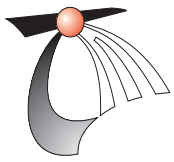
On the Design of Application Protocols

Marten van Sinderen



CTIT Ph. D-thesis series No. 95-04

P.O. Box 217 - 7500 AE Enschede - The Netherlands
telephone +31-53-893779 / fax +31-53-333815

Centre for
Telematics and
Information
Technology

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Sinderen, Marten Jelle van

On the Design of Application Protocols / Marten Jelle van Sinderen.
[S.1. : s.n.]. - Ill. - (CTIT Ph. D-thesis series, ISSN 1381-3617; no. 95-04)
Thesis University of Twente, Enschede. - With ref.
ISBN 90-365-0730-8
Subject headings: distributed systems; application protocols.

On the Design of Application Protocols

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. Th.J.A. Popma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 10 maart 1995 te 16:45 uur

door
Marten Jelle van Sinderen

geboren op 6 maart 1958
te Ternaard

Dit proefschrift is goedgekeurd door de promotoren
prof. dr. ir. C.A. Vissers
prof. ir. E.F. Michiels

Preface

In the last decades, much effort has been spent on the design and provision of sophisticated communication infrastructures. The development of end-user oriented distributed system applications, leaning on top of these communication infrastructures, so far has attracted little attention. This is regrettable, since communication infrastructures can only become useful and profitable if they can be deployed in the context of a sufficient number of distributed applications.

Two important factors determine the success of distributed applications: (1) the provision of high quality application services and protocols at short time scales; and (2) the availability of standards for these services and protocols that can be used for the construction of ‘open’ distributed systems. The achievement of both (1) and (2) can be supported by a suitable design methodology.

A design methodology entails a systematic approach to carry out complex designs, and therefore should incorporate proper concepts that enable the effective structuring of such designs. Concepts currently used for the design and structuring of application protocols appear to be inadequate for this purpose. Also a step-wise design approach that would help to master complexity and shorten development times is currently lacking.

Standards are necessary since individual users of distributed system applications prefer to be independent on any particular manufacturer or vendor when procuring products, while manufacturers prefer to have maximum implementation freedom when developing such products. An ‘open’ protocol standard defines necessary and sufficient conditions for system parts to interact, such that the system parts can be implemented independently of each other.

ISO[1] and ITU-TSS[2] base the development and definition of protocol standards on a ‘reference model’, called the Reference Model for Open Systems Interconnection (OSI-RM). This model comprises a rudimentary form of a design approach and a reference architecture that can be derived with this approach. According to the OSI-RM, the overall application protocol functionality is distributed over three hierarchical protocol layers. Each layer has been assigned a specific functionality, except the highest layer, the Application Layer, which is made responsible for all remaining protocol functions. Because the

1. International Organization for Standardization.

2. International Telecommunication Union - Telecommunication Standardization Section.

functionality of the Application Layer is not delimited it cannot, as opposed to the other layers, be covered by a single protocol standard or a fixed set of protocol standards. Several identified sets of Application Layer protocol functions are defined by separate Application Service Elements (ASEs).

The appropriateness of the OSI-RM for the development and definition of application protocol standards can be criticized on a number of points:

- the reference architecture defined by the OSI-RM is not flexible enough to adequately cope with the diversity of interaction requirements of distributed applications.
- some design concepts are not clearly defined, thus prohibiting their effective application to structuring problems;
- the relationship between high level application requirements and proposed application protocol solutions is unclear;
- the development of application protocol standards generally takes a long time.

This thesis aims at the development of a methodology for the design of application protocols, including application protocol standards, and so addresses the problems mentioned above. The following contributions are made to achieve this aim:

- design quality criteria are proposed that can be used to guide design decisions and to evaluate designs;
- OSI design decisions and design concepts with respect to application protocols are evaluated;
- general-purpose, elementary design concepts are proposed;
- milestones in the application protocol design process are presented;
- behaviour composition and structuring techniques are developed that can be used to represent design results corresponding to the identified milestones;
- design methods are proposed to support the correct performance of design steps between milestones;
- a flexible reference architecture is proposed.

A (potential) result of the design methodology is that layered application protocol hierarchies can be avoided if they are not required by the class of distributed applications that must be supported.

This thesis is structured as follows:

- *Chapter 1* (Introduction) presents a global problem description for this thesis, the scope and objectives of this thesis, and the approach followed in this thesis. It further introduces some general concepts related to the design of distributed systems.

- *Chapter 2* (Design Quality Criteria) discusses quality criteria that can be used to guide the design of application protocols and to evaluate the quality of already designed application protocols.
- *Chapter 3* (OSI Upper Layer Architecture and Model: State of the Art) presents the architecture and concepts defined by the OSI-RM with respect to application protocol standards. It also contains a brief description of the most important application protocol standards that were developed in the context of the OSI-RM.
- *Chapter 4* (OSI Upper Layer Architecture and Model: Evaluation) evaluates the architecture and concepts defined by the OSI-RM with respect to application protocol standards. It also discusses the relation between the quality of application protocol standards and the nature of standardization, and the implementation freedom supported by protocol standards.
- *Chapter 5* (Design Framework) presents a general framework for the design of application protocols. It identifies elementary concepts for distributed systems design, and abstraction levels at which distributed systems and system parts can be represented. The abstraction levels are used to define an application protocol design trajectory consisting of a sequence of design steps between different milestones in the application protocol design process.
- *Chapter 6* (Design Model) discusses a model for the representation and manipulation of behaviours, such that it can be used in the application protocol design trajectory. The model is based on the elementary design concepts, identified in Chapter 5, and defines additional concepts and rules in order to allow the composition of behaviours. It also includes techniques for the composition of structured behaviours and requirements for behaviour decomposition and refinement.
- *Chapter 7* (Application Protocol Reference Architecture) proposes a flexible reference architecture for application protocols. The reference architecture is based on the design quality criteria, the design framework and the design model, and presents specific design methods for structuring application protocols. Furthermore, some generic application protocol structures are discussed, and some application protocol functions that can be used as building blocks for the support of many classes of distributed system applications are characterized.
- *Chapter 8* (Suggestions for Further Work) presents some suggestions for further work.
- *Chapter 9* (Summary of Conclusions) presents a summary of the conclusions drawn in the previous chapters.

Acknowledgements

There are a number of people who, in one way or another, contributed to this Ph.D. thesis. First of all, I like to thank my supervisors, Chris Vissers and Eddie Michiels. Chris Vissers has created much of the architectural foundation on which the research of this thesis has been based. Despite his busy schedule, especially during the last months of the writing of this thesis, he still found time to guard the architectural consistency of the results of my research, and provided many useful suggestions for improvements. Eddie Michiels' help was indispensable in understanding the details and background of the standards referenced in this thesis. His explanations, larded with first hand anecdotes, were always useful and enjoyable at the same time.

I am indebted to Luís Ferreira Pires for his contribution to the work on behaviour definition based on causality relations. This work resulted in many joint papers, which in turn were taken as the starting point for Chapters 5 and 6 of this thesis. I also like to thank Dick Quartel. He has been involved in many discussions on the subject of causality relations, and provided useful feedback through the elaboration of examples.

I further like to thank the members of my discipline group, Application Protocol Systems (APS). Ing Widya, Robert Huis in 't Veld, and, recently, Andre van den Bos are responsible for a pleasant working environment. In addition, Ing Widya was kind enough to take over some of my tasks during periods in which the Ph.D. work prevented me from performing these.

The Tele-Informatics and Open System (TIOS) research group has been, and still is, a stimulating multi-disciplinary environment with a pleasant working atmosphere. Joint sailing, skiing and bicycle trips contributed to this atmosphere, and should therefore be treasured as a tradition to be continued in the future. Especially, I like to thank Aiko Pras for the many sociable discussions and useless but amusing bets. Rom Langerak is gratefully acknowledged for providing additional motivation to continue the training of long distance runs.

A Ph.D. thesis cannot be successfully completed without proper support of the home front. Klarie provided just the right mix of support and diversion that kept me aware of other things in life. Rutger and Maarten were so kind to leave me working with my own 'television set' for such a long time.

Marten van Sinderen
Enschede, February 1995

Table of Contents

Chapter 1

Introduction

This chapter presents a global problem description for this thesis, the scope and objectives of this thesis, and the approach followed in this thesis. It also briefly indicates the relevance of the work presented in this thesis and some areas and projects where related work is carried out. Finally, some general concepts are presented, including the concepts of architecture, implementation, service, and protocol, and their relevance to application protocol design. Although these concepts are often used in the context of distributed systems design, their use is not always consistent and the relationship between these concepts is not always clear. The interpretation of the concepts presented in this chapter will be adopted in the following chapters.

The structure of this chapter is as follows: section 1 presents the background of the work on application protocols; section 2 presents the problems related to the design of application protocols; section 3 mentions the role of international standardization organizations in the development of application protocols; section 4 defines the scope and objectives of this thesis; section 5 discusses the approach that is followed in this thesis; section 6 indicates existing work that is related to the work presented in this thesis; and section 7 presents some general concepts.

1.1 Background

Systems of interconnected computers, hereafter called distributed systems, play an important and still increasing role in modern society (see [SA91] and [Hartmanis92], for example). Examples of applications of distributed systems that illustrate this role are: computer supported cooperative work, process control, on-line transaction processing, product data interchange, electronic data interchange, electronic mail, tele-working, teleconferencing, and point-of-sale. Distributed systems are the result of the fusion of computing and communications. In both areas, dramatic technological advances have been made in the last few decades which made their fusion practical and cost-effective. In parallel, progress was made in the area of requirements engineering and system engi-

neering, which enabled to manage the complexity involved in the design of systems based on these new technologies.

The success of distributed systems is measured by the acceptance of distributed system applications by users and by the profits made by providers and manufacturers. By these standards, the first generation of distributed system products was successful. Today, users want higher degrees of functionality and performance, and want functions and performance to be more precisely tailored to their specific needs. The second generation of distributed systems is therefore much more sophisticated and diverse. New applications are no longer paced by breakthroughs in technology, but by the choice and deployment of available technologies ([Dorros92]). In other words, the complexity of distributed systems has grown, and will continue to grow, which makes a systematic approach to the design of distributed systems increasingly important.

An important aspect of distributed system design is protocol design. Protocols define the interworking aspects of distributed systems, i.e. the rules that must be obeyed by individual computer systems in order to achieve meaningful interaction. Protocols can be divided into three classes, viz. application protocols, data transfer protocols, and transmission protocols. Application protocols are concerned with the interworking of application processes allocated to different computer systems. Data transfer protocols are concerned with the transfer of binary encoded information, or data. Transmission protocols are concerned with the transmission of physical signals through various transmission media that are used to connect the computer systems.

New application requirements with respect to distributed systems directly affect application protocols. Data transfer protocols and transmission protocols are not always and not so drastically affected. The design of application protocols is therefore a key concern in the design of new distributed applications. Their design should be efficient and effective, and the resulting application protocols should adequately support the user requirements.

1.2 Problem description

There are several problems related to the design of application protocols:

- mastering the *complexity* of application protocol design.

  The design of application protocols generally involves many difficult design decisions. In order to master the complexity of the design process, the design should be carried out in a systematic way, separating orthogonal concerns of the design so that design and design validation can progress in discrete steps. Such a systematic approach is called a *design methodology*. An effective design methodology provides guidance to the designer, helps the designer to control and improve the quality of design solutions, and shortens the development time of design solutions.

- need of common *prescriptions* for application protocol implementations.

  Computers in a distributed system may differ in many respects. They may be based on different hardware architectures, operating systems, compilers, etc. In order to permit the system parts (application processes) of such a heterogeneous distributed system to interwork, they must implement a common *application protocol architecture*[1]. It is useful to define application protocol architectures explicitly. Arbitrary computer systems can then join a distributed system if they correctly implement a common (set of) application protocol architecture(s). Application protocol architectures prescribe implementations, i.e. they define the required characteristics of application protocol implementations. They should not, however, unnecessarily constrain implementation freedom. Application protocol architectures must therefore be defined at the right level of abstraction, which poses requirements on the design concepts and the specification language that are used in the design/definition process.

- need for *modular* application protocol solutions.

  Application protocols are generally so complex that they need to be structured. Preferably, application protocol structures should be designed that contain re-useable application protocol components. Re-useable application protocol components are defined such that they can be used in more than one application context. The availability of re-useable application protocol components may considerably reduce the development time of new application protocols. The determination of effective structuring techniques, or modularities, for application protocols is, however, very difficult.

- *coordinating* the development of application protocol solutions.

  The range and variety of distributed system applications is unlimited. Without coordination, different manufacturers will most likely identify different, often partially overlapping, sets of user requirements for which they develop different application protocols. And even if different manufacturers take the same set of user requirements as a starting point, they may still develop different application protocols since there are generally many conceivable application protocol solutions that all adequately support a given set of user requirements. In order to prevent the development of many different, i.e. mutually incompatible, application protocol solutions, some sort of coordination should take place. The objective of such a coordination effort should be the classification of user requirements, and the development of common application protocol architectures (see above). A framework that provides guidelines to designers in this respect is often called a *reference architecture*.

1. The system parts must also implement common data transfer and transmission architectures.

A design methodology is usually concerned with all of the afore mentioned problems. Hence, it usually comprises a definition of design concepts, structuring techniques, and a reference architecture. A reference architecture is often understood as a rather rigid, thus inflexible framework with pre-defined relations between different types of functions. It can, however, also be understood as a set of guidelines to structure functions and of methods for achieving such structures. It is then the freedom of the designer to define specific compositions of functions. Such a reference architecture may also identify generic components that may be useful in many specific architectures, and relationships between these components, provided that only relationships are prescribed that apply to any specific architecture that incorporates the related components.

1.3 The role of standardization

International standardization organizations can play an important role in coordinating the development of application protocol architectures and ratifying these as standards. They provide a suitable forum for stake-holders (manufacturers, vendors, users) to meet and to agree upon commonly acceptable solutions. Active organizations is this area are, among others, the International Organization for Standardization (ISO) and the Telecommunication Standardization Section of the International Telecommunication Union (ITU-TSS)[1]. ISO has developed a coordinating framework for the definition of protocol standards, the Reference Model for Open Systems Interconnection (OSI-RM) ([IS7498:84], [Day83]). This framework was also adopted by the ITU-TSS. The OSI-RM comprises both a loosely defined set of design concepts and a protocol reference architecture.

The OSI-RM has been very influential in establishing a ‘common language’ and a ‘common model’ for discussing protocol issues. The OSI-RM also led to the development of a large number of protocol standards, including application protocol standards. The application protocol standards based on the OSI-RM have been less successful than originally expected by standards’ designers and user organizations. User complaints about products based on application protocol standards include lack of functionality, lack of performance, and lack of flexibility. The reasons behind these complaints relate to the quality of the implementations, the quality of the standards, the nature of the standardization process, and only partially to the suitability of (the concepts and the reference architecture of) the OSI-RM. Nonetheless, the OSI-RM is an important enabling, or disabling, factor for the success of OSI products.

As mentioned in the previous section, the development of application protocols requires an application protocol design methodology, with design concepts for the design/ definition of application protocol architectures, proper structuring techniques, and a

1. Formerly called the International Telegraph and Telephone Consultative Committee (CCITT).

flexible application protocol reference architecture. We believe that the current OSI-RM does not sufficiently satisfy these requirements.

1.4 Scope and objectives

This thesis considers the design of application protocols, in particular application protocol architectures. It is not our intention to propose specific application protocol solutions to support particular sets of user requirements. It is also not our intension to propose protocol implementation methods as part of the application protocol design process.

The main objective of this thesis is to develop an *application protocol design methodology.* This methodology should help the designer to carry out the design process in a systematic way, and to achieve short development times and good quality of application protocols. The following objectives are considered part of the main objective:

- propose *design concepts.*

  The expression of designs in the application protocol design process should be based on a set of elementary design concepts.

- propose related *milestones* and *design steps.*

  Milestones are design results in the application protocol design process that reflect important design decisions and for that reason may be explicitly considered in instances of the design process. One milestone is the application protocol architecture. Milestones should be related by design steps.

- propose *structuring techniques* and *design methods.*

  Structuring techniques are necessary to represent complex application protocols and complex intermediate design results. In addition, structuring techniques are necessary to support the definition of re-useable application protocol components. Design methods are needed to systematically handle design concerns of design steps and to ensure that design steps are performed correctly. Structuring techniques and design methods related to the final step(s) in the design of an application protocol architecture constitute the basis for a flexible application protocol reference architecture.

1.5 Approach

The OSI-RM has been more successful as a ‘discussion model’ than as a ‘reference model’. That is, the OSI-RM is often used as a basis for discussing protocol issues, but protocol products claiming conformance to standards based on the OSI-RM have not gained wide acceptance so far (except in a few cases). We believe that the success of the

OSI-RM as a discussion model is mainly based on the identification of two levels of abstraction, viz. the service level and the protocol level. The service level is a higher abstraction level than the protocol level and is instrumental to the separation of concerns that leads to a layered protocol architecture. On the other hand, we believe that:

- the service level and its related concepts are not clearly defined by the OSI-RM; and
- the structuring technique of layering alone is not sufficient to cope with the complexity of application protocols.

The lack of success of OSI products is, in our opinion, partly due to this 'incompleteness' and to the inflexibility of the reference architecture defined by the OSI-RM. There is a need for an application protocol design methodology, with clearly defined design concepts, milestones and design steps, and structuring techniques and design methods, and a more flexible reference architecture.

This does not mean that we should abandon the OSI-RM and start from scratch. Rather, the OSI-RM state of the art should be carefully evaluated in order to determine what precisely are its strong and weak points. The application protocol design methodology and the application protocol reference architecture should adopt those aspects of the OSI-RM which are considered useful and at the same time incorporate solutions to compensate for the shortcomings of the OSI-RM.

The approach adopted in this thesis is:

- determine which design quality criteria are useful to guide the designer in taking design decisions. Such quality criteria are considered part of the application protocol design methodology. They will also be used, however, to evaluate the OSI-RM (see below).
- present the state of the art of the OSI-RM. Here we only want to consider aspects of the OSI-RM that relate to application protocols. These aspects include the design concepts of the OSI-RM, part of its reference architecture, and important design choices incorporated by application service and protocol standards.
- evaluate the OSI-RM state of the art with respect to application protocols. The evaluation should address the following questions:
  - are the design objectives of the service level and the protocol level clearly defined?;
  - are the design concepts clearly defined?;
  - are the structuring techniques adequate for supporting the design objectives?; and
  - do the application service and protocol standards comply with our design quality criteria?

  Moreover, the evaluation should indicate what are the shortcomings of the OSI-RM.

- introduce a design framework for the design of application protocols. The design framework identifies elementary design concepts and abstraction levels, and defines their role in the design process. It further identifies milestones (corresponding to distinct abstraction levels) in the design process and design steps that relate these milestones. Conclusions from the evaluation of the service level and the protocol level as defined by the OSI-RM are relevant at this point.
- introduce a design model for the design of application protocols. The design model defines how elementary design concepts can be composed with the purpose of representing behaviour. It also defines techniques for the composition of structured behaviours and requirements for the decomposition and refinement of behaviours. Conclusions from the evaluation of OSI structuring techniques and design concepts are relevant at this point.
- introduce a reference architecture for specific application protocol architectures. This reference architecture provides (further) guidelines for structuring application protocol architectures, and is based on the design quality criteria, the design framework and the design model mentioned above. It also identifies application protocol components that can be used a generic building blocks in application protocol design. Conclusions from the evaluation of the OSI reference architecture are relevant at this point.

Figure 1.1 illustrates this approach. It also indicates the relationship with the chapters of this thesis.

1.6 Related work

Architectural issues related to the development of application service and protocol standards have been given separate attention since the completion of the first version of the OSI-RM in 1984. It was recognized that the Upper Layer Architecture (ULA), i.e. the part of the OSI reference architecture that is concerned with application protocols, deserved additional attention. This lead, among others, to the development of the Application Layer Structure (ALS, [IS9545:89]) to facilitate the structuring of application protocols assigned to the OSI Application Layer. The attention was thus mainly focused on a refined application protocol reference architecture; the other problems identified in section 1.2 were hardly considered. We believe that the improvements that can be achieved by such a refinement are rather limited. More substantial improvements are expected if a reference architecture is proposed that is more flexible and that is part of a design methodology.

Another development initiated by the ISO, and joined by the ITU-TSS, is the work on Open Distributed Processing (ODP) which started in 1987. This work aims at the creation of a Reference Model for Open Distributed Processing (ODP-RM) ([IS10746:94], [Linington92]). The scope of ODP is wider than that of OSI: it is not limited to interworking aspects of distributed systems, but it also considers distribution transparency and

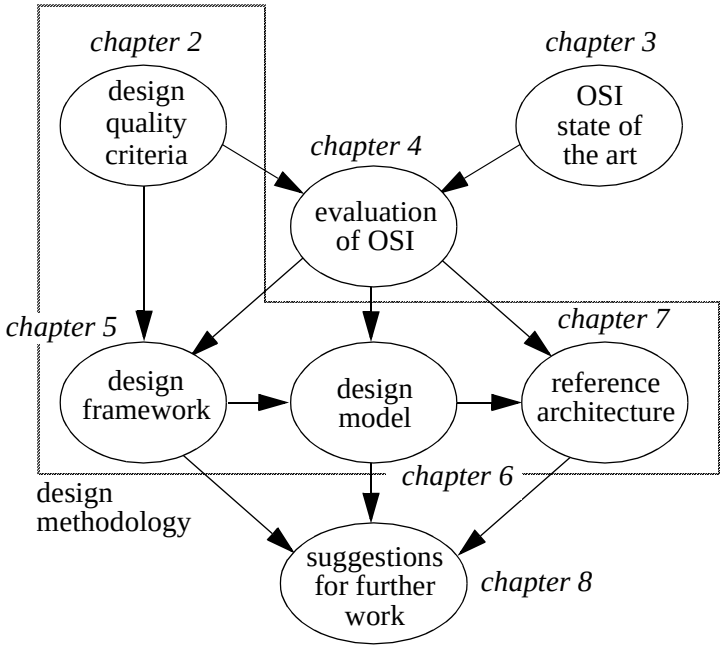


Figure 1.1: Approach adopted in this thesis

application portability. For this reason, the ODP-RM must consider both higher and lower levels of abstractions compared to the OSI-RM[1]. The ODP-RM includes a framework of abstractions and related design concepts. The abstractions, or viewpoints in ODP terminology, in the framework of abstractions are not abstraction levels. Their relationship is hard to define precisely, and (consequently) there is no design methodology defined that provides guidance to the designer using these abstractions. Also the relationship with the service and protocol level, as defined by the OSI-RM and used in the ALS, is still under discussion.

In the telecommunications arena, work is carried out in the area of architectures for Intelligent Networks (IN; this term was introduced by Bell Communications Research Inc. in 1985). For example, ITU-TSS works on standards for IN ([Kung92]), various telecommunications research laboratories have formed the Telecommunications Information Networking Architecture (TINA) consortium to study IN related issues and to organize international workshops on IN ([Barr92]), and Bellcore is developing an IN architecture called the Information Networking Architecture (INA) ([Natarajan92]). All these initiatives have in common that they aim at easing the introduction of new telecommunications services. Service logic, switching control and switching functions are separately considered in these new architectures in order to obtain a better flexibility, and several functional entities have been introduced to exploit this separation. A distinct concern of these archi-

1. The term ‘open’ therefore has another meaning in the ODP-RM than in the OSI-RM.

tectures is the support of ‘service engineering’, i.e. the creation of new services by combining existing functional entities. Service engineering is performed by the provider, acting for the user, or by the user himself, and enables faster introduction of new services that better support specific user requirements. The work on IN has been mainly system-oriented, in the sense that the development of architectures starts out from characteristics of system implementations. Although different abstractions of system implementations have been identified in most approaches, no top-down design methodology has been associated with these abstractions. The identification of re-useable components for service engineering is usually based on object-oriented techniques.

Under the Research on Advanced Communications for Europe (RACE) programme, a number of projects are conducting research on service engineering ([Campolargo94]). Examples are the Cassiopeia project (R2049), the SCORE project (R2017), and the BOOST project (R2076), all of which started in 1991. As with IN, the aim is to ease the introduction of new services. The architecture that supports service engineering has been called the Open Services Architecture (OSA). Not surprisingly, these projects have been based on early results of the work on IN and on ODP. The same remarks as those that have been made with respect to IN and ODP also apply here.

The design of protocols has been subject of considerable research effort (see [Probert91], for example). Among others, a number of methods have been proposed that start from a service definition to design a protocol that provides the given service. The state of the art of protocol design has advanced considerably during the last decade, also due to progress made in the area of formal methods and support tools. Most of the methods are, however, oriented towards the design of protocols in general or towards the design of data transfer protocols in particular. The design of application protocols, as a separate area, has hardly attracted any attention (papers that explicitly consider the design of application protocols include [Clark90], [Feldhoffer92], [Solvie92], and [Box93]). Furthermore, the abstraction spectrum covered by these methods is generally rather limited. For example, the design of application protocols in the context of the design of distributed system applications is usually not considered. Furthermore, many methods are based on the use of a particular specification model or language which in some respects constrains the designer. Although it is expected that any practical method will impose such constraints, it should be clear to the designer when and where such constraints apply.

1.7 General concepts

This section presents some general concepts that will be used in this thesis. These concepts are well-known in the sense that they are often used in the context of distributed systems design. Nevertheless their use is not always consistent and the relationship between the concepts is not always clear. The interpretation of concepts given in this section will be used throughout this thesis. Some concepts will be further refined in later

chapters. In addition, alternative interpretations (in particular, interpretations used in the context of the OSI-RM) of some concepts will be considered in later chapters.

- *design process*: a set of activities aiming at the conception of a (in our case, technical) system. The definition of the *conceived system* should allow the manufacturing of a *real system*. A design process generally starts from a set of *user requirements*, i.e. requirements on the system that relate to the workings of the system as far as observable and relevant to the future users of the system. The formulation of user requirements may also be considered part of the design process. In order to determine user requirements it is necessary to identify *user needs*, i.e. improvements to the work procedure and work environment of users which can be accomplished through the introduction of a system. Figure 1.2 depicts the design process.

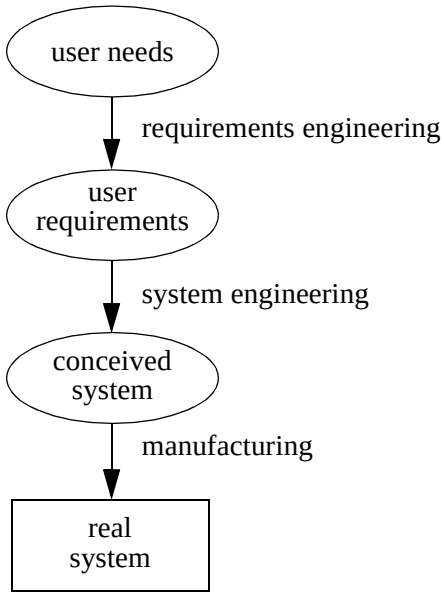


Figure 1.2: Design process

- *design*: an abstract representation of the system that is the object of concern in a design process. Systems may be represented at different *abstraction levels*. Consequently, a design process may have as intermediate results designs at different abstraction levels.

- *design phase*: a distinguished phase in a design process. It is possible to divide the design process into a number of consecutive phases, where each next phase produces a design at a lower abstraction level (Blaauw76]):
  - *architectural phase*: part of the design process that aims at transforming user requirements into a system architecture. A (system) *architecture* is a design that

defines the function of the system, as perceived by its users, that satisfies the user requirements.

- *implementation phase*: part of the design process that aims at transforming a system architecture into a system implementation. A (system) *implementation* is a design that defines the method embodied by the system. This method implements the function defined by the architecture.
- *realization phase*: part of the design process that aims at transforming a system implementation into a system realization. A (system) *realization* is a design that defines the means employed by the system. These means are used to realize the method defined by the implementation.

The concepts of architecture and implementation can be applied recursively in the design process: an implementation may act as an architecture when further details of the system are being considered in the design process. A general characterization of the relationship between architecture and implementation is that an architecture defines the 'what' of the system, while the corresponding implementation defines the 'how' of the system. We distinguish between two types of architecture-implementation relations:

- *horizontal implementation*: a relation where the implementation refines the interfaces defined by the architecture, but does not consider the internal workings of the system. For example, the implementation may define concrete representations of the information values that are exchanged between the system and its environment.
- *vertical implementation*: a relation where the implementation introduces a possible internal workings of the system, consistent with the function of the system as defined by the architecture. For example, the implementation may define a composition of the system in terms of interconnected system parts, where each system part is represented by its architecture.

The concept of realization is applied only once in the design process; it is the lowest level design that can be considered in the design process. Figure 1.3 illustrates the use of design phases in a design process.

- *distributed system*: a distributed system is a system that, at some stage in the design process, will be represented by a composition of interconnected system parts[1]. There are two extreme approaches to the design of distributed systems: either the interactions between system parts are separately considered during design, or they are not separately considered. The design of the interactions between system parts is called interac-

1. This definition relates to an objective of the design process. It does not consider particular properties that distinguishes a distributed system from a non-distributed system.

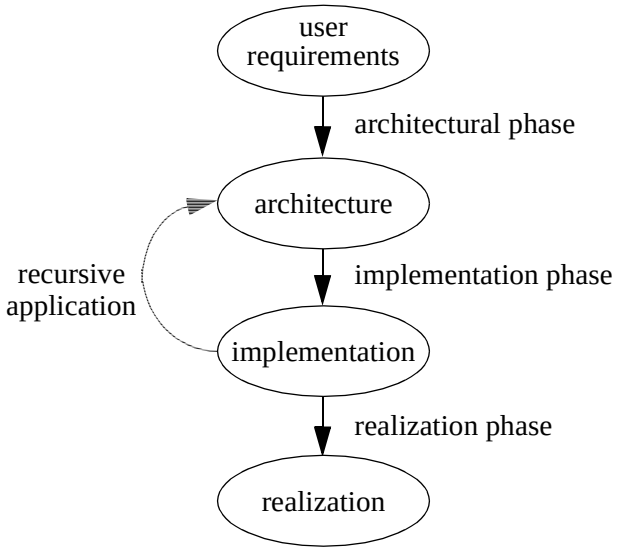


Figure 1.3: Design phases

tion system design. An *interaction system* is formed by the set of related interactions between two or more system parts.

