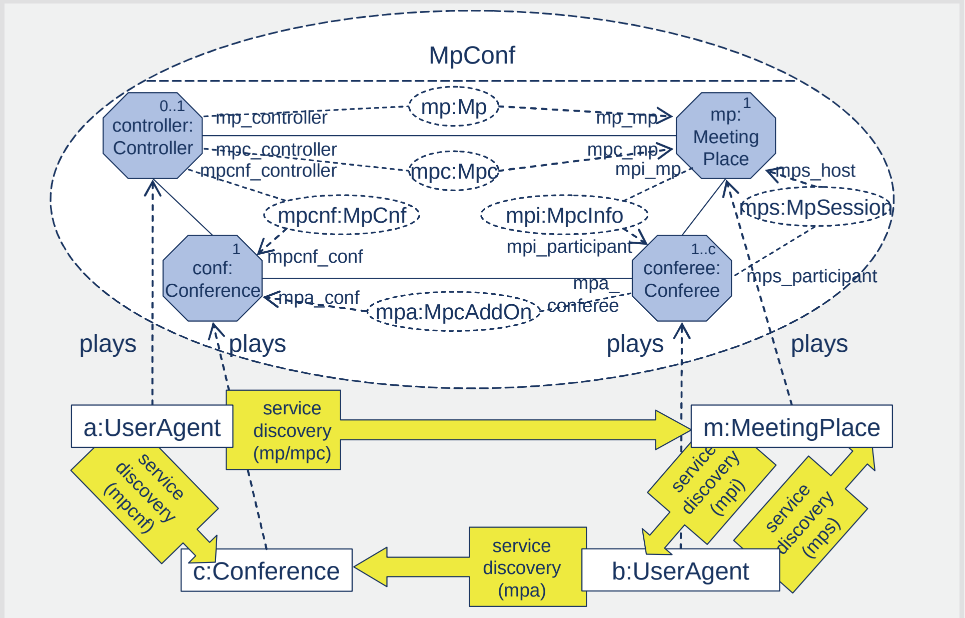


Figure 7.2 : Discovering compatible actors: multi-role service

1. The discovery mechanism looks for actors that can play live subtypes of the *Callee* role. In this example both actors are of the same actor type, *UserAgent*. This need not be the case.
2. “Discovering callers” is not generally recognised as a useful service discovery feature. However, specifying the opposite, i.e. who should *not* be able to call one, is a traditional service feature known as *Call Screening*.

Figure 7.2 shows how the service roles of *MpConf* are bound to four actors, and illustrates *discovery of compatible actors* for each. For each actor, the discovery mechanism looks for other actors that can play live subtypes of its dual role. E.g. *a:UserAgent* looks for actor types that can play the *mp_mp* and *mpc_mp* roles, as well as actor types that can play the *mpcnf_conf* role. This is according to the semantic connectors that *MpConf* is composed of, and the semantic interfaces that actor *a* plays.

Actor types can play a number of semantic interface roles, either simultaneously and/or consecutively. It is the total portfolio of role-playing capabilities that is of concern to service discovery. This determines what services are available to each actor type, depending on the existence of actor types in its environment with the necessary role-playing capabilities. For instance, if no actor can play *mp_mp*, then actor *a* cannot initiate the semantic connector *mp*.

However, this does not imply that the services can be successfully initiated at a particular point of time, as the actors discovered may not be in a situation (state) in which to grant requests, e.g. the actor *m* may not be in a state where it can play *mpc_mp*. This is because the discovery mechanism does not take actor states into consideration, nor goal sequences. The next section addresses a mechanism that does.

7.3 Discovery of service opportunities

In this section we present an approach to service discovery where we take goal sequences and the goal achievement of actors into consideration. This entails finding other actors in the environment of an actor with an enabled responding role that is compatible with an enabled initiating role of the actor.

Discovery of service opportunities is evaluated during service execution. With such a mechanism in place, an end user can be made aware of new service opportunities as they arise, e.g. through information presented via the user interface.

The opportunities for actors to initiate semantic connectors evolve as goals are achieved by the actor itself and by other actors in its environment. What roles that are enabled is a function of goal achievement and the goal sequences, as discussed in section 5.4.2.1 on page 129.

Definition: Service opportunity
An actor has a service opportunity with a set of actors in its environment when there exists a non-empty intersection between its current set of enabled initiating roles and the set of enabled compatible roles offered by the actors in the environment.

That a service opportunity arises means that the roles the actor is capable of playing become enabled due to the achievement of certain goals within itself and in actors in its environment. It does not imply that an actor learns new roles.

The sequence of goals is described in actor goal sequences. Along with timely information about the status of goal achievement of actor instances, this information can be used to determine what opportunities exist at any particular point in time.

Discovery of service opportunities requires that the service discovery mechanism knows what roles each actor instance is currently playing, what goals have been achieved, and which roles are enabled at any point in time, as expressed by the actor goal sequences. The mechanism distinguishes between initiating and responding roles, assuming that service opportunities are only of interest to actors that are enabled to play initiating roles.

We outline an algorithm for discovering service opportunities in Algorithm 7.2 on page 173. In the following section we present the technique by way of an example.

7.3.1 Example: multi-role service

Consider the actor goal sequences depicted in Figure 7.3.

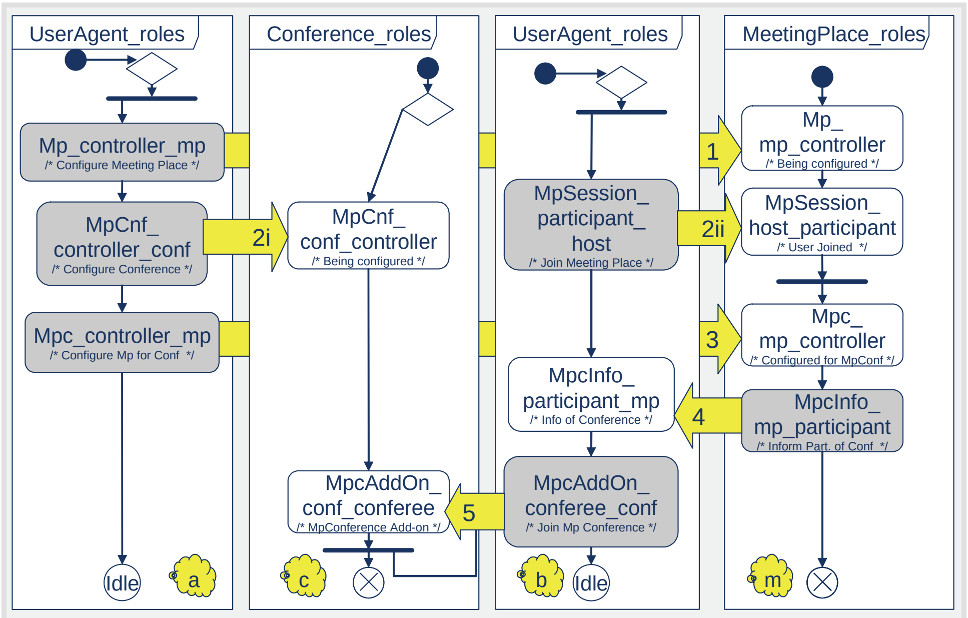


Figure 7.3 : Exploiting actor goal sequences to discover service opportunities

Figure 7.3 contains portions of three actor goal sequences presented in section 5.3.1.2 on page 125. Here the goal sequence *UserAgent_roles* is referred to twice and with a different portion, in accordance with Figure 7.2 where two instances of *UserAgent* are involved, here playing different roles. The actors *a*, *c*, *b* and *m* named in Figure 7.2 play (from left to right) the actor goal sequences in Figure 7.3, as indicated by callout clouds.

The numerated steps below refer to the labels in the block arrows of Figure 7.3.

1. At the outset the actors *a*, *c*, *b* and *m* have not achieved any goals. However, *c* “offers” the enabled role *MpCnf_conf_controller*, and *m* offers *Mp_mp_controller*. The latter implies that *a* can initiate *Mp_controller_mp* toward *m*. Note that no actor is yet enabled to play the *MpCnf_controller_conf* role with *c*.

2. Given that the goal of the semantic connector *Mp* is achieved by *a* and *m* in step 1, *a* is now enabled to play *MpCnf_controller_conf*, while *m* now offers *MpSession_host_participant*. Hence both *a* and *b* have service opportunities that can be taken advantage of:

   i. *a* initiates *MpCnf_controller_conf* to start configuring the conference.

   ii. *b* initiates *MpSession_participant_host* to join the meeting place.

3. Achieving the goal of the semantic connector *MpSession* enables *m* to offer the *Mpc_mp_controller* role, an opportunity which *a* can subsequently take advantage of, provided it has achieved the goal of the semantic connector *MpCnf*. Note that both goals of step 2 must be achieved before *a* gains this opportunity.

4. The goal of the semantic connector *MpSession* having been achieved in step 2ii enables *b* to offer the *MpcInfo_participant_mp* role, while *m* is enabled to play *MpcInfo_mp_participant* as the result of achieving the goal of the semantic connector *Mpc*.

5. Having achieved the goal of the semantic connector *MpcInfo*, *b* is enabled to play *MpcAddOn_conferee_conf*, while the semantic interface *MpcAddOn_conferee_conf* has been offered by *c* since step 2. Thus *b* is in a position to join the meeting place conference by initiating the sub-service *MpcAddOn*.

   Achieving the goal of the semantic connector *MpcAddOn* results in the collaboration goal of the composite service *MpConf* being achieved.

The example illuminates a number of central points:

- That actors offer responding roles means that peer actors can play the corresponding initiating roles. I.e. if a responding role is not offered, progress for its corresponding initiating role is not possible. Conversely, if initiating roles are not enabled, progress for its corresponding responding role(s) is not possible.

  An example: In step 2 above, if there is no *UserAgent* like *b* that succeeds in joining the meeting place, then *a* cannot proceed to configure the meeting place conference. This is in accordance to the service logic defined by the goal sequence diagram of Figure 5.12. Recall from Figure 3.11 that *Mpc* is initiated by the signal *ConfigMpConf* from the environment (i.e. from the user); this step in the service logic will only achieve its goal when the user withholds this initiative until the conditions are fulfilled.

  Presenting the service opportunity to the end user when conditions are fulfilled (at least one other user has joined the meeting place) is an example of context-sensitivity.

- Service opportunities are exactly that: opportunities. The fact that roles are enabled does not force them to be played. For instance there is nothing compelling *a* to initiate *Mp* in step 1.

The example demonstrates how actor goal sequences can be exploited to support context-sensitive service discovery.

7.4 Role learning

Role learning is inspired by the Internet, where a web client will send a generic request to a resource identified in a URL, and receives HTML code that defines "services" available to the client, sometimes in the form of software plug-ins.

Given knowledge of what responding roles are offered by its environment, an actor can gain information about functional possibilities that it was not initially designed to handle. It may be possible for an actor to adapt so that it can initiate more services towards its environment. It can for example search for service roles that give a better match against the responding roles in its environment, i.e. service roles that can achieve more goals.

In this section we look at principles and mechanisms that can be put to use for actors to learn about new role behaviour from their environment. We shall see how initiating actors can learn functional capabilities from their peers, i.e. functionality that goes beyond what the actor's initiating role is capable of.

Consider the initiating role in Figure 7.4.

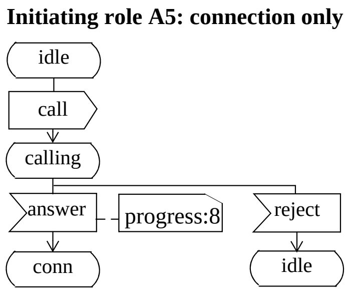


Figure 7.4 : Initiating role A5 before learning

An actor playing the initiating role *A5* in Figure 7.4 is capable of initiating calls. Imagine that it interacts with an actor playing the responding role *B1* shown in Figure 7.5.

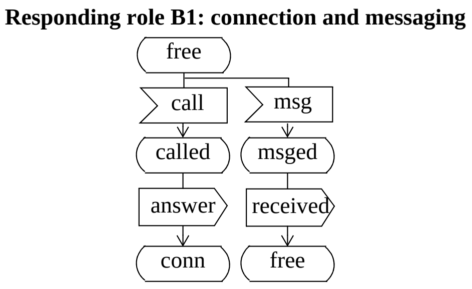


Figure 7.5 : Responding role B1

Using the techniques to validate basic liveness properties presented in section 6.1, one can validate that *A5* and *B1* will play well together, and can achieve the call setup goal indicated by the progress label in Figure 7.4.

However, it is apparent that the messaging capabilities of *B1* are "wasted" on the actor playing *A5*, since it is not capable of initiating messaging. An actor would play "better" with the role *B1*, i.e. achieve more goals, if it played the interface behaviour of the semantic role *A6*, see Figure 7.6.

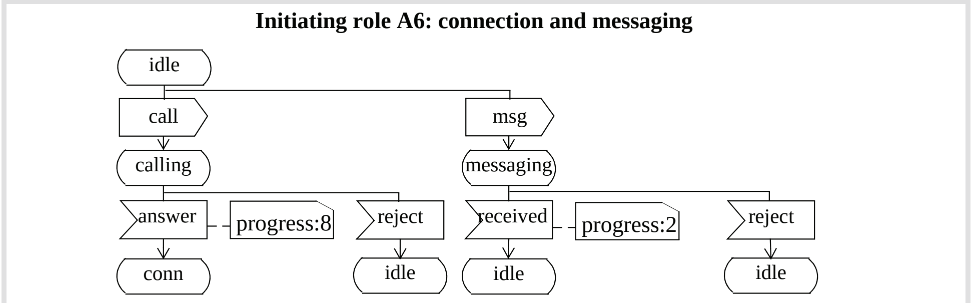


Figure 7.6 : Initiating role A6 has more capabilities than A5

It is desirable for an actor to learn a service role with a semantic interface that is compatible with a given opposite semantic interface. Such a role learning mechanism is illustrated in the communication diagram in Figure 7.7.

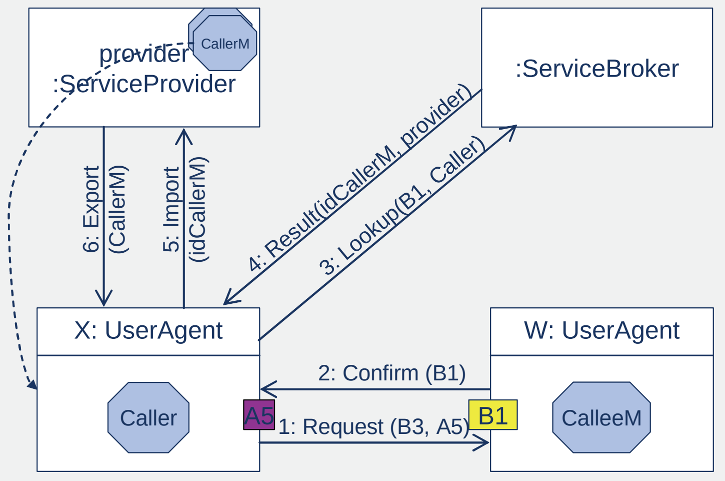


Figure 7.7 : Learning a new service role

Here the initiating actor *X* issues a role request[3] to *W*, stating that a live subtype of *B3* is requested (*B3* is described in Figure 4.7 on page 77). In addition to informing *X* that the two can play well together, the role confirmation includes a description of the responding role (*B1*). Since *B1* is different from *B3*, *X* approaches a service broker[4] in search of a service role that it can play, to look for a new role whereby *X* can achieve more of the goals of *B1*. It includes a description of *Caller* to characterise its current functionality.

3. See section 2.4.5 for an introduction to the role request mechanism. An alternative way of learning is to preform a comparison between an actor's semantic interfaces and the semantic interfaces of its contacts. An actor can thereby learn new roles from known actors in its environment prior to interacting with them.
4. The mechanism assumes the existence of a service broker, or *trader* as it is called in [RM-ODP 1998].

Here the broker[5] provides a link to a service provider that offers the service role *CallerM* that *X* subsequently downloads. *X* is from then on capable of playing the interface role *A6*.

Note that the learning of new service roles, steps 3 to 6 in Figure 7.7, can be performed in parallel or independently of the actual service invocation that takes place after step 2.

Role learning means that the behaviour of *X* is changed. This may influence other semantic interfaces in addition to *A5*; in the example the user interface of *X* must be updated so that messaging can be invoked. This is why the *Lookup* includes information about the actor's capabilities (i.e. the service role *Caller*).[6]

One consequence of learning new roles should be noted: if the requesting actor *X*, after having learnt the service role *CallerM*, sends new requests according to the new role *A6*, it may encounter role rejections from peers that only play *B3*.[7] *X* could find it necessary to be able to play the old behaviour as well as the new. This implies that learning new behaviour does not necessarily mean that old behaviour has no further use.[8]

7.5 Mechanisms

The approach is based on the assumption that service structures, goals and goal sequences are known to the service discovery mechanism, as well as knowledge of compatible opposite roles and live subtyping relationships. Combined with the information about what roles actor types can play or are playing, this information can be used to:

1. determine the totality of useful semantic connectors between various actor types; this enables an actor to find other actors in its environment with which to play (*discovery of compatible actors*);
2. discover actors in the environment of a given actor that currently are in a state that enables goals to be reached (*discovery of service opportunities*).

Below we sketch mechanisms that can support these forms of service discovery.

7.5.1 Transformation of activity diagrams to transition charts

We decided to transform actor goal sequences into transition charts, using the set-based notation of [Floch 2003] to represent transition charts. Our motivation for this is that the algorithms can reuse the existing work of [Alsnes 2004] and [Korda 2004]. Here we discuss how this can be done.