Whether or not interaction system design is part of the design process depends on the user requirements and on the objectives of the designer. In the following situations, interaction system design should be considered:

- if the relation between system parts is complex. In this case, proper attention should be given to the design of the relation between system parts. This is possible if this relation is made a separate object of design, i.e. if the interaction system of the system parts is considered separately. Consideration of the interaction system is possible at different abstraction levels in order to cope with the complexity of the relation. The design of the interaction system implies explicit attention to design choices that concern the effectiveness and efficiency of interactions.
- if it is easier to define an architecture of an interaction system than the architectures of the system parts that interact. This may be the case if the functionality of the system parts is still in part unknown, or if the architectures of the system parts are relatively complex because it must take account of the characteristics of the means of interconnection between the system parts.
- if it is more likely that interactions are changed than just the contributions to interactions by individual system parts. This is the case if improved performance or reliability will be achieved by alternative interaction mechanisms rather than by faster or more reliable processors. An interaction mechanism can only be replaced by another, functionally equivalent interaction mechanism if the function of the mechanism is clearly indicated in the design. This is naturally supported with interaction system design.

The two approaches to distributed system design allow two alternative views on distributed systems, viz. a view in which the system parts are recognized as separate objects of design and a view in which the interaction systems are recognized as separate objects of design ([Vissers77]). Any distributed system can be understood and represented using either view. This is depicted in Figure 1.4. It is also possible that both views are combined. For example, general-purpose or generic interaction systems may be defined by standardization organizations. Users can extend these systems using either approach. Figure 1.4 also depicts the combined view.

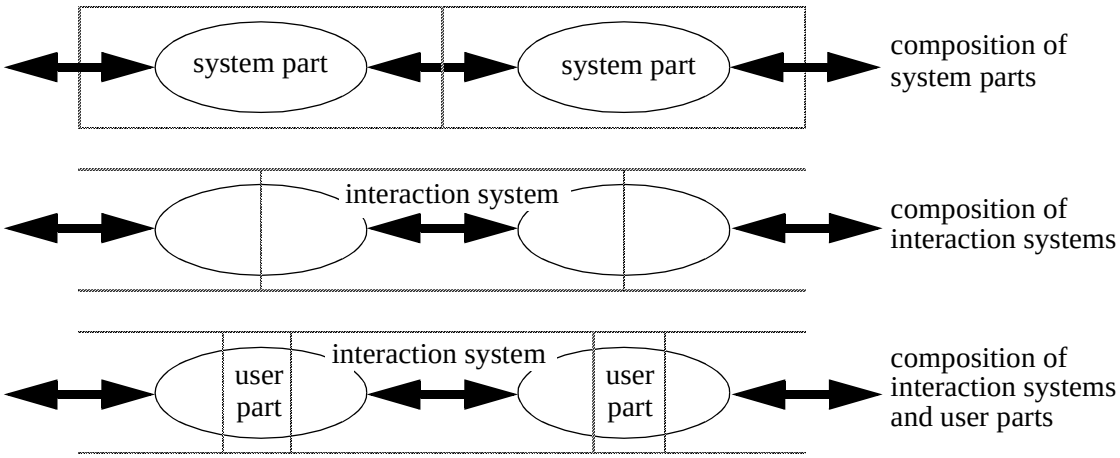


Figure 1.4: Different views on a distributed system

- *service*: architecture of an interaction system of system parts. According to our definition of architecture, a service defines the function of an interaction system as perceived by its users. The boundary between an interaction system and its users is internal to the system parts that are involved in the interaction system. The functions offered at this boundary can therefore be defined in *abstract terms* by the service. The users of an interaction system are called *service users*.

- *protocol*: implementation of an interaction system of system parts. According to our definition of implementation, a protocol defines the method embodied by the interaction system such that the service is provided to the service users. This means that the protocol defines a set of related distributed interactions between the system parts. Distributed interactions are defined by the contributions made by the system parts to these interactions. The information that is exchanged between system parts as a result of these contributions must be defined in *concrete terms*. Only part of each system part is involved in an interaction system. This part is called a *protocol entity.*

Protocol mechanisms can be divided into three classes:

- *application protocols*: these protocols define distributed interactions that directly support the establishment of information values relevant to the application service users. The mechanism for establishment may vary from simple passing of information values to complex processing of information values. Applications protocols rely on data transfer protocols to transfer binary encoded information values.
- *data transfer protocols*: these protocols define distributed interactions that enable the transfer of binary encoded information. The transfer should be performed in accordance to the quality agreed with the data transfer service users. Examples of quality parameters are reliability, throughput, and transfer delay. Data transfer protocols rely on transmission protocols to transfer sequences of bits across transmission media.
- *transmission protocols*: these protocols define distributed interactions that enable the transfer of sequences of bits across particular transmission media. The quality of the transfer depends on the distance that needs to be bridged and the transmission media that are used for this.

The term *communication protocol* is often used, sometimes to denote the general concept of protocol, and sometimes to denote just data transfer protocols. We think that the term is confusing in both cases: application protocols, for example, generally do more than just information transfer, while data transfer protocols only provide ‘transparent’ data transfer, i.e. they are not concerned with the meaning of the data. We will therefore not use this term further in this thesis.

- *interaction system design process*: a set of activities aiming at the conception of an interaction system. The architectural phase of this design process yields a service, and the implementation phase yields a protocol. A protocol defines the contributions of protocol entities to interactions. These contributions are called *protocol actions*. Protocol actions, and thus protocols, can be defined at different abstraction levels. The highest level is one that only represents the function of protocol actions. At this level, a *protocol architecture* is defined. At subsequent lower levels, the mechanism embodied by the protocol entity and the means employed by the protocol entity can be considered, which leads to the definition of a *protocol implementation* and a *protocol realization*, respectively. The design phases involved are called the *protocol design phase*, the *protocol implementation phase*, and the *protocol realization phase*, respectively. Figure 1.5 illustrates the phases in the design of an interaction system. The following should be noted with respect to these phases in relation to the design phases in the system part design process:
  - a service already defines aspects of the implementation of the system parts whose interactions are considered. It can only be defined if the designer has the characteristics of the system parts in mind;

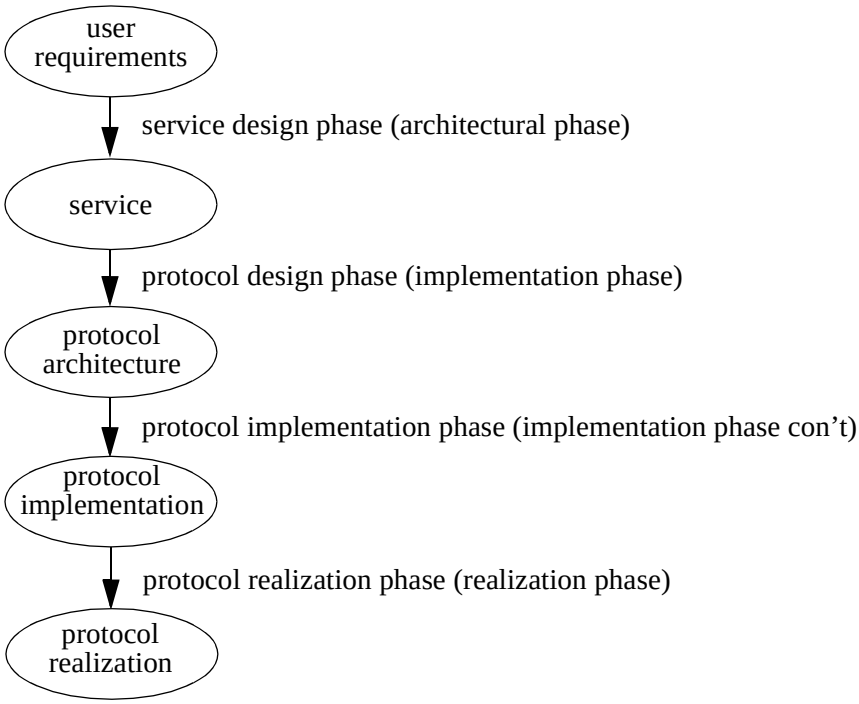


Figure 1.5: Design phases in the design of an interaction system

- a protocol architecture defines further aspects of the implementation of the system parts. Given the protocol, the implementation of the system parts can be completed in isolation;
- connected system part architectures imply a definition of the interactions between the system parts. Thus a system part architecture can only be defined if the designer has the characteristics of the interactions with other system parts in mind;
- the protocol implementation phase and the protocol realization phase consider the same aspects as the system part implementation phase and the system part realization phase, respectively. Hence, these phases are the same in nature (i.e., have the same design objectives).

Figure 1.6 shows the interaction system design process where the different protocol classes are recognized. The shaded areas in the figure mark the scope of this thesis.

References

[Barr92] Barr, W.J., Boyd, T., and Post, M.J., The telecommunications information networking architecture initiative, In: *Open Distributed Processing*, Meer, J. de, Heymer, V, and Roth, R. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 81-98.

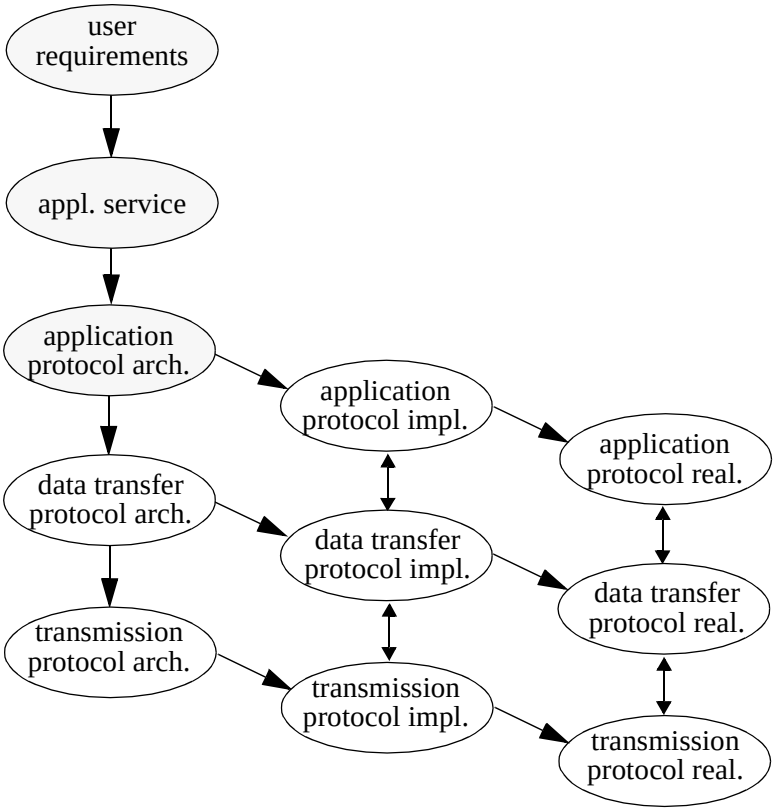


Figure 1.6: Recognition of protocol classes in the design of an interaction system (the shaded areas mark the scope of this thesis)

[Blaauw76] Blaauw, G.A., *Digital system implementation*, Prentice-Hall, Inc., 1976.

[Box93] Box, D.F., Schmidt, D.C., and Suda, T., ADAPTIVE - an object-oriented framework for flexible and adaptive communication protocols, In: *High Performance Networking, IV*, Danthine, A, and Spaniol, O. (editors), Elsevier Science Publishers B.V. (North-Holland), 1993, 367-382.

[Campolargo94] Campolargo, M., Service engineering in RACE and its relation to ODP, In: *Open Distributed Processing, II*, Meer, J. de, Mahr, B., and Storp, S. (editors), Elsevier Science Publishers B.V. (North-Holland), 1994, 117-126.

[Clark90] Clark, D.D., and Tennenhouse, D.L., Architectural considerations for a new generation of protocols, *Computer Communication Review*, Vol. 20, No. 4, September 1990, 200-208.

[Day83] Day, J.D., and Zimmermann, H., The OSI reference model, *Proceedings of the IEEE*, Vol. 71, No. 12, December 1983, 1334-1340.

[Dorros92] Dorros, I., Telecommunications in the US: diversity, change and success, *International Switching Symposium*, Yokohama, Japan, October 1992, 20-26.

[Feldhoffer92] Feldhoffer, M., Communication support for distributed applications, In: *Open Distributed Processing*, Meer, J. de, Heymer, V, and Roth, R. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 215-228.

[Hartmanis92] Hartmanis, J., and Lin, H., *Computing the future: a broader agenda for computer science and engineering*, National Academy Press, 1992.

[IS7498:84] ISO, *Open Systems Interconnection - basic reference model*, International Standard ISO 7498 Part 1, 1984.

[IS9545:89] ISO, *Application layer structure*, International Standard ISO 9545, 1989.

[IS10746:94] ISO, *Basic reference model of open distributed processing*, Draft International Standard ISO 10746 Part 1-4, 1994.

[Kung92] Kung, R., Rational for intelligent networks, In: *Open Distributed Processing*, Meer, J. de, Heymer, V, and Roth, R. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 69-79.

[Linington92] Linington, P.F., Introduction to the open distributed processing basic reference model, In: *Open Distributed Processing*, Meer, J. de, Heymer, V, and Roth, R. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 3-13.

[Natarajan92] Natarajan, N., and Slawsky, A framework architecture for information networks, *IEEE Communications Magazine*, April 1992, 102-109.

[Probert91] Probert, R.L., and Saleh, K., Synthesis of communication protocols: survey and assessment, *IEEE Transactions on Computers*, Vol. 40, No. 4, April 1991, 468-475.

[SA91] *Scientific American*, special issue on Communications, Computers and Networks, September 1991, Vol. 265, No. 3.

[Solvie92] Solvie, G., A flexible open systems architecture satisfying modern communication requirements, In: *International Workshop on Advanced Communications and Applications for High Speed Networks*, Munich, Germany, March 16-19, 1992, 383-392.

[Vissers77] Vissers, C.A., *Interface: definition, design, and description of the relation of digital system parts*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1977.

Chapter 2

Design Quality Criteria

This chapter discusses what qualities are desirable of an application protocol design, and what quality criteria can be used when evaluating or engineering an application protocol design. We recognize that design quality must be enabled by the design model that is used in composing designs, and that quality achievement can be supported through a design methodology. The quality criteria presented are not peculiar to application protocol design, but are applicable to design in many other areas, including data transfer protocol design. We illustrate the criteria with examples taken from the OSI Session Layer standards.

The purpose of this chapter is to provide a basis for evaluating the quality of designs and for successfully pursuing quality when engineering designs. This objective is motivated by the fact that quality achievement is one of the major requirements on the design methodology that is developed in this thesis.

The structure of this chapter is as follows: section 1 introduces the problem of design evaluation: what are desirable qualities and what are adequate measures of quality? Efficiency and ease of use are considered as two possible qualities that could be evaluated. Ease of use is taken as the principal yardstick for assessing overall quality; section 3 introduces criteria for evaluating ease of use; section 4 discusses quality in relation to the use of a design model, a specification language, and a design methodology; and section 5 presents the conclusions of this chapter.

2.1 Design evaluation

Design starts out from user requirements, or from user needs, that have to be satisfied, subject to a cost function. Possibly a minimum set of user requirements and a maximum cost are set. The task of the designer is to maximize the number of requirements that are satisfied by the design and to minimize the cost during the design process, given these boundary conditions. It seems natural, therefore, to evaluate a design on basis of requirements-satisfaction and cost. There is, however, a major problem with (high level) design

evaluation. A design represents a class of possible implementations and realizations. Hence, it must be evaluated independently of any *specific* implementation or realization: only the design choices incorporated in the present design should be considered. Design (implementation) choices should not be anticipated since alternative choices may exist that would change the conclusion of the evaluation. This makes the selection of appropriate criteria for the evaluation of a design generally much harder than that of a realized system.

User requirements normally include requirements related to *functionality* and requirements related to *non-functional aspects*, such as performance, availability, etc. The cost factors most often observed during design are *development* cost and *manufacturing* cost. Development costs concern all phases of design, viz. the architecture, implementation, and realization phase. *Maintenance* constitutes another important cost factor: the costs of modifying the realized system, and possibly its design, during its lifetime.

In this section we consider two possible qualities that could be evaluated since they both consider requirements and costs, although in different extents. These qualities are efficiency and ease of use.

2.1.1 Efficiency

One approach to design evaluation is to consider *efficiency* as the most important quality of a design. The efficiency of a design is concerned with the performance of derived real systems and the costs to manufacture these systems[1]. As mentioned above, performance and costs should be measured independently of specific implementations or realizations of the design. Such measures may be hard to find and may be rather inexact, depending on the level of abstraction of the design. For example, the efficiency of a service is harder to estimate than the efficiency of a protocol. Efficiency evaluation of a service must rely on the number and the complexity of service primitives involved, the complexity of the relationships between service primitives, and the amount of service state information. Efficiency evaluation of a protocol (or a protocol stack) can be based on more detailed information: the number of protocol data units (PDUs) exchanged, the amount of protocol control information in PDUs, the complexity of the encoding and decoding rules, and the protocol state that is maintained.

It should be noted that a protocol architecture still does not permit precise estimations of performance and costs since protocol actions may be implemented in many different ways, and with different kinds of efficiency. Most approaches express protocol efficiency

1. We may distinguish between many forms of efficiency. For example, it is possible to take account of the relative importance of different aspects of performance or of cost related factors such as expectations about the number of product sales, the time before replacement, etc. Such refinements are not considered here since they are not relevant to the present discussion.

as a combination of system efficiency, which is inversely proportional to the computational complexity of protocol actions, and network efficiency, which is inversely proportional to the amount of control information that is exchanged. A possible approach to evaluating system efficiency is by assigning weights to protocol actions, representing the relative computational complexity of the actions. In [Ravindran93], for example, application protocol efficiency is evaluated based on the relative complexity of state transitions and the number of messages exchanged.

In general, however, the relationship between aspects directly represented by a design and its efficiency is not straightforward. Moreover, there are no general principles or rules which can be used to evaluate the efficiency of a design solution, independent of the level of abstraction.

Although efficiency is an important quality of a design, it is not the only quality, and it is even debatable whether it is the most important quality when overall costs are considered. For example, efficiency is not concerned with overall development and maintenance costs, neither is it concerned with functionality as a design variable.

2.1.2 Ease of use

Ease of use of a design denotes the quality of the design to be used in a straightforward way. A design has different types of users. Assuming that the design is an *architecture*, e.g. a service, these users are (see also Figure 2.1):

- the architect, who has to conceive and maintain the architecture;
- the implementor, who has to interpret the architecture in order to transform it into an implementation; and
- the system user, for example an application programmer, who sees the realized system through its architecture and thus uses the architecture to learn and operate the system.

Ease of use has a different meaning for each of these types of users:

- the architect likes the architecture to be robust to changes that concern its implementation, its realization, or the way it is used. For example, it should be possible to change the architecture's implementation, e.g. to replace it by a more efficient implementation, without changing the architecture. On the other hand, if changes are necessary that effect the architecture, e.g. to satisfy additional user requirements, the architect chooses the architecture to be easily adaptable, without requiring major redesign;
- the implementor prefers an architecture that is precise and unambiguous, without unnecessarily constraining implementation options; and
- the system user chooses an architecture that is easy to learn and easy to use, since this will facilitate job performance and improve productivity.

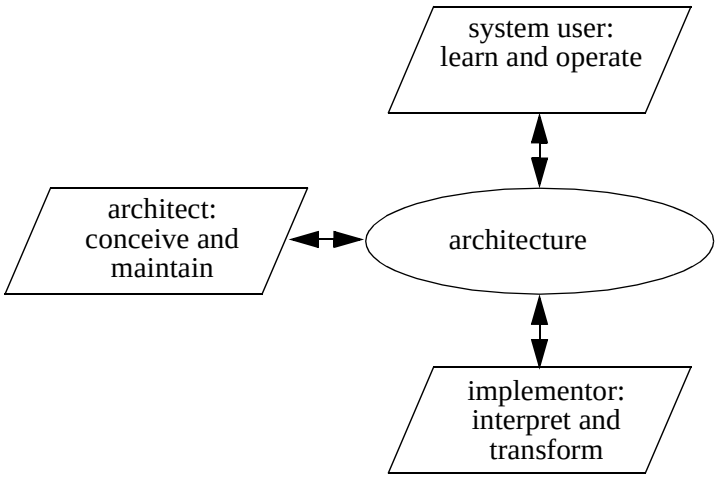


Figure 2.1: Different types of users of an architecture

Similar arguments apply to an implementation, e.g. a protocol that implements a service, since an implementation has to be maintained as well, must be transformed into a realization, and is used through the architecture that it implements. Notice that an implementation may act as an architecture for a lower level implementation. For example, a protocol architecture defines protocol actions as integrated functions. A protocol implementation replaces these integrated functions by implementations, i.e. specific applications of an underlying computer system or operating system architecture.

Ease of use will certainly optimize functionality: omitting required functions will decrease job performance for (some of) the system user(s). One the other hand, providing too many, too specialized functions will again hamper ease of use. It is unclear at first glance, however, how ease of use will effect cost and performance optimization. The achievement of ease of use of a design requires attention and effort of the designer, and thus even appears to increase development costs.

We believe, however, that the overall costs of developing, using and maintaining a system will be less if ease of use is pursued during design. This is confirmed by experience with designing all kinds of (complex) systems. With respect to an architecture, the following arguments apply:

- if the architecture is robust and adaptable, it is cheap to maintain. Moreover, it facilitates future changes of the implementation or realization that would improve performance;
- if the architecture does not constrain implementation, the implementor has maximum freedom to optimize cost and performance of the implementation; and

- if the architecture is easy to learn and use, user education is cheaper and user productivity is higher.

Again, similar arguments apply to an implementation.

Optimizing ease of use during design thus leads to an optimization of overall functionality, overall performance, and overall, long-term costs. For this reason we will take ease of use as the principal yardstick for evaluating designs.

2.2 Quality criteria

In this section we discuss some criteria for evaluating ease of use. The criteria are based on a decomposition of ease of use into a number of sub-qualities. A design that is easy to use should be correct, clearly defined, and ‘clean’.

The necessity of a *correct* design is obvious. A design should not contain errors or otherwise the derived realized system will not work or will not work properly. Since correctness depends on the user requirements, no general criteria can be provided.

A *clearly defined* design is a design that is represented without ambiguities and inconsistencies, in terms of the elementary design concepts available to the designer. It should also be represented at the *right abstraction level*. Also this quality is obvious. Since clearness depends on the contents of the design, no general criteria can be provided in this case either. Clearness also depends on the choice of elementary design concepts, which may be furnished by a predefined design model. The impact of a design model on ease of use will be discussed later in section 2.3.

Cleanliness ([Blaauw85], [Roman85]) is concerned with the conceptual integrity of the design, which in turn is strongly related to the choice of functions that collectively determine the functionality defined by the design. A *clean* design is well structured, well balanced, well fit. Hence, cleanliness is basically an aesthetic quality, and therefore to some extent a matter of taste, not completely arbitrary but also not rigidly defined. It is possible to formulate a number general principles or rules which can act as cleanliness criteria. Designs can be evaluated with respect to their compliance to these criteria, although, as mentioned before, there are no clear boundaries between ‘bad’, ‘good’, and ‘better’. Still cleanliness criteria can be very useful in evaluating the overall quality of a design. Cleanliness criteria are not restricted to application protocol design, but have proven to be useful in many design domains. For example, in [Blaauw75] they were used to assess the quality of computer architectures, in [Vissers88] they guided the development of styles for service and protocol specification, and in [Scollo93] they were taken as touchstone in the engineering of logics for data type specification.

The importance of cleanliness is often underestimated, whereas correctness and clearness are always important objectives during design. For example, cleanliness is often sacrificed for efficiency reasons ([White93]). In the case of service and protocol standards, cleanliness is normally not improved after the first release of a standard. Reported defects related to errors, ambiguities and inconsistencies, on the other hand, are always corrected (which is not surprising since such defects have an immediate effect on the interworking of real systems).

Each of the cleanliness rules will be explained in a separate subsection below. These explanations will be in general terms, referring to two levels of design, viz. architecture and implementation. When architecture is mentioned, this can also be understood as service, or, more specifically, application service. When implementation is mentioned, this can be understood as protocol, or application protocol. Similarly, architecture functions can be understood as service functions (service elements), and implementation functions as protocol functions (elements of protocol procedure). Also, small examples of violations of the cleanliness rules are presented. All examples are taken from the first version of the OSI Session Layer standards, defined by ISO: the basic connection oriented Session Service definition ([IS8326:87]), and the basic connection oriented Session Protocol specification ([IS8327:87]). The examples are based on our experience with these standards through their formal specification in LOTOS ([Scollo87, Sinderen89]).

2.2.1 Consistency

"do not include what is conflicting with previous design choices"

A consistent design exhibits regularity and suggests a single, coherent line of reasoning behind it. This regularity makes it possible to anticipate parts of the design once other parts are known ([Blaauw85]). Hence a consistent design confirms our expectation and, conversely, an inconsistent design contradicts our expectation.

For example, an architecture or implementation should define uniform constraints on the use of functions and function features, independently of when they are used and in what context they are used, whenever possible. Consistency is needed above all because people are involved in using a design. Its main purpose is to facilitate understanding, learning, and using a design.

An example of a violation of the consistency rule can be found in the constraints on the Synchronization Point Serial Number (SPSN) parameter in the Resynchronize function and the Activity Resume function of the Session Service and Protocol. A SPSN is used by the Session Service users to identify a common synchronization point; the Session Protocol constrains the value of each SPSN that is used by the Session Service users. The Resynchronize function is provided to assist orderly re-establishment of communication. It sets the session connection to an agreed, hence defined, state. The value of the SPSN

that is agreed between the users is the next SPSN that will be used. The Activity Resume function is provided to indicate that a previously interrupted activity is resumed. It also sets the session connection to a new defined state. The value of the SPSN that is agreed between the users is the next SPSN to be used *minus one*. It would have been consistent, however, if this value had the same meaning as in the Resynchronize function.

2.2.2 Orthogonality

"do not link what is independent"

Orthogonality calls for the separation of aspects of a system that are independent of each other. For example, functions that are independently needed by system users should be separated in the architecture. Similarly, implementation functions that are independently needed in support of the architecture's functionality should be separated, i.e. assigned to different entities in the implementation.

The principle of orthogonality complies with that of *separation of concerns*. A special case of this principle is *encapsulation*, or *information hiding* ([Parnas72]), which is used in structuring implementations. Encapsulation leads to a structure of entities, where each entity hides as much as possible of its inner workings from the others. Encapsulation is the leading principle in object oriented approaches to system design ([Booch86]).

The rule stated above does not prohibit the separation of aspects that are largely, but not completely, independent. The benefits of orthogonality still apply to some extent in these cases. On the other hand, the complementary rule to the one stated above should also be respected:

"do not separate what is dependent"

Thus, functions that are strongly related should *not* be separated in an architecture, or assigned to separate entities in an implementation. Otherwise complex relationships between functions, and complex interactions between entities, have to be defined.

Orthogonality primarily supports maintenance: it permits to make changes to functions without (much) effecting other functions, and to make changes to entities' implementations without (much) effecting other entities. Orthogonality also facilitates learning and using a design: orthogonal functions can be understood on their own and can be used independently. Finally, orthogonality supports concurrency of design: a subdivision of a design into orthogonal functions permits the independent implementation of these functions.

An example of a violation of the (first) orthogonality rule can be found in the constraints associated with the Resynchronize Type parameter in the Resynchronize

function of the Session Service and Protocol. The Resynchronize Type parameter is used to indicate one of three possible resynchronize options (abandon, restart, or set). The choice of an option effects the Resynchronize function in two respects. First, each option implies different constraints on the value of the Synchronization Point Serial Number (SPSN) that can be used by the Session Service users. And second, each option implies a different 'precedence' which is used by the Session Protocol to resolve collisions of function invocations that involve at least one invocation of the Resynchronize function. Since the semantics of synchronization points is transparent to the Session Protocol, the range of synchronization points from which the users may select, and the precedence of a resynchronization attempt can better be treated as independent aspects at this level, which can be served by separate parameters in the Resynchronize function.

2.2.3 Propriety

"do not introduce what is immaterial"

Aspects that are defined by a design should be proper to the purpose of the system, where the purpose of the system is determined by the user requirements. For example, functions which are not required by the system users and whose usefulness is not clear are better omitted in the architecture.

Propriety also demands that aspects that are defined by a design should be proper to the purpose of the design, i.e. correspond to the *level of abstraction* of the design. Thus, an architecture should not define aspects of the implementation that are irrelevant to the system user. And an implementation should not define aspects of functions that are only relevant to a lower level implementation, or a realization. This allows a clean division of work, with proper attention of the designers to problems that are their concern and maximum freedom to resolve these problems.

Propriety implies *parsimony.* Not only should designs only define relevant functions, corresponding to essential, well understood user and domain requirements, they also should define each function only once. Adherence to parsimony avoids abundance in the architecture, thus aiding its comprehensibility, and avoids redundancy in the implementation, contributing to efficiency.

An example of a violation of the propriety/parsimony rule can be found in the basic concatenation function of the Session Protocol. Basic concatenation yields a concatenated sequence of two session protocol data units (SPDUs) which is mapped onto a single transport service data unit (TSDU). The first SPDU in a basic concatenation sequence is a so called category 0 SPDU (either a GIVE TOKENS SPDU or a PLEASE TOKENS SPDU), the second SPDU is a category 2 SPDU (a class of various types, including the DATA TRANSFER SPDU). Category 2 SPDUs are *never* mapped one-to-one onto a TSDU. This means that if a category 2 SPDU has to be sent, it is either appended to a category 0 SPDU

which also has to be sent, or it is appended to a ‘dummy’ category 0 SPDU which is generated for the purpose of sending the category 2 SPDU. In the latter case the basic concatenation function does not serve any real purpose (the constraint that a category 2 SPDU is never sent alone is inherited from a predecessor of the Session Protocol, CCITT Recommendation T.62, and has been retained for compatibility reasons).

2.2.4 Generality

“do not restrict what is inherent”

Aspects covered by a design should be defined in their most general form. Thus, functions of the architecture should be general purpose or generic, as opposed to being tailored to the current conception of how the architecture will be used. Also entities of an implementation should be general purpose or generic, permitting their use in several contexts, and, consequently, permitting the composition of several more complex or more specialized functions. An implementation that allows alternative compositions of entities can adapt to different user environments, or a changing environment, and thus supports *flexibility* and *adaptability.*

Generality implies *open-endedness*, i.e. the property of allowing future extensions. An open-ended system does not require modifications of its architecture, or, worse, its implementation, in order to be used in supporting an extended functionality. Instead, the original architecture and implementation can be extended with appropriate functions and entities, respectively. Generality thus facilitates maintenance.

Generality also allows *re-use* of functions of an architecture and of entities of an implementation. Generic functions or entities of a design can be re-used in other, new designs that possess some degree of similarity with the existing one. This property clearly aids cost-effectiveness of design.

In order to be general, a design should be *complete*. For example, an architecture should include all functions that are proper to its purpose. If not, at least some intended user environments can not use the architecture. Also each function should be complete, permitting all possible uses that are proper to its purpose. A complementary rule to the one above should therefore be respected as well ([Scollo93]):

“do not forget what is relevant”

To illustrate generality, we use the example that was used in subsection 2.2.2 to illustrate orthogonality. The constraints associated with the Resynchronize Type parameter in the Resynchronize function of the Session Service and Protocol imply that only specific combinations of resynchronization point ranges and resynchronization preferences can be used. For example, it is not possible to use a Synchronization Point Serial Number with a

value that is greater than the value that is most recently used, and a low precedence. Hence, the Resynchronize function is not complete in this respect, and may not be appropriate to some user environments. Another consequence of the provision of rather arbitrary options is that some options may never, or hardly ever used (indeed, the set option is not used by any of the currently defined Application Layer protocols).

2.3 Quality and design engineering

This section discusses the relationship between design engineering, i.e. the activity of developing a design, and the achievement of quality characterizing a clean design. Three elements of the design process will be considered, viz. the design model, the specification language, and the design methodology used in the design.

2.3.1 Relation to design models

Often a designer has a predefined set of concepts at his disposal which form the basic building blocks in which designs have to be expressed ([Gotzhein93], [Ferreira93]). These concepts are the elementary *design concepts*, or architectural concepts, which together with rules for their composition form a design model or design language (to be distinguished from a specification language, see subsection 2.3.2). Since complex designs need to be structured, a design model may also include *structuring techniques* that yield particularly useful structures in the considered design domain. Figure 2.2 illustrates the application of a design model.

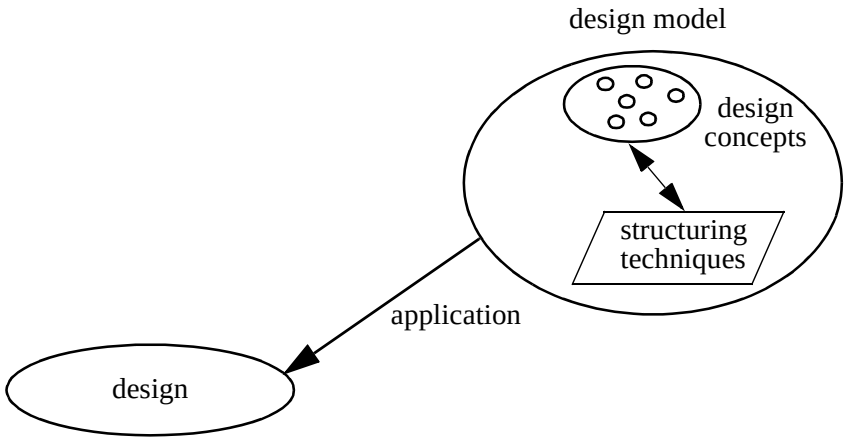


Figure 2.2: Application of a design model

Since a design is expressed using the design model, the quality of a design is indirectly determined by the suitability of the design model. What, then, constitutes a suitable design model? Specific requirements on a design model are determined by the design domain, in our case application protocol design. A general requirement with respect to a design

model is ease of use, and the same criteria apply as have been proposed for the evaluation of design quality:

- *consistency*: design concepts should be consistent in their representation of, or abstraction from, aspects of the 'real world';
- *orthogonality*: distinct design concepts should be used to represent different, independent aspects;
- *propriety*: design concepts should be proper to the needs of the design domain as well as straight to the point;
- *generality*: design concepts should be general purpose in the design domain, and the complete set of design concepts should completely cover the needs of the design domain.

Compliance to these criteria leads to a 'clean' design model which satisfies often stated requirements on design models, such as relevance, appropriateness, and completeness (see [Vissers93], for example). A clean design model defines a minimal set of general purpose design concepts that completely covers the needs of the design domain, and so enables designs to be efficiently and effectively expressed. In practice, however, it turns out to be very difficult to determine the right design concepts, mainly because the needs of the design domain are not clear from the start. Only through experience one finds that certain aspects, that are considered proper to the design domain, are cumbersome or even impossible to express with the design concepts available. An iterative process of application and improvement based on application feedback is usually needed to establish the right concepts.

A design domain is generally not focused on a single level of abstraction, but, instead, covers an abstraction spectrum with several distinguished abstraction levels. Design concepts which are proper at one abstraction level may not be proper at another abstraction level. A design model may therefore be subdivided into a number of sub-models, corresponding to abstraction levels, with distinct or partially overlapping sets of design concepts. For example, the concept of PDU is proper at the protocol level, but not at the service level. The concept of service primitive, on the other hand, is proper at both the service level and the protocol level. (One may, of course, decide that a more general purpose concept is needed, of which the concepts of PDU and service primitive are special cases.)

Another desirable quality of the design model, besides cleanliness, is clearness. All design concepts identified should be precisely formulated, without ambiguities or inconsistencies. Sometimes a mapping is defined between elementary design concepts and their representation in a formal specification language in order to satisfy this requirement. Such a mapping determines the so called *architectural semantics* of the specification language ([Turner87], [Schot90]). When defining an architectural semantics one should be careful

about preserving the cleanliness of the design model. For example, design concepts as originally intended should not be inadvertently changed due to limitations of the specification language. Also structures can be represented using a formal specification language (as demonstrated in [Turner95]), although, in practice, the motivation for this is usually not clarification of the structuring techniques but illustration of the applicability of the specification language.

The structuring techniques provided by a design model should not limit the applicability of the design concepts, but should merely aid in composing structures that can be used for separation of concerns (see orthogonality, subsection 2.2.2). Examples of component types, or *high level* design concepts, that are useful in the protocol design domain are service provider, protocol layer, and protocol entity.

2.3.2 Relation to specification languages

A design has to be represented using some specification language. There is a broad consensus among designers that preferably a *formal* specification language should be used for this, provided that the language satisfies besides formality also other important requirements that determine its effective use in practice (see [Weber93] and [Vissers93], for example). Formality enables the precise and unambiguous formulation of the design and permits evaluation of the correctness of the design using language based validation tools. A formal specification language has an underlying mathematical model which determines the choice of (elementary) language elements. This fact, and the fact that the application domain for which the specification language has been defined is usually not the same as the design domain at hand, makes that there is not a one-to-one relationship between elementary design concepts and elementary language elements. It is therefore useful to distinguish between a design and its representation, where the latter is referred to as the design specification. This is illustrated in Figure 2.3.

A formal specification language should be sufficiently expressive in order to support the needs of the design domain. A useful test is the definition of an architectural semantics, i.e. a mapping between the elementary design concepts and specification language constructs that can be used for the representation of these concepts. If this is not possible, a subset of the design concepts may be considered, in case this subset permits expression of designs at least one abstraction level distinguished in the design domain.

Sufficient expressiveness, also called appropriateness ([Roman85, Vissers93]), is the principal requirement on formal specification languages. Other requirements are compositionality, which allows the support of structuring techniques, and the support of a design methodology. Experience with the application of formal specification languages in service and protocol specification indicates that neither of these requirements are completely satisfied by current formal specification languages ([Vissers93]). For example, the formal specification of the OSI Session Service ([IS8326:87]) and Session Protocol

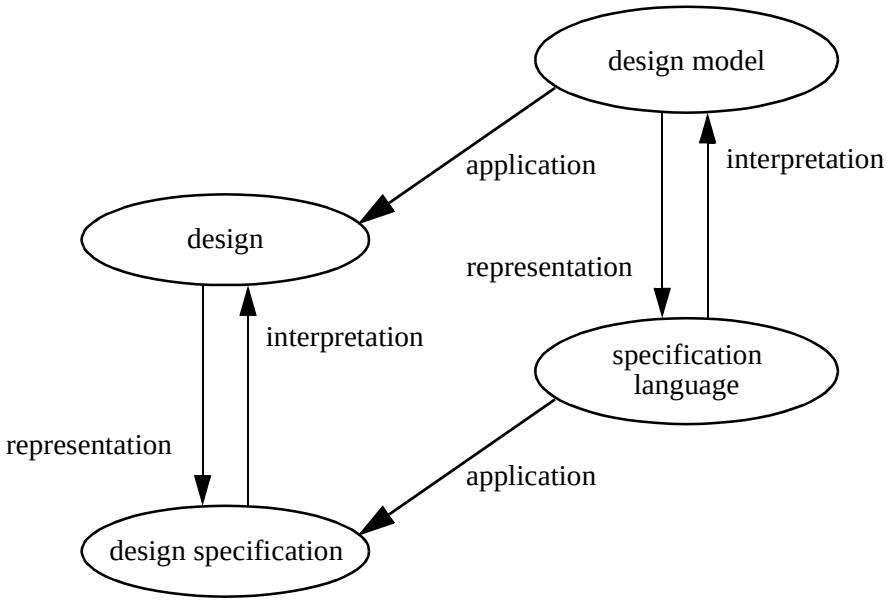


Figure 2.3: Application of a specification language

([IS8327:87]) in LOTOS took about 5 years to complete, which was partly due to lack of resources and to lack of quality of the standards, but certainly also partly to blame on the inappropriateness of the specification language.

2.3.3 Relation to design methodologies

A design methodology is used to *systematically* transform a set of user requirements into a design that satisfies these requirements. Since the 'gap' between requirements and the conceptual solution embodied by the final design is potentially wide, a design methodology should indicate how this gap can be bridged, e.g. by setting out a design trajectory that can be followed by the designer. Each step along this trajectory should serve a well defined purpose, thus permitting division of work and sometimes concurrency of design. The subdivision of the design process into a design (architectural) phase, implementation phase, and realization phase is an example of a design trajectory. Another example is the subdivision into a service design phase and a protocol design phase. Design steps of a design trajectory may also be recursive, in the sense that subsequent steps aim at similar types of transformations. For example, a design step may be concerned with the *decomposition* of an architecture into a composition of functional entities, each of which is again defined by an architecture. The architectures of the functional entities may again be decomposed in the same way.

A design process involves many decisions from the designer, where each decision typically requires a selection from many design alternatives that satisfy the given user requirements. A design methodology should maintain a consistent set of principles or rules

throughout the design process that guide this selection and so ensure conceptual integrity of the design ([Rechtin92]). The *quality criteria* proposed in section 2.2 form such a set. Design decisions are further restricted by technical engineering constraints. A design model is needed that supports the expression of designs at any point along the design trajectory. Since many different types of designs are possible along the design trajectory, the design model may consist of a number of sub-models, each with a different set of elementary design concepts. Different sub-models may require different (formal) specification languages. Figure 2.4 illustrates the application of a design methodology and its 'inputs': user requirements, design model, quality criteria, and technical engineering constraints.

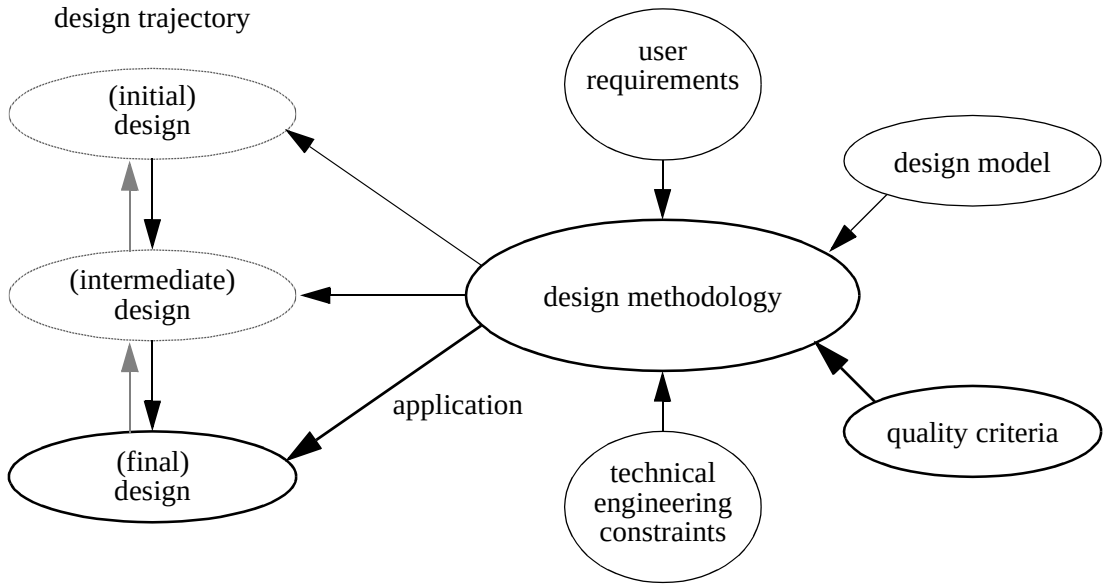


Figure 2.4: Application of a design methodology

Figure 2.4 also shows the design trajectory, which can be followed top-down, but which should also allow back tracking to recover from poor design decisions. The concept of design trajectory helps the designer to position and relate design results achieved during the design process. It should not constrain the designer in applying practical design strategies. Figure 2.5 shows two particular uses of a design trajectory in the design process: a combination of top-down design and back tracking to find optimal design solutions, which can be characterized as tree search, and a cyclic top-down design approach which starts with the design of key functions in the first cycle and considers an extended functionality in each next cycle.

Systematizing the transformation of user requirements into a design solution can be used to improve the quality of the design process, and thus the quality of the final design, in a number of ways. We illustrate this by considering a *step-wise refinement* design meth-

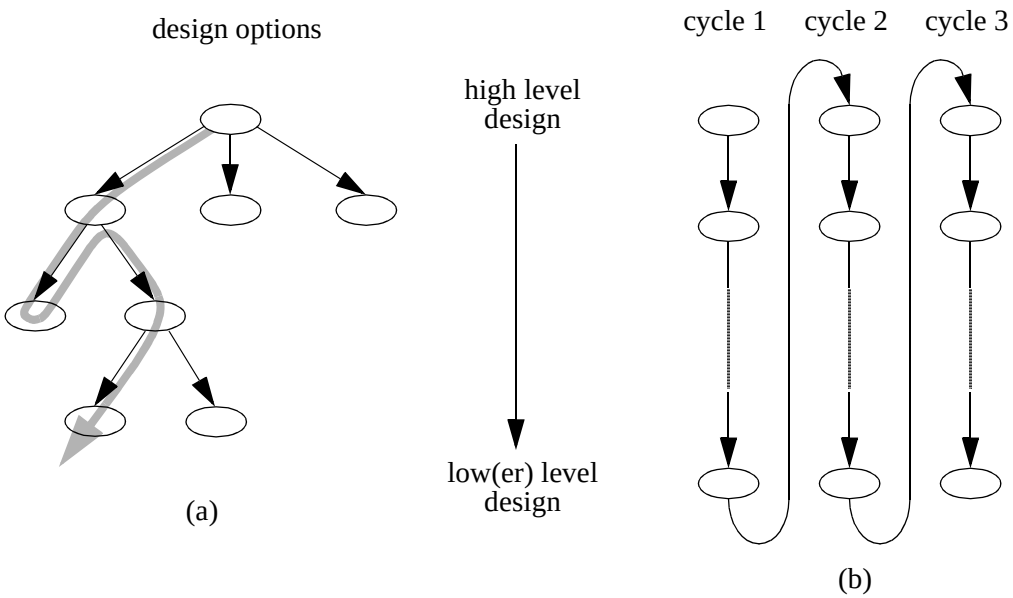


Figure 2.5: Applications of a design trajectory in the design process: (a) tree search, and (b) cyclic approach

odology. Step-wise refinement implies that the design trajectory is organized as a *sequence* of design steps, where the result of one design step is taken as the starting point for the next design step. Each step produces an (intermediate) design which *implements* the design produced by the previous step. This implies that subsequent steps along the design trajectory produce designs with decreasing *abstraction levels*. The advantages of such an approach in the light of quality pursuit are:

- *division of work*: each design step is in principle a unit of work. Although each step depends on the result of the previous step, work may start well before the completion of the previous step based on general directions taken by previous steps. Since each design step has a well defined purpose, with particular design concerns that must be handled in relation to this purpose, also an effective application of dedicated expertise is possible. Furthermore, each step can be validated independently of other steps, thus permitting intermediate and partial validation of the correctness of the design.
- *conceptual integrity*: all design decisions related to a certain abstraction level are taken in a single step by one (team of) designer(s). Quality criteria, such as proposed in section 2.2, can be consistently applied to guide design choices and to produce a clean (intermediate) design. High level design decisions will correctly propagate to lower levels because of the step-wise refinement approach. Consequently, conceptual integrity can be preserved during the design process and can give the impression that the final design, although an heterogeneous group effort, is developed by a small, single-minded group of people.
- *design for re-use*: application of the orthogonality rule and the generality rule in all design steps yields re-useable components (functions or entities) at each abstraction

level. This will enable the identification of similarities among designs at the highest possible level, hence at the earliest possible time in the design process. Specialization of general purpose or generic components at the proper abstraction level will shorten the development time of the final design.

- *domain interaction*: during the design of a system it is necessary to be aware of the relationships with other design domains. Clear boundaries between design domains will lead to a cleaner design from an integrated domain perspective. These boundaries should be provided at the proper, i.e. the highest possible, abstraction level in order to simplify domain interaction and, consequently, to support trade-offs across design domains. Different design domains may have boundaries defined at different abstraction levels. For example, application protocol design is related to (application) software design and (network or distributed) operating system design. The relation to the former is best considered at an application service or application interface level; the relation to operating system design is best considered at an application protocol architecture (with integrated protocol actions) or implementation (with protocol actions defined in terms of operating system calls) level.

The relation to quality is obvious: division of work will concentrate attention and speed up development; conceptual integrity will provide a well balanced, clean design which is easy to maintain, implement, learn, and operate; design for re-use will reduce design effort; and clear domain interaction will facilitate integration of design products from different domains and avoids redundant solutions.

All of the above mentioned advantages rely on the fact that the design methodology employs a set of abstraction levels and exploits the relationships between these abstraction levels. We will therefore consider as desirable characteristics of a design methodology:

- definition of *abstraction levels*: each abstraction level should correspond to a well defined purpose of a design step, i.e. to an identifiable need in the design process. There should be a highest abstraction level that supports the expression of user requirements; there should be a lowest abstraction level that allows the expression of the final design. For each abstraction level a design (sub-) model must be defined that will enable expression of designs at that abstraction level.
- definition of *relationship between abstraction levels*: each next (lower) abstraction level should be related to the previous (higher) abstraction level. This implies that their design models, if different, are related and that of each construct expressed at the lower abstraction level it is in principle possible to establish whether it is consistent with a corresponding construct at the higher abstraction level. This latter relationship is called conformance: if a lower level construct conforms to a higher level construct it is a correct implementation of that construct, and, conversely, the higher level construct is an abstraction of the lower level construct.

A design methodology may incorporate many strategies, methods, and techniques for attacking design problems. Strategies, methods, and techniques, in turn, may include the use of heuristics and quality criteria in order to support identification of, and selection from, design alternatives. Heuristics are rules of thumb, based on experience, and particularly valuable when no algorithmic techniques are applicable, e.g. because of the complexity of the design problem at hand (a situation which is not unusual in distributed systems design). Like the quality criteria for cleanliness, heuristics are not precise and subject to personal judgement, but nonetheless of great value, especially to high level design engineering ([Rechtin92]).

The efficiency of a design process and the quality (in particular correctness and unambiguity) of designs may be improved by the use of formal methods that support design transformation or design validation ([Sinderen92], [Turner93]). Formal methods are most convenient if they are based on the formal specification languages that are also used to represent design results.

2.4 Conclusion

Design engineering should aim at optimizing satisfaction of user requirements related to functionality and performance and at minimizing costs. It is difficult to establish criteria for evaluating the efficiency (performance to cost ratio) of a design in quantitative terms. And even if efficiency if this is possible, this would not provide insight in the overall costs of the design, related to its development, use, and maintenance.

Another quality that can be considered is ease of use, where ease of use is related to conception and maintenance, interpretation and implementation, learning and using a design. Ease of use demands correctness, clearness, and cleanliness. With respect to cleanliness a number of general design quality criteria are presented, which are summarized in Figure 2.6. Although these criteria can be considered as essentially aesthetic criteria, they are also of practical use, especially when pursuing adequate functionality and low overall costs.

The following elements of a design environment influence design quality:

- *design model*: a design model defines the elementary design concepts that are used for the expression of a design, composition rules for these concepts and possibly structuring techniques. If the design model is not suitable for the design domain, the quality of designs expressed with this model will also be poor.
- *specification language*: design concepts must be represented using a (general purpose) design language. Preferably, a formal specification language is used for this purpose. A formal specification language should be capable of representing the elementary design concepts and the structuring techniques defined by the design model. In practice, this is often not the case.

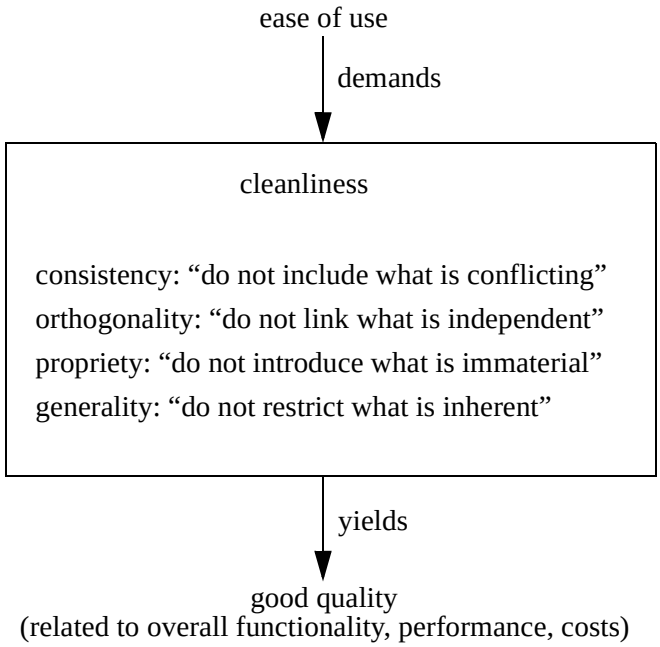


Figure 2.6: Summary of design quality criteria

- *design methodology*: a design methodology serves to provide methods for transforming user requirements into a satisfactory design. The quality criteria proposed here are well suited to application during design engineering. A design methodology that exploits multiple related abstraction levels has some potential advantages that contribute to design efficiency and design quality.

Consequently, the quality criteria proposed in this chapter can be used for design evaluation, but can also be used in a design methodology to support design engineering.

References

[Blaauw85] Blaauw, G.A., and Brooks, F.P. Jr., *Computer architecture*, Lecture notes, University of Twente, The Netherlands, and University of Carolina at Chapel Hill, U.S.A., August 1985.

[Booch86] Booch, G. Object-oriented development, *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, 211- 221.

[Ferreira93] Ferreira Pires, L., Sinderen, M. van, and Vissers, C.A., Advanced design concepts for open distributed systems, In: *Proceedings of the 4th Workshop on*

Future Trends of Distributed Computing Systems, IEEE Computer Society Press, 1993, 419-425.

[Gotzhein93] Gotzhein, R., *Open distributed systems: on concepts, methods, and design from a logical point of view*, Vieweg, 1993.

[IS8326:87] ISO, *Basic connection oriented session service definition*, International Standard ISO 8326 (first edition), 1987.

[IS8327:87] ISO, *Basic connection oriented session protocol specification*, International Standard ISO 8327 (first edition), 1987.

[Parnas72] Parnas, D.L., On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, Vol. 15, No. 12, December 1972, 1053-1058.

[Ravindran93] Ravindran, K. , and Lin, X.T., Structural complexity and execution efficiency of distributed application protocols, *Computer Communication Review*, Vol. 23, No. 4, October 1993, 160-169.

[Rechtin92] Rechtin, E., The art of systems architecting, *IEEE Spectrum*, October 1992, 66-69.

[Roman85] Roman, G.-C., A taxonomy of current issues in requirements engineering, *IEEE Computer*, April 1985, 14-22.

[Schot90] Schot, J., *The role of architectural semantics in the formal approach of distributed systems design*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1990.

[Scollo87] Scollo, G., and Sinderen, M. van, On the architectural design of the formal specification of the session standards in LOTOS, In: *Protocol Specification, Testing, and Verification, VI*, Sarikaya, B., and Bochmann, G. von (editors), Elsevier Science Publishers B.V. (North-Holland), 1987, 3-14.

[Scollo93] Scollo, G., *On the engineering of logics*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1993.

[Sinderen89] Sinderen, M. van, Ajubi, I., and Caneschi, F., The application of LOTOS for the formal description of the OSI session layer, In: *Formal description Techniques, I*, Turner, K.J. (editor), Elsevier Science Publishers B.V. (North-Holland), 1989, 263-278.

[Sinderen92] Sinderen, M. van, and Ferreira Pires, L., Protocol design and implementation using formal methods, *The Computer Journal*, Vol. 35, No. 5, 1992, 478-491.

[Turner87] Turner, K.J., An architectural semantics for LOTOS, In: *Protocol Specification, Testing, and Verification, VII*, Rudin, H., and West, C.H. (editors), Elseviers Science Publishers B.V. (North-Holland), 1987, 15-28.

[Turner93] Turner, K.J (editor), *Using formal description techniques: an introduction to Estelle, LOTOS and SDL*, John Wiley & Sons, 1993.

[Turner95] Turner, K.J., and Sinderen, M. van, LOTOS specification style for OSI, In: *LOTOSphere: software development with LOTOS*, Bolognesi, T., Lagemaat, J. van de, and Vissers, C.A. (editors), Kluwer Academic Publishers, 1995, 137-160.

[Vissers88] Vissers, C.A., Scollo, G., and Sinderen, M. van, Architecture and specification style in formal descriptions of distributed systems, In: *Protocol Specification, Testing, and Verification, VIII*, Aggerwal, S., and Sabnani, K. (editors), Elsevier Science Publishers B.V. (North-Holland), 1988, 189-204.

[Vissers93] Vissers, C.A., Sinderen, M. van, and Ferreira Pires, L., What makes industries believe in formal methods, In: *Protocol Specification, Testing, and Verification, XIII*, Danthine, A., Leduc, G., and Wolper, P. (editors), Elsevier Science Publishers B.V. (North-Holland), 1993, 3-26.

[Weber93] Weber-Wulff, D., Selling formal methods to industry, In: *Formal Methods Europe, I*, Woodcock, J.C.P., and Larsen, P.G., Lecture Notes in Computer Science 670, Springer Verlag, 1993, 671-678.

[White93] White, S., et al., Systems engineering of computer-based systems, *IEEE Computer*, November 1993, 54-65.

Chapter 3

OSI Upper Layer Architecture and Model: State of the Art

This chapter discusses aspects of the OSI-RM related to the development and definition of application protocols. The OSI-RM has been adopted by ISO and ITU-TSS as the major framework for coordinating protocol standardization efforts. In addition, the wide acceptance of the terminology defined in the OSI-RM has contributed to the establishment of a 'common language' for discussing protocol engineering concepts across different protocol architecture communities.

The OSI-RM comprises both a (loosely defined) design model and a reference architecture for protocol design. The design model defines such general concepts as service, service primitive, service data unit, service access point, protocol, protocol entity, and protocol data unit. It also defines the basic structuring technique for OSI: layering. Extensions to this design model were later defined by the OSI Application Layer Structure (OSI-ALS). The OSI-ALS is mainly concerned with additional structuring techniques that must enable a greater flexibility in combining Application Layer standards.

The reference architecture defines a layered subdivision of the overall protocol functionality. This results in a global characterization of four lower protocol layers and three upper protocol layers. The lower layers are concerned with data transfer and transmission functions, whereas the upper layers are concerned with distributed application interactions. We will refer to the design model aspects related to the upper protocol layers as the OSI Upper Layer Model (OSI-ULM) and to the definition of the three upper protocol layers as the OSI Upper Layer Architecture (OSI-ULA).

Requirements with respect to application protocols continuously evolve and therefore require the OSI-ULA to be open-ended. The structuring techniques of the OSI-ULM determine to what extent it is possible to support incremental extension of application protocol functionality and re-use of available application protocol standards.

The purpose of this chapter is to present the OSI-ULM and the OSI-ULA so that their description can be used as a reference for evaluation and comparison. The chapter explains the structuring techniques, and related concepts, that are adopted for application protocol development in the context of OSI. It also explains the subdivision of application protocol functionality according to the OSI-ULA.

The structure of this chapter is as follows: section 1 discusses structuring techniques that have been used for the composition of application protocol standards; section 2 provides an overview of the most important application service and protocol standards of the present OSI-ULA; section 3 presents more details on the Session Layer standards; section 4 presents more details on the Presentation Layer standards; section 5 presents two of the 'building block' standards of the Application Layer; section 6 presents more details on the Distributed Transaction Processing standard; and section 7 presents the conclusions of this chapter.

3.1 Application protocol structuring techniques

We identify three techniques that are available in the OSI-ULM for structuring application protocols:

- *layer composition* (or layering): This is the basic structuring technique in OSI that effects a separation of concerns through a subdivision of the overall protocol functionality into hierarchically related protocol layers. A protocol layer consists of protocol entities that interact peer to peer using the service provided by the underlying layer. Since this lower level service provides transparent data transfer, concerns of the protocol that uses the lower level service can be separated from concerns of the protocol that provides the lower level service. The enhanced functionality accomplished through the interaction of protocol entities in a layer is provided to the next higher layer. Hence, protocol layers only interact directly with adjacent protocol layers in the protocol hierarchy. The principles of layering are described in the OSI-RM ([IS7498:84]). The OSI-RM identifies a fixed number of layers (7), the upper three of which are concerned with application protocols (application-oriented protocols, in OSI terminology). The OSI-RM also outlines the assignment of protocol functions to different layers.
- *application service element composition*: This structuring technique is used in the highest layer of the OSI-RM, the Application Layer, to cope with the broad scope and open-ended nature of this layer. It allows to separate different protocol concerns based on the identification of distinct classes of distributed system applications. Application protocol functions that are to some extent independent of each other are assigned to different Application Service Elements (ASEs). ASEs may be composed in different ways, dependent on the specific class of distributed system applications that is supported. This structuring technique is described in the OSI-ALS ([IS9545:89]).

- *composition of cooperating main service and auxiliary service*: This structuring technique assumes an assignment of application protocol functions to ASEs, as with the previous structuring technique. It is limited, however, to cases where according to the previous technique a hierarchical composition of ASEs would apply. Instead of a hierarchical composition, where one of the ASEs uses the service provided by the other ASE, one of the ASEs directly includes service functions and protocol functions of the other ASE in its own service functions and protocol functions, respectively. The ASE that defines the inclusion (that defines references to service primitives and protocol actions of the other ASE) is called the cooperating main service. The other ASE is called here the auxiliary service (no separate OSI term is available in this case). The application of this structuring technique is rather exceptional. The only application of this technique is currently found in the File Transfer, Access and Management ASE ([IS8571:88]), which describes a possible use of the Commitment, Concurrency and Recovery ASE ([IS9804:90], [IS9805:90]) as auxiliary service.