We let actor goal sequences be represented by transition charts, where inputs represent responding roles, outputs represent initiating roles, and states represent the achievement of preceding goals. Potential transitions out of a state represent service opportunities,

5. Progress calculations can be performed by the broker to select between alternative service provider candidates.
6. The lookup request may convey additional information about the device, its network connections and so on.
7. A solution can be to include the semantic interface (*B3*) in role rejection, and let the initiating actor search for alternative service roles (i.e. *Caller*) in its own role portfolio, or re-learn *Caller* from a service provider.
8. This issue has implications on the life-cycle of service logic in general, e.g. how to know when behaviour is obsolete and can be removed. General solutions to this issue must be sought, such as caching service software, and discarding the oldest when space on the device runs out. This is a topic for further work.

while the firing of a transition represents an activation of the semantic connector named by the signal event.

The transformation from activity diagrams to transition charts is as follows:

- the activity start (which is unique) is mapped to an initial state s_0;
- initiating roles are mapped to an output event with the same name as the role;
- responding roles are mapped to an input event with the same name as the role;
- each signal event is preceded by a state, followed by a next state, as follows:
    - an edge between actions is mapped to a unique state s_i;
    - a decision node gives succeeding events with the same preceding state;
    - a merge node gives preceding events with the same next state;
- a fork node is treated as a decision node, while a join node is treated as a merge node.

Figure 7.8 provides an example that illustrates the transformation of activity diagrams to transition charts. The transition chart is here represented by a state transition diagram.

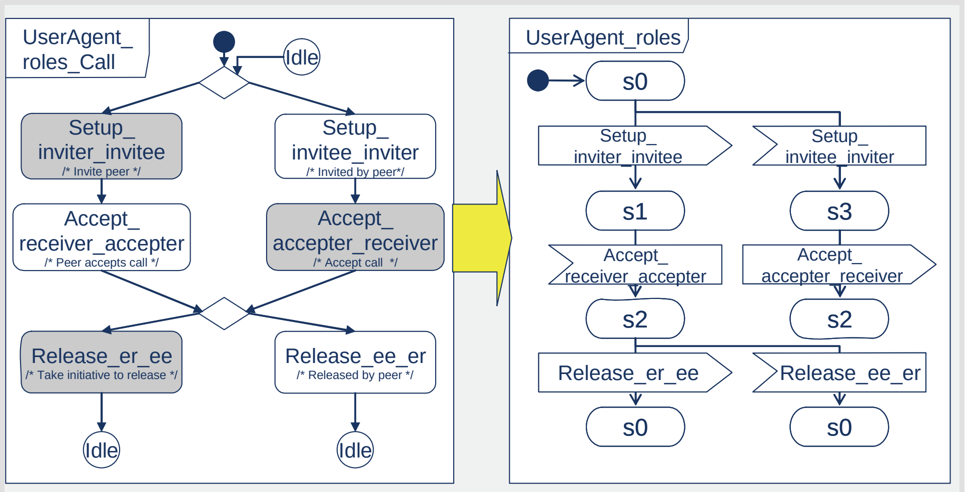


Figure 7.8 : From actor goal sequences to transition charts

7.5.1.1 Actors playing several roles simultaneously

Join and fork nodes model simultaneous role-playing capabilities. This can be modelled by hierarchical states, but not by a simple transition chart. Instead we opt to use transition charts and model that an actor is in several states simultaneously. For the algorithms this choice means that it is easy to support the semantics of fork nodes. Not so with join nodes.

A join node “fires” if it is offered tokens on all its incoming edges. Supporting this using simple transition charts is not straightforward. One solution is to map each edge leading to a unique state; all these states have an empty transition to a new state that “fires” spontaneously when an actor is in all the preceding states simultaneously, see the transitions to the state *s3* in Figure 7.9, where a condition *{and}* is inserted.

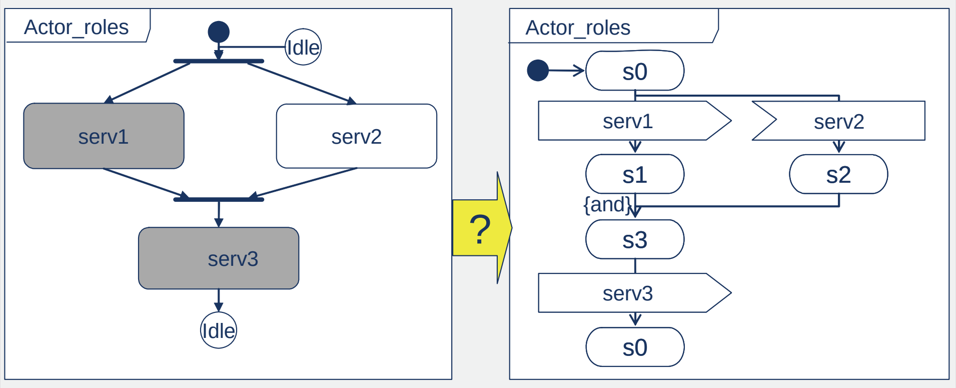


Figure 7.9 : Possible mapping of join nodes

However, since join nodes do not seem to be necessary we proposed doing without them (see the method rule "Model actor goal sequences" on page 127). If present, they are treated as merge nodes in the transformation from actor goal sequences to transition charts.

If later research reveals that join nodes are needed, then a different representation than transition charts must be used. An attractive alternative is to use the UML model directly, e.g. in the Eclipse Modelling Framework used by Ramses, see [Birkeland 2005].

7.5.1.2 Set-based notation

The set-based notation of [Floch 2003] was introduced to perform validation of role behaviour. It can be used to represent transition charts, and we simplified it so suit our limited requirements, which is to represent actor goal sequences.

The simplification consists of removing constructs from the set-based notation that are not needed. These include τ-events, σ-states, exit states, equivalent states, equivocal transitions, entry and exit points, and deferred triggers. In addition we only require one initial state s_o, not a set of initial states.

The transition charts are defined by:

- a finite set $S = \{s_0, s_1, s_2,..., s_n\}$ of states, where s_0 is the initial state;
- a finite set $E = \{e_1, e_2,..., e_r\}$ of events that trigger transitions. This set is the union of the disjoint sets:
    - $I = \{i_1, i_2,..., i_s\}$, a set of inputs, representing responding roles;
    - $O = \{o_1, o_2,..., o_t\}$, a set of outputs, representing initiating roles;
- a state transition relation T. To each pair (s, e) of S x E, T associates a set of zero or one immediate successor state (a subset of S);
    - If $T(s, e)$ is empty, there exists no transition from the state s for the event e. Otherwise it gives the successor state for the event e in the preceding state s.

7.5.2 Data structures

Data structures that model the initiating and responding roles of an actor type are shown in Figure 7.10 below. The figure shows both the UML model and examples of how the model can be instantiated.

The design criteria for the data structures are as follows:

- the number of initiating or responding roles for each actor type is a fraction of the number of semantic connectors that exist;
- the number of roles an actor plays simultaneously is small (actor goal sequences will not result in many states), thus the number of states offering more than one responding role is not large;
- it should be easy to determine whether an initiating role can find a role compatible with semantic interfaces played by other actor types;
- it should be easy to determine whether an instance of a requested actor type has an enabled role in its present state(s).

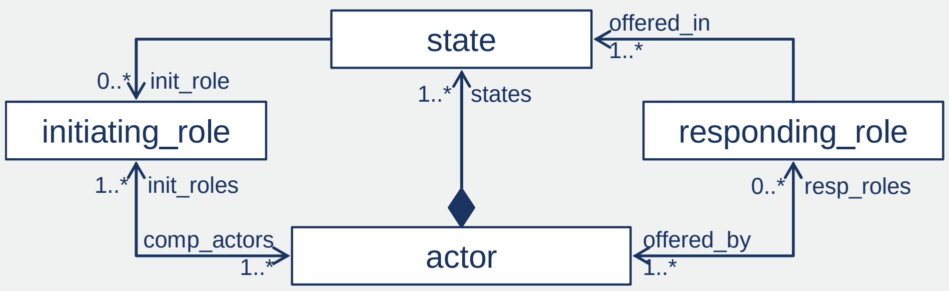
The following list describes the algorithms needed to build the data structures shown in Figure 7.10:

i. The table of initiating roles per state can be built up by traversing the transition chart of the actor type, and only including the output events for each state found. This is trivial, and no algorithm is included here. The data structure is fixed as long as the actor goal sequence remains unchanged. If new functionality is downloaded, or functionality is removed, the data must be rebuilt;

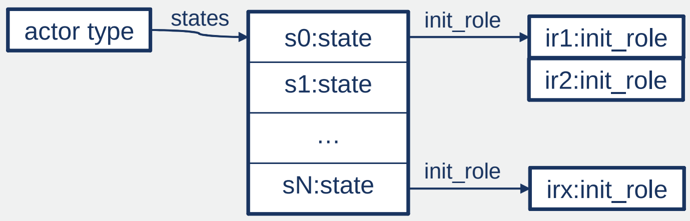
ii. The list of responding roles is built up by traversing the transition chart of the actor type, adding each new input event in the list of responding roles, and adding the preceding state if not already included. The data is fixed as long as the actor goal sequence remains unchanged, but must be rebuilt if functionality is added or removed;

iii. The list of compatible role players for the initiating roles of an actor type identifies what actor types can play well with it in given states. It is built by Algorithm 7.1;

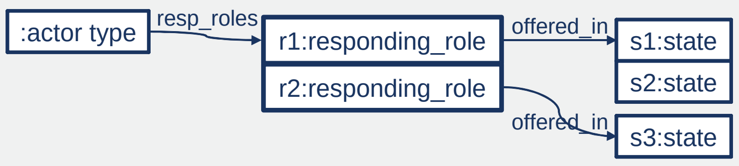
iv. The global table of responding roles is built by traversing all the structures of table ii, and, for each responding role found, adding the actor type to the list that offers the role.



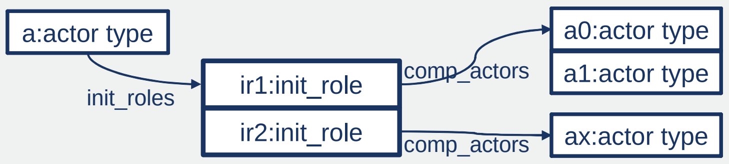
UML data model



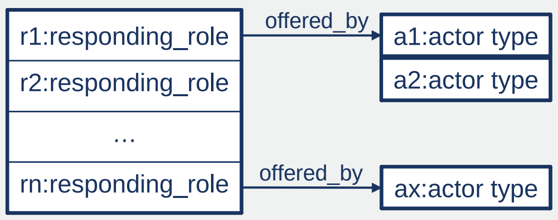
i) Actor type’s table of states listing its initiating roles for each state



ii) Actor type’s list of responding roles, listing states in which each is offered



iii) Actor type’s list of compatible actors for its initiating roles



iv) Global table of responding roles, listing actor types that offer the role

Figure 7.10 : Data structures for service discovery

7.5.3 Algorithms

In the following we sketch the pseudocode for the algorithms that define the service discovery mechanisms. The algorithms use the data structures presented in Figure 7.10. The actor goal sequences are represented using the set-based notation. Some functions are not detailed, such as:

- identifying a semantic connector from the name of a semantic interface;
- finding an opposite role for a semantic interface. The opposite role is denoted $\overline{\text{role}}$ in the algorithms; it can be a live subtype of the dual role of the semantic interface;
- determining whether an interface role is an initiating or responding role (represented by an output event or input event, respectively, in the set-based notation).

We present two algorithms below:

1. Algorithm 7.1 performs discovery of compatible actors for an actor type; building table iii in Figure 7.10. The algorithm can be run periodically as a background process, with a frequency that depends on the (rate of) change of role behaviour in the environment. This is due to the fact that it only needs to be run when roles change.
2. Algorithm 7.2 finds service opportunities for an actor instance, and is invoked as a background process, with a frequency proportional to the rate of state changes in the actor's environment, and for every goal achieved by the actor.

Algorithm 7.1 builds the contents of *list_iii* used by Algorithm 7.2; *list_iii* lists actor types that play compatible roles. In addition to its use in Algorithm 7.2, it can be used to present actors with a list of available services corresponding to the enabled initiating roles, and for listing which actors these service can be initiated toward.

Algorithm 7.1: Find compatible actors for an actor type

```
main()
{
    list_iii = {}/* List of compatible actors, initially empty*/

    for each state of table_i
    {
        for each init_role in list_i
        {
            /* Find actor types with a compatible opposite $\overline{\text{role}}$ */

            for each actor type that offers responding role in table_iv
            {
                /* Check safeness/liveness properties */

                if Compatible(init_role, $\overline{\text{role}}$) /* Determined by Algorithm 6.2 on page 141 */
                {
                        add (actor type) to list_iii for init_role
                }   /* ignore actor types that give no progress */
            }   /* skip actor types that can't play responding role */
        }   /* loop of the actor's initiating roles in state */
    }   /*loop of actor's states */
}   /* Find compatible actors for an actor type */
```

Algorithm 7.2: Find service opportunities for an actor instance

```
main()
{
    opportunity_list = {}/* List of service opportunities */
    /* Format: {role, actor} */

    state := current state of actor

    for each init_role in table_i
    {
        for each actor type in list_iii /* Built by Algorithm 7.1 */
        {
            state_list := actor->resp_roles[$\overline{role}$].offered_in

            for each instance of actor type
            {
                /* check whether instance can play responding $\overline{role}$ */
                if instance.states in state_list
                {
                    add (init_role, instance) to opportunity_list
                }    /* skip actors that can't play responding role */
            }    /* loop of instances of actor type */
        }    /* loop of actor types in environment */
            /* ignoring actor types that give no progress */
    }    /* loop of initiating roles */
}    /* Find service opportunities for an actor */
```

Algorithm 7.2 supports the fact that actors can play multiple interface roles simultaneously.

7.6 Scalability issues

A discovery mechanism used at runtime must scale if it is to be of practical use. We have suggested a number of techniques to help achieve scalability:

- *Discovery of compatible actors* can be performed as a background process, possibly using agent technology. It can take place between actor types e.g. in different administration domains, rather than between actor instances; hence scaling problems are reduced. We assume that systems are populated by many instances of a limited number of types. In addition the mechanism constitutes a pairwise comparison of semantic interfaces. Both principles imply that the computational requirements are reduced.
    - It is based on knowledge gained at design time, such as the validation of actors against specified semantic interfaces. This implies that the computation required at runtime is reduced, since is does not need to calculate compatibility at runtime, unless validation for different systems is done against different versions of semantic connectors. In that case, runtime session validation is required;[9]

9. Once done, runtime session validation does not need to be repeated as long as interface behaviour is fixed.

  - The mechanisms suggested assume it is only of interest to actors playing initiating roles; they seek actors that can play compatible roles, while actors playing responding roles do not. E.g. a *UserAgent* actor playing *Callee* needs to find actors that can play *Caller*, while a *Callee* does not need to find actors that can play *Caller*.

- *Discovery of service opportunities* can be limited to the actors involved in active sessions, based on knowledge of actor goal sequences for the actor types. Scaling problems are reduced if only active sessions are evaluated.

Evaluating the scalability of the suggested discovery mechanisms, in particular the discovery of service opportunities, is an issue for further work.

8 Conclusion

In this chapter we summarize the achievements of our work, and present plans and suggestions for further work.

8.1 Main contributions

Our work presents an approach to service modelling, service validation and service discovery. It focuses on the cross-cutting aspects of services by defining semantic connectors and using these systematically to compose, validate and discover services:

- Semantic connectors encapsulate a reusable unit of structure and behaviour. They describe collaboration and role goals, as well as collaboration and role behaviour;
- Arbitrarily complex service structures can be composed of them.

The approach achieves a number of objectives:

- A modular approach to compositional service specification is supported. It exploits semantic connectors and semantic interfaces as building blocks for compositional service design;
- It provides a systematic way of defining goals to express goal fulfilment of collaborations and roles;
- Goal interactions combined with *state orientation* in collaboration state diagrams and in role state diagrams provide different perspectives of collaborative behaviour;
- Goal sequences capture behavioural compositions:
    - Collaboration goal sequences express horizontal and vertical relationships between goals of semantic connectors bound to service roles;
    - Role goal sequences express vertical goal relationships between service roles; they constrain actors playing the service roles;
    - Actor goal sequences express the role-playing capabilities of actors.
- Goal sequences express horizontal and vertical goal relationships in a succinct manner, and lend themselves to advanced context-dependent service discovery;
- A modular approach to service validation is enabled. Validation can be performed on the type level at design time; once performed it does not need to be applied to actor

instances, or repeated at runtime. This implies that the validation approach scales well;

- The validation techniques are closely related to service goals and service discovery. However they can also be applied independently;
- The validation of basic liveness properties has been implemented in prototype tools, thereby demonstrating its feasibility.
- Semantic connectors facilitate the discovery of compatible actors. Combined with goal sequences they also enable the discovery of context-dependent service opportunities. Role learning is also supported:
  - i. *Discovery of compatible actors* determines which other actors a given actor can achieve goals with;
  - ii. *Discovery of service opportunities* determines what services are offered by actors in the environment of a given actor at a particular point of time;
  - iii. *Role learning* enables an actor to learn new roles from its environment.
- The work is based on the latest version of the UML modelling language, exploiting recent additions to UML2, including collaborations, collaboration uses, and interaction overview diagrams;
  - UML2 collaborations are chosen due to the flexible role binding mechanisms of collaboration uses, and their ability to express interactions of cooperating objects and the state behaviour of collaborations and roles;
  - UML2 collaborations are shown to be the “first class modelling citizens” that have been wanting, see e.g. [Krüger et alia 2004];
  - Collaboration goal sequences use interaction overview diagrams.
- The approach addresses the needs of convergent services. However, many elements may also be useful for purely client-server systems, such as defining goals and goal sequences.

In summary we contend that we have achieved our initial ambition. This was not the case while we were still grappling with UML associations as the main modelling element to represent connections, as presented in Appendix A; with UML2 collaborations we have found a modelling approach that lends itself nicely to a compositional approach to service design, service validation and service discovery.

8.2 Planned further work

The approach presented here is novel and has not yet been put to extensive use. However, the principles have been adopted by research projects and by a number of doctoral studies that aim to validate the approach and to advance it further.

Below we present plans for further work.

8.2.1 SIMS - Semantic Interfaces for Mobile Services

The EU-funded project 27610 *Semantic Interfaces for Mobile Services* (SIMS), running from January 2006 to mid-2008, takes our work as a starting point, as well as that of [Floch 2003] and the related implementation work of [Korda 2004], [Alsnes 2004] and [Birkeland 2005]. SIMS will:

- evaluate the feasibility of the approach in an industrial setting;
- provide design and validation tool support for the approach;
- prototype the service discovery mechanisms in middleware, and validate its scalability;
- define a runtime representation of semantic connectors supported by an ontology;
- suggest techniques for composition of object behaviour (i.e. synthesis of actor behaviour from semantic interfaces);
- develop demonstration services.

Below we detail two issues of concern to SIMS: service ontologies and tool development.

8.2.1.1 Service ontologies

For concepts and information to be shared across system boundaries there has to be a common understanding of the service definitions.. This can be represented by an ontology[1], a mechanism being adopted by the IT community, coining the term “semantic web” [W3C 2004].

Concepts that are candidates to be included in a service ontology include:

- service names (e.g. *Call*, *Messaging*, *Mp*, *MpConf* etc.);
- names of service roles and interface roles (e.g. *caller*, *callee*, *msger*, *msgee* etc.);
- role goal names (e.g. *Call_callee_caller*, *Call_caller_callee* etc.);
- elements (attributes and enumerated values) used in the goal expressions;
- live subtype relationships as determined by the validation techniques.

SIMS will as a minimum support the derivation of ontology derived artefacts for role goals. This entails that service goals can be expressed in terms of concepts defined in a domain potential shared by all network operators and users. An ontology can support the search for candidate semantic interfaces for validation, as service designers can more easily find semantic interfaces that are conceptually “close”. It can also aid the search for reusable service roles characterized by their goal achievement capabilities.

1. Ontology is a term from philosophy referring to a systematic account of existence. The term has been adopted by the artificial intelligence (AI) community to specify a *conceptualization*, which means sharing and reusing knowledge by defining a vocabulary so that AI agents can communicate about a domain of discourse without operating on a globally shared theory. An AI agent *commits* to an ontology if its observable actions are consistent with the definitions in the ontology. *Ontological commitments* are agreements to use the shared vocabulary in a coherent and consistent manner following axioms that constrain the possible interpretations for the defined terms. [Gruber 1993]

8.2.1.2 Tool development

In our work we have initiated and supervised the development of prototype tools for progress checking according to the approach suggested in this thesis [Alsnes 2004], and have given advice during the transformation of the safety checking algorithms into an Eclipse plug-in to the Ramses UML tool suite carried out by [Birkeland 2005][2].

Work is being undertaken in SIMS to implement validation algorithms. Tool components to be worked on include:

- Design tool support for derivation steps of the method:
  - "Derive service-specific progress labels" on page 87;
  - "Derive role goal interactions" on page 123;
- Tool support for the notational extensions suggested to UML;
- Validation tools integrated in a service development platform,[3] i.e. tools that:
  - check whether interface roles are live or not;
  - check for the live subtype relationship at design time;
  - check for consistency between auxiliary behaviour descriptions (e.g. state orientation in state machines of collaborations and roles, goal sequences versus roles);
- Validation tools that support connector validation at runtime:
  - support for runtime checks against libraries of live subtypes;
  - validate connectors at runtime if necessary, updating subtype libraries accordingly;

8.2.2 Planned doctoral work

Doctoral work is currently being carried out that takes our work further:

- Humberto Castejón has provided a token passing-based[4] formal semantics for collaboration goal sequences [Castejón and Bræk 2006b], and demonstrated that a number of errors can be detected and corrected by analysing collaborations and goal sequences at a high level [Castejón and Bræk 2006a];
- Frank Krämer is exploring activity diagrams for more complete definitions of collaboration behaviour [Kraemer and Herrmann 2006], as well as formal definition of semantics using the compositional Temporal Logic of Actions (cTLA) [Kraemer et alia 2006]. This enables him to formally reason about service specifications using semantic connectors, and to define refinement steps from abstract collaborations to executable state machines;
- Fritjof Engelhardtsen has researched how calculus for Communicating Concurrent

2. [Birkeland 2005] did not port the progress calculation algorithms implemented by [Alsnes 2004]. However, these have since been integrated with Eclipse and the Ramses tool suite [RAMSES 2006].
3. The Ramses tool platform at the Teleservice Laboratory at NTNU [RAMSES 2006].
4. In our work we have used interleaving semantics for collaboration goal sequences and role goal sequences, and token passing semantics in actor goal sequences.

Systems (CCS) and stuck-free conformance can be used as a formal fundament for semantic interfaces [Engelhardtsen and Prinz 2006];

- Judith Rossebø is exploring collaborations as a framework of authentication and authorization patterns for ensuring availability in service composition [Rossebø and Bræk 2006a][Rossebø and Bræk 2006b];
- Haldor Samset is applying the service concept used in our work to that of the service oriented architecture (SOA).

8.2.2.1 Formalising goal sequences to describe complete behaviour

Goal sequence diagrams illustrate how connections, i.e. structural relationships, evolve as a function of time and event occurrences. Goal sequence diagrams are concise: they focus on the achievement of goals, and not on the detailed events needed to achieve them.

There does not seem to be any limitation preventing description of the complete behaviour of services in this way. Indeed, the work of Castejón and Krämer has shown that goal sequence diagrams can be formalised and analysed:

- [Castejón and Bræk 2006b] proposes a formal semantics for collaboration goal sequences by means of hierarchical coloured Petri-nets, and shows how tools can be used to analyse goal sequences automatically in order to detect implied scenarios. Implied scenarios can be symptoms of errors;[5]
- [Kraemer et alia 2006] uses compositional Temporal Logic of Actions (cTLA) to formally reason about service specifications and their refinement. They bridge the gap between UML for modeling and design, cTLA specifications used for reasoning, and the efficient execution of services in order to prove important properties formally.[6]

8.3 Other areas for further work

Additional areas for future work include the following:

1. Investigating whether goal sequences can benefit the discovery and resolution of feature interaction problems. I.e. see whether conflicts can be detected and their resolution expressed in terms of goal dependencies and goal sequences. This can be:
    - at the level of actors (e.g. the conflict between service roles played by an actor that initiates a call at the same time as a call is received);
    - at the level of service roles (e.g. the conflict between a call being simultaneously released by two parties of the same call).
2. Find ways of expressing conditions that roles place on the actors playing them, i.e. to express service goals in terms of what actors are allowed to achieve them:

5. For instance in the *MpConf* example they would detect that *mpc* could be initiated by the controller before a participant has joined the meeting place. This may not be clear to the service designer. A resolution of this is for *mp* to accept the initiative from the *controller*, but in such cases to return *MpCnfNak* rather that *MpCnfNak* (see Figure 3.11 on page 52).
6. For instance they clarify formally that the executable code is a correct refinement of the executable service model in spite of the practical limitations of execution frameworks such as finite message buffers.

- A role should be played only by an identified actor, and none other (e.g. calling a peer without allowing any forwarding);
- A role should be played only by an identified actor or some other actor designated by the former (e.g. for call forwarding);
- A role should not be played by a set of identified actors (e.g. a screening list);
- A role can be played by any actor in a group (e.g. group hunt);
- A role can be played by any actor that is capable (e.g. a client-server service).

The latter point may be addressed by organisational roles, as discussed below.

8.3.1 Using organisational roles to assign actor behaviour dynamically

What services an actor should play can depend on what role the actor plays in a wider context. One can speak of an actor having an *organisational role*, and that to certain organisational roles there are associated certain services.

An organisational role is a role in an organisation being played by one or more actors. Organisational roles may be addressed, have responsibilities, credentials and access rights. Organisational roles can be assigned to actors on a semi-permanent basis.

Characterizing actors by their organisational roles may be a way of extending context-based service deployment and service discovery. Organisational roles that are assigned to an actor can vary as a function of time and responsibility. In many work situations certain persons play particular organisational roles, such as nurse on duty, commanding officer, etc. Which person plays these roles at any particular moment varies during the course of a day or week. The services they perform can vary in these different roles. Also, other peer actors may require that certain interface roles are played by an actor for some desired services to be performed. E.g. it should be possible to place a “priority call to nurse in charge” without the caller having to know which person or which device is the addressee.

Assigning organisational roles to actors implies that service roles are also assigned. A promising future for services can be dynamic assignment of organisational roles to actors, combined with dynamic role learning and discovery of service opportunities.

9

Appendix A - Alternative UML modelling

In this appendix we discuss an alternative approach to the modelling of service structures that was initially attempted, which was to exploit associations (or rather association classes). It is included as auxiliary material for sake of discussion.

9.1 Modelling services as UML association classes

Before we chose to represent services as collaborations, we investigated other modelling elements in UML2, but which turned out to be too restricted:

- UML interfaces can describe a classifier in terms of operations, but are not well suited to describing two-way asynchronous signal transfer; defining a required interface for sent signals and a provided interface for consumed signals is possible, but the relationships between the two cannot be described in one place separate from the classifier. For a discussion of this, see section 3.3.4;
- We evaluated thoroughly the use of UML association classes, and identified a number of issues. The fact that associations and association classes are inflexible in terms of role binding to actors is the main obstacle to their systematic use; the discussion of this is detailed below. One motivation for exploring this option was the existence of tool support for these constructs.

The modelling of objects and their relations in class diagrams is by many held as the most essential and important modelling view in UML. Seeing services as collaborations or joint actions, it would be natural to consider modelling service structures as associations, or association classes, between classes. In UML, *“associations are the ‘glue’ that holds together a system model. Without associations, there is only a set of isolated objects”*.[1]

However, as we shall see, associations do not fulfil our requirements. The main reason is the lack of flexibility at the association ends, resulting in a rigid binding between association classes representing services and classes representing actors. Other limitations regarding n-ary association classes are also discussed: it seems these can be overcome.

Note that associations have a dual use in systems modelling: modelling knowledge aspects, i.e. entity-relationship modelling of IT systems, and modelling the communication aspect representing signalling channels and call paths in ICT systems. It is the latter that concerns us.

1. According to [UML2 Ref] p. 174.

9.1.1 Collaboration goals in association classes

Rather than using UML *associations* to model services, we focused on the capabilities of *association classes*, which combine the collective properties of associations and classes.

The motivation for pursuing association classes rather than associations is that the UML *class* properties, which association classes contain by inheritance, can be used to provide a placeholder for boolean attributes representing collaboration goals, to which one can attach predicates that characterize goal achievement.[2]

Association classes inherit the model elements and semantics of UML *associations*, which can be used to represent links between instances of the associated classes.

The approach to modelling service structures using association classes is as follows:

- Let *classes* represent the actor types involved in a service. Let these classifiers have attributes and states that are service specific, including *goal assertions*;
- Let *association classes* represent service types, thereby naming the services. Let association classes capture the involvement of the end classifiers in connectors by attachment, and by identifying *role names*[3]. The association ends thus provide a name and a type[4] for the role the actor plays in the service;
- Let the instances of the association classes define connections between actor instances; i.e. the links represent connections between the involved actors;
- Let the multiplicity of the association ends enumerate the roles played, thus defining cardinality constraints on connections;
- Let the association class contain boolean *goal* attributes representing the *collaboration goals*, i.e. the status of the collaboration goal achievement (*True* or *False*);
- Let service-specific *attributes* in the associated classes express the situations (states or other conditions) used to express *goal assertions*, in terms of knowledge (data available to the class, including its own state, timer values etc.);
- Let goal assertions be expressed in OCL.

We exemplify below the attempted approach, first with simple, two-party services, then with a multi-party case.

9.1.2 Two-party services

The outline of a generic model of two-party services using binary association classes is illustrated in Figure 9.1.

Here the service is represented by the association class, the actors by classes and the service roles by association end names. Association ends define the multiplicity of the actors playing the roles.

2. An alternative is to define a stereotype of association containing collaboration goals.
3. The UML1 term *role name* is not used in UML2; [UML2 Ref] still uses the term, while the proper UML2 term is *association end name*, which is actually the *name* of the *property* of the *member end* of an *association*.
4. The type of the property is the type of the end of the association. The type is not visible in the class diagrams.

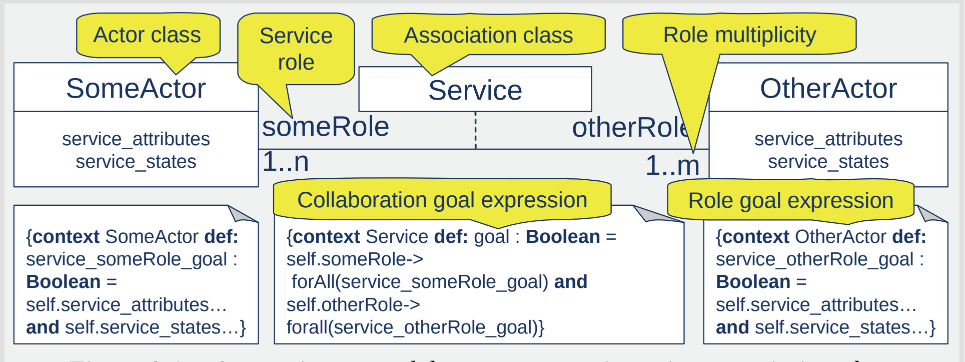

Actor class
Service role
Association class
Role multiplicity
SomeActor
service_attributes
service_states
Service
someRole
otherRole
1..n
1..m
OtherActor
service_attributes
service_states
Collaboration goal expression
Role goal expression
{context SomeActor def: service_someRole_goal : Boolean = self.service_attributes… and self.service_states…}
{context Service def: goal : Boolean = self.someRole-> forAll(service_someRole_goal) and self.otherRole-> forall(service_otherRole_goal)}
{context OtherActor def: service_otherRole_goal : Boolean = self.service_attributes… and self.service_states…}

Figure 9.1 : Attempting to model two-party service using association class

We define the goals of each role in OCL. Goal assertions are expressed using local attributes and states, while the collaboration goal lodged in the association class is a conjunction of the role goal expressions, as indicated in the bottom of Figure 9.1.

Examples of two-party services are presented below, each highlighting different types of services, and thus different aspects of the approach.

9.1.2.1 Example 1: Call

The century-old classical two-party service, the telephone call, highlights the establishment of a connection between two peers or users, involving media streams, i.e. the voice data exchanged between the peers.

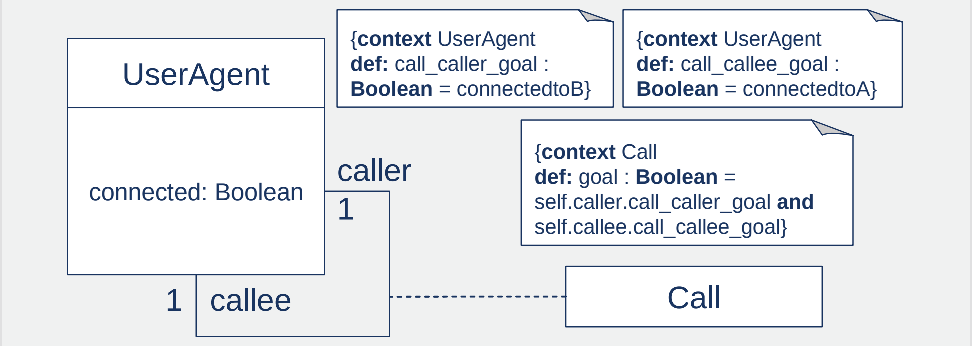

UserAgent
connected: Boolean
{context UserAgent def: call_caller_goal : Boolean = connectedtoB}
{context UserAgent def: call_callee_goal : Boolean = connectedtoA}
{context Call def: goal : Boolean = self.caller.call_caller_goal and self.callee.call_callee_goal}
caller
1
1
callee
Call

Figure 9.2 : The Call service modelled by an association class

Figure 9.2 is a model of the *Call* service, abstracting away all implementation details. It focuses solely on the relationships between the peers, who in this case are modelled as instances of *UserAgent*.

The collaboration goal is captured by the Boolean attribute *goal* defined in OCL within a note. It is a conjunction of two role goals. In this case the *UserAgent* actors play the traditional role *caller* (the user placing the call) and *callee* (the user being called).

Both actor classes have a service-specific attribute named *connected*. Their respective goals are reached when this attribute evaluates to *True*.

Multiplicity constraints express that exactly one *caller* and one *callee* participate in a call. They do not say how many simultaneous calls are available for each actor. Whether a *UserAgent* has the capacity to engage in two simultaneous calls or not is not modelled by the association class.

The service structure in Figure 9.2 can be used to express constraints for the *Call* service:

> {**context** Call **inv** uniquePeers :
> self.callee <> self.caller}

and the collaboration goal:

> {**context** Call **def:** goal : **Boolean** =
> self.caller.call_caller_goal **and** self.callee.call_callee_goal}

The first constraint is an example of a general constraint. The invariant states that the *caller* and the *callee* must be unique, i.e. a *UserAgent* may not call itself. This is an example of using OCL to express general statements about valid models.

The second OCL expression is the definition of a collaboration goal. It simply states that the *Call* service as a whole has achieved its goal when the role players have achieved their goals simultaneously.[5]

9.1.2.2 Example 2: Messaging

A model of a rudimentary messaging service is depicted in Figure 9.3.