Each of these structuring techniques is further discussed in a separate subsection below. Related concepts are explained together with the structuring technique in which they are used.

3.1.1 Layer composition

Layering is the basic structuring technique that was used to define the OSI layered protocol architecture (see [Day83] and [Linington83], for example). The overall problem considered by OSI, i.e. the problem of allowing systems to exchange information and to cooperate in distributed applications, is decomposed into manageable pieces, called protocol layers. Each layer is concerned with a layer-specific function which is distributed over all participating systems. The distributed portions of a layer function are assigned to *protocol entities*. A layer *protocol* defines the required interactions of the protocol entities in the layer. The function that results from this interaction and that is offered to a higher layer constitutes a layer *service*. A protocol uses the lower level service, i.e. the service provided by the lower layer, in order to realize the exchange of information between the protocol entities.

In total 7 hierarchically related layers are thus identified in the OSI-RM. The lowest layer is the Physical Layer (layer 1), which uses a combination of available physical media for exchanging information[1]. The highest layer is the Application Layer (layer 7), which does not offer its service to a higher layer[2]. The layered architecture of the OSI-RM is shown in Figure 3.1 (only end-systems, and end-to-end protocols, are shown; a ‘complete’ architecture would also contain intermediate systems).

1. The collection of physical media can also be regarded as a layer (layer 0, to be consistent with the OSI numbering convention), although no protocol can be associated with this layer since its function cannot be distributed over protocol entities.

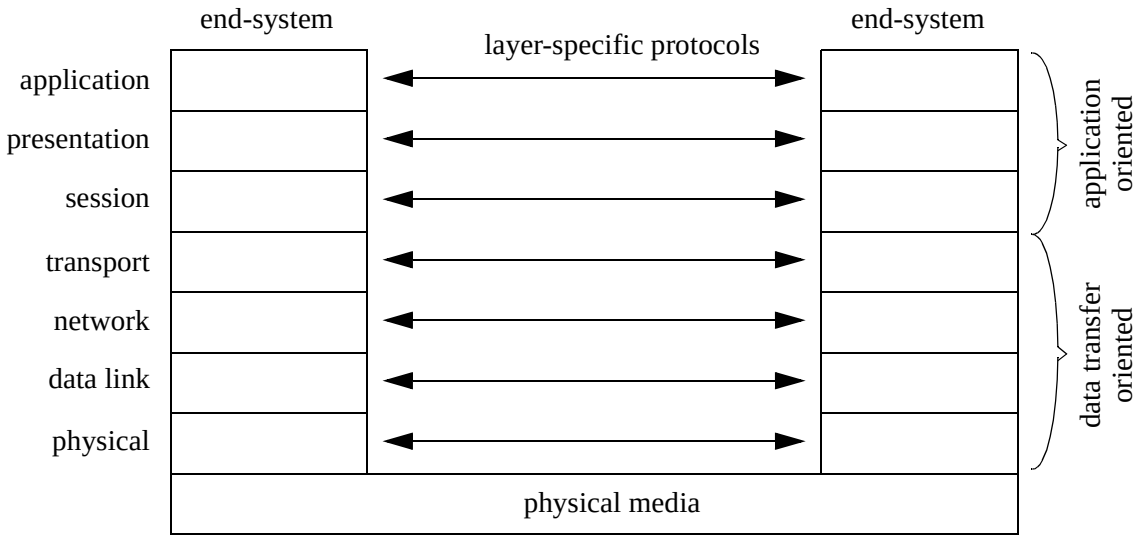


Figure 3.1: Layered architecture of the OSI-RM

A protocol uses the lower level service offered by a lower level protocol in order to implement the service offered to a higher level protocol. Thus the service provided by a protocol is actually the result of the combined behaviour of a stack of protocols. The total functionality of a stack of protocols is assigned to a *service provider.* Since a service provider can be defined at each distinguished level, it is possible to consider an arbitrary service provider, say service provider 'N', as the composition of a layer, layer 'N', and the underlying service provider, service provider 'N-1'. This is depicted in Figure 3.2.

The (N)-protocol exchanges units of information, called (N)-*protocol data units* (PDUs), between (N)-protocol entities via the (N-1)-service provider. The use made of the (N-1)-service provider by the (N)-protocol is defined in terms of abstract interactions, called (N-1)-*service primitives* (SPs). A service is thus defined in terms of SPs, and a protocol in terms of PDUs *and* SPs (to be precise, an (N)-protocol is defined in terms of (N)-PDUs, (N)-SPs, and (N-1)-SPs). Whereas PDUs have a defined format and encoding[1], SPs are merely defined as sets of parameters with no prescribed representation. An (N)-PDU is always conveyed in a data parameter of an (N-1)-SP. Since the meaning of

2. The Application Layer can be considered as being 'open-ended'. This means that new functions will be included when their need has been recognized by the international community. These functions will use existing service functions, both service functions of ASEs in the Application Layer and service functions provided by the Presentation Layer. Also user defined protocols will use these service functions.

1. Except in the Application Layer, where PDUs only have a defined format.

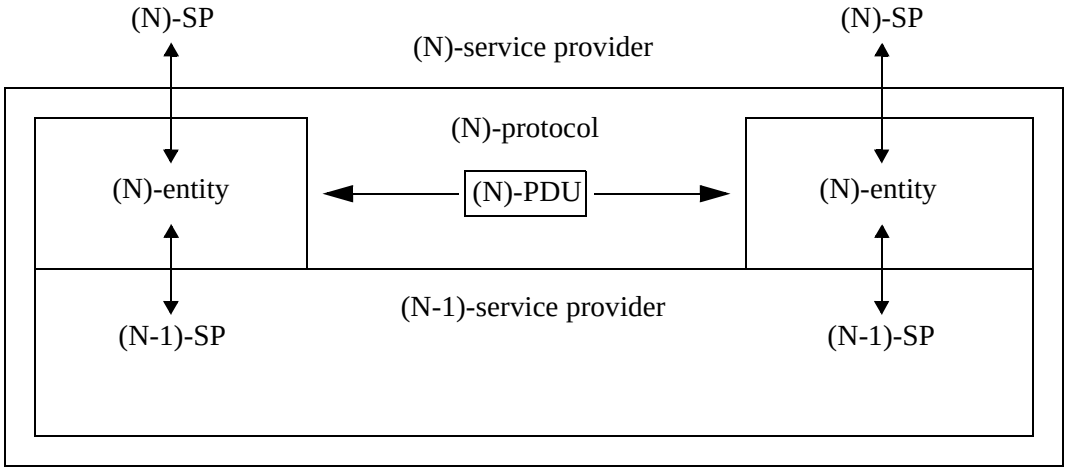


Figure 3.2: Model of an arbitrary layer (layer 'N') in the OSI-RM

an (N)-PDU is only of concern to the (N)-protocol, a data parameter is transferred transparently by the (N-1)-service provider[1], preserving the representation defined by the (N)-protocol. Such a parameter is called an (N)-*service data unit* (SDU). An (N)-PDU normally consists of (N)-*protocol control information* (PCI) and (N)-*user data*. (N)-user data in an (N)-PDU represents the data (corresponding to all or part of an (N)-SDU) that is transparently transferred by the (N)-protocol.

The definition of a protocol in this way permits the implementation of peer protocol entities by different (and isolated) implementation teams. Adjacent protocol entities, however, cannot be implemented in isolation by different implementation teams, unless their abstract interface defined in terms of SPs is first replaced by a concrete interface.

3.1.2 Application service element composition

An *Application Service Element* (ASE) is a separately defined (standardized) part of an application entity. An ASE is always considered together with a peer ASE. The cooperation of these ASEs is defined by an ASE protocol and the function realized by this protocol and offered to a higher level ASE protocol (*not* a higher level layer) is defined by an ASE service. The definition of an ASE protocol and an ASE service is done in the same way as the definition of a 'normal' protocol and service. An important distinction, however, is the scope of an ASE protocol and an ASE service. An ASE protocol is concerned with a particular part or aspect of the interaction of application entities, and is

1. An exception is the Presentation Service provider. The Presentation Layer is responsible for the coding of application PDUs in transit between end-systems. The Presentation Service provider preserves the meaning of information during transfer, not necessarily the representation of the information.

therefore not necessarily involved in all information exchanges between these application entities.

Consequently, an ASE protocol may sometimes use several lower level services, and an ASE service may be defined as being part of a more comprehensive application service. Figure 3.3 depicts an example of a composition of different ASE protocols.

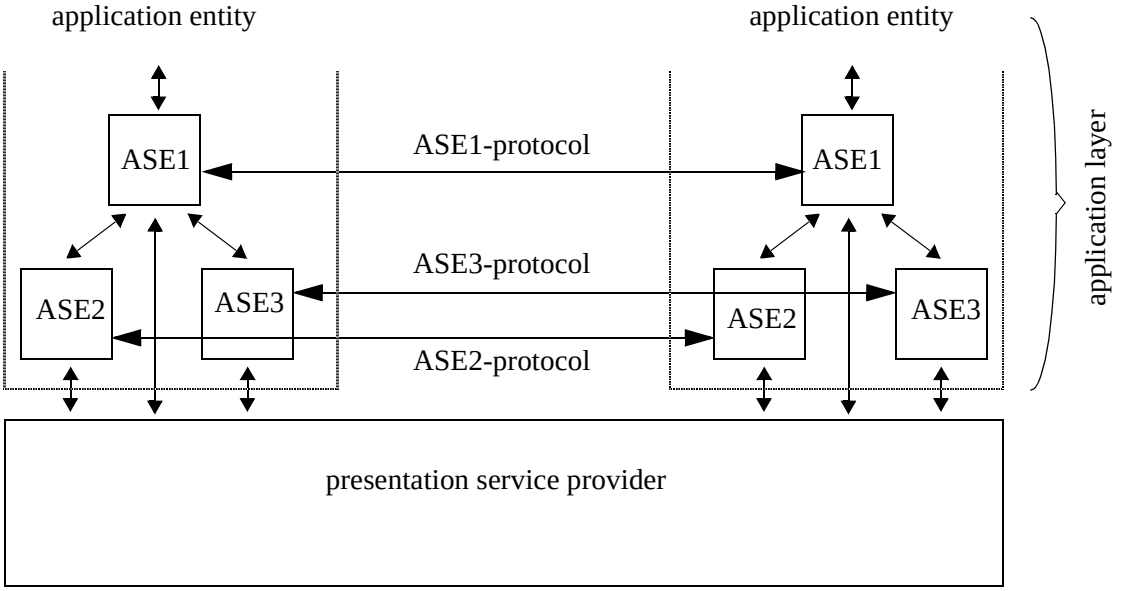


Figure 3.3: Example of application service element composition

The potentially complex relationships between ASE protocols cannot be represented by the technique of layer composition alone. Consider, for example, the composition depicted in Figure 3.3. If ASE1 is involved in request service primitives with successively ASE2, the underlying presentation entity, and ASE3, it is assumed that its peer ASE participates in corresponding indication service primitives in the same order. However, if no control is exercised on the ordering of service primitives in which ASE2, ASE3 and the presentation entities are involved it is unlikely that this order will be preserved.

An auxiliary element is introduced for this purpose, called the *Single Association Control Function* (SACF). A SACF represents the rules for controlling the relationship between service primitives in which ASEs are involved and service primitives in which the presentation entities are involved. The relationship between these service primitives must be such that the order in which ASEs send (receive) information through service primitives corresponds to the order in which the presentation entities are involved in service primitives that convey this information. Notice that there must be a specific SACF

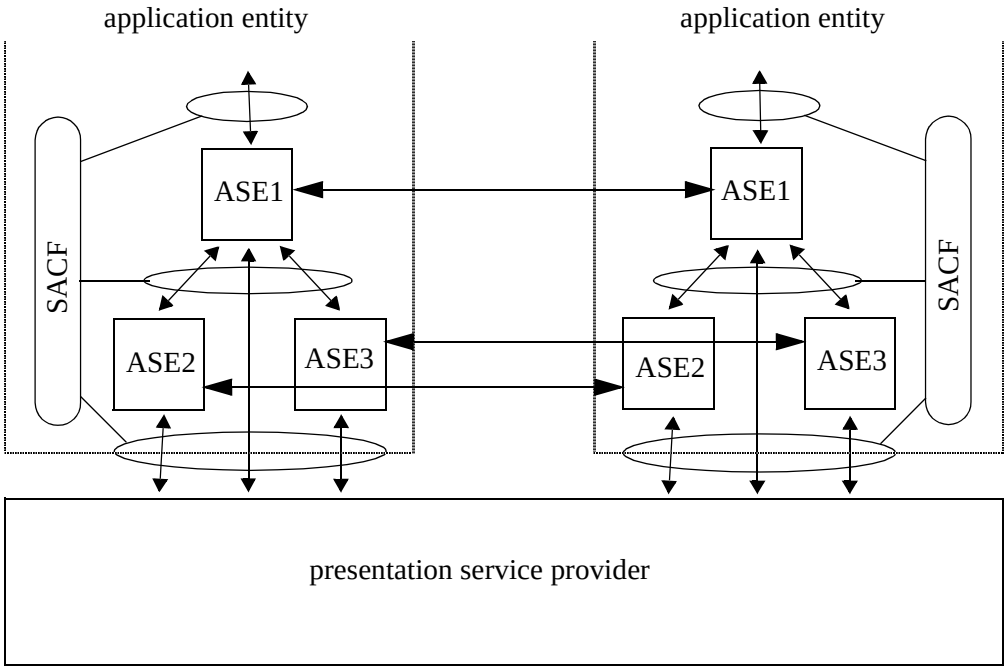


Figure 3.4: SACF for a specific combination of ASEs

for each combination of ASEs. Figure 3.4 depicts an example of a SACF and a combination of ASEs.

ASE protocols only define the cooperation of two peer ASEs, i.e. they are limited to peer-to-peer relationships. Many distributed applications, however, require multi-peer relationships between application entities. For this reason another element is introduced, called the *Multiple Association Control Function*[1] (MACF). A MACF locally relates and constrains multiple point-to-point application services, each of which is provided by one or more ASE protocols that all use the same presentation connection. In addition, a MACF may interact with peer MACFs to perform distributed coordination of application services[2]. One or more MACFs provide an application service that can be used for multi-peer distributed processing. Note that there must be a specific MACF for each combination of lower level application services. Figure 3.5 depicts an example of a MACF that locally

1. SACF and MACF are concepts defined in the first version of the Application Layer Structure ([IS9545:89]). A revision of this standard defines the more general concept of Control Function. We use here the original ALS concepts since these have been used in the Application Layer standards considered in this chapter.

2. Whether the interaction between peer MACFs should be defined by a MACF protocol is not clear from the Application Layer Structure. The only MACF that is explicitly defined so far (the MACF for Distributed Transaction Processing, in [IS10026:92]), is only concerned with local coordination.

coordinates the use of two lower level application services and coordinates its own operation with two peer MACFs (assuming the definition of a MACF protocol).

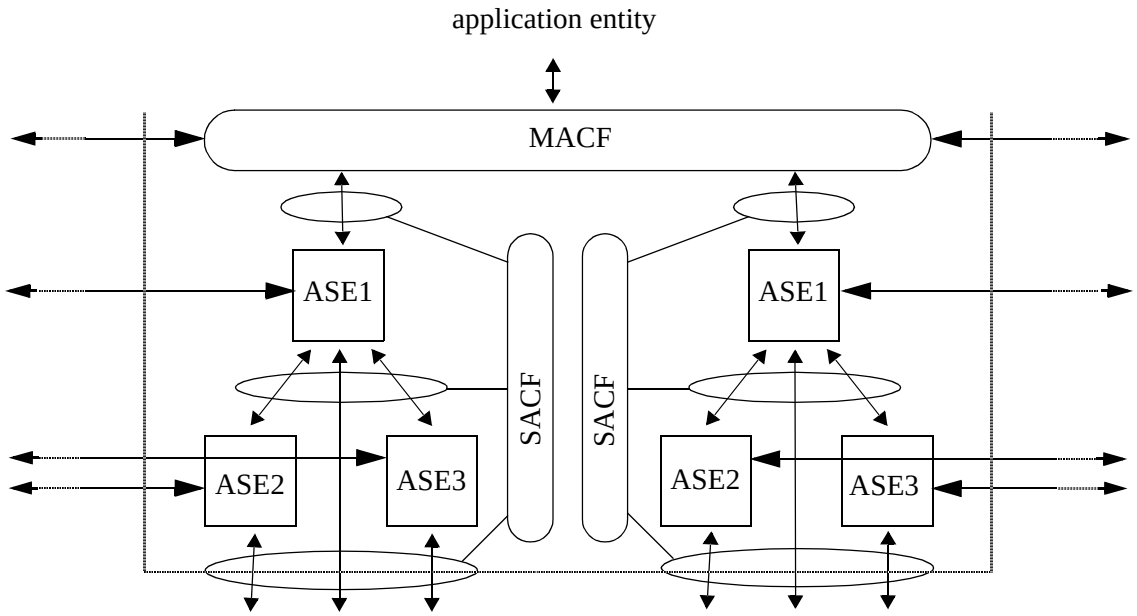


Figure 3.5: Example of control functions required by application service element composition

A specific SACF and a specific MACF may be defined in an application protocol framework standard. A framework standard defines a specific ASE and defines how other ASEs, defined in other standards, may be used in combination with this ASE. An example of an application protocol framework standard is the Distributed Transaction Processing standard ([IS10026:92]).

3.1.3 Composition of cooperating main service and auxiliary service

The *cooperative main service* approach is not defined in general terms by the OSI-RM or the OSI-ALS. The following is an attempted generalization of the cooperative main service approach, based on the only present application of this approach, namely the inclusion of the Commitment, Concurrency and Recovery Service Element ([IS9804:90], [IS9805:90]) by the File Transfer, Access and Management standard ([IS8571:88]). For convenience we introduce the following terms:

- *main ASE*: the cooperating main service (element); and
- *auxiliary ASE*: the ASE that is referenced by the main ASE.

We will use the prefixes ‘main’ and ‘auxiliary’ consistently in order to indicate aspects of the main ASE and the auxiliary ASE, respectively.

The main ASE service defines which main service primitives are able to carry the semantics of an auxiliary service primitive and which parameter of these main service primitives is used for this purpose. The mapping between main service primitives and auxiliary service primitives must be such that the constraints that are defined for the main ASE service and the auxiliary ASE service do not conflict. For example, the ordering of auxiliary service primitives may not conflict with the ordering of main service primitives that carry the semantics of these auxiliary service primitives. The interpretation of the main ASE service is that of a conjunction of the constraints defined on main service primitives (in the main ASE service) and the constraints defined on the auxiliary service primitives (in the auxiliary ASE service).

Similarly, the main ASE protocol defines which main PDUs are able to carry the semantics of an auxiliary PDU and which field in these main PDUs is used for this purpose. The format of a main PDU that carries an auxiliary PDU is given by the format defined in the main ASE protocol, with the field that carries the auxiliary PDU replaced by the format of this PDU (as defined in the auxiliary ASE protocol). The constraints and actions defined in the main ASE protocol may not conflict with those defined in the auxiliary ASE protocol. The interpretation of the main ASE protocol is that of the conjunction of constraints and the cumulation of actions defined in the main ASE protocol and the auxiliary ASE protocol.

Figure 3.6 depicts an example of the composition of a main ASE (the cooperating main service) and an auxiliary ASE.

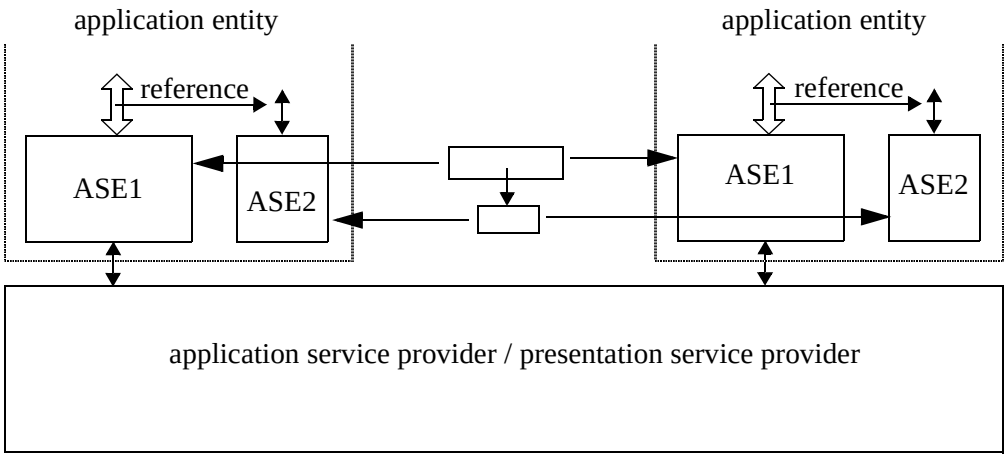


Figure 3.6: Example of the composition of a main ASE (ASE1) and an auxiliary ASE (ASE2)

3.2 Application protocol standards

Although the development of application protocols was initially lagging behind the development of data transfer protocols, at present a rich set of application protocol standards is available that provides support to a wide variety of distributed system applications. Figure 3.7 presents an overview of the most important application protocol standards (each with an associated service standard) that are currently completed. It also shows the relationships between these standards.

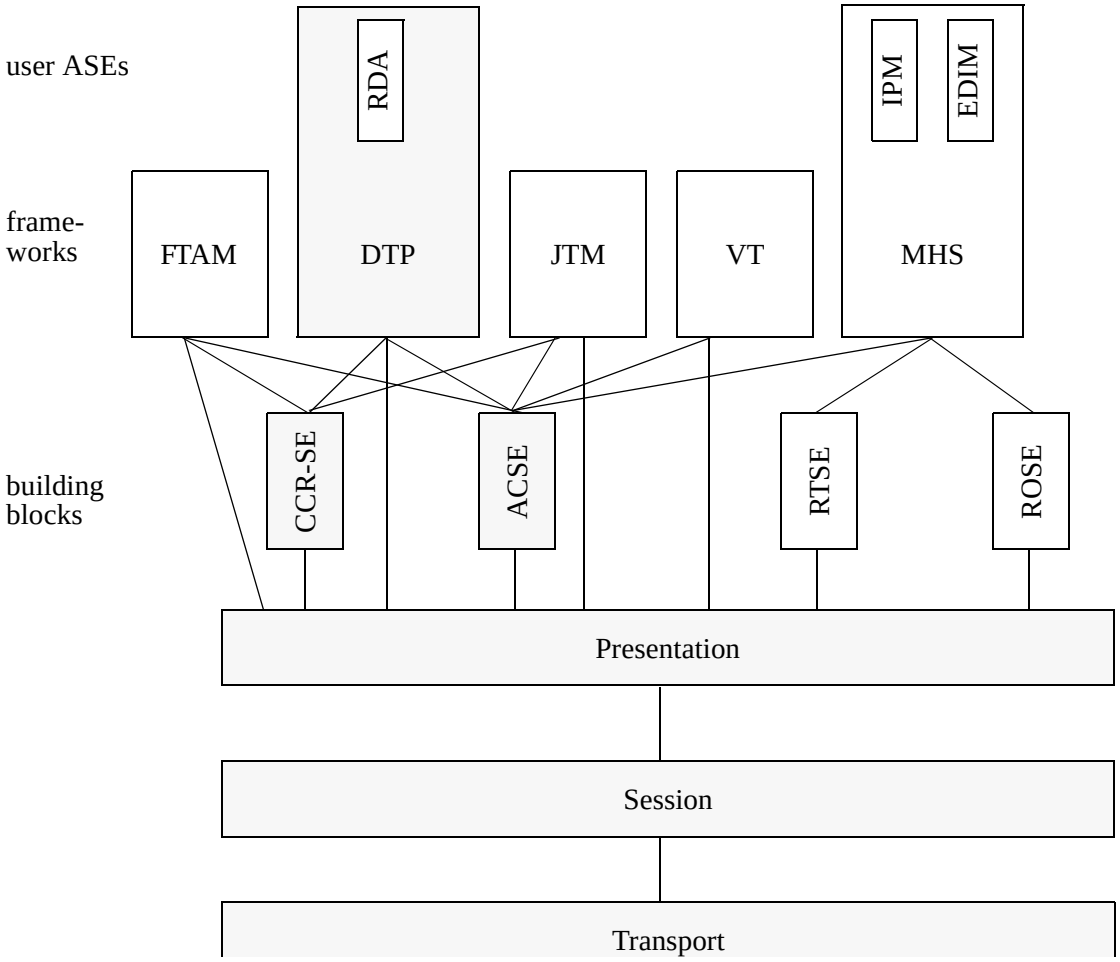


Figure 3.7: Application protocol standards

All application protocols use, directly or indirectly, the service provided by the Transport Layer. The connection oriented Transport Service ([IS8072:86]) is a relatively simple service that comprises the following functions:

- *connection establishment*: this function provides confirmed connection establishment that allows the negotiation of quality characteristics (‘quality of service’) of the connection, including the expedited data transfer option (see below).
- *data transfer*: there are two data transfer functions, viz. ‘normal’ and ‘expedited’ data transfer. Normal data transfer is always available and provides in-sequence transfer of transport SDUs (TSDUs). Expedited data transfer is only available if selected during connection establishment. It provides in sequence transfer of expedited TSDUs (of limited length). Expedited transfer is never slower than normal transfer, and possibly, but not necessarily, faster than normal transfer.
- *connection release*: this function provides unconditional release of the connection.

In addition, a connectionless-mode Transport Service is defined (in an addendum to [IS8072:86]) which comprises a single function: the transfer of self-contained TSDUs. With this type of service, TSDUs are not related and the transfer of TSDUs does not require previously established arrangements of the service provider and the service users.

All application protocols indicated in Figure 3.7 offer a connection oriented (or association oriented) service, except the Message Transfer (MT) service of the Message Handling System (MHS) standard, which provides a (kind of) connectionless-mode service. In addition, connectionless-mode services, and supporting protocols, are defined for the Session Layer, the Presentation Layer, and the Association Control Service Element (ACSE). These services are very similar to the connectionless-mode Transport Service. Until now, none of the application protocol framework standards makes use of lower level connectionless-mode services.

It is not possible to give a comprehensive overview of all present application service and protocol standards. We will therefore consider a representative ‘profile’ of the OSI-ULA, which permits an illustration of the subdivision of the application protocol functionality and which can be used as a reference for evaluation and comparison in later chapters. This profile is indicated in Figure 3.7 by the shaded boxes, and consists of:

- the Session Layer standards ([IS8326:87], [IS8327:87]) and Presentation Layer standards ([IS8822:88], [IS8823:88]): these standards illustrate the layering approach. The Presentation Protocol uses the Session Service, and Application Layer protocols use the Presentation Service.
- the Association Control Service Element (ACSE, [IS8649:88], [IS8650:88]) and the Commitment, Concurrency and Recovery Service Element (CCR-SE, [9804:90], [IS9805:90]): these standards are examples of the most general building blocks of the Application Layer. The ACSE protocol and the CCR-SE protocol only use the Presentation Service, while the ACSE service and the CCR-SE service are used by many

other Application Layer protocols and thus support many distinguished classes of distributed system applications.

- the Distributed Transaction Processing (DTP) standard ([IS10026:92]): this standard is an example of an application protocol framework standard. It defines an ASE for the support of distributed transaction processing, the TP-ASE, and defines how this ASE can be used in combination with lower level ASEs and with 'user' ASEs. A user ASE is an ASE which can fill certain functional holes in the framework standard. There may be several ASEs which can perform the role of user ASE in a framework standard. The DTP standard includes the definition of a SACF (since it must define the combination of the TP-ASE and lower level ASEs) and of a MACF (since it is concerned with multi-peer interactions).

These standards are further explained in the following sections.

3.3 Session Layer

The main functions of the Session Layer are support of the synchronization needs of application entities and support of the controlled use of transfer facilities by application entities. *Synchronization* entails the following functions:

- transfer of synchronization points (identified by serial numbers) for synchronization and resynchronization purposes;
- generating serial numbers of synchronization points (except when the application entities want to agree upon a new initial synchronization point);
- separation of data sent before and after a synchronization point (in case of synchronization) and purging of undelivered data sent before a synchronization point (in case of resynchronization); and
- signalling that an identified activity of an application entity is started, interrupted, resumed, discarded, or ended.

Figure 3.8 illustrates the separation and purging of data accomplished by the Session protocol.

The *controlled use of transfer facilities* (dialogue control) entails the following functions:

- management and transfer of tokens (representing the exclusive right to invoke a specific service); and
- constraining the use of services according to the token states.

Figure 3.9 illustrates these functions.