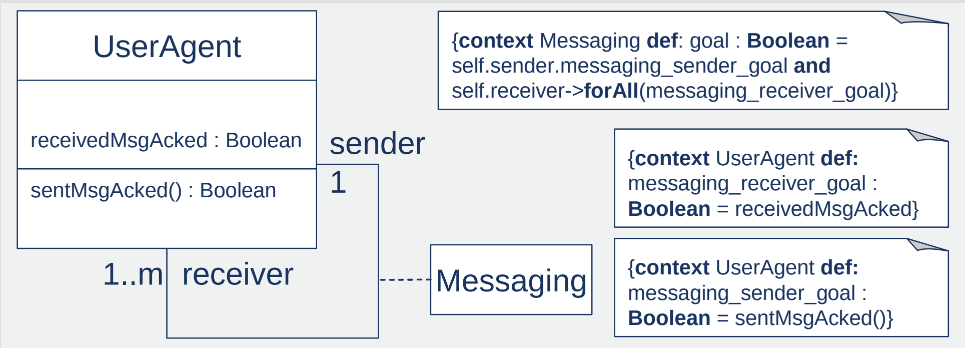

UserAgent
receivedMsgAcked : Boolean
sentMsgAcked() : Boolean
sender
1
1..m
receiver
Messaging
{context Messaging def: goal : Boolean = self.sender.messaging_sender_goal and self.receiver->forAll(messaging_receiver_goal)}
{context UserAgent def: messaging_receiver_goal : Boolean = receivedMsgAcked}
{context UserAgent def: messaging_sender_goal : Boolean = sentMsgAcked()}

Figure 9.3 : Messaging service modelled by an association class

The sender's goal is achieved when the receivers have acknowledged the reception of a sent signal (details not shown). The receivers' goals are achieved when a signal has been received, and an acknowledgement sent. Cardinality constraints express that a signal can be sent to multiple receivers (multicast or broadcast).

9.1.3 Multi-party services

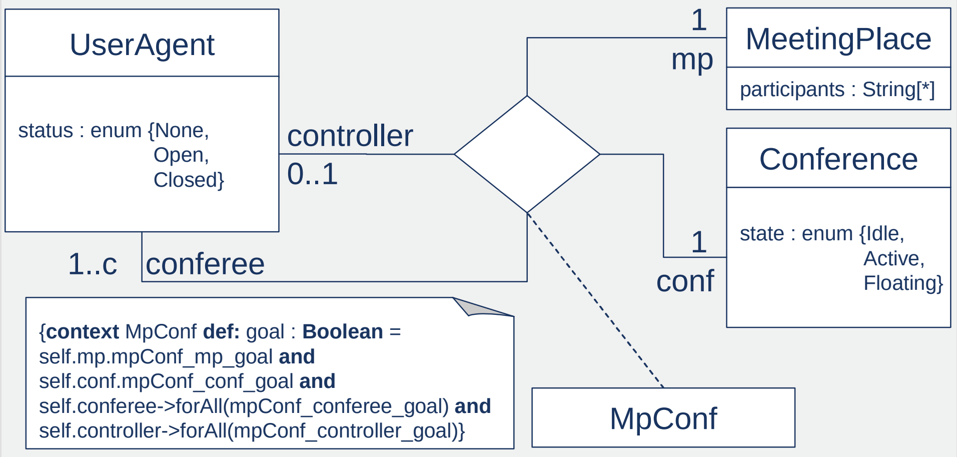
We have so far presented services involving two actors modelled by binary association classes. We know that some services involve more than two actors and/or more than two actor types. Such services were indicated in Figure 1.3 on page 4.

5. Meaning that both role goals are *True* at the same time sometime during the connection.

The question now is how we can model a multiparty service using associations or association classes.

9.1.3.1 Modelling multiparty services using n-ary association classes

We first attempted to represent a multiparty service by an n-ary association class, which combines the properties of an n-ary association (depicted as a diamond) with class properties detailed in the class symbol attached to the diamond[6], see Figure 9.4.



{**context** MpConf **def:** goal : **Boolean** =
self.mp.mpConf_mp_goal **and**
self.conf.mpConf_conf_goal **and**
self.conferee->forAll(mpConf_conferee_goal) **and**
self.controller->forAll(mpConf_controller_goal)}

Figure 9.4 : Meeting Place Conference as an N-ary association class

A Meeting Place Conference involves one Meeting Place and a number of its participants as *conferees* of a conference. There is a Conference *controller* (usually the Meeting Place controller) that “owns” the conference; the controller is allowed to leave, placing the conference in a Floating state. There must be at least one conferee in a conference. Only Meeting Place *participants* are welcome to join the service.

The role goal expressions for each of the four roles are suppressed to save diagram space; only the collaboration goal is included as a conjunction of the role goals.

The n-ary association class represents the whole service. All service roles and actor types that can take part in the service are joined by the n-ary association class. Compared with a binary association class most things simply scale up, e.g. the collaboration goal being a conjunction of all the goals of the roles, as shown in the note in Figure 9.4.

However, one issue that complicates the situation is related to the interpretation of multiplicity of n-ary associations, and thus also to n-ary association classes. A discussion on this is found in section 9.1.4.3.

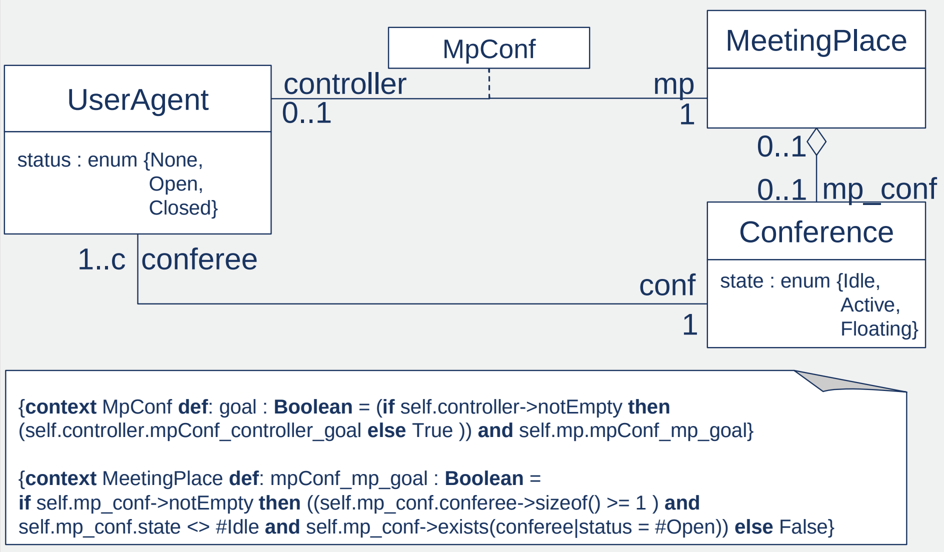
We suggested overcoming these limitations by imposing our own semantics on n-ary association classes, letting multiplicity express the legal range of instances that can play a role, enabling both optional (0..1) and multiple (n>1) ranges to be expressed.

6. This is the graphical symbol according to [UML2 Ref]; the UML standard does not define the precise layout.

9.1.3.2 Modelling multiparty services using a set of associations

In Figure 9.4 we represented the multiparty service *MpConf* using an n-ary association class. Alternatively we could revert to collections of binary associations, which have a clearer semantics than n-ary associations, see discussion in section 9.1.4.3.

In Figure 9.5 we have revised the *MpConf* service from Figure 9.4.



{**context** MpConf **def**: goal : **Boolean** = (**if** self.controller->notEmpty **then** (self.controller.mpConf_controller_goal **else** True)) **and** self.mp.mpConf_mp_goal}

{**context** MeetingPlace **def**: mpConf_mp_goal : **Boolean** = **if** self.mp_conf->notEmpty **then** ((self.mp_conf.conferee->sizeof() >= 1) **and** self.mp_conf.state <> #Idle **and** self.mp_conf->exists(conferee|status = #Open)) **else** False}

Figure 9.5 : Meeting Place Conference as a set of associations

The goals for three of the four service roles are suppressed to save diagram space. Note how the collaboration goal uses the navigation capabilities of OCL to reach all role goals: it tests whether the *controller* is present; if it isn't, the overall goal value is determined only by the *mp* role goal. This goal assertion first checks whether a *Conference* is linked to the *Meeting Place*; if it isn't, then the goal is not reached. If a *Conference* is in place, it checks that there is at least one *conferee* present, that the conference status is *Active* or *Floating* (i.e. not *Idle*), and that at least one of the *conferees* is in the *Open* state, implying that its media stream is connected to the conference. Given all these conditions, the service goal is fulfilled.

Here the *MpConf* service uses solely binary associations to model relationships between the roles and actor types involved in the service. The association classes used in this case express more general relationships between the actor types; for instance that conferences may or may not be contained in a Meeting Place, and that a conference has at least one conferee. However, this information is not a necessary part of the service structure.

More importantly we note that the actors playing the *conferee* and *conf* roles are not visible in the overall goal expression. A correct evaluation of the collaboration goal relies on *MeetingPlace* including a goal expression *mpConf_mp_goal* containing a correct evaluation of the other goal assertions.

9.1.3.3 N-ary association classes versus sets of binary associations

We examined two ways of modelling multiparty services, firstly using n-ary association classes, and secondly using a set of binary associations.

The main drawback to using a set of binary associations is that they conceal the fact that a service involves multiple roles. This can be seen in Figure 9.5, where it is possible only to understand the implications of the *conferee* role if one studies the OCL expressions carefully. This does not seem satisfactory. N-ary association classes, as in Figure 9.4, on the other hand, are capable of explicitly naming all the roles involved in the service.

9.1.4 Discussion

Associations and association classes go some way towards providing the expressive power and formalism needed to model the horizontal inter-relationships between service roles. They seem to name services and roles adequately, to identify role players, and to offer placeholders for simple service goal expressions.

However, associations and association classes proved to be too limited for our modelling purposes:

- The important shortcomings that make them unsuitable are firstly their limited ability to express collaboration behaviour, and secondly their lacking role-playing flexibility;
- There are also some idiosyncrasies concerning multiplicity in association classes and n-ary associations, although there seem to be ways of overcoming them.

The following subsections discuss each issue in particular.

9.1.4.1 Role binding flexibility

The decisive argument against using association classes to model services is the lack of flexibility at the association ends, in terms of a loose binding to the classifiers, i.e. to actor types. This was pointed out in [Bræk 1999], and is still the case in UML2. The metaclass *Property* that bridges an association and a class is either contained in the class, meaning it is an attribute owned by the class, or it is contained by the association[7]. A property may be associated to a classifier, but only with optional (0..1) multiplicity, and is included in the UML metamodel to capture the property's optional redefinition context, not to enable different classifiers to associate with a property.

In Figure 2.5 on page 30 we summarized the relationships between services, roles and actors, and specified the requirement for a role to be played by several actor types. With associations or association classes it is not possible to express the fact that different kinds of actors can play the "role" at the association end. In fact, an association end isn't really a *role*. It is just called that in approximate terms when speaking of the expressive power of class diagrams, and is the result of the imprecise use of the term "role" in earlier versions of UML[8].

7. *When a property is owned by a class it represents an attribute. In this case it relates an instance of the class to a value or set of values of the type of the attribute. When a property is owned by an association it represents a non-navigable end of the association. In this case the type of the property is the type of the end of the association.* [UML 2.0 Adopted] p. 89.

Instead we turned our attention to a new modelling form that has been introduced by UML2, namely *collaborations*. As we see from chapter 3, UML2 collaborations support the kind of flexibility between roles and role players that we seek in service structures:

- Compare the *Call* service in Figure 9.2 on page 183 with the same service modelled in a collaboration in Figure 3.7 on page 50. Firstly, it is clear from the latter that two roles are involved in a *Call*. But more importantly, the role players have not been identified. Any actor that is able to play the *A* and *B* roles of *Call* could potentially be acceptable;
- Compare the *MpConf* service modelled in an n-ary association class in Figure 9.4 on page 185 with the same service modelled in a collaboration in Figure 3.12 on page 53. In the latter the role players have not been identified. Modelling multiparty service structures with collaborations is not hampered by the limitations of (n-ary) association classes. That the *controller* role is optional, as indicated by the multiplicity (0..1), and that several *conferees* can be present, is not a problem for service structures modelled by collaborations.

This does not mean that one should categorically rule out the use of association classes to model services. We recognize that for the near future tool support for the new elements of UML2 will be lacking in functionality. In addition, many practitioners may feel comfortable considering the properties that bind association ends to classes as roles.

Note that [ARTS 2003] discuss whether collaboration uses are too limited when used to compose systems[9]. However, since synthesis of behaviour is not an issue in this thesis, these reservations are not of concern to our work.

9.1.4.2 Multiplicity of association classes

One limitation in the UML concerns multiplicity in the use of association classes: "*in an instance of an association class, there is only one instance of the associated classifiers at each end, i.e. from the instance point of view, the multiplicity of the associations ends are '1'*" [UML 2.0] p. 43. So there is no way of expressing the fact that there are "one or more *receivers*" in terms of the association class *Messaging*. The OCL constraints will not navigate properly.[10]

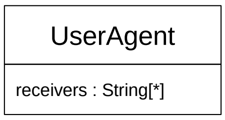
There are ways of overcoming this restriction in UML. One is to use a two-step navigation of the association class. For instance, in the *Messaging* service introduced in section 9.1.2.2, to get hold of the set of signal receivers and not just one receiver, we navigate first to the sender, and then to the set of receivers, using OCL navigation mechanisms.

8. The UML reference manual [UML2 Ref] uses the term *rolename* for the name of a particular association end. The official term is *association end name*. OCL 2.0 also talks of classes playing roles defined by associations; these" role names" are used to determine the navigation path in OCL navigation expressions [OCL 2.0] p. A-3.
9. *Constraints: [1] All the client elements of a roleBinding are in one classifier and all supplier elements of a roleBinding are in one collaboration and they are compatible. [2] Every role in the collaboration is bound within the collaboration use to a connectable element within the classifier or operation. [3] The connectors in the classifier connect according to the connectors in the collaboration.* [UML 2.0] p. 167.
10. "Navigation from an association class to one of the objects on the association will always deliver exactly one object. This is a result of the definition of AssociationClass. Therefore, the result of this navigation is exactly one object, although it can be used as a Set using the arrow (->)." [OCL 2.0] p. 19.

Using this approach, the service goal of Figure 9.3 can be rewritten as follows:[11]

```
{context Messaging def: goal : Boolean =
self.sender.messaging_sender_goal and
self.sender.receiver->forAll(messaging_receiver_goal)}
```

Another way round it is to include attributes in the entity classes representing the sets, e.g.:



and express the goal satisfaction in terms of such attributes. This leads us to disregard the restriction in OCL regarding multiplicity of roles for association classes, and take the view that more than one instance can be modelled.

9.1.4.3 n-ary associations

Another modelling limitation is concerned with the multiplicities of n-ary associations. What do the multiplicities in, for instance, Figure 9.4 on page 185 really mean? We intended to express that in any given *MeetingPlaceConference*, there must be exactly one *Conference* and one *MeetingPlace*, with one or more *conferees* present, while the presence of the single *controller* is optional. However, the interpretation of n-ary associations in general is not clear, with semantics as divergent as *actual tuples*, *potential tuples* (with its "bouncing effect of the one") and "*limping links*" [Gonzalo et al 2002]. Indeed, UML2 states: "*the lower multiplicity for an end of an n-ary association of 1 (or more) implies that one link (or more) must exist for every possible combination of values for the other ends*" [UML 2.0] p.38.

In our case this means that if we want to express that at least one conferee must participate in a *MeetingPlaceConference*, and use an n-ary association to do so, we end up saying that a connection (a link) for this *UserAgent* must exist for every possible combination of *MeetingPlace* and *Conference*. This is certainly not the case in the world we are modelling: not all *MeetingPlaces* have *Conferences*, and not all *UserAgents* can be members of all *MeetingPlaces*.

We let multiplicity express the legal range of instances that can play a role, and let both optional (0..1) and multiple (n>1) ranges be expressed. We suggest the semantics of n-ary association to be "actual tuples", and viewed from the vantage point of the service, in this case the association class.

We suggest not following the "zero-forbidden effect" of actual tuples noted by [Gonzalo et al 2002]: the multiplicity of the *controller* in Figure 9.4 is 0..1, implying that in the context of the *MpConf* service represented by the association class, there can be at most one *UserAgent* playing *controller*, but that it needn't be there, i.e. it is free to stop playing the *controller* role. This fact is explicitly expressed in the OCL goal expression[12], where the goal assertion for *controller* is only evaluated if it is present. This interpretation of the semantics is chosen to reflect the reality we are modelling.

11. The underlined text marks the change. Since this modification is trivial, we have refrained from using it in the OCL examples, to keep the expressions simple. The intentions of the OCL expressions should be clear.

9.1.4.4 Expressing collaboration behaviour and collaboration goals

We modelled the satisfaction of the collaboration goal by including *goal* attributes in the association class representing the service. In the association class we can only express simple collaboration goals in the form of predicates in terms of the roles and their goals. It is not possible to attach behaviour to associations; all they represent are structural properties, such as the participation of the end classifiers in links. It is in principle possible to relate behaviour to association classes, since they inherit from class. But there seems to be no meaningful way of expressing the behaviour of an association class either separate from or related to the behaviour of the classes at the association ends[13].

This implies that all service behaviour must be expressed in terms of the participating classes. Understanding the cross-cutting behaviour of the service is not possible.

An additional limitation is that the signalling interface at the association ends cannot be properly defined. Only provided and required operations can be defined, and this does not satisfy the requirements of the service modelling.

9.1.4.5 Modelling services as associations versus collaborations

A UML2 collaboration models a collection of objects that play roles within a transient context in order to implement a desired functionality.

The difference between associations and collaborations is pointedly discussed in the second edition of the UML reference manual:

> *Contrast the restricted scope of a role with an association: An association describes a relationship that is globally meaningful for a class in all contexts, whether or not an object actually participates in the association. A collaboration defines relationships that are restricted to a context and that are meaningless outside of that context.*[14]

Collaborations being constrained by a context fits in well with the concept that communicating peers play certain roles in a certain context; in another connection between the same actors the roles (e.g. *caller* and *callee*) may be swapped. It is the flexibility to specify the context that is the attraction of UML2 collaborations. Conversely, the lack of support for flexible role binding is a limiting factor of UML associations.

On the other hand, the “ability to play a role” is something more permanent than a connection, and can well be represented by an association, e.g. that *he* can call *her.* This kind of entity-relationship modelling (representing knowledge) lies outside the scope of our work.

Compared with associations and association classes, UML2 collaborations offer the desired flexibility between roles and actors, and support the definition of overall behaviour of the service in addition to role behaviour.

---

12. Note that OCL has its own mechanism for navigation of n-ary associations. In OCL, navigation operations start from an object of a source class and retrieve all connected objects of a target class. A n-ary association induces a total of n * (n - 1) directed navigation operations, because OCL navigation operations only consider two classes of an association at a time [OCL 2.0] p. A-15.
13. The UML documents do not mention behaviour of association classes, nor discuss the semantics of it.
14. [UML2 Ref] p. 228.

10
Appendix B - Open issues

In our thesis we have proposed a modular approach to service modelling based on semantic connectors and semantic interfaces. The modelling approach includes the expression of service goals, role behaviour and goal sequences. We have discussed how this enables a modular approach to validation, and suggested how service discovery can gain from it.

In this appendix we list a number of open issues that have been identified.

10.1 Service structures

In chapter 3 we suggested that services be modelled by collaborations composed of semantic connectors. Below we discuss two open issues regarding service structures:

- How to express interface roles that are external to a collaboration;
- How to express connections between more that two roles.

10.1.1 Connectors external to a collaboration

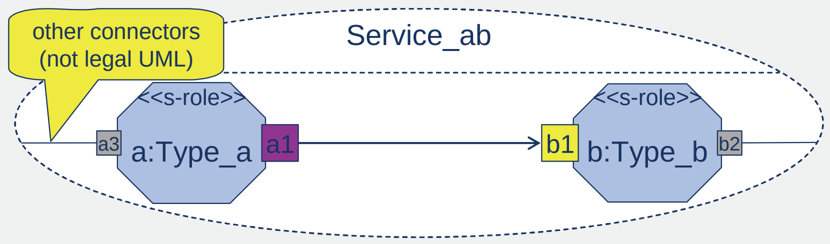


Figure 10.1 : Interface roles external to a collaboration

In Figure 10.1 we have indicated that the service roles *a* and *b* have interface roles and connectors that interact with entities outside the collaboration (semantic interfaces *a3* and *b2*). This is not legal in UML, and is included for sake of argument. Although modelling all possible relationships is needed in order to analyse and understand the full behaviour of a service role, it is not in general possible to include them all in a single UML2 collaboration: a collaboration describes interactions between the roles of that collaboration only; other roles and connectors must be defined in additional collaborations and referenced by collaboration uses. This issue was researched by [Fuglesang 2005], who suggested notational additions to UML to express external roles.

10.1.2 N-ary connections

We have restricted connections to being between two and only two roles. The motivation for this is to simplify validation and service discovery, while arguing on the other hand that arbitrarily complex service structures can be composed of semantic connectors.

However, there is a need to express n-ary connections as a single entity. The *MpConf* service is a good example. Three of the service roles are unary, while the *conferee* role is n-ary. Clearly there is a mismatch between the unary collaboration occurrences bound to *conferee*, and the multiplicity of this role.

To the best of our understanding, UML has no mechanism for specifying an array of collaboration uses. A possible solution to this in suggested in Figure 10.2.

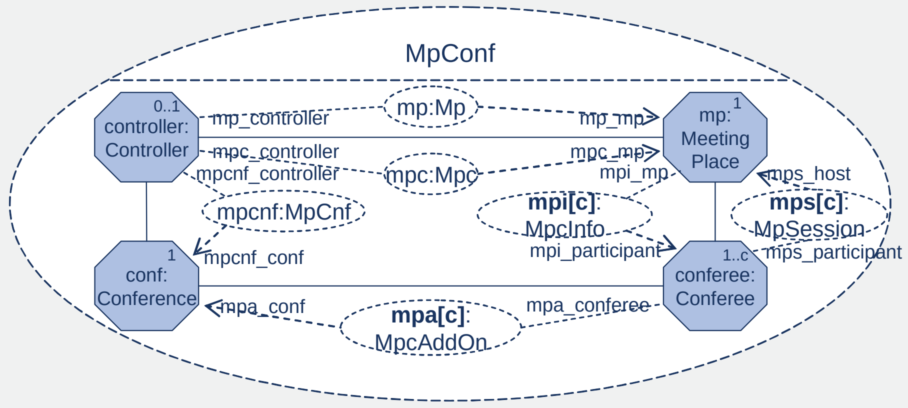


Figure 10.2 : Collaboration composed of sets of semantic connectors

Figure 10.2 uses a set notation for the collaboration uses *mps*, *mpi* and *mpa* that are bound to *conferee*. This is inspired by the way SDL defines sets of agents. However, while this technique seems workable in the collaborations, referring to members of sets of collaboration uses become cumbersome where interaction diagrams and interaction overview diagrams are concerned, and hence the same applies to the goal sequence diagrams we have suggested. A possible solution to this is shown in Figure 10.3.

The goal sequence diagram in Figure 10.3 contains loops for the n-ary collaboration uses (representing interaction uses). While interaction overview diagrams support loops, the loop controller construct and the identification of members of the set is not supported in UML2[1]. However, while this might seem an attractive extension for representing the multiplicity of connectors for n-ary roles, there are many issues that need to be resolved. For instance, the bottom two interaction uses do not require that the loops for these interaction uses only apply to set members that have achieved their goals.

Loops are simple in programming languages, but not in specifications. One challenge lies in the identification of objects in sets, and on referencing them consistently in different language constructs. Another is the parallelism introduced.

1. Loop constructs in UML interaction occurrences are in the form *loop (min, max)*; referencing a member of a set is not supported.

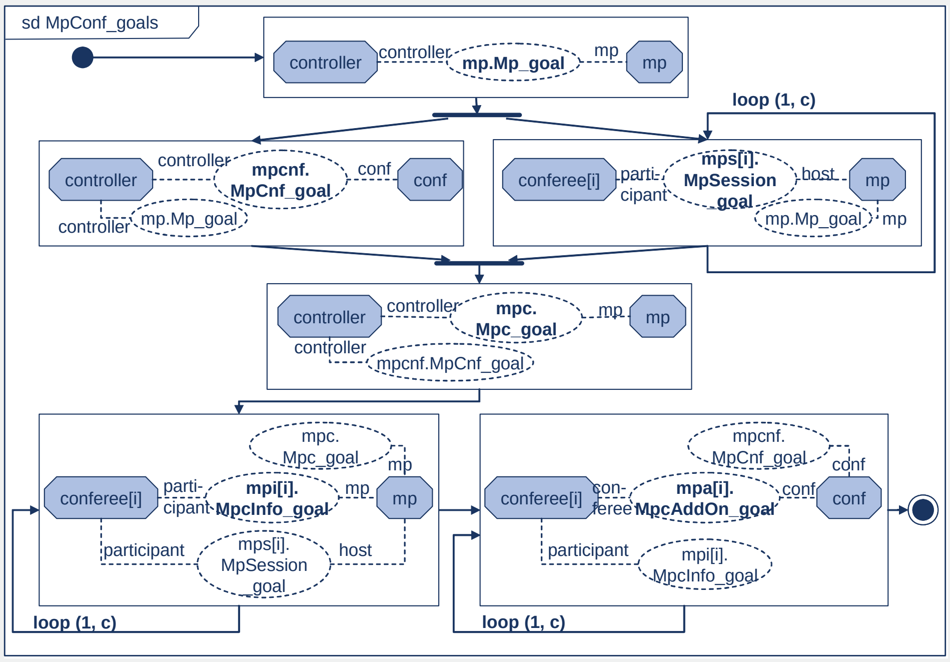


Figure 10.3 : Goal sequence diagram with loops

While both UML2 and MSC [MSC 2004] have loop operators in inline expressions of interaction fragments, and in UML2 interaction overviews, these cannot be used to index instance sets. The issue of instance sets seems to be complex; we note that they are not properly supported by MSC, despite the obvious need when describing SDL systems.

10.2 Goal sequences

In chapter 5 we have suggested goal sequence diagrams as a way of expressing a combination of horizontal and vertical relationships. We argue that they adequately describe essential service relationships, i.e. goal relationships. That actor goal sequences lend themselves to context-dependent service discovery is a further argument in their favour.

While many modelling objectives are achieved using goal sequences, both the diagram forms and the underlying semantics require further work before they reach the maturity needed for any widespread adoption. We discuss some of the open issues below.

10.2.1 Goal dependencies for service roles

Since a number of interface roles can be bound to a service role, service roles can obviously have more than one role goal. However, in role goal sequences we define the sequence of interface goals, not sequences of goals of service roles.

We have considered modelling goal dependencies for service roles, in the same way as goal sequences that describe goal dependencies for collaborations, interface roles and actors. An example is shown in Figure 10.4.

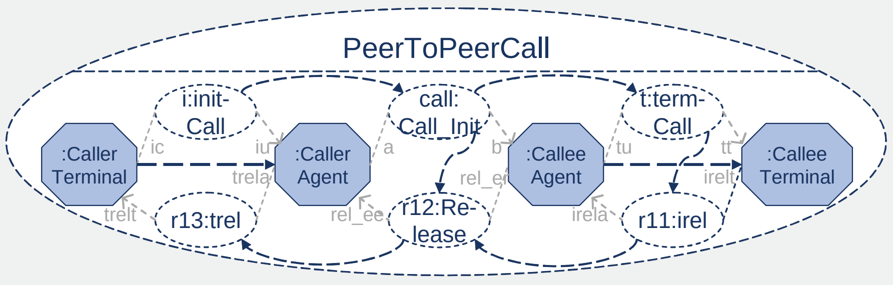


Figure 10.4 : Goal dependencies for service roles (not recommended)

Figure 10.4 shows goal dependencies for a call. Here four service roles are involved. Goal dependencies between semantic connectors are depicted in curved dashed arrows. The top row of dependencies shows how a call setup progresses from left to right, while the bottom row shows a call release initiated by the callee terminal progressing from right to left. Here we have introduced *service role goal dependencies*, using straight dashed arrows.

However, it seems that nothing is gained from describing these dependencies:

- If goal dependencies between service roles are *not* qualified by a goal name, see Figure 10.4, then they cannot capture both the goal dependencies of the initiating call (left to right) and the call release (right to left), since dependencies are unidirectional. I.e. they would express less than goal relationships between collaborations;
- If on the other hand goal dependencies *were* qualified[2] by a goal name, they would only be capable of capturing the same relationships as those expressed by goal dependencies between semantic connectors. I.e. they would add nothing new.

We have opted to do without modelling such dependencies in order to prevent unnecessary modelling elements and design steps. Some of the goal relationships that are relevant for service roles can instead be expressed by collaboration goal sequences.

However, it could be desirable to express the conflict resolution that service roles perform. The *Call* service provides classical examples of this: conflict resolution between making and receiving calls at the same time, and conflict resolution within a call, such as call release initiated by both parties simultaneously. If conflicts could be detected and their resolution expressed in terms of goal dependencies, then much could be gained.

10.2.2 Illustrating preceding goals

It is an open question whether it would be useful to distinguish between preceding and succeeding goal achievements of semantic connectors. We considered expressing this by:

- rendering the “current” collaboration use in boldface;
- rendering the preceding collaboration uses in normal typeface if the collaboration goal should *still be valid* when the succeeding goal or the composite goal is evaluated, and in grey-toned text and lines if otherwise.

2. Qualifying goal dependencies with a goal identity is exemplified in Figure 10.5, where the semantic connector goals setup.ringing and accept.connected are identified.

An example of where the preceding goal is not valid is shown in Figure 10.5.

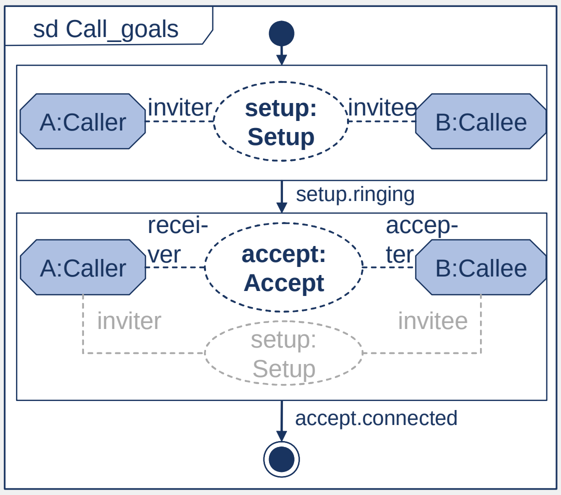


Figure 10.5 : Goal sequence diagram illustrating preceding goals (1)

Figure 10.5 illustrates the role binding between the instances of the preceding semantic connector *setup*, relative to the roles of the succeeding semantic connector *accept*, in the context of *Call*. The *ringing* goal of *setup* is no longer *True* when the goal of *accept* is achieved, so grey-toned text and lines are used when referencing *setup* in the bottom part.

An example of where the preceding goal is valid is given in Figure 10.6.

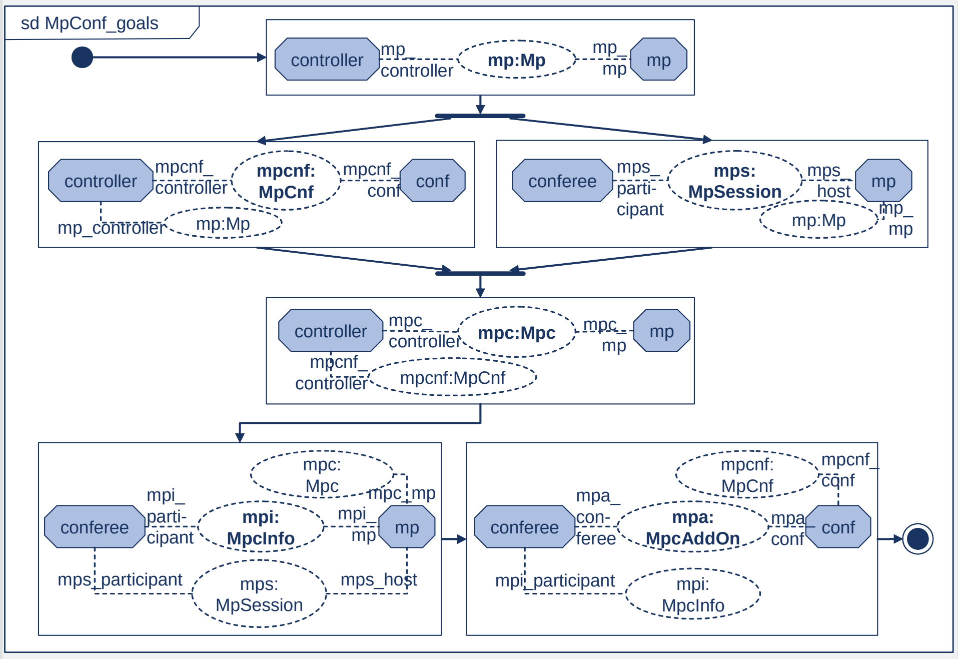


Figure 10.6 : Goal sequence diagram illustrating preceding goals (2)

Figure 10.6 illustrates the role bindings of the composite collaboration *MpConf* as it progresses towards its ultimate goal of achieving a conference. Here the goals of the subordinate semantic connectors are still valid, indicated by the use of normal typeface.

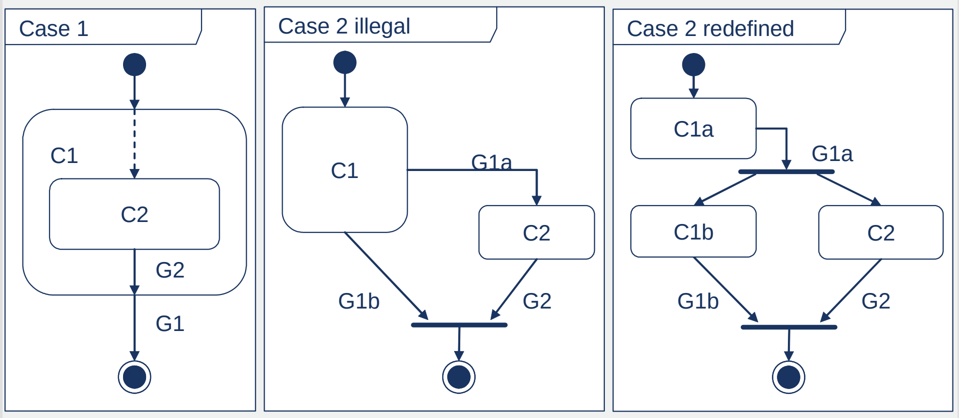
While one can argue that this makes the sequence of role bindings clear, it does not add anything new. The role binding is defined by the composite collaborations, and the goal sequence diagram is but an illustration of the sequence of role bindings as a function of goal achievement. Note that elements of goal sequence diagrams can only refer “backwards”, i.e. only references from succeeding semantic connectors to preceding semantic connectors are captured. This is necessary to be in line with the semantics of interaction overview diagrams.