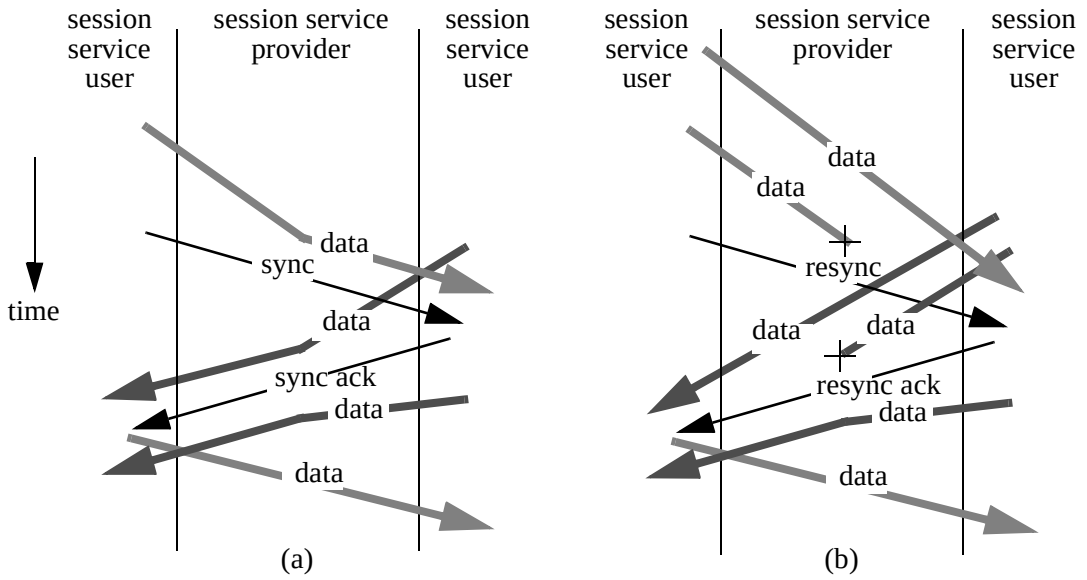


Figure 3.8: Separation of data (a) and purging of data (b)

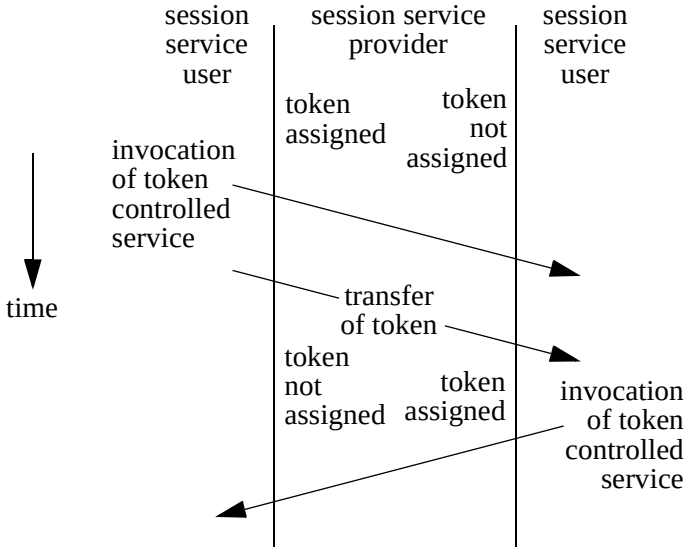


Figure 3.9: Use of a token controlled service

Other functions of the Session Layer include:

- establishment of a suitable session connection:

  - negotiation of session functionality, quality of service, and, if required, initial synchronization points and initial token settings;
  - establishing or re-using a suitable transport connection; if a new transport connection is established to support a session connection, the transport connection must have been established before the above mentioned negotiations can start (i.e., the session and transport connection are not established in 'one shot').
- matching and exploiting data transfer capabilities of the transport service:
  - segmentation/reassembly of session SDUs;
  - concatenation/separation of session PDUs.
- termination of the session connection without loss of data (orderly release), or with possible loss of data (abort):
  - termination or retaining the transport connection; if the transport connection is terminated, this must occur before the termination of the session connection can start.

Establishing, retaining, re-using, and terminating transport connections are internal functions, i.e. functions of the Session Protocol that have no visibility in the Session Service. Also segmentation/reassembly and concatenation/separation are internal functions. All other functions are visible in the Session Service.

The Session Service provides a *kernel* functionality consisting of the connection establishment function, the normal data transfer function, and the connection termination functions (including orderly release). All other functions of the Session Service are *optional* and have to be negotiated between the Session Service users. The Session Protocol only monitors this negotiation, i.e. it must support whatever functionality is agreed upon by the Session Service users. The session connection establishment function also comprises the negotiation of quality of service (QoS). Only two QoS parameters are of relevance to the Session Protocol:

- optimized dialogue control: a boolean which indicates whether (extended) concatenation of session PDUs should be performed; and
- extended control: a boolean which indicates whether the transport expedited flow option should be used for bypassing normal flow control.

All other QoS parameters are negotiated between the application entities and the Transport Service provider, and are thus merely passed through by the Session Protocol.

3.4 Presentation Layer

The Presentation Layer is concerned with the *representation of information in transit* between two application entities. It is not concerned with the interpretation of the information. The need for the presentation layer functionality stems from the fact that in an 'open' and thus potentially heterogeneous environment different conventions for the local representation of information may be used by different computer systems. A common representation must be agreed if information is exchanged between two systems. If the local representation used by a system differs from the agreed common representation, a translation to (at the sending side) or from (at the receiving side) the common representation must be performed.

An application PDU (APDU) is mapped onto a Presentation Data Value (PDV) in a presentation SDU (PSDU). Potentially multiple PDVs may be contained in an PSDU (thus permitting a kind of concatenation/separation at service level). Since the Application Layer is relieved from concerns about the common representation of information, and local representations are not within the scope of protocol standardization, Application Layer protocols define APDUs independent of any specific representation (using an abstract syntax notation). A set of APDUs associated with a specific Application Layer protocol constitutes the *abstract syntax* of that protocol. A specific representation of information units is called a concrete syntax. Thus, implementations of application entities will use a local concrete syntax to represent the APDUs, and a common representation used by the Presentation Protocol constitutes a concrete transfer syntax, or *transfer syntax* for short. Figure 3.10 depicts the relationship between an abstract syntax, local concrete syntax, and transfer syntax.

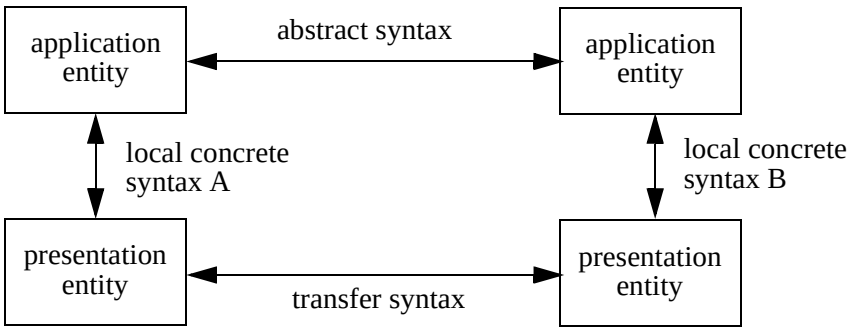


Figure 3.10: Relation between an abstract syntax, local concrete syntax, and transfer syntax

The main function of the Presentation Service is the transfer of information, as PDVs in PSDUs, independent of the local concrete syntaxes used for the representation of PDVs.

The main functions of the Presentation Protocol are:

- negotiation of transfer syntaxes to support the abstract syntaxes used by the application entities; and
- transformation from the local concrete syntax to the transfer syntax, and vice versa.

For each abstract syntax that is used by the application entities, a suitable transfer syntax must be determined by the Presentation Protocol. The negotiation of abstract syntaxes (between application entities) is performed in conjunction with the negotiation of transfer syntaxes (between presentation entities)[1]. This is possible during presentation connection establishment. Alternatively, the use of an optional service function, the *context management* function, can be selected by the application entities, which permits the addition or deletion of abstract syntaxes and supporting transfer syntaxes after the presentation connection has been established.

The coupling of an abstract syntax and a transfer syntax is called a *presentation context*. At each end of the presentation connection, the application entity and the presentation entity refer to a presentation context by means of a presentation context identification (PCId). The PCId of a presentation context is locally agreed between the application entity and the presentation entity during the negotiation of that context[2]. A PCId is necessary each time a PDV is passed from the application entity to the presentation entity (to permit the presentation entity to determine the associated transfer syntax), and vice versa (to permit the application entity to determine the associated abstract syntax).

The Presentation Service also provides all the functions of the Session Service (which are thus passed through by the Presentation Protocol), with the only difference that the Presentation Protocol determines the representation of PDVs contained in the user data parameter of Session Service primitives. Besides context management, application entities may also select a *context restoration* function (only if context management is also selected). When context restoration is selected, presentation entities record the presentation contexts in force each time a synchronization point is transferred between the application entities. The presentation contexts associated with a particular synchronization point are restored by the presentation entities if the application entities request resynchronization or activity resumption from this point.

1. Although the presentation protocol is not involved in the negotiation of abstract syntaxes, it must be informed of the set of abstract syntaxes that is being negotiated in order to perform the transfer syntax negotiation.

2. Presentation context identifications used at different ends of the presentation connection are related by presentation context identifiers exchanged in the Presentation Protocol.

The establishment and termination of the presentation connection is performed in conjunction with the establishment and termination of the session connection (‘one shot’ establishment and termination).

3.5 Application Layer building blocks

A number of Application Layer protocols are defined that act as general building blocks for other Application Layer protocols. At present, four such building blocks are defined. We only consider the building blocks that are used by the Distributed Transaction Processing standard, described in section 3.6.

3.5.1 Association Control

The Association Control Service Element (ACSE) defines functions for the establishment and termination of an association between two application entities, called an application association. An application association is a mutual agreement of two application entities concerning their cooperative relationship, which comprises, among others, the identification of ASEs that must be available to support the cooperation.

The ACSE provides the possibility for an application entity to respond to a request for cooperation from any other application entity. If the necessary ASEs are available in the addressed application entity, the response from the application entity can in principle be affirmative, leading to the establishment of an appropriate application association. If this is not the case, the response will be negative, and no application association will be established. This functionality is essential in an open environment, and must therefore always be present in any application entity.

An application association is established and terminated in conjunction with a presentation connection and a session connection. The possibility of negotiating or selecting optional functions of the Presentation Service and the Session Service is offered via the ACSE service to the initiator and acceptor of the application association.

3.5.2 Commitment, Concurrency and Recovery

The Commitment, Concurrency and Recovery Service Element (CCR-SE) defines functions for the coordination of two application entities such that the activity in which these application entities participate has atomic action properties. An activity with atomic action properties is called an *atomic activity.* Atomic action properties are:

- atomicity: the operations that are part of the atomic activity are either all performed or none of them is performed;
- consistency: the operations are correctly performed[1];

- isolation: intermediate results of the atomic activity are not accessible, except by operations of the activity itself;
- durability: the final results of the atomic activity are not effected by system or communication failures.

The operations of an atomic activity may access and manipulate different kinds of resources (e.g. information resources and physical objects).The collective state of the resources which is used by the atomic activity is referred to as *bound data*. The coordination provided by the CCR-SE ensures that bound data can either be released in the initial state (the state that existed before the start of the activity) or in the final state (the state that results from a successful performance of all operations) at the end of the atomic activity, and that the intermediate states of the bound data are not visible outside the atomic activity. This is illustrated in Figure 3.11. The operations that effect bound data and the (possible) changes effected are not considered by the CCR-SE.

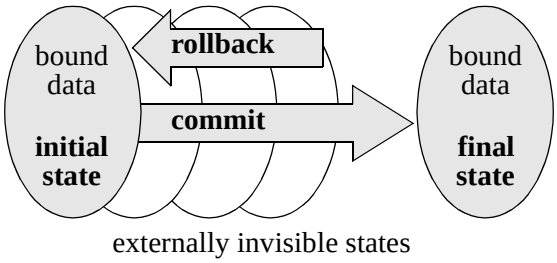


Figure 3.11: Transition of bound data in an atomic activity

The state and control data about an atomic activity that is maintained for recovery purposes is called *atomic action data*. The CCR-SE defines when atomic action data must be recorded, or logged, such that this data can survive system and communications failures.

The relationship between two CCR-SE service users is characterized as a superior-subordinate relationship. The superior user normally takes initiative for the coordination supported by CCR-SE. The coordination provided by the CCR-SE is based on a two-phase commit mechanism with presumed rollback. Two-phase commitment consists of two phases:

- voting phase: the subordinate is asked by the superior whether it can release its bound data in the final state (commitment offer);

1. Both according to the their specification and not conflicting consistency conditions which may be defined for the application.

- commitment or rollback phase: the superior orders the subordinate to release its bound data in the final state (commitment)[1] or to release its bound data in the initial state (rollback).

Presumed rollback implies an assignment of recovery responsibility to the CCR-SE service users such that the overhead for logging and removing logged data can be minimized in an implementation.

The CCR-SE assumes the availability of an application association which supports at least the exchange of CCR PDUs. This association must have been established, using the ACSE service, by another ASE. The use of the CCR-SE therefore always requires one or more other ASEs (besides the ACSE), defined by a framework standard that incorporates the CCR-SE as a building block ASE.

3.6 Distributed Transaction Processing

The Distributed Transaction Processing (DTP) standard defines functions that support the performance of transactions by two *or more* application entities. A transaction is an atomic activity, with the atomic properties mentioned in section 3.5.

DTP distinguishes three types of peer-to-peer relationships between application entities:

- *application association*: Application associations are established and terminated using the ACSE service. An application association may be idle, i.e. not in use by application entities performing DTP functions. DTP does not define when and how application associations are established.
- *dialogue*: A dialogue is a relationship between two application entities that perform DTP functions. A dialogue may survive system and communication failures (although it is currently not defined what atomic action data must be maintained). The exchange of information between application entities may be temporarily blocked because of such failures. The exchange of information on a dialogue is supported by an application association. Several application associations may be used consecutively by a dialogue (in case of communication failures).
- *transaction branch*: A transaction branch is a relationship between two application entities that perform DTP functions concerned with transaction coordination. A transaction branch implies the existence of a dialogue. Multiple consecutive transactions can be supported on a dialogue.

1. A commitment offer from the subordinate does not imply that the superior will order commitment. The atomic activity considered in the scope of CCR may be part of a larger atomic activity, involving many other application entities, each of which may be required to be able to release bound data in the final state before an order of commitment is given.

Multi-peer relationships between application entities can be built with dialogues and with transaction branches. In both cases these are modelled as trees, with application entities as nodes and dialogues or transaction branches as arcs. A *transaction tree* is by definition part of a *dialogue tree* (since a transaction branch requires a dialogue). DTP allows the use of different parts of a dialogue tree for independent transactions. Figure 3.12 illustrates the relationship between a dialogue tree and multiple transaction trees.

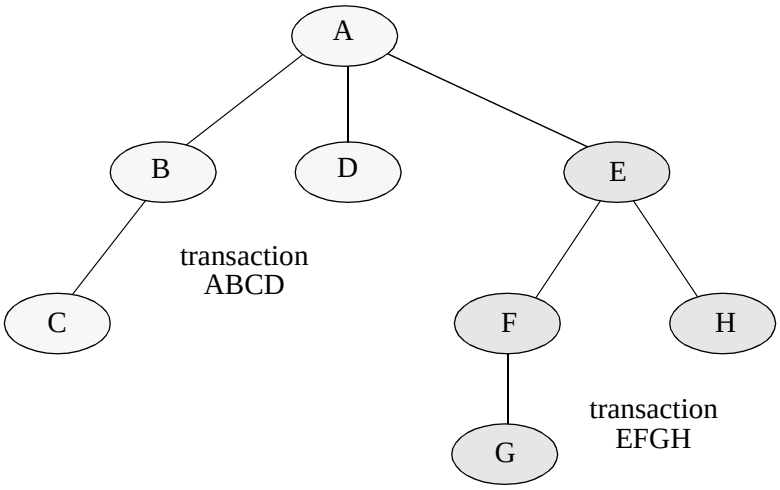


Figure 3.12: Dialogue tree supporting two concurrent transactions

The functions of DTP can be broadly classified as either being concerned with dialogues (on which transactions can be performed) or with transactions (on dialogues). Dialogue related functions include:

- establishment and termination of a dialogue between application entities (thus changing the dialogue tree);
- supporting the orderly use of the dialogue by application entities (requesting and granting control of the association);
- supporting the synchronization of processing activities of application entities (hand-shaking).

Transaction related functions include:

- *begin* transaction function: initiating a transaction branch between two application entities (thus changing the transaction tree).

  If the initiating application entity is already involved in a transaction, the transaction tree is extended, with the initiating application entity acting as superior for this new transaction branch and the accepting application entity acting as subordinate for this

branch. Otherwise, if the initiating entity was not involved in a transaction, a transaction tree is created consisting solely of this new transaction branch, with the initiating entity acting as the master of the transaction.

This function is only available if the ‘unchained transactions’ option has been selected. This option implies that transaction trees must be explicitly built in order to perform a transaction and will dissolve again after termination of the transaction. When the unchained transactions option is selected it is possible that at a given point in time two users of the dialogue are involved in the same transaction, in different transactions, or one or both of them are not involved in any transaction. If the unchained transactions option is not selected, the ‘chained transactions’ option must be selected. This option implies that all dialogue users are always involved in the same transactions. When a transaction terminates, a next transaction is implicitly started. Only when the dialogue is terminated, a transaction branch is pruned from the transaction tree. The following functions are available with either of the options.

- coordination functions:
  - *prepare* function: allows a superior to request completion of transaction processing in a transaction sub-tree (coordinated by a subordinate of this superior). All application entities in this sub-tree are requested to prepare the release of their bound data in the final state.
  - *ready* function: allows a subordinate to indicate to its superior that all application entities in the transaction sub-tree are ready to release their bound data in the final state.
  - *commit* function: allows a user to indicate that it has completed all processing for the current transaction and to request termination of the transaction with the release of bound data in the final state. When the user is a master, this function initiates the termination of the transaction; otherwise, this function is used to propagate termination of the transaction.
  - *rollback* function: allows a user to request the termination of the transaction with the release of bound data in the initial state.
  - *done* function: allows a user to indicate that is has released its bound data.
  - commitment/rollback *complete* function: this function is initiated by the DTP service provider to indicate to all users involved in the transaction that commitment/ rollback is complete.

A dialogue only survives failures if it supports a transaction. Otherwise, a failure will cause an abort of the dialogue. Furthermore, a dialogue can only be terminated if no transaction is in progress. A transaction branch can only be initiated on a dialogue if no current transaction branch exists on the dialogue. Termination of a dialogue may result in a new, pruned dialogue tree (if the dialogue was connected to a ‘leaf’ in the original dialogue

tree), or two new dialogue trees (if the dialogue connected two ‘non-leafs’ in the original dialogue tree). If the chained transactions option has been selected on the dialogue, similar changes apply to the transaction tree.

DTP is a rather complex standard. It includes the definition of a TP-ASE, a TP-SACF, and a TP-MACF. In addition, it uses the ACSE service, the CCR-SE service (optionally), and the Presentation Service (and via the Presentation Service, also the Session Service). Finally, it must be associated with one or more (undefined) user ASEs. The DTP standard therefore defines a number of internal services in order to facilitate the definition of the protocol. Figure 3.13 depicts the TP protocol entity architecture. This architecture also

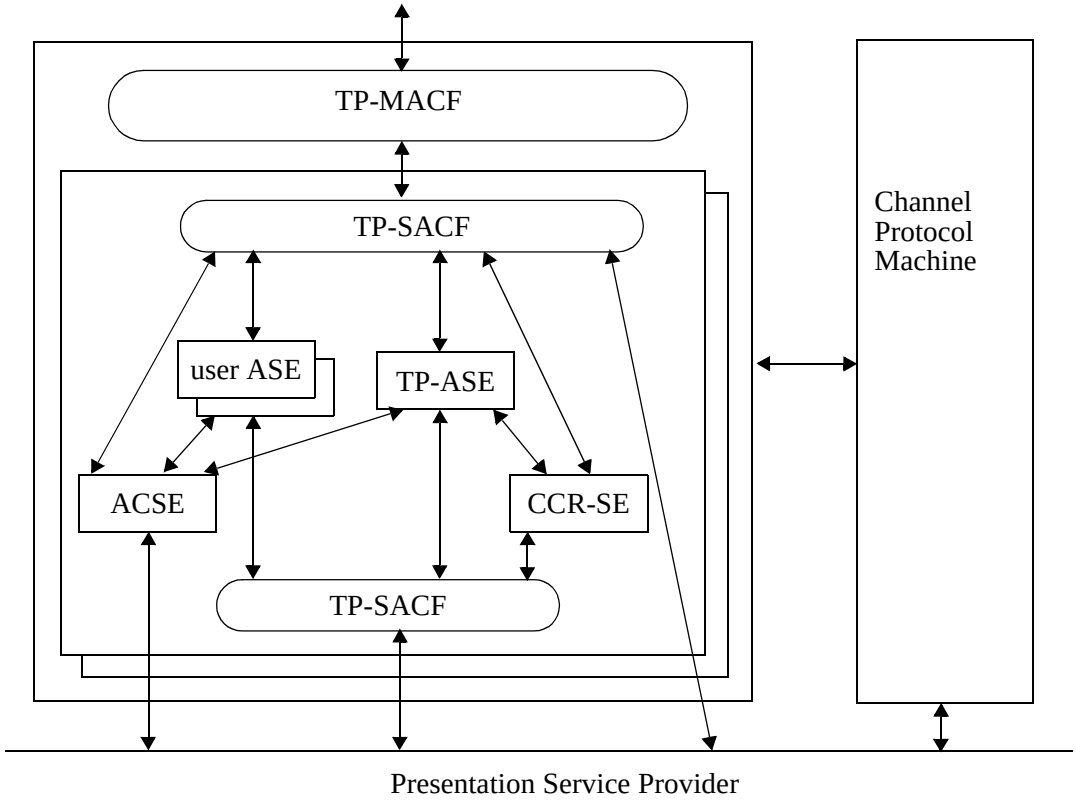


Figure 3.13: Architecture of a Transaction Processing protocol entity

includes a Channel Protocol Machine (CPM) which establishes and terminates ‘channels’ as appropriate for the purpose of recovery. The CPM includes the same elements as the Transaction Processing Protocol Machine (TPPM), except that no user ASEs are included. The TPPM is the only ‘user’ of the CPM.

3.7 Conclusion

The OSI Upper Layer Model (OSI-ULM) comprises three structuring techniques that are used to compose application protocol standards:

- layer composition;
- Application Service Element (ASE) composition; and
- composition of a cooperating main service and one or more auxiliary services.

Layer composition yields a vertical structure. Each layer protocol uses the service provided by the next lower layer and provides an enhanced service to the next higher layer. The OSI Upper Layer Architecture (OSI-ULA) defines three layers: the Session Layer, the Presentation Layer, and the Application Layer. The Session Layer and the Presentation Layer are each defined by one service and one protocol standard. The application layer is, however, further structured, according to the second (as a rule) and third (as an exception) mentioned structuring technique.

The Session Service and Protocol and also the Presentation Service and Protocol are structured in terms of functional units. A service functional unit is a grouping of related service functions (service elements) and a protocol functional unit is a grouping of protocol functions (elements of procedure). Usually, only one functional unit in the service and protocol is always selected. This functional unit is called the kernel functional unit. All other functional units have to be negotiated between the service users. The main purpose of functional units is to allow application entities to select the functions they actually require.

ASEs can be considered as functional units of ‘sub-layers’ of the Application Layer. The only, but important, difference is that ASEs are defined in separate (service and protocol) standards. Hence, the relationship between ASEs that are used ‘next to each other’ in the same sub-layer is not defined by these ASEs. An auxiliary element, called the Single Association Control Function (SACF), is introduced to define necessary relationships that characterize the desired composition. Since ASEs, like the Session Layer and the Presentation Layer, only cover peer-to-peer interaction, and because many distributed system applications require multi-peer interactions, yet another element is introduced. This element is called the Multiple Association Control Function (MACF). A MACF defines the local coordination of multiple application associations and possibly also the interactions between more than two application entities. A MACF uses the services of one or more ASEs and possibly also the Presentation Service. SACFs and MACFs are usually defined in so called application protocol framework standards, which define an application-specific ASE and which include references to one or more (building block) ASEs whose services can optionally be used by the application-specific ASE. The Distributed Transaction Processing standard is an example of a framework standard.

A cooperating main service can be considered as a ‘main’ ASE with ‘holes’ in its service and protocol that must be filled by an ‘auxiliary’ ASE. Thus, the main ASE service defines the relationship between its service functions with ‘dummy’ parameters and service functions of the auxiliary ASE service that are used to replace the dummy parameters. Similarly, the main ASE protocol defines the relationship between its PDUs with ‘dummy’ fields and PDUs of the auxiliary ASE protocol that are used to replace the dummy fields. The advantage of this composition is that it does not require the introduction of a SACF, since the relationship between the main ASE and the auxiliary ASE is defined by the main ASE. The disadvantage is that the composition possibilities are rather restricted.

The layering approach and the composition of ASEs is illustrated with descriptions of the Session Layer, the Presentation Layer, the Association Control Service Element, the Commitment, Concurrency and Recovery Service Element, and the Transaction Procession Application Service Element.

References

[Day83] Day, J.D., and Zimmermann, H., The OSI reference model, *Proceedings of the IEEE*, Vol. 71, No. 12, December 1983, 1334-1340.

[IS7498:84] ISO, *Open systems interconnection - basic reference model*, International Standard ISO 7498 Part 1, 1984.

[IS8072:86] ISO, *Transport service definition*, International Standard ISO 8072, 1986.

[IS8326:87] ISO, *Basic connection oriented session service definition*, International Standard ISO 8326, 1987.

[IS8327:87] ISO, *Basic connection oriented session protocol specification*, International Standard ISO 8327, 1987.

[IS8571:88] ISO, *File transfer, access and management*, Part 1: General introduction, Part 2: Virtual filestore definition, Part 3: File service definition, Part 4: File protocol specification, International Standard ISO 8571, 1988.

[IS8649:88] ISO, *Service definition for the association control service element*, International Standard ISO 8649, 1988.

[IS8650:88] ISO, *Protocol specification for the association control service element*, International Standard ISO 8650, 1988.

[IS8822:88] ISO, *Connection oriented presentation service definition*, International Standard ISO 8822, 1988.

[IS8823:88] ISO, *Connection oriented presentation protocol specification*, International Standard ISO 8823, 1988.

[IS9545:89] ISO, *Application layer structure*, International Standard ISO 9545, 1989.

[IS9804:90] ISO, *Service definition for the commitment, concurrency and recovery service element*, International Standard ISO 9804, 1990.

[IS9805:90] ISO, *Protocol specification for the commitment, concurrency and recovery service element*, International Standard ISO 9805, 1990.

[IS10026:92] ISO, *Distributed transaction processing*, Part 1: OSI TP model, Part 2: OSI TP service, Part 3: OSI TP protocol, International Standard ISO 10026, 1992.

[Linington83] Linington, P.F., Fundamentals of the layer service definitions and protocol specifications, *Proceedings of the IEEE*, Vol. 71, No. 12, December 1983, 1341-1345.

Chapter 4

OSI Upper Layer Architecture and Model: Evaluation

In this chapter the OSI upper layer architecture (OSI-ULA) and the OSI upper layer model (OSI-ULM) are evaluated. The evaluation is partially based on the design quality criteria proposed in Chapter 2 (Design Quality Criteria). The impact of standardization on the quality of application protocol standards is also reviewed. The last part of this chapter briefly discusses protocol implementation freedom and its relation to the quality (efficiency) of application protocol products.

The purpose of this chapter is to identify and analyse deficiencies of the OSI-ULA and of the OSI-ULM, such that pertinent improvements can be considered. Some potential improvements are explored or proposed in this chapter. In particular, some architectural alternatives are discussed, the concept of service is clarified, and design methods are presented that can be used to derive protocol layer compositions and ASE compositions.

The structure of this chapter is as follows: section 1 evaluates the OSI-ULA. It includes a discussion of each of the upper layers and of the Transport Service; section 2 discusses the extent to which OSI application protocol products have been successful, and discusses some general complaints of users concerning these products; section 3 presents problems related to the nature of standardization which may influence the quality of protocol standards; section 4 evaluates the OSI-ULM. It includes a discussion of the major unclarities of OSI design concepts; section 5 discusses design methods that can be used to derive application protocols structured in terms of layers and ASEs; section 6 discusses one particular aspect of implementation freedom offered by OSI protocols, namely that of multi-layer implementation; and section 7 presents the conclusions of this chapter.

4.1 Upper layer architecture

This section evaluates the OSI-ULA in terms of the composition of functional entities and the assignment of application protocol functions to these entities. The design quality criteria will be used as main criteria for judging the architecture.

4.1.1 General observations on the layers of the OSI-ULA

The Session Layer and the Presentation Layer can be characterized as ‘static’ application protocol layers. They consist of a fixed number of functions, some of which can be selected through negotiation during connection establishment. The presentation entities are the only direct users of the Session Service, and the application entities are the only direct users of the Presentation Service. No new functions can be added without changing the service and protocol standards, and no alternative composition of the layers is permitted.

The Application Layer, on the other hand, can be characterized as ‘dynamic’. It consists, among others, of ASEs, which are selected through negotiation during application association establishment. The ‘kernel’ functionality of the Application Layer only comprises the ACSE. Since the definition of each ASE is self-contained, new standard ASEs can be added to the Application Layer, without the need of changing existing ASE standards.

The need for a dynamic Application Layer comes from the unlimited variety of distributed system applications. Each time the need for standardized protocol support for a new class of applications is identified, a new ASE can be developed that is added to the Application Layer. If only fixed layers were permitted, this would result in the development of a new layer that is positioned on top of the layer hierarchy.

With a dynamic layer, different components can be grouped together in order to provide separate but related application service functions that support a specific class of distributed applications. A static layer, however, has the following characteristics (see also Figure 4.1):

- its service comprises data transfer and application service functions.

  Data transfer functions are always provided since at higher protocol levels new application protocol functions may be implemented that do not use the application service functions provided by this service.

  The application service functions provided by the service constitute only a small subset of all possible application service functions. Each higher layer may extend the set of application protocol functions.

application service functions		S1	S2	S3	S4	etc.
application protocol layers	P3	X	O	X	X	
	P2	X	O		O	
	P1		X		X	

X = functional enhancement
O = pass through

Figure 4.1: Provision of application service functions implemented by protocol functions of fixed, hierarchically related application protocol layers

- its protocol defines functional enhancements and ‘pass through’ of lower level service functions.

  A functional enhancement of a lower level service function is accomplished through the exchange of PCI. The PCI is exchanged by using the data parameter of the lower level service function. Non-data parameters of the lower level service function may not be used by the protocol but are merely mapped from and to corresponding non-data parameters of the service function that is provided. Figure 4.2 illustrates a typical functional enhancement with parameter mapping in a layered application protocol architecture. It shows the implementation of an application service function that defines a transformation of three parameter values. The parameter values *a1*, *b1* and *c1* of a request primitive are transformed into the parameter values *a3*, *b3* and *c3*, respectively, of a related indication primitive.

  Pass through of lower level service functions implies that no functional enhancements with respect to these functions are implemented by the protocol. The same functions are thus offered to the next higher layer. (The term ‘pass through’ has been coined in discussions concerning the Presentation Protocol, which defines the pass through of most Session Service functions.)

The pass through of service functions and the direct mapping of parameters implies that the definition of constraints on these functions and parameters either must be repeated or a reference must be made to the lower level service standard in which these constraints are defined. Hence, the separation of concerns effected by application protocol layer composition is only limited.

The above comments apply to a lesser extent to components of a dynamic application protocol layer. In particular, the protocols associated with these components generally also define direct mappings of some service primitive parameters.