It is not clear that this graphical expression form is worth the added complexity. For use in connection with service discovery it does not seem to be necessary to know the status of preceding service goals. On the other hand, if goal sequences are to be used to aid the construction of actor behaviour, then such information seems useful, as was pointed out in [Sanders et alia 2005b]. However, pursuing synthesis of behaviour may result in the demand for additional information in goal sequence diagrams beyond capturing the status of preceding goals, such as defining whether more events are possible after goal achievement.

10.2.3 Overlapping roles

We examined the properties of several UML concepts to describe goal relationships: dependencies, interaction overview diagrams and activity diagrams. These are good at expressing sequential and parallel relationships, but they fail to express finer relationships between intermediate goals, such as when two collaboration uses overlap.

In UML activity diagrams activities can be nested, so it is possible to show that, for example, *C1* only succeeds if *C2* also succeeds, by nesting *C2* inside *C1*, see Figure 10.7a.



a) Fully nested goals b) Partially nested goals (illegal UML) c) Reworking

Figure 10.7 : Nested goal dependencies

Consider another case, where *C1* starts, reaches a goal *G1a* that enables *C2*, and from then on both *C1* and *C2* run in parallel, with dependency on both reaching subsequent goals *G1b* and *G2*. This is not possible to model in UML, since activities cannot partially end; in Figure 10.7b *G1a* and *G1b* will fire simultaneously, according to the semantics of UML. To model this case correctly in UML, *C1* must be split in two, as in Figure 10.7c.

Splitting semantic connectors into fragments is a feasible work around, but not necessarily desirable. [Castejón 2005] has analysed Use Case Maps [UCM 2003] as an alternative to UML for modelling goal dependencies, and has shown that they provide better support in such cases.

[Krüger et alia 2004] support the composition of overlapping interactions using specific operators (e.g. *join*) to synchronize identically labelled signals and states. This is a different form of composition that we have not addressed.

10.2.4 Multiple role goals

Can semantic interfaces have multiple goals? In our presentation of goal sequences this was not discussed. If a semantic interface has several goals, it would be desirable to express the relationships between such goals in goal sequences.

The suggested approach to goal sequences can be developed to distinguish between multiple role goals for a given semantic interface. Goal dependencies can be adorned with the name of the goal achieved, see Figure 10.5 and Figure 10.8.

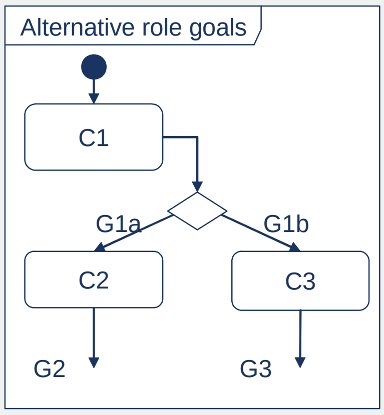


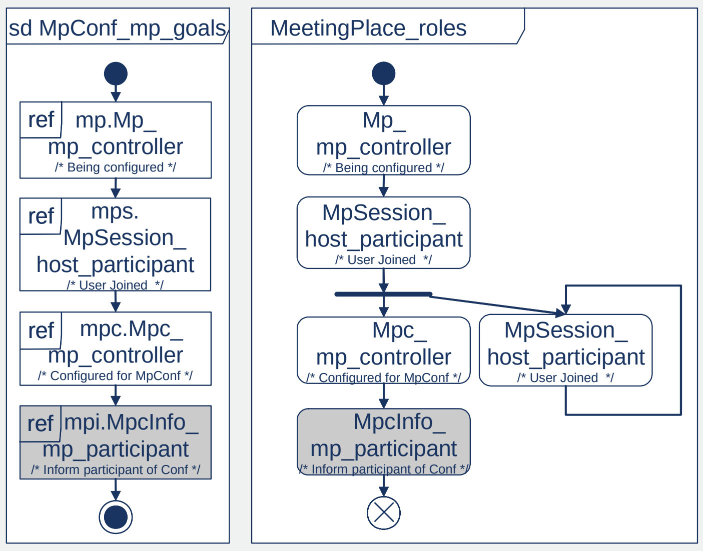
Figure 10.8 : Alternative role goals

Multiple role goals can be used to model alternative sequences, as in Figure 10.8, but not to express that a given semantic interface achieves a sequence of goals; the reason is that discussed in connection with overlapping roles above: we model semantic interfaces with activities, and they cannot have control flows that fire tokens at different times. Once a token has been fired, the activity firing has completed and will not fire again.

10.2.5 Modelling parallel role-playing

Actors are typically capable of playing several roles in parallel. Such role-playing flexibility was not expressed by the role goal sequences. One could argue that it should have been, since goal sequences are meant to describe properties adhered to by actors.

It is a methodological choice that we suggest that role goal sequences describe the goal relationships between roles, without describing the parallelism properties of role-playing Describing parallel behaviour is postponed to the actor goal sequences, see Figure 10.9.


sd MpConf_mp_goals
ref mp.Mp_ mp_controller
/* Being configured */
ref mps. MpSession_ host_participant
/* User Joined */
ref mpc.Mpc_ mp_controller
/* Configured for MpConf */
ref mpi.MpcInfo_ mp_participant
/* Inform participant of Conf */
MeetingPlace_roles
Mp_ mp_controller
/* Being configured */
MpSession_ host_participant
/* User Joined */
Mpc_ mp_controller
/* Configured for MpConf */
MpSession_ host_participant
/* User Joined */
MpcInfo_ mp_participant
/* Inform participant of Conf */

a) Role goal sequence b) Actor goal sequence

Figure 10.9 : Role goal sequences versus actor goal sequences

Compare for instance the role goal sequence and actor goal sequence of Figure 10.9. The role goal sequence does not define that several *MpSession_host_participant* roles can be played, nor that the *Mpc_mp_controller* role can be played in parallel (simultaneously) with *MpSession_host_participant*. This is instead expressed by the actor goal sequence, which defines the role-playing capabilities of the actor. This can include the capability of establishing several *MpSessions*. Not all actors need necessarily have this capability, though *MeetingPlace* does.

But why is *MpSession_host_participant* duplicated, while the other roles are not? If this is due to the nature of the *MpSession* service feature, should this not be captured in role goal sequences, rather than being postponed until the actor sequences? These are open questions that should be investigated.

10.2.6 Semantics of goal sequences

We have relied on trace semantics in interactions (e.g. collaboration goal sequences and role goal sequences, where goals pose constraints on lifelines representing service roles and interface roles, respectively), and token passing semantics in activities (e.g. actor goal sequences, where actions represent roles played by actors). As has been discussed, this does not pose very strict behavioural requirements on the actors; they are free to abandon preceding roles after initiating succeeding roles, and can chose to skip preceding goals when reaching succeeding goals without violating the goal sequences.

An alternative could be to incorporate some form of temporal logic in the goal sequences, some kind of MAY / MUST relationships between service goals in a sequence specifying the degrees of freedom between roles played by an actor. This is a candidate for future work, possibly exploiting elements of the approach of [Carrez et al 2004].

An open issue is the relationship between event goal expressed by state invariants on the lifelines of connector goal interactions, and constraints expressed by goal assertions. There may be a large set of valid event traces after goal achievement, despite that connector goal interactions should not specify events after goal achievement according to the method rule "Omit event occurrences after goal achievement" on page 116. Some valid post-goal interactions may maintain a goal assertion *True*, while others not.

An open issue is how to specify valid event traces after goal achievement in a succinct manner. Can this be achieved using interaction operators like *ignore* and *consider*? Also: Can interactions formalise relationships between state invariants and goal assertions?

10.2.7 State invariants to express goals

Connector goal interactions and role goal interactions use UML state invariants to indicate role goal achievement. In goal interactions we have considered distinguishing between *initial* and *terminal state invariants*, see Figure 10.10.

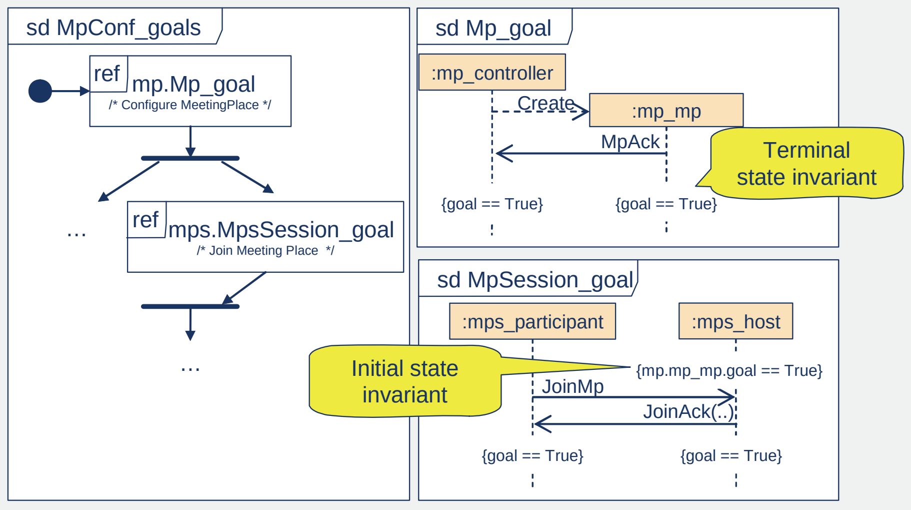


Figure 10.10 : State invariants as pre- and postconditions on role goals

These terms could be defined as follows:

Definition: Initial state invariant
An initial state invariant is a state invariant that is placed prior to any event occurrences on the lifeline of a goal interaction.

Definition: Terminal state invariant
A terminal state invariant is a state invariant that is placed after the last event occurrences on the lifeline of a goal interaction.

While terminal state invariants are useful for denoting goal achievement, the use of initial state invariants gives us both opportunities and challenges.

One of the opportunities lies in the use of initial state invariants to define invalid behaviour. Recall that the semantics of a UML interaction is given as a pair of sets of traces, where the two trace sets represent valid traces and invalid traces.[3]

State invariants in UML are evaluated immediately prior to the execution of the next event occurrence on the lifeline to which the constraint is attached.[4] E.g. in Figure 10.10 the state invariant *mp.mp_mp.goal == True* is evaluated when the event occurrence *JoinMp* is consumed by *mps_host*. Traces that have a state invariant with a false constraint are invalid traces. Thus, if the goal of the interface role *mp_mp* is not achieved when *JoinMp* is consumed by *host*, then the trace is invalid.

This implies that the event occurrences of *mps* in Figure 10.10 cannot occur prior to *mp* reaching its collaboration goal. This means that there is no way of reaching the role goal of *MpSession_host* without the role goal of *mp_mp* being valid first.

It is an open question whether this also implies that the succeeding semantic connector referenced in a collaboration goal sequence can *exist* prior to the preceding semantic connector reaching its goal. If the event occurrence defined in the goal interaction is the first event of the collaboration, then it seems reasonable that the collaboration cannot exist prior to the preceding semantic connector achieving its goal. Note however that a goal collaboration does not necessarily express all the events of a collaboration. I.e. there may be other initial event occurrences on the lifelines where a state invariant is placed.[5]

Could UML semantics demand that initial state invariants referenced in interaction overview diagrams are checked against terminal state invariants of preceding interaction uses for each lifeline? Consider the following method rule:

Method rule: Initial state invariants must be consistent with terminal state invariants
If a lifeline of a goal interaction has a set of initial state invariant(s), this set must be a subset of the set of terminal state invariants of all preceding goal interaction occurrences, if any. If not the goal sequence is invalid.

We exemplify this by referring to Figure 10.10. First we note that the collaboration *MpConf* has bound *mps_host* and *mp_mp* to the same lifeline, while *mps_participant* and *mp_controller* are bound to different roles. The goal interaction *MpSession_goal* has an

3. The union of these two sets does not necessarily cover the whole universe of traces; traces that are not included are inconclusive, i.e. one cannot know whether they are valid or invalid. See [UML 2.0] p. 468.
4. See [UML 2.0] p. 487.
5. Such events can be explicitly modelled by using the *ignore* construct in the interaction fragment.

initial state invariant *mp.mp_mp.goal == True* on *mps_host*, which when evaluated is indeed equal to (and thus a subset of) the terminal state invariant for the lifeline *mp_mp* in the goal interaction *Mp_goal*. The *mps_participant* lifeline has no initial state invariant, so this lifeline is also OK. We can conclude that the goal sequence *MpConf_goals* is valid.

A second refinement of UML semantics of interaction overview diagrams is to let terminal state invariants propagate to succeeding interaction occurrences:

Method rule: Terminal state invariants propagate to succeeding goal interactions
Terminal state invariants propagate to their respective lifelines in succeeding goal interactions referenced in goal sequence diagrams.

Again referring to Figure 10.10, this would mean that the lifeline *mps_host* in *MpSession_goal* would have an initial state invariant *mp.mp_mp.goal == True* propagated from the terminal state invariant *goal == True* related to the service role *mp*, which corresponds to the lifeline *mp_mp* in *Mp_goal*. (The propagated state invariant is in this case identical to the initial state invariant explicitly inserted). The lifeline *mps_participant* would not have any initial state invariant from the preceding interaction fragment, since *mps_participant* is bound to the service role *conferee* and not to *mp* or *controller.*

These possible refinements of UML semantics are based on the initial and terminal global conditions of MSCs in [MSC-92], and their interpretation by MSC documents. Combining two MSCs where the start condition of the second was a subset of the end condition of the first was legal. This was used to ensure correct combination of MSCs in high-level MSCs (HMSCs)[6]. We have in addition taken the interleaving semantics of UML interactions into consideration, as well as the semantics of state invariants.

Currently it is not clear from the UML2 semantics what the interpretation of state invariants is between interaction uses referenced in interaction overview diagrams. The refinements above seem to fit in well with both trace semantics and constraint resolution in UML2. The state invariants can be evaluated without side effects if they are confined to variables and states belonging to a (bound) lifeline.

Note that the method rule "Omit event occurrences after goal achievement" on page 116 enforces the rule that events occurring after goal achievement should not be included in goal interactions, ensuring that the interaction uses referenced in goal sequences have well-defined terminal state invariants. However, it may be that a collaboration use has additional event occurrences after a goal is achieved, although they are not included in goal interactions. An example are the *MpInfo* and *JoinInfo* signals of the collaboration uses *mp* and *mps*, which occur after the achievement of their role goals (see Figure 4.27 on page 97), i.e. interleaved with the event occurrences specified in the interaction *MpConf_goal* (see Figure 5.11 on page 115). This does not violate the semantics of UML.

Goal assertions can have a longer “lifetime” than state invariants; the latter are evaluated at a specific point of time, i.e. at the first event occurrence on the lifeline to which they are attached, after which they have no function. In goal sequences, such an “instantaneous” constraint validation corresponds to preceding collaboration uses rendered in grey

6. Initial and terminal conditions were replaced by *setting* and *guarding* conditions in [MSC-2000], which had a different syntax and semantics.

(as in Figure 10.5), i.e. preceding goals that do not have to be valid when a succeeding goal is reached. “Longer lifetime” goals could be included in the succeeding goal assertions, and be used in subsequent goal evaluations.

10.2.8 Alternative forms of goal sequence diagrams

The goal sequence diagrams we have suggested constitute a considerable departure from the interaction overview diagrams upon which they are based. It is not likely that UML tool vendors will offer support for such a diagram; replacing the name of the referenced interaction fragment with a graphic rendering the collaboration uses and role bindings will not be easy to implement in CASE tools.

Here we discuss two alternative forms: goal sequences expressed by state machine diagrams, and goal sequences expressed by a special form of interaction diagram.

10.2.8.1 Goal sequences expressed by collaboration state machines

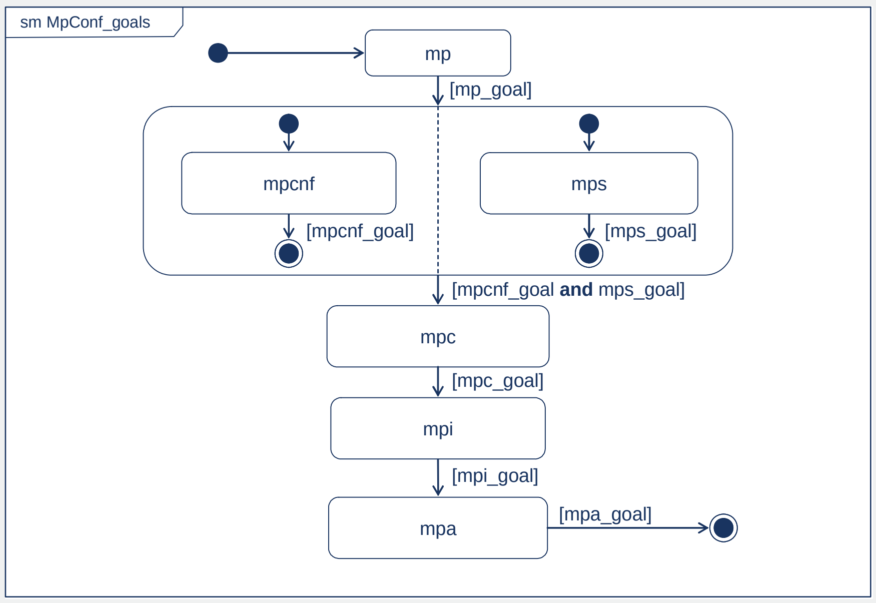


Figure 10.11 : Goal sequence diagram as a state machine

Collaborations in UML2 are behavioural classifiers, and can thus have associated behaviour defined by state machines. If we consider the semantic connectors to be in a state when they are active, and exit from this state when they achieve a goal, we could express goal sequences in a state machine diagram, see Figure 10.11.

Figure 10.11 defines the sequence of goals of the semantic connectors of *MpConf* in a state machine diagram. Each state corresponds to a semantic connector with the same name as the state. Transitions between the states are marked with conditions on the goal achievement of the preceding semantic connector.

An orthogonal composite state with two regions has been used to model two semantic connectors being active simultaneously; in this case the transition out of the composite state depends on both features reaching their goals.

10.2.8.2 Goal sequences expressed by a special form of interaction diagrams

This alternative has been suggested to us by Ina Schieferdecker, see Figure 10.12.

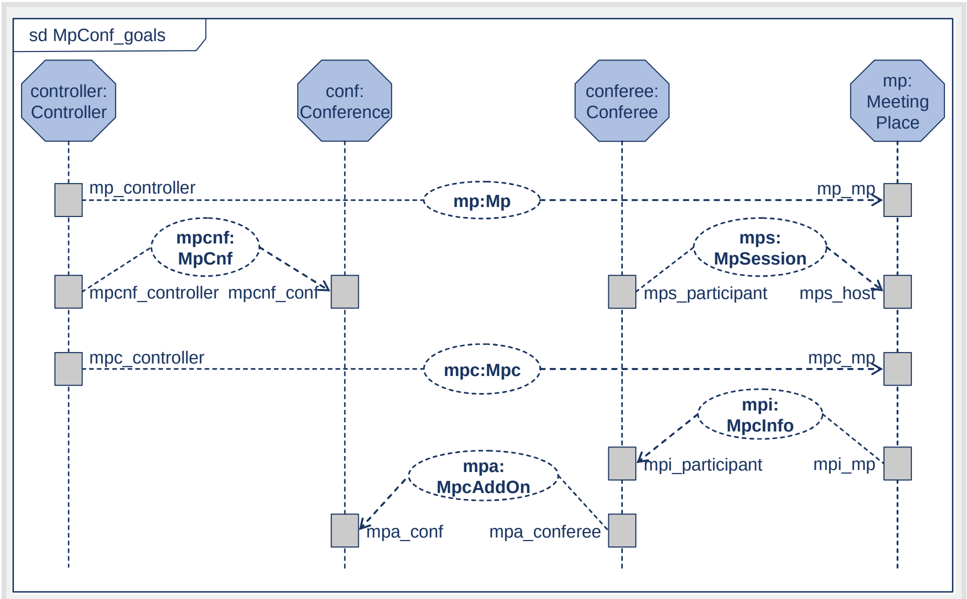


Figure 10.12 : Goal sequence diagram as a special form of interaction diagram

The goal sequence diagram in Figure 10.12 in effect combines interaction diagrams with collaboration uses, showing a sequence of role bindings to the lifelines of the interaction diagram that correspond to the roles of the composite collaboration.

10.2.8.3 Discussion

Both alternatives have interesting properties relative to goal sequence diagrams.

- Using state diagrams, no role goal interaction is referenced; this implies that goal relationships can be expressed at a conceptual level without having to specify interactions; this can be desirable at early stages of development. In Figure 10.11 no role binding is illustrated; this can be illustrated by additional graphics in the same way as that used in goal sequence diagrams. Without additional graphics, standard UML tools can be used to define goal dependencies; this is an advantage. However, since it lacks references to interactions, no validation opportunities arise; the diagram remains an expression of intentions that cannot be validated by tools. Modelling semantic connectors as states might also result in some conceptual confusion about designers.
- The special form of interaction diagram seems quite intuitive, given that the reader is familiar with the referencing of interaction occurrences in interaction diagrams. It conveys preceding role goals without any additional graphics; these can be discerned by

following the lifelines. However, it is not standard UML, and is thus unlikely to receive support from tool venders, unless it were to catch on and be endorsed by the OMG.

It seems worthwhile to pursue both alternatives.

10.3 Evolution or revolution?

The approach we present in our work marks a departure from earlier practices, which are:

- for the telecoms engineer: focus on designing state machines that communicate asynchronously via signals;
- for the software engineer: focus on designing objects that communicate synchronously via procedure calls.

Instead we advocate focusing on service composition from semantic connectors, goal sequences, and the binding of semantic interfaces to actors. Does this mark an evolution or a revolution? Semantic connectors, where one semantic interface plays an initiator role and the other a responding role, are in many ways “conveniently” close to the client-server paradigm of synchronous procedure calls; the asynchronous mechanisms that underlie our approach are conveniently concealed: a semantic connector can be (mis)understood as encompassing a synchronous pattern of behaviour[7].

It may be considered an advantage that semantic connectors are close to the conceptual understanding of computer programmers. In this way our approach may constitute a form of “modelling convergence” between the mind sets of computing and communication, and might enable the multitude of IT practitioners to embark on the development of services in symmetrically communicating systems.

Can one envision a division of responsibility in R&D work, where programmers design, implement and validate semantic connectors, while specialists (service engineers) design and validate composite collaborations? And can one assume that the construction of service roles and components could be automated by tools? This is as yet unknown.

7. This holds in particular if role behaviour strictly follows a query - response pattern between an initiating and responding role, i.e. no conflicting initiatives are modelled.

11

Appendix C - Glossary

In our thesis we have introduced and defined a number of terms. These are listed here in alphabetical order, for the benefit of the reader.

Actor: a computational object that can play service roles. An actor can play several service roles, both simultaneously and alternately.

Actor goal: a service goal defined for an actor. An actor goal is usually related to the role goal of a service role or a semantic interface an actor can play.

Actor goal sequence: describes the inter-relationships between preceding and succeeding goals of the interface roles played by an actor type.

Actual interface behaviour: the behaviour an actor or service role exhibits on an interface.

Alternative role: two roles are *alternatives* with respect to a particular connected role if they both can play with the connected role without violating the safety rules.

Basic interface role: an interface role without progress labels.

Collaboration goal: a predicate expressing when a goal is achieved seen from the perspective of the collaboration as a whole.

Collaboration goal sequence: describes the global ordering of goals of semantic connectors in the context of a composite service structure.

Compatible connected roles: a role is behaviourally *compatible* with another role if and only if the roles interact safely and their truncated roles contain all their respective progress labels.

Compliancy between a service role and a semantic interface: a service role is compliant with a semantic interface if its p-role projected over the connection represented by a semantic connector is a live subtype of the semantic interface.

Compliancy with a semantic interface: a service role or actor is compliant with a semantic interface if its p-role projected over the connection represented by the semantic connector is a live subtype of the semantic interface.

Connected role: an interface role is called a connected role with respect to an opposite interface role, if it intends to or actually interacts with that interface role over a connector.

Connector: a binding between two roles that can carry the interactions of a service.

Connector goal interaction: an interaction defining the goal achievement of a semantic connector.

Deadlock: occurs when two interface roles are unable to proceed because they wait endlessly for signals from each other.

Discovery of compatible actors: a service discovery mechanism by which an actor can determine which actors are capable of playing compatible roles.

Dual role: an interface role that interacts compatibly with a given interface role. The full behaviour of the given interface role can be reached interacting with the dual role.

Elementary collaboration: a service structure defining the roles and collaboration behaviour for a cooperation between two and only two interface roles.

Enabled role: a role is enabled for an actor if there exists a token on all incoming edges of the action representing the role in the actor goal sequence.

Event: a signal sent or consumed.

Event goal: related to the occurrence of an event: the event occurring implies that something useful is achieved at that point in the behaviour.

Goal assertion: expressed by goal expressions in structure descriptions, or by state invariants in behaviour descriptions.

Goal dependency: a dependency between two collaboration uses expressing the fact that goal achievement for the supplier element is a precondition for goal achievement of the client element. In UML a stereotyped *<<goal>>* dependency is used.

Goal expression: a predicate defined within the scope of a service structure. If evaluated to *True*, the goal of the element is currently achieved.

Goal sequence diagram: a graphical adaptation of the interaction overview diagram showing how goal achievements in semantic connectors of a composite service structure are ordered.

Goal state: a state or exit point of a state machine where a goal is achieved. Goal states can be expressed by goal assertions and/or by progress labels.

Graded progress label: a progress label with a numerical designation of the relative amount of progress; the higher the number the greater the relative progress.

Improper termination: occurs:

- when two interface roles do not terminate in a coordinated manner: no signal should be sent to a role that has terminated;
- when the exit conditions attached to the interface role terminations are not consistent with each other. Two exit conditions are consistent when they represent the same termination cases, or when one of the conditions represents a termination case that covers the termination case represented by the other condition.

Initial state invariant: a state invariant that is placed prior to any event occurrences on the lifeline of a goal interaction.

Initiating role: an interface role that can send a first signal over a connector.

Interaction safety: a pair of interface roles are said to interact safely when their interactions do not lead to any unspecified signal reception, deadlock or improper termination.

Interface role: describes the (actual or specified) interface behaviour of an actor or service role at a connector endpoint.

Level of progress: the level of progress of an interface role truncated by another interface role is the sum of the graded progress labels in its truncated role.

Live interface role: an interface role with progress label(s).

Live subtype: a safe and useful redefinition of an interface role. By useful it is implied that no progress is lost by the redefinition.

Progress ambiguity: occurs in an interface role when an event does not have consistent progress labelling for all occurrences of the event.

Progress label: marks an event goal or goal state in a service role or interface role.

Projection role (p-role): an interface role describing the actual interface behaviour of a service role visible at a connector endpoint.

Responding role: an interface role that does not send any first signal over a connector.

Role compatibility: a pair of roles is compatible if and only if the roles interact safely, and are capable of achieving service goals when doing so.

Role compliancy: a classifier is compliant with a role bound to it if the interface behaviour of the classifier is a live subtype of the interface behaviour of the bound role.

Role goal: a service goal defined for a collaboration role.

Role goal interaction: an interaction defining the goal achievement of an interface role.

Role goal sequence: describes the relationships between preceding and succeeding role goals for an interface role.

Safe interface role: an interface role that satisfies the safety rules.

Safe service role: a service role whose projections to p-roles all result in safe interface roles.

Safe subtype: a safe redefinition of an interface role. The redefinition may have less progress then the supertype.

Semantic connector: an elementary collaboration with a consistently defined pair of semantic interfaces and service goals, i.e. where:

- the semantic interfaces are dual roles, so by definition they are safe interface roles;
- goals are defined consistently: a goal in one role is matched by a goal in its opposite role;
- it optionally defines *collaboration goal(s)* that are reachable when the roles interact.

Semantic interface: an interface role describing specified interface behaviour. A semantic interface has at least one *event goal*.

Service: a collaboration between concurrent and potentially distributed service roles played by computational objects in order to provide some identified functionality to the environment.

Service goal: a property that characterizes a point in the behaviour of a collaboration between roles as having achieved something useful.

Service invocation: an instance of a service.

Service opportunity: an actor has a service opportunity with a set of actors in its environment when there exists a non-empty intersection between its current set of enabled initiating roles and the set of enabled compatible roles offered by the actors in the environment.

Service progress: the service progress of a role truncated by another role is the set of service-specific progress labels in its truncated role. An empty set implies no progress.

Service role: the part a computational object plays in a service. A service role can play several interface roles, both simultaneously and alternately.

Service structure: defines a collaboration by name, and identifies (names) the roles that collaborate to provide the service or service feature. The service structure also defines the multiplicity and type of the roles.

Service-specific progress label: a progress label that identifies a role goal.

Specified interface behaviour: the behaviour an actor or service role is specified to exhibit on an interface.

State-like goal: a predicate that can be evaluated at any time when a system is stable, i.e. in between the handling of events and outside the execution of operations. A state-like goal is a *goal expression* or a *goal assertion*.

Terminal state invariant: a state invariant that is placed after the last event occurrences on the lifeline of a goal interaction.

Truncated role: the sub-tree of an interface role that is reachable when playing with some specific connected interface role. Role *A* truncated by role *B* is written *A with B*.

Unspecified signal reception: occurs when an interface role consumes a signal that is not specified as input of the current role state.

References

[Alloy 2002] Daniel Jackson, “Micromodels of Software: Lightweight Modelling and Analysis with Alloy”, MIT Lab for Computer Science, Feb. 2002

[Alpern and Schneider 1985] B. Alpern and FB Schneider, “Defining liveness”, Information Processing Letters, vol. 21, 181-185, Oct. 1985

[AMIGOS 2004] Information at www.pats.no/projects/AVANTEL/AMIGOS.html (accessed August 2006)

[ARTS 2003] Øystein Haugen and Birger Møller-Pedersen, “The fine Arts of Service Modeling”, ARTS - Arena for Research on advanced Telecom Services, December 2003

[Alsnes 2004] Rune Alsnes, “Role Validation Tool”, M.Sc. thesis, NTNU, June 2004

[Bachman and Daya 1977] Bachman, C.W., Daya, M., “The role concept in data models”, Proc. of the 3rd Int. Conference on Very Large Data Bases, Tokyo, Japan, IEEE Computer Society (1977)

[Basic Call 1988] “Teleservices supported by an ISDN : Telephony”, ITU-T Recommendation I.241.1 (11/88)

[Birkeland 2005] Sebjørn Sæther Birkeland, “Behavioral Projections and Validation from UML 2.0 State Machines”, Project Assignment TTM4170, NTNU, December 2005

[Bræk 1977] Rolv Bræk, “Modelling Telecommunication Control Systems”, ELAB report STF44 A77229 (1977)

[Bræk 1979] Rolv Bræk, “Unified system modelling and implementation”, International Switching Symposium, 7-9 May 1979, CCIC, Paris, France Vol. 3, 1180-1187 (1979)

[Bræk and Emstad 1986] Rolv Bræk and Peder Emstad, “Telesystemering, 2. utgave”, Tapir, Trondheim, Norway ISBN 82-519-0721-7 (1986)

[Bræk and Haugen 1993] Rolv Bræk and Øystein Haugen, “Engineering Real Time Systems”, Prentice Hall, ISBN 0-13-034448-6 (1993)

[Bræk 1999] Rolv Bræk, “Using roles with types and objects for service development”, Proceedings of the Fifth International Conference on Intelligence in Networks (Smartnet’99), 265-278, Kluwer Academic Publishers (1999)

[Bræk 2004] Rolv Bræk, “MDA in perspective”, keynote speech at First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, University of Twente, Enschede, The Netherlands, March 17-18, 2004

[Bræk and Floch 2004] Rolv Bræk and Jacqueline Floch 2004, “ICT Convergence: Modeling Issues”, Proceedings of the 4th SDL and MSC Workshop (SAM), Ottawa, Canada, 2-4 June 2004, 237-256 (2004)

[Burmester et alia 2004] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling, “Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite”, SVERTS workshop, UML 2004, October, 11, 2004 in Lisbon, Portugal (2004)

[Carrez et al. 2004] Cyril Carrez, Alessandro Fantechi and Elie Najm, “Assembling Components with Behavioural Contracts”, Annales des Télécommunications Vol. 60 n°7-8, July-August 2005, Hermes-Lavoisier (2005)

[Castejón 2005] Humberto Nicolás Castejón, “Synthesizing State-machine Behavior from UML Collaborations and Use Case Maps”, in [SDL Forum 2005], 339-359

[Castejón and Bræk 2005] Humberto Nicolás Castejón and Rolv Bræk, “Dynamic Role Binding in a Service Oriented Architecture”, in IFIP International Conference on Intelligence in Communication Systems (INTELLCOMM 2005), October 2005, Montreal, Canada, Springer-Verlag (2005)

[Castejón and Bræk 2006a] Humberto Nicolás Castejón and Rolv Bræk, “A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios”, 5th Workshop on Scenarios and State Machines: Models, Algorithms and Tools, Workshop at ICSE, May 20-28, 2006, Shanghai, China (2006)

[Castejón and Bræk 2006b] Humberto Nicolás Castejón and Rolv Bræk, “Formalizing Collaboration Goal Sequences for Service Choreography”, Proceedings of FORTE'06, September 26-29, 2006, Paris, France, LNCS 4229, Springer-Verlag (2006)

[Cinderella] Cinderella SDL and MSC tools, information at www.cinderella.dk (accessed August 2006)

[Cohen and Levesque 1990] P. Cohen and H. Levesque, “Intention is choice with commitment”, Artificial Intelligence, 32(3):213-261, 1990

[CONF 1988] “Multiparty supplementary services: Conference calling (CONF) (ISDN)”, ITU-T Recommendation I.254.1 (11/88)

[CORBA 2001] “The Common Request Object Broker: Architecture and Specification. CORBA revision 2.5”, Object Management Group, Needham (MA), USA (2001)

[Diethelm et alia 2002] I. Diethelm, L. Geiger, T. Maier, A. Zündorf, “Turning Collaboration Diagram Strips into Storycharts”, Workshop on Scenarios and state machines: models, algorithms, and tools (SCESM), ICSE 2002, Orlando, Florida, USA (2002)

[Deubler et alia 2005] Martin Deubler, Michael Meisinger, Sabine Rittmann and Ingolf

Krüger, “Modeling Crosscutting Services with UML Sequence Diagrams”, MoDELS/UML 2005, Montego Bay, Jamaica, Oct. 2-7 2005, LNCS 3713, 522-536, Springer-Verlag (2005)

[Engelhardtsen and Prinz 2006] Fritjof Engelhardtsen and Andreas Prinz, “Application of Stuck-free Conformance to Service-role Composition”, in Proceedings of the 5th Workshop on System Analysis and Modelling (SAM'06), May 31-June 2, 2006, Kaiserslautern, Germany, LNCS 4320, 115-132, Springer-Verlag (2006)

[Estelle 1989] “Estelle: a formal description technique based on an extended state transition model”, ISO 9074 (1989)

[eODL 2003] “Extended Object Definition Language (eODL)”, ITU-T Recommendation Z.130 (07/2003)

[Ferraiolo et alia 2001] D. F. Ferraiolo, R. Sandhu, S. Gavrila, R. Kuhn and R. Chandramouli, “Proposed NIST Standard for Role-Based Access Control”, ACM Transactions on Information and System Security, Vol. 4(3), 224–274, August 2001

[Fisler and Krishnamurthi 2001] Kathi Fisler and Shriram Krishnamurthi, “Modular Verification of Collaboration-Based Software Designs”, Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (2001)