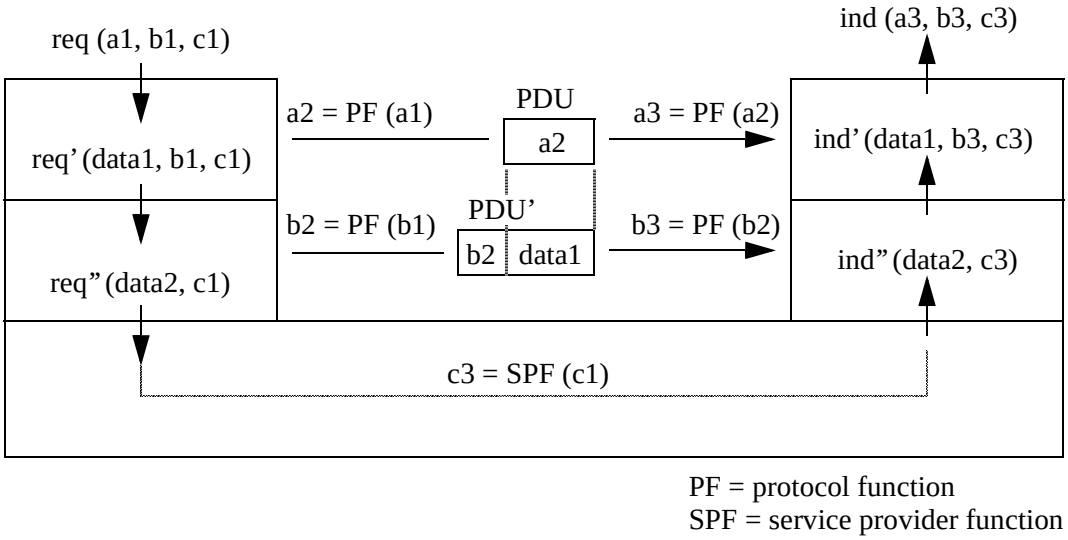


Figure 4.2: Example of a layered protocol mechanism implementing an application service function

The separation of concerns that can be accomplished with static application protocol layers decreases as we move up in the layer hierarchy. This can be illustrated by considering the incremental enhancement of service functions, starting from a data transfer service. The enhancement accomplished by the first layer (using bottom-up numbering) concerns only data transfer functions. Hence, only data transfer functions are passed through. The enhancement accomplished by the second layer concerns both the data transfer functions and the lower level application service functions. In this case, the data transfer functions and the lower level application service functions that are not enhanced are passed through. In addition, the lower level application service functions that are enhanced require the definition of mappings between parameters of lower level application service primitives and required application service primitives. And so on. The more sophisticated a lower level service, the greater is the extent of function pass through and parameter mapping.

This explains the fact that the Application Layer is a dynamic application protocol layer, and the Session Layer and the Presentation Layer are static application protocol layers: the need for dynamic composition of functions is stronger at the level of the Application Layer. Another, historical, reason is that application protocol standards were developed in a bottom-up fashion: at the time that the definition of the Session Layer standards was started the notion of dynamic composition of layer components was still immature (the first version of OSI-ALS was published in 1989; the first version of the Session Layer standards was published in 1987).

In retrospect, however, it seems inconsistent to restrict the application of dynamic composition of layer components to the Application Layer. For example, the Presentation Service and the Session Service are for a large part identical, due to the pass through of Session Service functions.

4.1.2 Application Layer

In this subsection we only consider the DTP standard and the ASE standards that are referenced by the DTP standard, viz. ACSE standards and CCR-ASE standards.

The DTP standard is based on the consideration of requirements of a distinguished class of distributed applications, namely distributed transaction processing applications. The design of the TP Service was heavily influenced by the LU 6.2 verbs and parameters of IBM's Advanced Program-to-Program Communications (APPC). Also certain characteristics of the TP-ASE protocol can be traced back to LU 6.2. On the other hand, existing ASEs, the CCR-ASE in particular, had to be incorporated, and the TP Protocol had to use the existing Presentation Service. The resulting composition of functional entities, with their assigned functionality, is therefore not very effective (some changes have been made to the CCR-ASE standards and to the Presentation Layer and Session Layer standards in order to improve this situation).

The TP-MACF does not define distributed interactions and, consequently, does not define PDUs. It does define local coordination of multiple associations. However, since it functions as a 'layer' between the TP Service users and the other components of the TP Protocol, this definition does not contribute to a separation of concerns (see also subsection 4.1.1). On the contrary, the definition of the TP Protocol would be more concise and easier to understand if the TP-MACF was not defined as a separate component.

The role of the TP-SACF does not comply with the general role described in the OSI-ALS. The TP-SACF includes functions such as discarding, queuing, bidding, association management, and concatenation/separation. The actual SACF functionality, as intended by the OSI-ALS, is explained by a statement in the TP Protocol standard that the TP Protocol machine executes action sequences as atomic units. An action sequence is a sequence of protocol actions related to the TP-MACF and one or more ASEs (ACSE, CCR-SE, and user-ASEs) that all result from a single input event to the TP Protocol machine (i.e., a TP request primitive or a Presentation indication primitive)[1].

1. The protocol standard states that: "An action sequence executes completely (...) before the protocol machine becomes available for handling any subsequent input events". And: "(...) any outgoing events created by actions of a state machine that are incoming events to other state machines, are processed by those state machines, and so on, until the only unprocessed events are outgoing events which are not incoming events (that is, they are events at the PSAP or TPSUI)".

Although the TP Service comprises a number of dialogue control and activity synchronization functions, no use is made of the Session Service dialogue control and synchronization functions. For example, the TP Service functions TP-Grant-Control (to grant control of the dialogue to the other user), TP-Request-Control (to request control of the dialogue from the other user), and TP-Handshake (to synchronize the processing of the users) correspond to the Sessions Service functions S-Token-Give, S-Token-Please, and S-Synchronize-Major, respectively. The above mentioned TP Service functions are, however, provided by TP-ASE protocol functions that use normal data transfer (i.e., S-Data).

The functions of the TP Service can be divided into two groups: dialogue related service functions and transaction related service functions. Functions from one group have only ordering relationships with functions from the other group. A possible further division of the dialogue related functions is one that distinguishes between dialogue establishment and termination, and dialogue control and activity synchronization. It would be interesting to investigate a structuring of the TP Service, with a corresponding structuring of the TP Protocol, that reflects this subdivision. Figure 4.3 illustrates a possible structure of the TP Protocol in this case (the distribution over application entities is not shown). The functionality of the lower level service depends on whether it is useful to introduce a hierarchical decomposition of the function groups mentioned. The TP-MACF functionality and the TP-SACF functionality are not separately represented. The ACSE and CCR-SE are represented, but their precise relationship with other functions is not indicated.

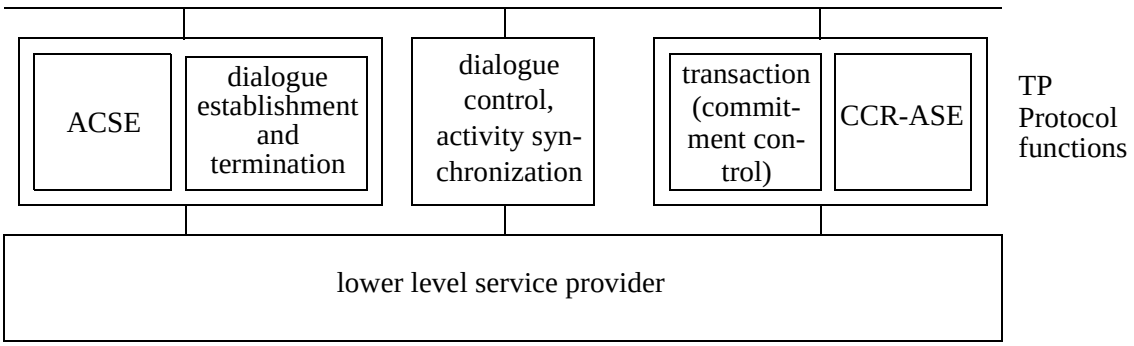


Figure 4.3: Alternative structure of the TP Protocol

4.1.3 Presentation Layer

The purpose of the Presentation Layer is to separate concerns related to the representation of PDUs from other application protocol concerns. The Presentation Service comprises two sets of functions: the actual Presentation Service functions (the functions that require interactions between presentation entities) and functions of the Session Service (the functions that are passed through by the Presentation Layer). The actual Presentation Service

functions allow the Presentation Service users to request the establishment of transfer syntaxes to support abstract syntaxes. The Presentation Protocol defines the negotiation that leads to the establishment of appropriate transfer syntaxes.

If a transfer syntax is established for an abstract syntax, PDUs defined according to the abstract syntax can be exchanged between application entities. The Presentation Protocol imposes the constraint that these PDUs can only be exchanged when represented according to the agreed transfer syntax[1]. This requires that the data parameter of Presentation Service primitives cannot be transparent to the Presentation Protocol: it consists of a list of Presentation Data Values (PDVs)[2], where each PDV may contain nested PDVs as a result of hierarchical Application Layer protocol functions.

PDVs must be interpreted by the Presentation Protocol in order to impose the transfer syntax constraints. The separation of transfer syntax constraints and abstract syntax constraints, where the latter are imposed by Application Layer protocols, violates the orthogonality criterion and is therefore not effective. The relationship between these constraints makes separate implementations of Presentation Protocol entities not very attractive. It is more effective to assign the enforcement of transfer syntax constraints to the Application Layer protocols that also define the abstract syntax constraints (see Figure 4.4).

Most Session Service functions are passed through by the Presentation Layer (apart from the code conversion of data parameters that may be performed by the Presentation Protocol; if we shift the transfer syntax constraints from the Presentation Protocol to the Application Layer protocols, as proposed above, even this function is no longer present). This prompts the question whether the Presentation Layer and the Session Layer cannot be reversed, or defined as components of a single layer. In either case, the Application Layer protocols can use the Session Service directly, and the Presentation Protocol can directly use the Transport Service. If the Session Protocol is used on top of the Presentation Service, it can use the Presentation Service to request a transfer syntax for its PDUs, which avoids the need of a separate approach for the encoding of session PDUs. In this respect, this alternative composition improves the consistency of the OSI-ULA.

1. Although the Presentation Layer standards suggest that the Presentation Protocol is responsible for code conversion (the transformation from the local concrete syntax to the transfer syntax, and vice versa), we believe that (1) this function is not a proper function of a protocol *architecture*, and (2) in a protocol *implementation*, this function may be performed at any time, by any entity, before and after data transfer. For this reason, we only discuss transfer syntax constraints, or coding constraints, which are imposed on PDUs.

2. The purpose of a *list* of PDVs in a data parameter is unclear. Neither the Presentation Layer standards, nor the OSI-ALS explain when it is useful to have more than one PDV in a single data parameter. One reason may be to effect some kind of concatenation/separation function. However, on basis of the orthogonality and propriety (parsimony) criteria, this is considered a poor design choice.

In the following, we briefly investigate the possibility of alternative compositions, by tracing the dependency of Presentation Protocol functions on Session Service functions, according to the current standards:

- the presentation connection establishment function, which includes the negotiation of transfer syntaxes, uses the session connection establishment function. The latter function effects the assignment of a newly established transport connection or the assignment of a retained transport connection. If the Presentation Protocol uses the Transport Service directly, it should include this transport connection management functionality.
- the context alteration function of the context management functional unit, which allows the negotiation of transfer syntaxes after the presentation connection has been established, uses the session typed data transfer function in order to be free from token restrictions. If the Presentation Protocol uses the Transport Service directly, the context alteration function can use the transport normal data transfer function instead. Moreover, the typed data functional unit does not have to be negotiated by the Presentation Protocol during presentation connection establishment if it wants to provide context alteration.
- if the context restoration functional unit is selected, the Presentation Protocol remembers the presentation context when a synchronization point is exchanged (and forgets old presentation contexts when a major synchronization point is exchanged) and restores the presentation context when a resynchronization is performed to that point. These functions can no longer be provided if the Presentation Protocol uses the Transport Service directly since it does no longer ‘see’ the session (re-) synchronization points. Instead, a resynchronize PDU should be mapped onto the data parameter of the context alteration function in order to allow for a re-negotiation of presentation contexts (if necessary) during resynchronization. (This is also consistent with the re-negotiation of token positions during resynchronization.)
- finally, the presentation connection release and abort functions use the corresponding session functions which effect termination or retainment of the transport connection without loss of data. If he Presentation Protocol uses the Transport service directly, it should include this transport connection management functionality.

Hence, alternative compositions of the Session Layer and the Presentation Layer require a new mechanism for presentation context restoration, but otherwise are clearly viable and, as mentioned above, also attractive. Figure 4.4 illustrates the Presentation Protocol in an alternative composition (the distribution over presentation entities is not shown). Note that the results of the transfer syntax negotiation should be passed to the Session Protocol and the (selected) Application Layer protocols. The latter protocols can then directly use the Transport Service. This suggests that it is possible to define the (revised) Presentation Layer functionality as a component of a dynamic layer.

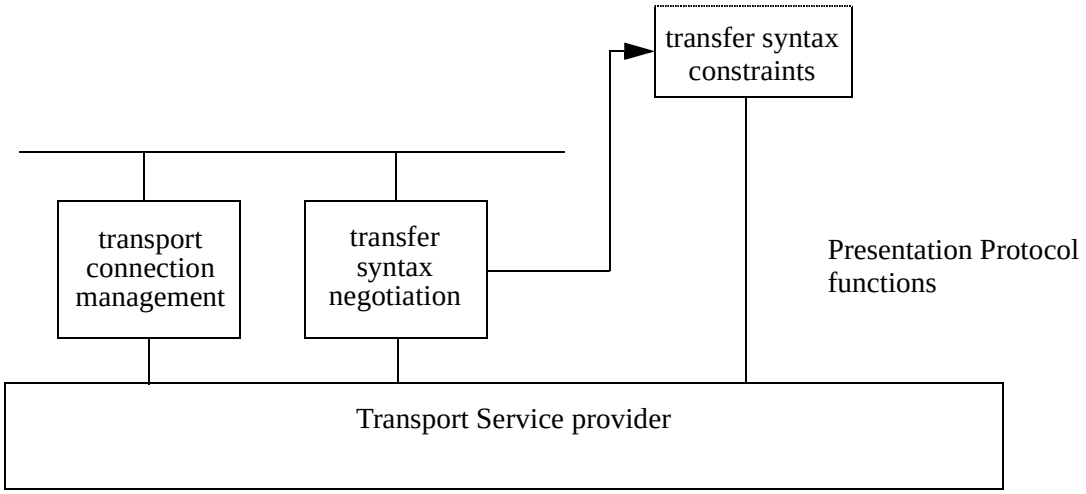


Figure 4.4: Alternative Presentation Protocol that directly uses the Transport Service

4.1.4 Session Layer

The purpose of the Session Layer is to separate concerns related to dialogue control and activity synchronization from other application protocol concerns. The Session Layer standards are heavily influenced by contributions from CCITT (T.62 recommendation) and ECMA (ECMA-75 standard). It is through the forced integration of these contributions that many violations of design quality criteria were introduced (see also Chapter 2, Design Quality Criteria), including:

- functional overlap of synchronization point services (ECMA-75) and activity management services (T.62);
- inappropriate definition of the initial synchronization point serial number negotiation, resulting from the fact that ECMA-75 sets an initial synchronization point serial number, while T.62 does not;
- absence of serial number wrapping, which leads to some strange service constraints;
- superfluity of the basic concatenation function, inherited from T.62;
- superfluity of a separate give control function, inherited from T.62;
- the complexity of exception reporting, due to the interrelationships with other functions. One would expect exception reporting to be a high priority function and therefore more independent from other functions than currently is the case.

Furthermore, the Session Protocol includes functions that perform segmentation/reassembly and a limited form of concatenation/separation. These functions optimize data

transfer, rather than supporting distributed application interaction, and are therefore not considered proper application protocol functions[1].

Figure 4.5 illustrates the Session Protocol in the alternative composition of the Session and Presentation Layer, as discussed in subsection 4.1.3 (the distribution over session entities is not shown). Apart from the negotiation of session parameters (functional units, initial synchronization point serial number, and token settings), two main sets of functions are distinguished, viz. dialogue control and activity synchronization functions. Dialogue control functions can be considered a 'clean' version of the session functions related to token constraints and token management. Activity synchronization can be considered a 'clean' version of the minor synchronize, major synchronize, resynchronize, and activity management functional units. The figure suggests that it is possible to define the (revised) Session Layer functionality as a component of a dynamic layer.

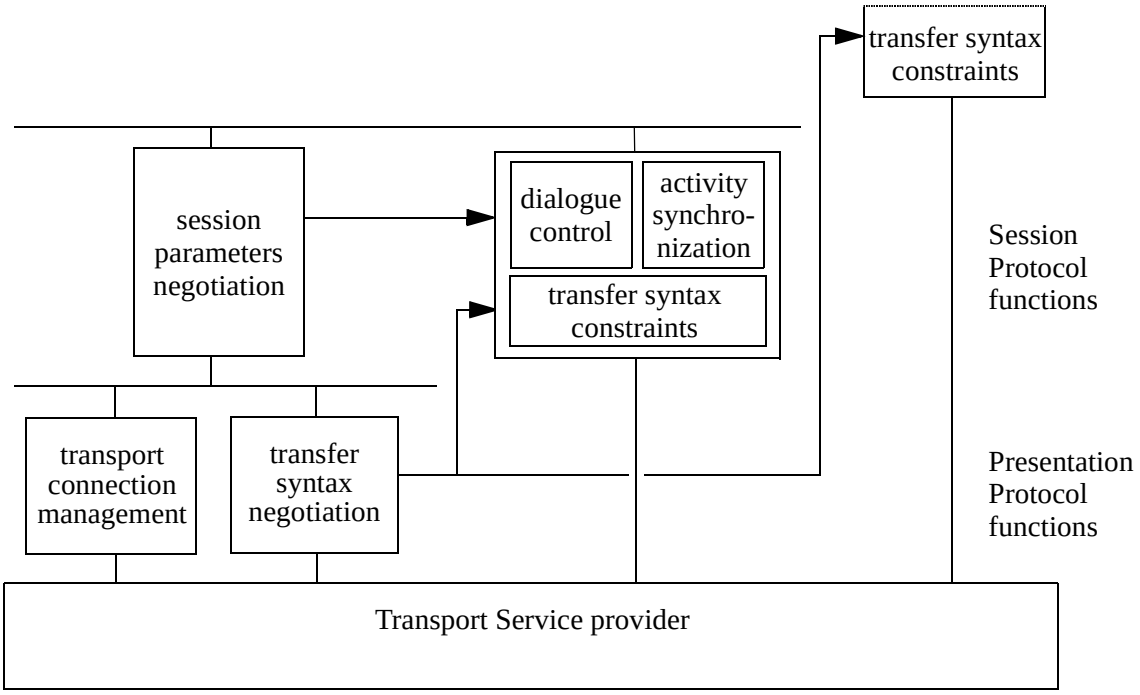


Figure 4.5: Alternative Session Protocol that directly uses Presentation and Transport Service

1. Session concatenation/separation is based on a categorization of session PDUs: depending on their categories, session PDUs can or cannot be concatenated. Since the categories of session PDUs cannot be distinguished at lower protocol levels, it might be argued that concatenation/separation *must* be a session protocol function. This argument can be refuted, however, by the observation that the categories themselves are rather arbitrary and, consequently, unnecessarily restrictive with respect to concatenation/separation.

4.1.5 Transport Service

The Transport Service provider is responsible for data transfer between end-systems, independently of the information contents of the data. Thus, the Transport Service provider separates concerns related to ‘transparent’ data transfer from information processing and transfer concerns. This separation widely considered as one of the key decisions incorporated by OSI ([Zimmermann83]). We support the principle of this division, based on the orthogonality criterion.

Two basic types of Transport Service have been defined: connectionless (-mode) and connection-oriented. Each of these types has specific characteristics which makes it more or less suitable, dependent on the class of distributed applications that must be supported.

There is no inherent distinction between connectionless and connection-oriented with respect to reliability, as is often claimed. This depends on the ‘quality of service’ (QoS) negotiated and the implementation, and realization, of the service. The main, implementation independent, distinctions are:

- connection-oriented has the overhead of connection establishment and connection termination compared to connectionless;
- during connection establishment certain characteristics of the connection are established which apply to all subsequent instances of SDU transfer. With connectionless, characteristics have to be established for each instance of SDU transfer. Hence, the transfer of a SDU on a connection requires less overhead than a connectionless SDU transfer. For example, QoS and address information can be omitted when a SDU is transferred on a connection, but must accompany each SDU when connectionless data transfer is used. On the other hand, this information can only be omitted since state information is maintained by the service provider. Connectionless data transfer does not require the service provider to maintain state information after a SDU is transferred.

The transport connection establishment function provides a data parameter which allows Transport Service users to transfer data during the connection establishment phase. The data parameter, however, only permits the transfer of 32 octets of data maximum, which makes its use rather restricted. In particular, this limited length prohibits ‘one-shot’ establishment of the application association and the transport connection. One shot establishment might provide an attractive alternative, next to step-wise connection establishment, for distributed applications that do not require the exchange of large amounts of information. Also the transport disconnect function provides a data parameter. This parameter has a length restriction of 64 octets. It is optionally used by the current version of the Session Protocol to convey additional information on the reason of session connection release.

The transport disconnect function unconditionally terminates the transport connection. The release cannot be rejected and is therefore not confirmed. Consequently, the use of this function may lead to loss of data in transit. Many distributed applications require an orderly release, i.e. a release that guarantees no loss of data and that can optionally be rejected. For this reason, the Session Layer provides this function. It seems more appropriate, however, to make orderly release an (optional) Transport Service function. With the provision of a data parameter with less severe length restrictions this would also permit the termination of the application association and the transport connection in one shot.

So called ‘gigabit’ distributed applications, such as full motion video conferencing and computer imaging, (will) require high throughput and varying emphasis on particular aspects of performance. Problems are often reported with ‘old’ data transfer protocols, including the Transport Protocol (see [LaPorta91], for example), when considering these applications. Although the reported problems are real, it should be noted that these problems do not necessarily relate to the Transport Service. The QoS parameter provided by the connection establishment function consists of a number of QoS sub-parameters, each of which can be negotiated independently. In addition, of some parameters (e.g., throughput and transit delay) an average as well as extreme values can be established (this is important for distributed applications that require constant throughput or transit delay). We think that the Transport Service is in principle appropriate and sufficiently general in this respect.

Both the connection-oriented and the connectionless Transport Service only provide point-to-point data transfer, or unicast. Some advanced distributed applications, such as electronic conferencing and group communication, require (point-to-) multipoint transfer, or multicast. Multicast is a proper data transfer function, since it can be provided independently of the information contents of the data. Although the concept of multi-endpoint connection is defined in the OSI-RM, no work has been done to standardize a connection-oriented multicast Transport Service. Also no connectionless multicast Transport Service has been standardized so far. Most non-interactive and non-real time distributed applications use therefore the Message Transfer service of the MHS standard if they require multicast. This service entails a general, application independent, ‘store-and-forward’ transfer of messages from a sender to one or more receivers. The messages are not interpreted or changed by the Message Transfer Protocol, except when code conversion is performed. Although the Message Transfer Protocol is positioned in the Application Layer, it seems more appropriate to position it at the Transport Protocol level. The Message Transfer Service can accordingly be considered as a special type of connectionless multicast Transport Service (note that the store-and-forward principle is a protocol characteristic, not a service characteristic).

4.2 Application protocol realizations

Many application protocol realizations conforming to OSI standards have been criticized by users. Support, on the other hand, has been expressed for the OSI-ALS as a coordinating architectural framework, that can solve the problem of proliferation of incompatible application protocol products.

This section briefly discusses the extent to which OSI application protocol products have been successful, and the general complaints of users concerning these protocol products.

4.2.1 Level of acceptance

It is often argued that OSI has not been very successful on basis of the observation that only few OSI protocol products are presently widely used ([Kalin92]). Apparently, most OSI protocol products are, according to the users, not as effective or efficient as proprietary protocol products. The advantage of being ‘open’ is not considered a sufficient compensation for the perceived disadvantages.

This observation also applies to OSI *application protocol* products. A notable exception is formed by realizations conforming to the Message Handling System (MHS) standard ([IS10021:92]), which are now widely used. MHS was first defined by ITU-TSS (at that time CCITT), and later adopted by ISO. The current MHS standard is jointly defined by ISO and ITU-TSS and conforms to the OSI-ALS. Also promising is the Distributed Transaction Processing standard ([IS10026:92]), since it has boosted from its start the interest of many manufacturers ([Mantelman89]), although its completion is too recent to draw definite conclusions. But most other standards are less successful. Their development took many years after which they were virtually neglected by manufacturers and users. An example is the Job Transfer and Manipulation (JTM) standard ([IS8831:89], [IS8832:89]), whose development took about 10 years and for which so far no commercial products have been announced.

It appears that after an initial optimism about OSI, which was expressed after the lower layer standards were nearly finished and many upper layer standards were scheduled in the next few years to come ([Rauch86a]), interest in OSI has rather dropped. An important factor that contributed to the decreasing interest in OSI is that most upper layer standards, in particular Application Layer standards, could not be completed according to the original schedule ([Rauch86b]), while propriety application protocol products were successfully launched.

Despite the previous observation, there are also arguments for assuming that the OSI market will grow in the near future. A sufficiently mature and rich set of Application Layer standards is, at last, available that potentially support a wide range of distributed

applications. The abundance of options provided by these standards are currently considerably reduced through a second round of standardization, called *functional standardization*, thus making standards more acceptable to users that prefer simple solutions for reasons of interworking reliability, performance, and cost effectiveness.

Parallel to the above development is the growing interest of users in ‘open’ distributed applications. For considerable time, user organizations were forced to develop their own distributed applications on top of data transfer systems, because suitable products were not available on the market. However, once these products became available, from various manufacturers, the problem of incompatible products arose. This problem, together with the growing necessity of internationalization of and cooperation between user organizations, created the need of standardization of application protocols from the point of view of users. The MAP and TOP initiatives can be seen as early exponents of this development ([Allan86]).

4.2.2 User complaints

Criticism on OSI application protocol products concern, besides high prices, the following three categories: lack of functionality, lack of efficiency, and lack of flexibility.

Lack of functionality

The complete set of standardized application protocols only supports a limited set of the distributed applications for which the users want support. In particular, there are currently no OSI application protocols that adequately support emerging, modern distributed applications, such as open distributed processing, distributed multimedia applications, and computer supported cooperative work. Often quoted functional deficiencies include:

- group interaction support ([Kündig91], [Schürmann90]);
- multimedia synchronization support ([Shepherd90]);
- distribution transparency selection support ([Rodden92]).

Lack of efficiency

OSI application protocol products are generally considered slower and less cost-effective than proprietary application protocol products. This is usually blamed on the layered architecture of the OSI-ULA and on the many options provided by OSI application protocols. The set of options of a particular application protocol enable the support of a broad class of distributed applications. Any specific distributed application will generally only require a small subset of these options. Although unnecessary options can be made inactive through negotiation, most implementations and realizations are structured to facilitate the activation of any combination of options, not to optimize performance. In

addition, the negotiation process itself may be rather complex and time consuming in realizations.

Some application requirements map onto data transfer requirements, which in turn are not adequately supported by current OSI data transfer protocols. Examples of such requirements are:

- multicast data transfer ([Ngoh89]);
- suitable performance ([LaPorta91]).

Lack of flexibility

The layered architecture of the OSI-ULA is considered not flexible enough to cope with requirements of new distributed applications and to profit from enhanced data transfer capabilities enabled by technological advances ([Solvie92]). Due to the diversity of distributed application requirements and the narrow range of (specific) requirements currently supported by Application Layer standards, proprietary application protocol products may be preferred over OSI application protocol products, or users may develop their own distributed applications on top of data transfer systems. Recent developments concerning the OSI-ALS (the *extended* Application Layer Structure, XALS) are considered useful, but not sufficient to solve the problem of lack of flexibility.

The following remarks can be made with respect to these user complaints:

- lack of functionality is partially due to the slow pace of standardization. This is discussed in section 4.3;
- lack of efficiency is sometimes due to a misunderstanding (or to under-exploitation) of the implementation freedom offered by OSI protocols. This is discussed in section 4.6. Another source of potential problems are standards that reflect political compromises (see section 4.3); and
- lack of flexibility is partially due to the static layers in OSI-ULA. Other inhibiting factors are the unclarity of some design concepts and the absence of design methods. The latter problems are discussed in section 4.4.

4.3 Problems of standardization

OSI standards have to be internationally agreed by all stakeholders: manufacturers, vendors, and users. Manufacturers and vendors are from two major industries, viz. from the computer industry or from the telecommunications industry, each of which has different interests.

The computer industry has a main interest in (distributed) information processing applications. It is also well aware of the market potential of distributed applications, based on their experience with selling application software products to users. The constituents of the computer industry, computer manufacturers and vendors, have, however, different captive markets to protect and therefore try to influence application protocol standards in order to make these standards easier to incorporate in their own environments. The telecommunications industry has less experience with distributed information processing applications. It has a main interest in data transfer infrastructures, and 'value added' services implemented on top of these infrastructures.

Computer manufacturers are eager to compete each other in the area of advanced application services, characterized by high-speed, low-cost data transfer requirements. The PTTs, as representatives of the telecommunications industry, are less eager to realize high-speed and low-cost data transfer services since their markets are geographically partitioned and thus lack the push of competition (although this situation is now changing as a consequence of the ongoing deregulation of PTTs). Value added services generally further reduce effective throughput and for that reason were sometimes considered as 'nuisance added' services rather than 'value added' services by the computer industry ([Chesson91]).

Users want advanced as well as less advanced application services, since they have widely varying work requirements. Also, they want to be able to 'shop around', independently of manufacturers and vendors. But users are generally not capable of forcing standardization in a direction that is to their best benefit. This is partly due to the fact that users are not sufficiently organized in powerful user groups, and partly because it is often unclear what is the users' best benefit.

The consequences of the mix of interests involved in OSI application protocol standardization seem to be as follows:

- *slow pace*: standardization progresses at a slow pace. This is partially so because of the procedures that have to be followed in order to guarantee the broad involvement of interested parties and the acceptance of standards. The other important reason is that the different interest groups have to come to an agreement concerning the contents of standards. This is a difficult process since the standards should be robust to technological changes, but the functions that can be conceived often depend on the technological state of the art. In addition, progress may be intentionally slowed down because of the potential threat of standards to captive markets of some of the (manufacturer and vendor) participants. The consequence is that, at the time of completion, a standard may have become obsolete, since it was based on assumptions no longer valid or since it offers a functionality no longer wanted. Technology-driven developments may therefore occur more rapidly than standardization can follow.
- *progress by compromise*: agreements on controversial issues are often reached by political compromises that do not embody the best technical solutions. Either defini-

tions of parts of standards are intentionally kept vague, to conceal differences of opinions, or multiple views are supported by the introduction of independent options in standards. In the first case, this may lead to interpretation problems. Different interpretations by implementors may lead to protocol products that are unable to interwork, despite the fact that conformance to the same standard is claimed. In the second case, an abundance of options is offered by protocol products that confuse the user, and that deteriorate efficiency.