[Floch and Bræk 2000] Jacqueline Floch and Rolv Bræk 2000, “Toward Dynamic Composition of Hybrid Communication services”, Proceedings of the Sixth International Conference on Intelligence in Networks (Smartnet 2000), 73-92, Kluwer Academic Publishers (2000)

[Floch 2003] Jacqueline Floch, “Towards Plug-and-Play Services: Design and Validation using Roles”, Ph.D. thesis 2003:47 NTNU (2003)

[Floch and Bræk 2003a] Jacqueline Floch and Rolv Bræk, “Using Projections for the Detection of Anomalous Behaviours”, in [SDL Forum 2003], 36-54

[Floch and Bræk 2003b] Jacqueline Floch and Rolv Bræk, “Using SDL for Modelling Behaviour Composition”, in [SDL Forum 2003], 251-268

[Floch and Bræk 2005] Jacqueline Floch and Rolv Bræk, “A Compositional Approach to Service Validation”, in [SDL Forum 2005], 281-297

[Fuglesang 2005] Marie S. Fuglesang, “Service Modelling using UML 2.0 Collaborations”, M.Sc. thesis, NTNU, June 2005

[Gonzalo et alia 2002] G. Gonzalo, J. Llorens and P. Martínes 2002, “The meaning of multiplicity of n-ary associations in UML”, Systems and Software Modeling, Vol. 1(2), 86 - 97, Springer-Verlag, Heidelberg, Germany, December 2002

[GRL 2003] “Goal-Oriented Requirement Language (GRL)”, ITU-T Draft Recommendation Z.151, Sept. 2003

[Gruber 1993] T. R. Gruber, “A translation approach to portable ontologies”, Knowledge Acquisition, 5(2):199-220 (1993)

[Harel 1987] David Harel "Statecharts: A visual formalism for complex systems", Science of Computer Programming, vol. 8, no. 3, 231-274, Elsevier (1987)

[Hennie 1968] Hennie, F.C., "Finite-state models for logical machines", John Wiley & Sons, Library of Congress Catalog Card Number: 67-29935 (1968)

[Herrmann and Krumm 2000] Peter Herrmann and Heiko Krumm, "A framework for modeling transfer protocols", Computer Networks Vol. 34, 317-337, Elsevier Science (2000)

[Herrmann 2003] Peter Herrmann, "Formal Security Policy Verification of Distributed Component-Structured Software", in Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2003), Berlin, Germany, LNCS 2767, 257–272, Springer-Verlag (2003)

[Holzmann 1991] Gerard J. Holzmann, "Design and Validation of Computer Protocols", Prentice Hall 1991, ISBN 0-13-539925-4 (1991)

[Holzmann 2003] Gerard J. Holzmann, "The SPIN Model Checker", Addison-Wesley, Boston, USA, September 2003, ISBN 0-32-22862-6 (2003)

[IN 1992] "Intelligent Network: Global Functional Plane Architecture", Recommendation I.329 / Q.1203, October 1992

[IN 1993] "Intelligent Network: Introduction to Intelligent Network Capability Set 1", ITU-T Recommendation Q.1211, March 1993

[IN 1997] "Intelligent Network: Introduction to Intelligent Network Capability Set 2", Recommendation Q.1221, September 1997

[IN 1999] "Intelligent Network: Introduction to Intelligent Network Capability Set 3", Recommendation Q.1231, December 1999

[IN 2001] "Intelligent Network: Introduction to Intelligent Network Capability Set 4", Recommendation Q.1241, July 2001

[ISDN 1988] "Introduction to Stage 2 Service Descriptions for Supplementary Services", ITU-T Recommendation Q.80 (11/88)

[Jacobsen et alia 1992] Jacobsen, I., Christerson, M., Jonsson, P., and Övergaard, G., "Object-Oriented Software Engineering: A Case Driven Approach", Addison-Wesley (1992)

[JAIN 2004] "Java APIs for Integrated Networks (JAIN)", java.sun.com/products/jain (accessed August 2006)

[JINI 2004] DJ - Discovery and Join, Jini Technology Core Platform Specification, www.sun.com/software/jini/specs (accessed August 2006)

[Jones 2005] Steve Jones, "Toward an Acceptable Definition of Service," IEEE Software, vol. 22, no. 3, 87-93, May/June, 2005

[Korda 2004] Dragana Korda, "Service-Role Validation", Diploma thesis, University of Banja Luka, May 2004

[Krüger 2003] I. Krüger, “Modeling and Synthesis with MSC Extensions for Broadcasting, Overlapping, Preemptive, and Triggered Collaborations”, Workshop on Scenarios and State Machines at ICSE 2003 (2003)

[Krüger et alia 2004] I. H. Krüger, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, S. Rittmann, J. Ahluwalia, “Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering”, in Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS) (2004)

[Kraemer et alia 2006] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk, “Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services”, Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), Oct. 29th–Nov. 3rd, 2006, Montpellier, France. LNCS 4276, 1613-1632, Springer–Verlag (2006)

[Kraemer and Herrmann 2006] Frank Alexander Kraemer and Peter Herrmann, “Service Specification by Composition of Collaborations --- An Example”, Proceedings of the 2nd International Workshop on Service Composition (Sercomp), Hong Kong, December 2006 (2006)

[Lam and Shankar 1984] S. S. Lam and A.U. Shankar, “Protocol Verification via Projections”, IEEE Transactions on Software Engineering, vol. 10(4), 325-342, July 1984

[Lamsweerde 2001] Axel van Lamsweerde, “Goal-Oriented Requirements Engineering: A Guided Tour”, Proceedings of the 5th IEEE International Symposium on Requirements Engineering, Toronto, August 2001, 249-263 (2001)

[Lamsweerde and Letier 2003] Axel van Lamsweerde and Emmanuel Letier, “From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering”, in Radical Innovations of Software & systems Engineering, Proceedings of the Monterey’02 Workshop, Venice, Italy, LNCS 2941, Springer-Verlag (2003)

[KAOS 2003] Axel van Lamsweerde, “From System Goals to Software Architecture”, in Formal Methods for Software Architectures, LNCS 2804, Springer-Verlag (2003)

[LSC 2001] Werner Damm and David Harel, “LSCs: Breathing Life into Message Sequence Charts”, Formal Methods in System Design, 19, 45–80, 2001 Kluwer Academic Publishers (2001)

[Mencl 2004] Vladimir Mencl, “Specifying Component Behavior with Port State Machines”, Proceedings of CV-UML workshop (Oct. 21, 2003, part of UML 2003) F. de Boer and M. Bonsan-gue (ed.), Elsevier Science (2004)

[MSC-92] Message Sequence Charts (MSC), ITU-T Recommendation Z.120, Sept. 1994

[MSC-96] Message Sequence Charts (MSC), ITU-T Recommendation Z.120, Oct. 1996

[MSC Semantics] Message Sequence Chart Annex B: Formal semantics of Message Sequence Charts, ITU-T Recommendation Z.120 Annex B (04/1998)

[MSC-2000] Message Sequence Chart (MSC), ITU-T Recommendation Z.120 (11/2001)

[MSC-2004] Message Sequence Chart (MSC), ITU-T Recommendation Z.120 (04/2004) (Prepublished recommendation)

[Myklopoulos et alia 1999] J. Myklopoulos, L. Chung, E. Yu 1999, “From object-oriented to goal-oriented requirements analysis”, Communications of the ACM, Vol. 42 (1), January 1999

[OCL 2.0] “UML 2.0 OCL Final Adopted specification”, ptc/03-10-14, Object Management Group, Needham (MA), USA, October 2003

[OCL 2.0 2006] Object Constrain Language, OMG Available Specification, Version 2.0, formal/06-05-01, Object Management Group, Needham (MA), USA, May 2006

[OOram 1995] Trygve Reenskaug, Per Wold and Odd Arild Lehne, “Working with Objects, The OOram Software Engineering Method”, Prentice Hall, ISBN 1-884777-10-4 (1995)

[OORASS 1992] Reenskaug, T., Andersen, E.P., Berre, A.J., Hurlen, A.J., Landmark, A., Lehne, O.A., Nordhagen, E., Ness-Ulseth, E. Oftedal, G., Skar, A.L., and Stenslet, P., “OORASS: Seamless support for the creation and maintenance of object oriented systems”, Journal of object-oriented programming, vol.5, no. 6, 27-41 (1992)

[OSA 2003] Open Service Access (OSA); Application Programming Interface (API); Part 3: Framework, ETSI ES 202 915-3 V1.2.1 (2003)

[RAMSES 2006] Information available at www.item.ntnu.no/lab/pats/wiki/index.php (accessed August 2006)

[ROOM 1994] Bran Selic, Garth Gullekson, and Paul T. Ward, “Real-Time Object-Oriented Modeling”, Wiley (1994)

[RM-ODP 1998] “Open Distributed Processing: Reference Model”, ISO/IEC 10746-1, 1998, Also known as Information technology - Open distributed processing - Reference Model: Overview, ITU-T Recommendation X.901 (08/97)

[Rossebø and Bræk 2006a] Judith Rossebø and Rolv Bræk, “Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition”, in Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES’06), Vienna, Austria, pages 206–215. IEEE Computer Society Press (2006)

[Rossebø and Bræk 2006b] Judith Rossebø and Rolv Bræk, “A Policy-driven Approach to Dynamic Composition of Authentication and Authorization Patterns and Services”, to appear in Journal of Computers, Academy Publisher Vol. 1 Issue no. 8, (2006)

[Rössler et alia 2001] Frank Rössler, Birgit Geppert, and Reinhard Gotzhein, “Collaboration-based Design of SDL Systems”, in [SDL Forum 2001], 72-89 (2001)

[Rössler 2002] Frank Rössler, “Collaboration-based Design of Communication Systems in SDL”, Ph.D. thesis D 386, Kaiserslautern, Feb 2002

[Rössler et alia 2003] Frank Rössler, Birgit Geppert, and Reinhard Gotzhein, “CoSDL - An Experimental Language for Collaboration Specification”, Telecommunications and Beyond: The Broader Applicability of SDL and MSC, E. Sherratt (Ed.), LNCS 2599, 1-20, Springer-Verlag (2003)

[Sanders 2002] Richard Torbjørn Sanders, “Service-Centred Approach to Telecom Service Development”, Proceedings of the 8th EUNICE and IFIP Workshop on Adaptable Networks and Teleservices, Trondheim, Norway, Sept. 2-4 2002, 95-101 (2002)

[Sanders et alia 2003] Richard Torbjørn Sanders, Jacqueline Floch and Rolv Bræk, “Dynamic Behaviour Arbitration using Role Negotiation”, Proceedings of the 9th EUNICE and IFIP Workshop on Next Generation Networks, Balatonfüred, Hungary, 8-10 Sept. 2003, 76-81 (2003)

[Sanders and Bræk 2004a] Richard Torbjørn Sanders, Rolv Bræk, “Discovering Service Opportunities by Evaluating Service Goals”, Proceedings of the 10th EUNICE and IFIP Workshop on Advances in Fixed and Mobile Networks, Tampere, Finland, 14-16 June 2004, 165-172 (2004)

[Sanders and Bræk 2004b] Richard Torbjørn Sanders, Rolv Bræk, “Modeling Peer-to-peer Service Goals in UML”, Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China, 26-30 September, 2004, 144-154, IEEE Computer Society Press (2005)

[Sanders et alia 2005a] Richard T. Sanders, Rolv Bræk, Gregor van Bochmann, and Daniel Amyot, “Service Discovery and Component Reuse with Semantic Interfaces”, in [SDL Forum 2005], 85-102 (2005)

[Sanders et alia 2005b] Richard Torbjørn Sanders, Humberto Nicolás Castejón, Frank Alexander Kraemer and Rolv Bræk, “Using UML 2.0 Collaborations for Compositional Service Engineering”, Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005), Montego Bay, Jamaica, Oct. 2-7 2005, LNCS 3713, 460-475, Springer-Verlag (2005)

[Sassen & Macmillan 2005] Anne-Marie Sassen and Charles Macmillan, “The service engineering area: An overview of its current state and a vision of its future”, Information Society and Media Directorate-General, European Commission, July 2005

[SDL-88] Specification and Description Language (SDL), ITU-T Recommendation Z.100, Blue Book (1988)

[SDL-92] Specification and Description Language (SDL), ITU-T Recommendation Z.100, (03/93)

[SDL-96] Specification and Description Language (SDL) Addendum 1, Z.100 Addendum 1 (10/96)

[SDL-2000] Specification and Description Language (SDL), ITU-T Recommendation Z.100 (08/2002)

[SDL-2000 Corr] Specification and Description Language (SDL): Corrigendum 1, ITU-T Recommendation Z.100 (03/2003)

[SDL Method 1997] SDL+ Methodology: Use of MSC and SDL (with ASN.1), ITU-T Recommendation Z.100 Supplement 1 (05/97)

[SDL Semantics] SDL Formal semantics, ITU-T Recommendation Z.100 Annexes F1-3

(11/2000)

[SDL Forum 2001] Rick Reed and Jeanne Reed (editors), Proceedings of the 2001 SDL Forum, Copenhagen, Denmark, June 27-29, 2001, LNCS 2078, Springer-Verlag (2001)

[SDL Forum 2003] Rick Reed and Jeanne Reed (editors), Proceedings of the 11th SDL Forum, Stuttgart, Germany, July 1-4, 2003, LNCS 2708, Springer-Verlag (2003)

[SDL Forum 2005] Andreas Prinz, Rick Reed and Jeanne Reed (editors), Proceedings of the 12th SDL Forum, Grimstad, Norway, 21-24 June, 2005, LNCS 3530, Springer-Verlag (2005)

[ServiceFrame 2002] Rolv Bræk, Knut Eilif Husa, and Geir Melby, “ServiceFrame: WhitePaper”, Ericsson Norarc, 2002, Available at www.item.ntnu.no/lab/nettint1/ServiceFrame/ServiceFrame.html (Accessed August 2006)

[SOM 1981] Rolv Bræk, O. Hell and F. Sandvik, “SOM - A SDL Compatible specification and Design Methodology. Experiences from 5 years of Extensive Use”, Proc. of the 4th Int. Conference on Software Engineering for Telecommunication Switching Systems, 111-117, IEE, Coventry, UK 20-24 July 1981, ISBN 0 85296242 8 (1981)

[Singh and Huhns 2005] Munindar P. Singh and Michael N. Hunhs, “Service-Oriented Computing”, Wiley & Sons, Chicester, UK (2005)

[SLP 1999] Service Location Protocol, Version 2, IETF RFC 2608, June 1999

[SLP 2002] Vendor Extensions for Service Location Protocol, Version 2, IETF RFC 3224, January 2002

[Sties and Kellerer 2001] P. Sties and W. Kellerer, “A Generic and Implementation Independent Service Description Model”, Proc. of the 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01), Mesa, Arizona, USA, April 2001

[Telektronikk 4/2000] Rolv Bræk (editor), “Languages for Telecommunications Applications”, Telektronikk, vol. 2, no. 4, Telenor, ISSN 0085-7130 (2000)

[Telelogic] Telelogic Tau SDL suite, Information available at www.telelogic.com (accessed August 2006)

[TIMe 1999] Rolv Bræk, Joe Gorman, Øystein Haugen, Geir Melby, Birger Møller-Pedersen, and Richard Sanders, “TIMe: The Integrated Method. Version 4.0”, SINTEF, Trondheim, Norway, www.sintef.no/time, July 1999 (accessed August 2006)

[TINA 1995] “Overall Concepts and Principles of TINA”, version 1.0, February 1995

[TINA 1999] Yuji Inoue, Martine Lapierre and Cesare Mossoto (editors), “The TINA Book. A Co-operative Solution for a Competitive World”, Prentice Hall (1999)

[TIPHON 2003] TIPHON Protocol Framework Definition; Part 1: Meta-protocol design rules, development method and mapping guidelines, ETSI TS 101 882-1 V4.1.1 (2003-09)

[TTCN 2003]Testing and Test Control Notation version 3 (TTCN-3): Core language, ITU-T Recommendation Z.140 (04/2003)

[W3C 2004] The World Wide Web Consortium (W3C): The Semantic Web, www.w3c.org/2001/sw (accessed August 2006)

[UCM 2003] URN - Use Case Maps notation (UCM), ITU-T Draft Recommendation Z.152, September 2003

[UDDI 2004] Universal Description, Discovery and Integration (UDDI) protocol, Version 3, www.uddi.org (accessed August 2006)

[UML 2.0 Adopted] UML 2.0 Superstructure Specification, OMG Adopted Specification, ptc/03-08-02, Object Management Group, Needham (MA), USA, August 2003

[UML 2.0 Revised] UML 2.0 Superstructure Specification, Revised Final Adopted Specification, ptc/04-10-02, Object Management Group, Needham (MA), USA, Oct. 8 2004 (Convenience document)

[UML 2.0] Unified Modeling Language: Superstructure Specification, version 2.0, formal/05-07-04, Object Management Group, Needham (MA), USA, August 2005

[UML 2.0 Infra] UML 2.0 Infrastructure Specification, OMG Adopted Specification, formal/05-07-05, Object Management Group, Needham (MA), USA, March 2006

[UML2 Ref] James Rumbaugh, Ivar Jacobson and Grady Booch, "The Unified Modeling Language Reference Guide, Second Edition", Addison-Wesley, Boston, USA, ISBN 0-321-24562-8, July 2004

[Z.109 1999] SDL combined with UML, ITU-T Recommendation Z.109, November 1999

[Aagesen et alia 1999] F.A. Aagesen, B. Helvik, V. Wuwongse, H. Meling, R. Bræk, U. Johansen 1999, "Toward a Plug and Play Architecture for Telecommunications", Proceedings of the Fifth International Conference on Intelligence in Networks (Smartnet'99), Kluwer Academic Publishers (1999)