- *gap between 'open' and 'closed'*: as long as no suitable protocol standards are available that can support the data transfer requirements of more advanced distributed applications, standardization of application protocols that directly support these applications will not commence, or will be based on non-optimal data transfer services. This, combined with the slow pace of standardization, results in a gap between 'open' and 'closed' distributed applications. Computer manufacturers will be able to demonstrate advanced and well performing distributed applications based on proprietary solutions, and less advanced and less performing distributed applications that are based on OSI standards. OSI protocol products appear to be based on technologies of the past, and are thus beaten by other protocol products that incorporate technologies of the present.

4.4 Upper layer model

The main cause of the design quality deficiencies discussed in section 4.1 can be traced back to the poor definition of the concept of service and of some of the OSI-ALS concepts. As a consequence, service decomposition methods are not defined, and protocols are not designed on basis of required services, with proper consideration of design quality criteria. This section presents the problems related to the interpretation of some OSI design concepts and discusses the absence of OSI protocol design methods.

4.4.1 Unclear design concepts

The importance of the service concept in the design of protocol standards has generally been underestimated ([Vissers85]). If a service is defined as an abstraction of a protocol, the design of a service can be used as an intermediate step in the design of a protocol.

As mentioned in Chapter 1 (Introduction), a service is considered an architecture of an interaction system of system parts. It defines what is accomplished by the interactions between the system parts, without defining how the interactions are implemented. The boundary between the interaction system and its environment is internal to the system parts. The intersection of this environment and a system part is called service user. The interaction system itself is called service provider.

Since the implementation of a system part may not implement the internal boundary in terms of concrete interfaces, the service should abstract from the implementation of inter-

actions between the service user of this system part and the service provider. This implies that the service represents the interactions between a service user and the service provider as integrated interactions. These integrated interactions are called service primitives. From this we can draw the conclusion that service and service provider are distinct concepts. A service does not define the individual responsibilities of the service users and the service provider, since it is defined in terms of service primitives. If the service provider is implemented as a separate functional entity, the service primitives have to be mapped on concrete interfaces, requiring different responsibilities at either side of each concrete interface. Consequently, the external behaviour of the service provider may be different from the behaviour defined by the service. Part of the service behaviour may be implemented by the service users, or even by a separate interface process. A service can therefore also be considered as the maximum architecture of a service provider (a definition to that effect can be found in [Schot92]). Another consistent interpretation of service is: an integrated interaction system of service users and the service provider (this definition is proposed in [Ferreira94]). The term ‘integrated’ denotes here that only integrated interactions between the service users and the service provider are represented.

A protocol is considered an implementation of an interaction system of system parts (see Chapter 1, Introduction). It defines the interactions between the system parts, in terms of contributions of the system parts to the interactions. The contributions are defined with protocol actions that effect the sending and receipt of PDUs. The intersection of a system part and the service provider is called protocol entity. Protocol standards, such as OSI standards, must be protocol architectures: they should define what contributions are made to interactions, not how these contributions, and thus the protocol entities, can be implemented in hardware or software.

It is probably because the role of service in protocol design has never been fully appreciated that the definition of the service concept was never very clear in OSI. After its introduction in the OSI-RM, several attempts have been made to clarify the concept. One such attempt led to the Conventions for the Definition of OSI Service ([IS10731:91]). However, the following problems can still be recognized:

- *distinction between service and service provider*: According to [IS10731:91], a service is "a capability of a service provider which is provided to service users (...)". Furthermore, a service is defined in terms of service primitives, where a service primitive is "an (...) interaction between a service user and its service provider". These statements do not distinguish the concepts of service and service provider. As a consequence it is unclear whether (1) the service behaviour pertains solely to the service provider, or (2) the service behaviour is determined by (constraints imposed by) the service users and the service provider. And even if the second interpretation is intended, it is unclear whether or not the constraints imposed by the service users and the service provider on service primitives should be integrated or separately defined by the service.
- *generality of service primitive*: Service primitives have often been interpreted as a special kind of interaction, namely one that effects the passing of information in one

direction only, either from user to provider or from provider to user. Although this interpretation clearly limits the usefulness and the applicability of the service concept, it prevailed for a long time. Each time this interpretation interfered with the requirements with respect to a service standard, the discussion of the service concept was restarted, and the standard was delayed. This happened, for example, during the definition of the Session Service, where the service provider determines the value of synchronization point serial numbers in certain request primitives ([Caneschi86]). In order to partially resolve this problem, [IS10731:91] now contains the explanation that a service primitive "has associated with it a direction (...) indicating the *main flow of information*". Information associated with a particular service primitive parameter "may be passed in the direction opposite to that of the service primitive".

Services should represent service primitives in the most abstract way, in order to allow maximum freedom for the protocol implementor and manufacturer. A service primitive can therefore best be seen as an integrated interaction in which values of information are established. A service primitive abstracts from any particular distribution of constraints that apply to the values of information. It also abstracts from any particular representation of the information values. The concept of service primitive should not be restricted to special kinds of integrated interactions, since this would limit the type of services that can be defined. Hence, there should be no restrictions on the number of information values that can be established in a service primitive, on the constraints that apply to any of these values, and on the relationships between service primitives.

4.4.2 Absence of design methods

The OSI-RM does not define a design methodology. Consequently, the OSI-ULA could not be designed according to a design methodology, e.g. in a top-down fashion. Instead, the OSI upper layers are the result of the layer composition and ASE composition techniques, discussed in Chapter 3 (OSI Upper Layer Architecture and Model: State of the Art). The development of these layers followed a bottom-up approach, starting from the Transport Service, which was one of the first stable OSI standards. Although the OSI-RM describes a number of principles for layer composition which aid the separation of protocol concerns, these principles were rarely explicitly applied, while on the other hand layer standards were often influenced by political factors and their historical context (see [Caneschi86], for example, concerning the architectural choices incorporated by the Session Layer standards). Corresponding service and protocol standards were at best developed at the same time, but never were the protocol standards derived from the corresponding service standards.

The absence of design methods in OSI is partly related to the lack of recognition of the service concept, and consequently to the poor definition of this concept. The relationship between a service and a protocol architecture (i.e., an OSI protocol) is not precisely defined in the OSI-RM. But also the relationship between an application service and a

distributed information processing application, and the relationship between a protocol architecture and a protocol implementation are not clearly defined. The former is illustrated by the problematic coordination of OSI (in particular, OSI-ALS) and Open Distributed Processing ([IS10746:93]). The latter is illustrated by misinterpretations of the concepts of service primitive and abstract interface (as opposed to a concrete interface).

The unclear relationship between distributed information processing, application service and application protocol are the reason why the Application Layer concepts, defined in the OSI-ALS, are easily misinterpreted. These concepts, ASE, SACF, MACF, etc., did not result from a refinement of the service or the protocol concept, and instances of these concepts are not derived through a structuring or decomposition of a given service or protocol.

The fact that service and protocol can be considered as related abstraction levels has long been recognized by many researchers outside OSI, and formed the basis of numerous proposed design methods (see [Vissers77], [Khendek89], [Saleh91], and [Sinderen92], for example). One might expect that such methods would have been incorporated in the OSI standards development practice, even if they were not described in the OSI-RM. This has not happened, however, maybe because most of the proposed methods are based on specific specification models and languages.

4.5 Application service structuring and decomposition

This section discusses design methods that can be used to derive application protocols structured in terms of layers and ASEs. These design methods are based on the concepts of service and protocol, as presented in section 4.4. The design methods are also used here to clarify the concepts of layer and ASE, and the OSI structuring techniques discussed in Chapter 3 (OSI Upper Layer Architecture and Model: State of the Art). The design methods are motivated by design objectives.

First the design method that leads to layered protocol architectures is discussed, and subsequently the design method that leads to application protocols composed of ASEs.

4.5.1 Layer protocol design

Design objective

The objective of layer protocol design is to design protocols step-wise, where each step considers part of the distributed interaction concerns and leads to the definition of a protocol layer. A protocol layer has an associated (layer) protocol that uses a lower level service and provides the required service. A sequence of steps leads to a hierarchical composition of protocol layers.

Lower level services should be identified that can be re-used in other layer compositions to support other (classes) of distributed applications.

Design method

The starting point is a given service, called the required service. The first design step considers the relationship between service primitives of the required service that occur at different service access points (SAPs) and separates aspects of this relationship that will first be implemented from aspects whose implementation is deferred to later design steps. The implementation of the former aspects is defined by distributed interactions, and constitutes a layer protocol between protocol entities. The latter aspects are defined by a lower level service. The layer protocol is the only user of the lower level service: PDUs that are exchanged between the protocol entities are transferred as data via the lower level service; other functions of the lower level service are used or passed through by the layer protocol in order to provide the required service. The required service is thus decomposed into a layer protocol and a lower level service.

The lower level service can be treated as a required service in the next design step and be decomposed in the same way. This can be repeated until a general purpose lower level service is identified whose implementation is already defined. Each design step in this method replaces a required service by a layer protocol and a lower level service. The final result is a layered protocol architecture. A particular application of this method would yield the OSI-ULA and, if further pursued, the OSI 7 layer protocol architecture. Figure 4.6 illustrates a single design step according to this method.

Each protocol layer separates particular concerns from other protocol layers. The separation of concerns is achieved through the mapping of PDUs to data parameters (SDUs) of lower level service primitives. Data is transparently transferred by the lower level protocol. Non-data parameters of service primitives are used to represent information that is relevant to two protocol levels, viz. the layer protocol that uses the service and the layer protocol that provides the service. The number of non-data parameters in service primitives of the lower level service should be minimized. This simplifies the lower level service and consequently improves the separation of concerns, or orthogonality, of the protocol layer. In any way, the lower level service should be much simpler (to implement) than the required service, otherwise the separation would not be very effective from an engineering point of view. Subsection 4.1.1 mentioned some limitations of layer composition when applied to application protocol layers. These limitations should also be observed when applying this method.

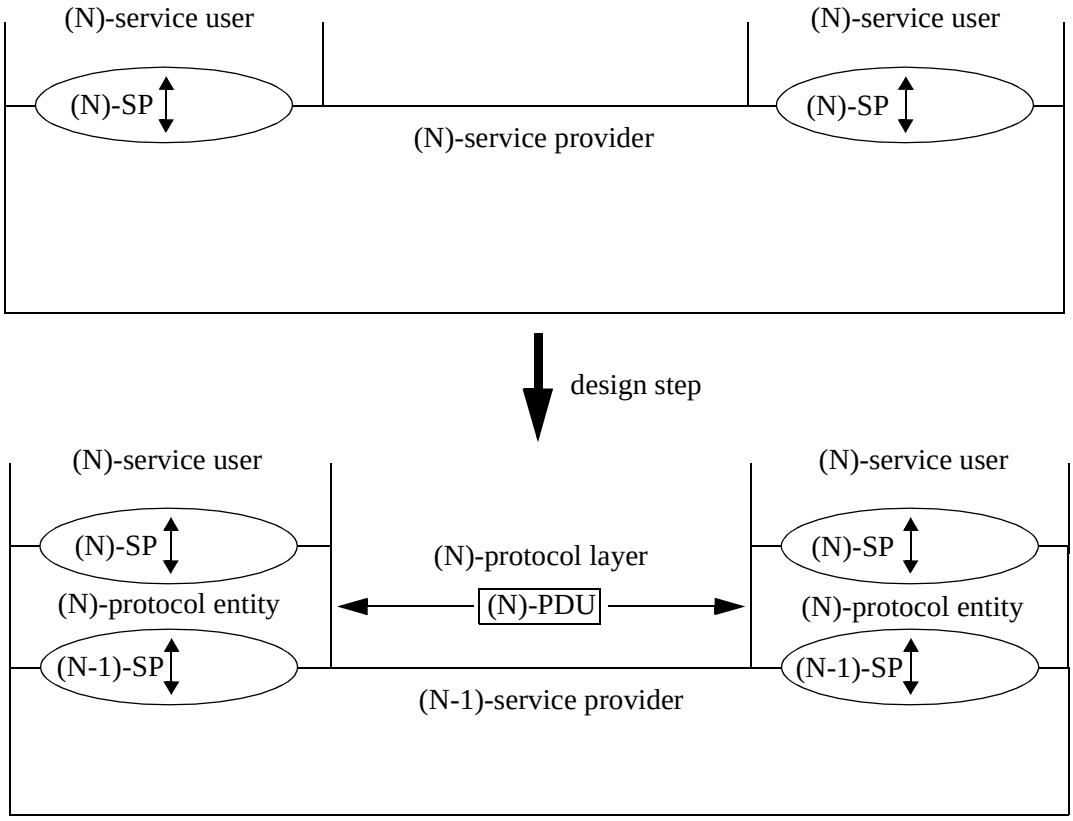


Figure 4.6: Design step that decomposes a required service into a layer protocol and a lower level service

4.5.2 Application service element protocol design

Objective

The design objective of ASE protocol design is to structure an application protocol layer as a composition of ASEs, where each ASE has an associated ASE protocol that provides a subset of the functions of the required service.

In particular, ASEs should be identified that can be re-used in other compositions to support other (classes of) distributed applications.

Design method

The starting point is a required service. This service is structured on basis on the identification of subsets of service functions, or ASE services, that can be recognized and defined as relatively self-contained, or orthogonal, units. An additional important criterion

for identifying ASE services is that some of the ASE services can be re-used as components in other required services. The level of ‘self-containedness’ of an ASE service is determined by its relationship to other ASE services in the composition of the required service. This composition must define possible ordering relations and parameter value dependencies between service primitives of different ASE services. Complex relationships will complicate the definition of the composition.

The composition can be relatively easily defined if the relationship between ASE services is such that it can be represented by additional constraints on service primitives at each service access point (SAP). These additional constraints can then be represented by separate (synchronized) behaviour components in the composition of the required service. Thus, (parameter values of) a service primitive of some ASE service should not depend on (parameter values of) service primitives of other ASE services, unless the latter service primitives occurred previously at the same SAP.

Most required services will require that the order of indication primitives depends on the order of the corresponding request primitives. This may result in another relationship between ASE services, which we will refer to as the sequencing relationship. Consider, for example, the situation where the order of two indication primitives, say *ind1* and *ind2*, depends on the order in which the corresponding requests, say *req1* and *req2*, occur. If *req1* and *req2* may occur in any order, then their sequencing relationship must be preserved by the service in order for *ind1* and *ind2* to occur in the right order. If *req1* and *ind1* belong to one ASE service and *req2* and *ind2* belong to another ASE service, this sequencing relationship must be defined between the ASE services. In general, it is difficult to represent this relationship as an additional constraint in the composition. (An alternative solution for defining the relationship between ASE services will be presented in Chapter 7, Application Protocol Reference Architecture.)

An ASE service should be defined as any ‘normal’, ‘complete’ application service, except that is must be possible to define its relationship to other ASE services. Figure 4.7 illustrates the structuring of a required service as a composition of two ASE services. The local relationships between the ASE services are represented by a thick, horizontal double headed arrow at each SAP; the sequencing relationship is represented by a thick, vertical double headed arrow.

The next step is to decompose the ASE services, according to the design method presented in the previous subsection. Figure 4.7 also illustrates this step. We have assumed here that the decomposition of both ASE services yield the same lower level service, such that the ASE protocols can share one lower level service. The relationship between the ASE services may imply that additional relationships must be defined between the ASE protocols, viz. ordering relations and value dependencies between PDUs sent/received by collocated ASE protocol entities. These relationships can be seen as the implementation of the sequencing relationship in the composition of the ASE services. Another reason for defining local relationships between the protocol actions of ASEs

concerns efficiency: an ASE can be immediately enabled after the receipt of a PDU by another ASE, instead of being blocked until the indication corresponding to the received PDU has occurred.

The relationship between protocol actions of collocated ASE protocol entities is represented in the Figure 4.7 by a thick, double headed arrow between the PDUs sent/received.

According to the OSI-ALS, the relationships between ASEs are represented by SACFs. No distinction is made, however, between composing ASE services and composing ASE protocols. In addition, SACFs are restricted to defining additional constraints on service primitive occurrences, possibly relating such occurrences at different service levels. As mentioned above, composing ASE protocols generally requires the definition of relations between PDUs of these protocols.

Only a simple form of ASE composition can be derived with the method described here, which suffices to clarify the concept of ASE. According to the OSI-ALS, ASEs may be composed ‘next to each other’, but also ‘on top of each other’ (as sub-layers of the Application Layer). This method only supports the first composition. The second composition requires an extension which is discussed in Chapter 7 (Application Protocol Reference Architecture). We did also not consider composing ASEs according to the cooperating main service approach. The support of this form of composition is briefly discussed in Chapter 8 (Issues for Further Work). (Both discussions are based on design concepts that will be introduced in the next two chapters.)

4.6 Application protocol implementation approaches

The definition of OSI protocols in terms of service primitives (integrated interactions) and protocol actions (send/receive PDU actions, defined independently of how PDUs are processed) is intentional in order to leave maximum freedom to the implementor. The layer structure should therefore not be interpreted as a requirement to introduce concrete interfaces at each layer boundary. Especially in the OSI-ULA, layer composition is meant to separate design concerns and to allow concurrent standardization efforts. Although a layered implementation structure, with concrete interfaces between adjacent entities, was recommended in the early days of OSI ([Zimmermann83]), it is now generally concluded that, in order to satisfy the interworking requirements of advanced distributed applications, layering is not the most effective modularity for implementations ([Clark90]). Many initial OSI protocol products were slow mainly because no efforts were made, during implementation and realization phases, to optimize efficiency ([Strauss87]).

This section discusses one particular aspect of implementation freedom offered by protocol architectures, namely that of multi-layer implementation. It further presents some approaches proposed in the literature to optimize the implementation of layered (in particular, application) protocol architectures.

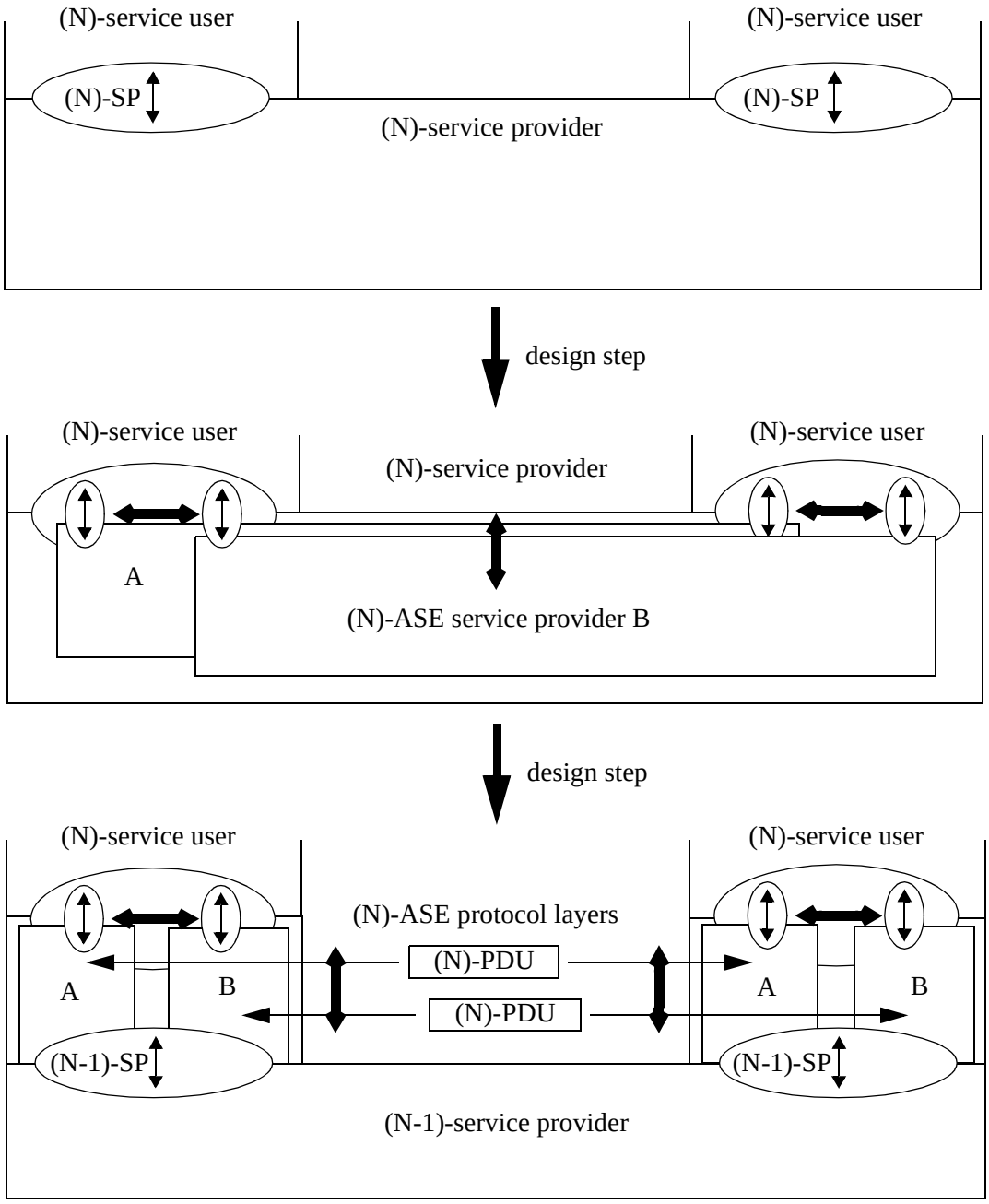


Figure 4.7: Design steps that decompose a required service into ASE layer protocols and a lower level service

4.6.1 Multi-layer implementation

The layered structure of the OSI-RM is often mentioned as one of the reasons for the low efficiency of OSI protocol products. There is, however, no requirement that implementations and realizations must be structured accordingly. The definition of abstract interfaces between protocol layers (or between service users and service provider) plays an important role in this respect. If the implementor wants to preserve the layer structure, i.e. if he wants to introduce implementation components or modules corresponding to protocol entities, he has to refine the abstract interfaces. The resulting concrete interfaces should define how service primitives are implemented, including an assignment of responsibilities to layer components and concrete representations of the information established. If the implementor wants to integrate layer functions, he can replace the abstract interface by a mechanism that implements the combined protocol actions of the layers involved. We will refer to the latter as a *multi-layer* implementation.

Figure 4.8 illustrates the principles of a multi-layer implementation. It shows the representation of a layered sender architecture, as might be defined by OSI, and the representation of an integrated sender architecture that can be derived from the layered architecture. In the layered sender architecture, the protocol functions of different layers produce different portions of protocol control information (PCI). The protocol functions are separated by the mapping of PDUs onto SDUs. A layered implementation may be derived from this architecture that performs the protocol functions associated with different layers in sequence, possibly with read/write actions to pass results between the functions. The integrated sender architecture defines a single layer. The protocol functions of this layer produce PCI that corresponds to the concatenation of the portions of PCI defined in the layered architecture. A multi-layer implementation may be derived from the layered architecture via this architecture, such that parallel processing is optimized and read/write actions are avoided. For example, the layered protocol presented in Figure 4.2 could be implemented using multi-layer implementation, permitting combined processing of *a1* and *b1*, and of *a2* and *b2*.

The integration of protocol functions in multi-layer implementations is not always easy or possible, but depends on the relationship between PDU exchanges defined in the protocol architecture. The following functions, in particular, constrain integration possibilities:

- *demultiplexing*: demultiplexing is the reverse function of multiplexing. It identifies PDUs that belong to different connections in a stream of PDUs that is received on a single lower level connection. Demultiplexing requires the extraction of some of the PCI in each received PDU in order to determine the connection to which it belongs. Only if the connection is determined, the processing of the PDU can proceed, since the protocol functions involved in the processing generally depend on the local state of the connection. This makes it hard to integrate the layer where demultiplexing is performed and the lower layer ([Clark90]). Although the OSI-ULA does not contain

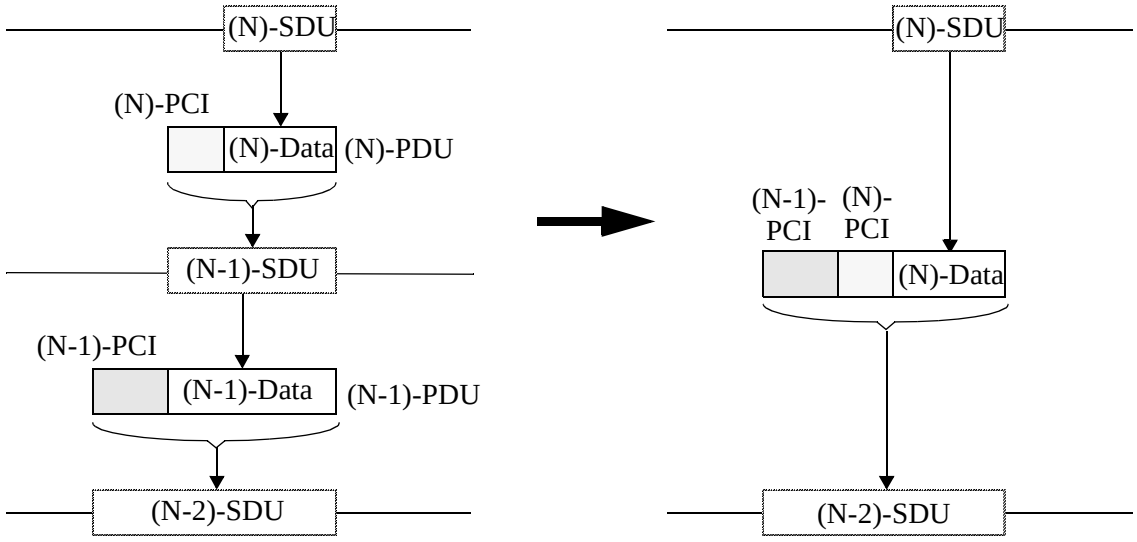


Figure 4.8: Layered sender architecture and integrated sender architecture

genuine demultiplexing functions, it defines similar functions which pose the same problems with respect to layer integration ([Tennenhouse89]). Examples mentioned in [Tennenhouse89] are: coordination of application associations (performed by a MACF), and local 'routing' of PDUs that are received on a single presentation connection that is used by multiple ASEs (this 'routing' function is based on presentation context identifications).

- *reassembly and separation*: reassembly and separation are the reverse functions of segmentation (or fragmentation) and concatenation, respectively. Reassembly is concerned with the reconstruction of a complete SDU from received PDUs, each of which represents a segment of the SDU. Separation deals with the identification of multiple PDUs that were received as a concatenated sequence. The problems with respect to layer integration are similar as with demultiplexing. Reassembly hampers the integration of the layer where reassembly is performed and the higher layer; separation hampers the integration of the layer where separation is performed and the lower layer. Segmenting/reassembly and concatenation/separation are all incorporated in the Session Layer (although session concatenation/separation is restricted to specific SPDUs; in particular, at most one DATA TRANSFER SPDU is sent in a concatenated sequence). The Reliable Transfer Service Element, one of the building blocks of MHS, also contains a segmenting/reassembly function.

Although multiplexing/demultiplexing, segmentation/reassembly, and concatenation/separation are all useful functions, since they allow sharing or optimize the use of data transfer resources, they should not be repeated in many different layers ([Kent87], [Tennenhouse89], [Feldmeier90]). The OSI-RM includes these functions in the Applica-

tion Layer and the Session Layer, and also in some of the lower layers. The purpose of these functions suggest that the most appropriate place is the Transport Layer, since this layer is responsible for optimizing data transfer[1]. In addition, a concatenation and separation function may be included in application protocols if this function is used to reduce the protocol overhead by combining application-specific protocol procedures.

4.6.2 Related work on implementation approaches

Several approaches to achieve efficient implementations of layered protocol architectures have been proposed, including:

- *header prediction* ([Clark89]): the receiving path is made faster by anticipating the PCI contained in PDUs;
- *integrated message processing* ([Clark90]): to the extent possible, protocol manipulation steps at different layers are performed in one integrated processing loop;
- *lazy message evaluation* ([O'Malley91]): the application of protocol functions is delayed until it is necessary to apply them (e.g., because of the results of protocol processing are required by another function), or until it is convenient to apply them (e.g., because the processor is idle);
- *layer bypassing* ([Woodside91]): sending and receiving path include tests on PDUs that indicates whether a fast or normal (layered) path should be followed.

The potential danger of these approaches is that efficiency is achieved at the cost of clearly defined implementation structure. The implementations may be efficient but hard to understand, and thus difficult to maintain. To minimize this danger, efforts should be made to preserve the separation of protocol functions defined in the protocol architecture, while reducing the sequentiality of function applications.

The above mentioned implementation approaches illustrate that layering as a structuring technique for protocol architectures does not necessarily deteriorate efficiency. Whether good efficiency is really possible depends on the functions defined by the layer protocols, not on the principle of layering.

4.6.3 Example of protocol overhead

A protocol architecture prescribes what PDUs are exchanged, and the order in which they are exchanged. Different protocol solutions may be designed that define different PDU exchanges and different PDU contents, but still provide the same service functions.

1. It should be noted, however, that the current OSI Transport Protocol does not include a concatenation/separation function, nor a blocking/deblocking function. The OSI-RM, on the other hand, identifies these functions as proper Transport Layer functions.

Depending on the amount of information that needs to be exchanged, the implementation of one protocol architecture may be more efficient than the implementation of another protocol architecture.

Consider, for example, a distributed application that operates in an OSI environment, but that does not require any of the presentation-specific or session-specific functions, except the functions for connection establishment and termination. In this case, a normal data transfer requires the construction of presentation PDUs (PPDUs) and session PDUs (SPDUs), according to the OSI-ULA, as follows:

- a P-DATA request results in the preparation of a S-DATA User Data parameter. The S-DATA User Data parameter must represent, in the agreed transfer syntaxes, the presentation data values contained in the User Data parameter of the P-DATA request. After the S-DATA User Data parameter has been established, a S-DATA request is issued with this parameter. No presentation PCI, and thus no overhead, is generated in support of the P-DATA function (the TRANSFER DATA PPDU mentioned in the standard is in fact a ‘ghost’ PPDU).
- a S-DATA request results in the preparation of a basic concatenated sequence of a ‘dummy’ GIVE TOKENS SPDU and a DATA TRANSFER SPDU. The GIVE TOKENS SPDU is encoded in at least 2 octets; the encoding of the DATA TRANSFER SPDU also requires an overhead of 2 octets (in both cases due to the inclusion of an identifier field and a length indicator field). After the concatenated sequence has been established, a T-DATA request is issued with the concatenated sequence as User Data parameter.
- at the receiving side the reverse mappings take place.

Figure 4.9 illustrates the send process.

Each time the application wants to transfer an application PDU, 4 redundant octets are also transferred (by the Session Protocol). In addition, an implementation has to perform the concatenation and separation defined by the Session Protocol. In this case, an alternative protocol architecture can easily be conceived that avoids this overhead.

4.7 Conclusion

Evaluation of the OSI-ULA shows that many architectural choices incorporated by the upper layers can be criticized on basis of design quality criteria. One general comment concerns the composition of application protocol layers. There is no architectural justification for restricting ‘dynamic’ composition of application protocol components to the Application Layer. The complexity of the OSI-ULA is partly due to the ‘static’ Presentation Layer and Session Layer. From this we can conclude that the designer of application protocols should not be forced to use a fixed protocol hierarchy, but should be free to

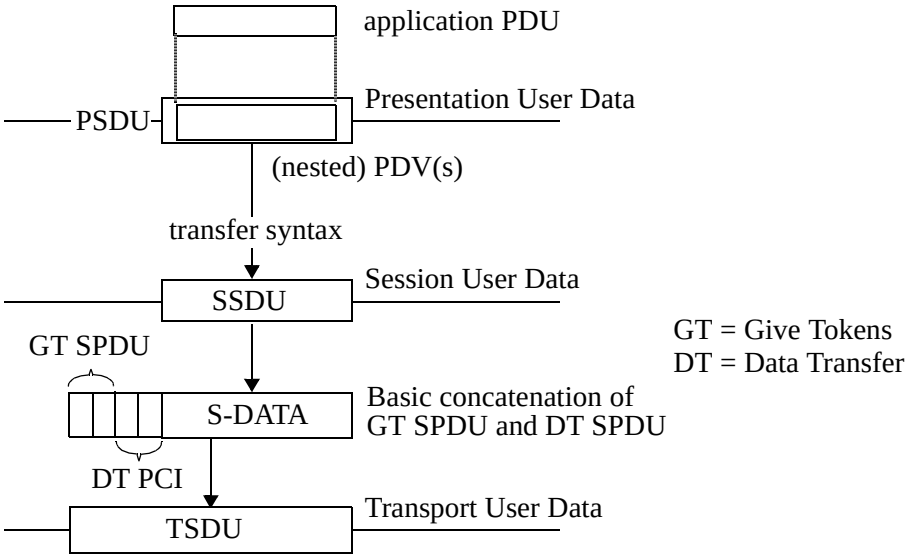


Figure 4.9: Transfer of an application PDU according to the OSI-ULA

determine the most effective composition of application protocols. This implies that fixed protocol layers, i.e. layers that are used to support any distributed application, should not be prescribed. Layering is, however, still useful as a structuring technique when designing application protocols that support specific classes of distributed applications.

Other important conclusions are:

- the definition of transfer syntax constraints on PDUs by the Presentation Layer is violating the orthogonality criterion;
- Segmentation/reassembly and concatenation/separation are not proper application protocol functions. They should be defined by the lower layers;
- the definition of activity synchronization functions by the Session Layer is violating the propriety and generality criteria.

OSI application protocol products have not been very successful so far. This is not only because of the quality deficiencies of the application protocol standards, but also because of ‘poor’ initial implementations and realizations of these standards and because of certain characteristics of the standardization process.

Many quality deficiencies of application protocol standards can be traced back to a misunderstanding of the role of the service concept. In addition, and to some extent as a consequence, some of the Application Layer concepts are poorly defined. This is shown by the evaluation of the OSI-ULM. Unclarity of the service concept hampers the defini-

tion of service decomposition methods, which aim at (step-wise) designing protocols on basis of required services, with proper consideration of design quality criteria. Two such methods are outlined here, after the service concept has been clarified.

A general comment concerning the OSI-ULM is that the relationship between levels of abstraction is not clearly defined. This applies to the service-protocol relationship, as mentioned above, but also to the relationship between application services and distributed information processing, and the relationship between protocol architecture and protocol implementation.

References

[Allan86] Allan, R., Factory communication: MAP promises to pull the pieces together, *Electronic Design*, May 1986, 103-112.

[Caneschi86] Caneschi, F., Hints on the interpretation of the ISO session layer, *Computer Communication Review*, Vol. 16, No. 4, July/August 1986, 34-72.

[Chesson91] Chesson, G., The challenge of workstations to networks, *Computer Networks and ISDN Systems*, 23 (1991) 15-18.

[Clark89] Clark, D., Jacobson, V., Romkey, J., and Salwen, H., An analysis of the TCP processing overhead, *IEEE Communications Magazine*, Vol. 27, No. 6, June 1989, 23-29.

[Clark90] Clark, D.D., and Tennenhouse, D.L., Architectural considerations for a new generation of protocols, *Computer Communication Review*, Vol. 20, No. 4, September 1990, 200-208.

[Feldmeier90] Feldmeijer, D.C., Multiplexing issues in communication system design, *Computer Communication Review*, Vol. 20, No. 4, September 1990, 209-219.

[Ferreira94] Ferreira Pires, L., *Architectural notes: a framework for distributed systems development*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1994.

[IS8831:89] ISO, *Job transfer and manipulation concepts and services*, International Standard ISO 8831, 1989.

[IS8832:89] ISO, *Specification of the basic class protocol for job transfer and manipulation*, International Standard ISO 8832, 1989.

[IS10021:92] ISO, *Message Handling Systems*, Part 1: System and service overview, International Standard ISO 10021-1, 1992.

[IS10026:92] ISO, *Distributed transaction processing*, Part 1: OSI TP model, Part 2: OSI TP service, Part 3: OSI TP protocol, International Standard ISO 10026, 1992.

[IS10731:91] ISO, *Conventions for the definition of OSI services*, Committee Draft Standard ISO 10731, 1991.

[IS10746:93] ISO, *Basic reference model of open distributed processing*, Part 1: overview, Draft International Standard ISO 10746, 1993.

[Kalin92] Kalin, T., and Barber, D., Has the OSI opportunity been fully realised?, *Computer Networks and ISDN Systems*, 23 (1992) 227-239.

[Kent87] Kent, C.A., and Mogul, J.C., Fragmentation considered harmful, *Computer Communication Review*, Vol. 17, No. 5, August 1987, 390-401.

[Khendek89] Khendek, F., Bochmann, G. von, and Kant, C., New results on deriving protocol specifications from service specifications, *Computer Communication Review*, Vol. 19, No. 4, September 1989, 136-145.

[Kündig91] Kündig, A., From computer communication to computer supported co-operative work, In: *5th Annual European Computer Conference (CompEuro '91)*, IEEE Computer Society Press, 1991, 184-190.

[LaPorta91] La Porta, T.F., and Schwartz, M., Architectures, features, and implementation of high-speed transport protocols, *IEEE Network Magazine*, May 1991, 14-22.

[Mantelman89] Mantelman, L., The birth of OSI-TP: a new way to link OLTP networks, *Data Communications International*, September 1989, 76-85.

[Ngoh89] Ngoh, L.H., and Hopkins, T.P., Transport protocol requirements for distributed multimedia information systems, *The Computer Journal*, Vol. 32, No. 3, 1989, 252-261.

[O'Malley91] O'Malley, S.W., and Peterson, L.L., A highly layered architecture for high-speed networks, In: *Protocols for High-Speed Networks, II*, Johnson, M.J. (editor), Elsevier Science Publishers B.V. (North-Holland), 1991, 141-156.

[Rauch86a] Rauch-Hindin, W., OSI standards spur product profusion, *Mini-Micro Systems*, June 1986, 67-82.

[Rauch86b] Rauch-Hindin, W., Upper level OSI protocols near completion, *Mini-Micro Systems*, July 1986, 53-66.

[Rodden92] Rodden, T., and Blair, G.S., Distributed systems support for computer supported cooperative work, *Computer Communications*, Vol. 15, No. 8, October 1992, 527-538.

[Saleh91] Saleh, K., and Probert, R., Automatic synthesis of protocol specifications from service specifications, In: *Proceedings of the 10th International Phoenix Conference on Computers and Communications*, IEEE Computer Society Press, 1991.

[Schot92] Schot, J., *The role of architectural semantics in the formal approach towards distributed systems design*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1992.

[Schürmann90] Schürmann, G., and Holzmann-Kaiser, U., Distributed multimedia information handling and processing, *IEEE Network Magazine*, November 1990, 23-31.

[Shepherd90] Shepherd, D., and Salmony, M., Extending OSI to support synchronization required by multimedia applications, *Computer Communications*, Vol. 13, No. 13, September 1990, 399-406.

[Sinderen92] Sinderen, M. van, Ferreira Pires, L., and Vissers, C.A., Protocol design and implementation using formal methods, *The Computer Journal*, Vol. 35, No. 5, 1992, 478-491.

[Solvie92] Solvie, G., A flexible open systems architecture satisfying modern communication requirements, In: *International Workshop on Advanced Communications and Applications for High Speed Networks*, Munich, Germany, March 16-19, 1992, 383-392.

[Strauss87] Strauss, P, Rozenberg, R., and Borsook, P., OSI throughput performance: breakthrough or bottleneck?, *Data Communications*, May 1987, 53-56.

[Tennenhouse89] Tennenhouse, D.L., Layered multiplexing considered harmful, In: *Protocols for High-Speed Networks*, Rudin, H., and Williamson, R. (editors), Elsevier Science Publishers B.V. (North-Holland), 1989, 143-148.

[Vissers77] Vissers, C.A., *Interface - definition, design, and description of the relation of digital system parts*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1977.

[Vissers85] Vissers, C.A., and Logrippo, L., On the importance of the service concept in the design of data communications protocols, In: *Protocol Specification, Testing, and Verification, V*, Diaz, M. (editor), Elsevier Science Publishers (North-Holland), 1986, 3-17.

[Woodside91] Woodside, C.M., Ravindran, K., and Franks, R.G., The protocol bypass concept for high speed OSI data transfer, In: *Protocols for High-Speed Networks, II*, Johnson, M.J. (editor), Elsevier Science Publishers B.V. (North-Holland), 1991, 107-122.

[Zimmermann83] Zimmermann, H., On protocol engineering, In: *Information Processing '83*, Mason, R.E.A. (editor), Elsevier Science Publishers B.V. (North-Holland), 1983, 283-292.

Chapter 5

Design Framework

This chapter presents a general framework for the design of application protocols. The framework identifies and defines two dimensions in which design concerns can be structured. One dimension distinguishes between two related domains of specification, viz. the behaviour domain and the entity domain. Behaviour is defined in the behaviour domain, while the assignment of behaviour to functional entities is defined in the entity domain. The other dimension distinguishes between relative abstraction levels at which functional entities can be considered. Based on these abstraction levels, an application protocol design trajectory is defined, consisting of a sequence of design steps between design milestones, where each design step addresses a specific set of design concerns.

The purpose of the framework is twofold. First, it provides a means for handling the complexity of the application protocol design process by indicating how different design concerns can be separated. Second, by recognizing different design concerns, it also allows the identification of elementary design, or architectural, concepts that are necessary for the representation of behaviour and functional entities along the design trajectory.

The structure of this chapter is as follows: section 1 discusses the behaviour domain and the entity domain; section 2 discusses three abstraction levels at which any functional entity can be represented; section 3 presents the milestones of the application protocol design trajectory; sections 4 illustrates the use of the application protocol design trajectory with a simple design example; and section 5 presents the conclusions of this chapter.

5.1 Domains of distributed system specification

In most approaches towards the design of distributed systems a distinction is made between the concept of entity and the concept of behaviour (see [Gotzhein93], for example). An entity is thought of as a logical or physical component of a system, characterized by its behaviour. Hence, an entity is seen as a carrier of behaviour. Having defined this relationship, it is common practice to concentrate either on the definition of entities or on the definition of behaviour (in [Kramer90], these approaches are called the constructive approach and the specification-driven approach, respectively). According to the first

approach, a system is composed of entities, where each entity has some associated behaviour. In the second approach the behaviour of the system is composed of behaviour components, where each behaviour component may be allocated to an entity.

We argue that structuring in terms of behaviour components and in terms of functional entities serve different design objectives, which should not be confused. Consequently, attention should be drawn to both of them. This section discusses two domains of system specification, viz. the behaviour domain and the entity domain, which allow to separate the concerns of behaviour definition and entity definition.

5.1.1 Specification concerns and objectives

Behaviour and entity structure

From a designer's point of view, a system is characterized by its behaviour and its entity structure. The entity structure is of concern if parts of the system are implemented under different implementation authorities, possibly in different places and at different times. It defines the boundaries which must be observed by the implementor who only implements a part of the system.

The specification of the entity structure of a system allows teams of implementors to work in isolation, or at least to minimize mutual contact. Any component in this structure, including the system itself, is called a functional entity, or *entity* for short. An entity is an object of implementation which plays a defined role in the environment in which it is embedded. This role is determined by the behaviour assigned to the entity, which defines the role, or responsibility, of the entity in *interactions* between the entity and its environment. Interactions occur at defined logical locations, called *interaction points*.

The behaviour of an entity may also define internal units of activity, called *actions*. Actions are not be directly visible to the entity's environment[1]. Since an entity can again be structured as a composition of component entities, it is useful during design to be able to associate logical locations with actions. These locations are called *action points*. If an entity is decomposed, some actions are transformed into interactions and associated action points are transformed into interaction points.

The specification of an entity structure implies the specification of a behaviour structure. It is sometimes desirable to structure behaviour, beyond the structure that corresponds to an entity structure. Such a structure serves to obtain conciseness of the behaviour specification, or to improve its comprehensibility. As such, behaviour structures may have considerable impact on the quality of derived implementations.

1. The environment should not have access to arbitrary results of the entity's behaviour, since then the role of the entity is no longer defined. This is sometimes referred to as encapsulation of information.

External and internal behaviour

The behaviour of an entity may be defined at different abstraction levels. A broad distinction can be made between the definition of behaviour in terms of (contributions to) interactions with the environment, and the definition of behaviour in terms of both interactions and internal actions. The former is called external behaviour, the latter internal behaviour. The *external* behaviour of an entity defines *what* is the role of the entity in its environment, independent of how the entity performs this role. *Internal* behaviour of an entity defines *how* an entity performs its role.

In many approaches external behaviour of an entity is referred to as *observable* behaviour, since it defines the behaviour as observed by the environment (i.e. the users) of the entity. From the designer's or implementor's point of view, however, all behaviour is 'observable' since it is specified by the designer and interpreted by the implementor. For this reason, we will only use the term external behaviour.

The external behaviour of an entity is often used as the starting point for deriving implementations of that entity. It can be considered as the *architecture* of that entity since it defines an abstraction of all possible implementations that are capable of performing a required role. In contrast, the internal behaviour of the entity defines a possible implementation (and an abstraction of a subset of more concrete implementations). Figure 5.1 depicts the relationship between an architecture and its implementations.

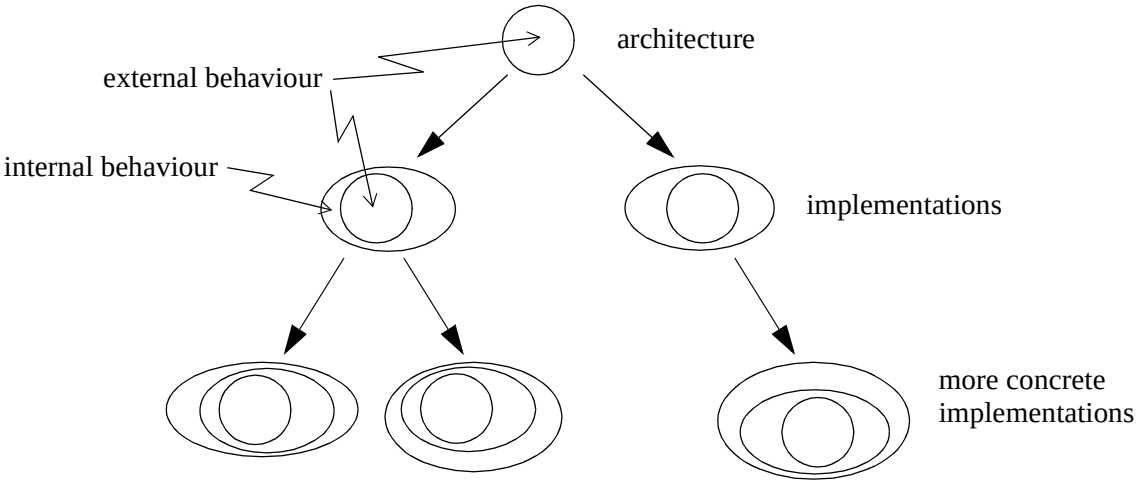


Figure 5.1: An architecture as the common abstraction of various implementations

Entities can be identified in a (top-down) design process as the result of refining and structuring the internal behaviour of a system. It is also possible to use pre-defined general purpose entities in a (bottom-up) design process, in which case the internal behaviour of a system is composed from the external behaviours of its component entities.

Prescription versus description

A specification of an entity or behaviour can either be prescriptive or descriptive. A prescriptive specification, or *prescription*, defines characteristics that must be implemented. Hence, a prescription is the proper specification for implementors that must built conforming implementations (the term requirements specification is therefore also used [Gotzhein93]). A descriptive specification, or *description*, on the other hand, does not contain explicit statements on the characteristics that must be implemented. It explains an architecture, possibly in terms of *a possible* implementation, rather than prescribing an implementation.

Descriptions are, however, often used in the design process instead of prescriptions if it is easier to define an architecture in terms of a possible implementation than in terms of abstract implementation requirements. Another reason for using descriptions is that descriptions are sometimes easier to understand than prescriptions. In both cases, this generally depends on the specification language that is used to express specifications. For example, most present formal description techniques (FDTs) are too much implementation-oriented to express prescriptions at the required level of abstraction (see [Ferreira93] and [Vissers93], for example). These languages force the designer to take design decisions which unnecessarily limit the freedom of the implementor.

Figure 5.2 illustrates the difference of a prescription and a description: it shows an architecture as a prescription for possible implementations, and a description of this architecture. The description is formulated in terms of implementation solutions which are incorporated by a valid implementation 1 but not by another, also valid implementation 2.

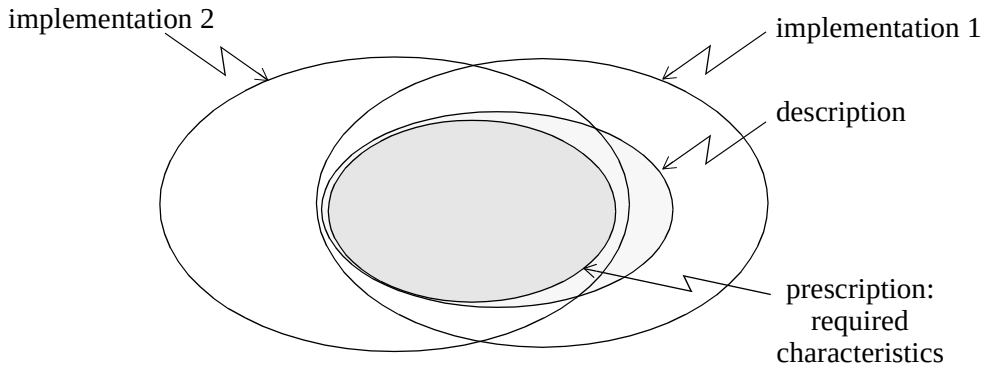


Figure 5.2: A prescription and description of an architecture

5.1.2 Behaviour domain

In the behaviour domain, actions and interactions, and the relations between them, can be used to represent behaviour. Behaviours though, especially complex ones, need to be well structured in order to make them intelligible. Structuring behaviour may also serve to

prepare the identification and definition of entities, i.e. the assignment of behaviour to entities.

Figure 5.3 depicts an example of this use of behaviour structuring. Given an arbitrary entity *E*, first an internal behaviour *B* is defined, with interactions at interaction points *ip1* and *ip2*, and actions at action points *ap1*, *ap2* and *ap3*. This behaviour is subsequently structured as a composition of component behaviours *B1*, *B2*, and *B3*. The component behaviours are related by sharing actions: *B1* and *B2* share the actions at action point *ap1*; *B1* and *B3* share the actions at action point *ap2*; and *B2* and *B3* share actions the actions at action point *ap3*. Each component behaviour defines its role in these actions (hence, shared actions have a distributed representation). Next, the component behaviours are assigned to different entities: *B1* is assigned to *E1*; *B2* is assigned to *E2*; and *B3* is assigned to *E3*. The behaviour assignment implies a transformation of (distributed) actions into interactions, and of action points into interaction points: *a1*, *a2*, and *a3* are replaced by the interaction points *ip3*, *ip4*, and *ip5*, respectively.

5.1.3 Entity domain

In the entity domain, aspects related to the entity structure of a system are considered. These aspects involve the identification of entities, and their interconnection. An entity is delimited by interaction points and contains action points. Interaction points are shared with other entities, forming the common means of interaction between these entities. Each action point, however, can belong to only one entity.

Figure 5.4 illustrates the representation of an entity according to an integrated system perspective and according to a distributed system perspective. As opposed to the integrated system perspective, the distributed system perspective considers a decomposition of the entity in terms of interconnected component entities (these perspective are further discussed in section 5.2).

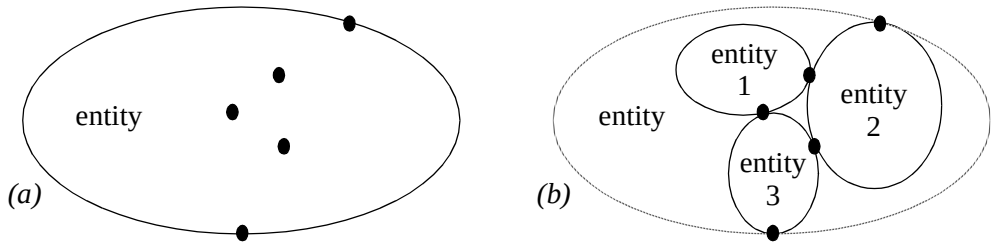


Figure 5.4: Representation of an entity (a) according to an integrated system perspective, and (b) according to a distributed system perspective

The identification of an entity represents a decision on the structure of the system. This structure should be preserved during the design process since it designates possible implementation boundaries. Carrying out the implementation of an entity in isolation requires

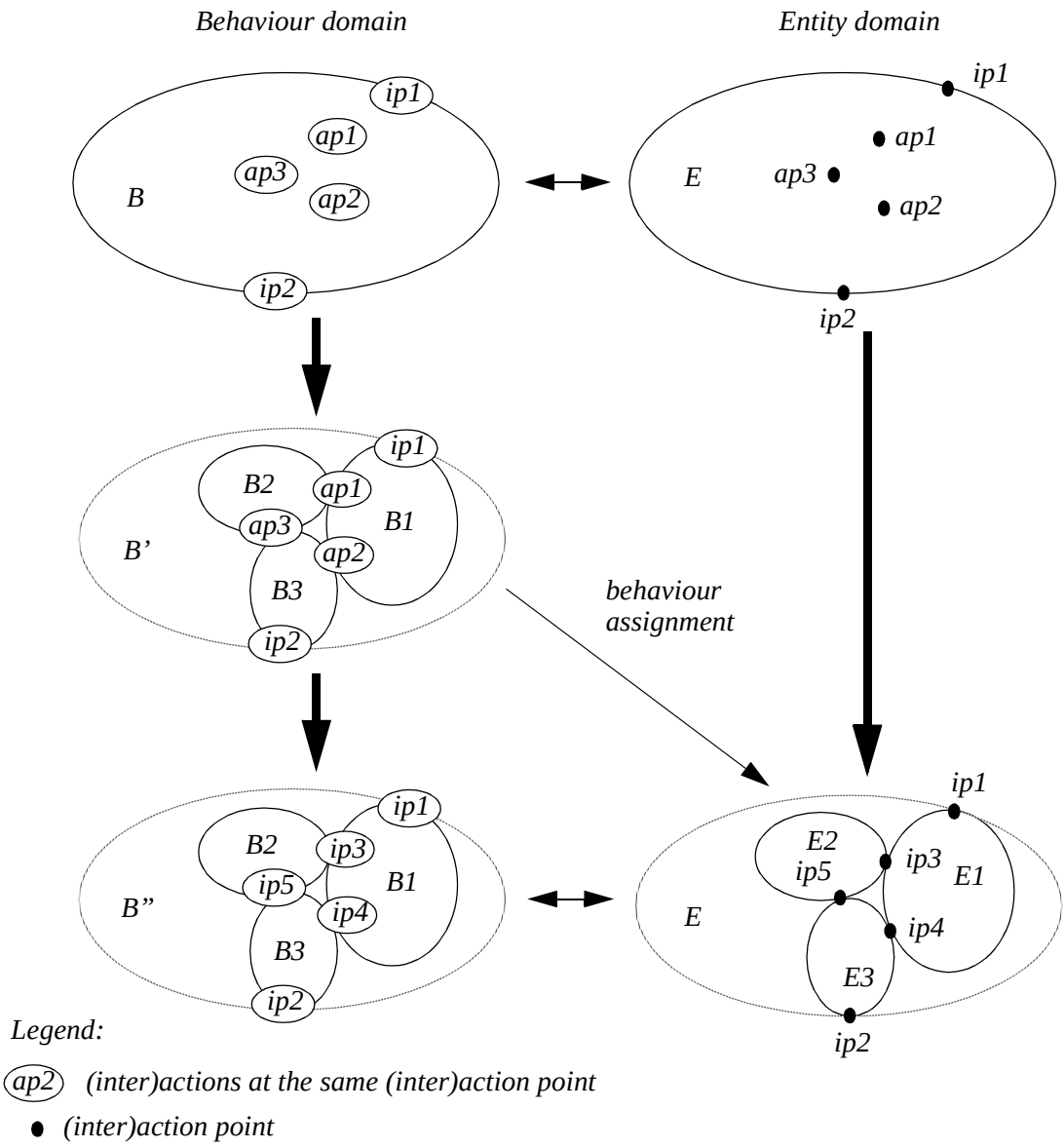


Figure 5.3: Preparing entity definition by way of behaviour refinement and structuring

that the final design defines the external behaviour of the entity at a concrete level (i.e, with concrete interfaces).

As illustrated by Figure 5.3, there is a mapping from the behaviour domain to the entity domain, such that behaviours are assigned to entities, and actions and interactions are assigned to action points and interaction points. The composition of behaviours assigned to entities has to be compatible to the composition of these entities, i.e. the resulting behaviour should be the behaviour assigned to the composite entity. Figure 5.5 depicts the aspects considered by the entity domain and by the behaviour domain.

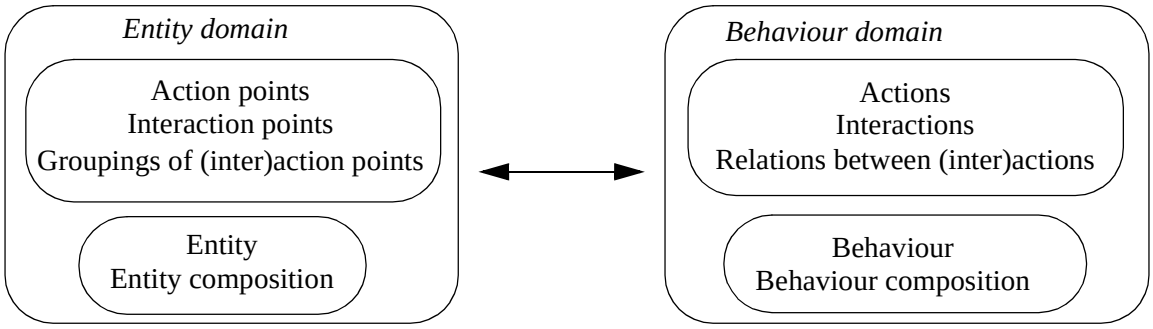


Figure 5.5: Aspects considered by the entity domain and by the behaviour domain

5.2 Abstraction levels in distributed system design

In a design process, abstractions of a system serve to focus on certain aspects of the system and to disregard other aspects. Which abstraction is used, i.e. which aspects are considered and which are disregarded, is determined by the needs of the designer. If multiple abstractions are used in a design process, it should be possible to verify the consistency of system representations based on these abstractions. This requires that relationships are defined between the abstractions.

This section proposes the use of abstractions that are hierarchically related. Such abstractions are called *abstraction levels*. Separate attention is paid to the use of the proposed abstraction levels in interaction system design.

5.2.1 Purpose of abstraction levels

Two abstraction levels are related to each other by the fact that at the lower abstraction level additional aspects of a system are considered compared to the aspects considered at the higher abstraction level. The characteristics of the system represented at the higher abstraction level (in the higher level design) must be preserved in representation at the lower abstraction level (in the lower level design).

The transformation of a given design, at some abstraction level, into a lower level design constitutes a design step. Hence, the design process can be represented by a sequence of design steps, forming a design trajectory, where each design step is concerned with the transformation of a given design into a lower level design. In this way, a set of abstraction levels can be effectively used to support *step-wise design*.

Figure 5.6 illustrates the use of abstraction levels in a design process.

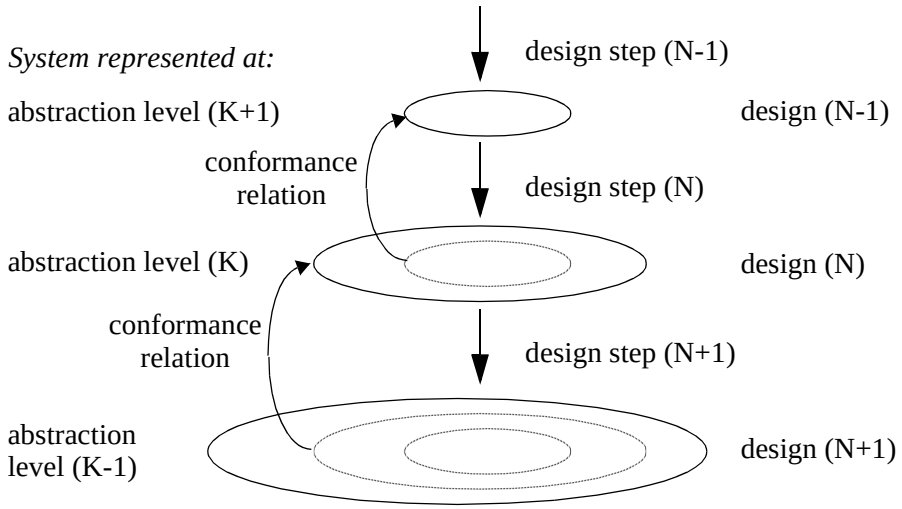


Figure 5.6: Abstraction levels in a design process

Step-wise design, based on the definition of abstraction levels, is attractive when designing complex systems for a number of reasons:

- *separation of design concerns*: each design step deals with specific design concerns, related to the additional aspects of the system that are considered. It is thus possible to systematically deal with all requirements and constraints involved in the design of complex systems. The design steps of the design trajectory correspond to a separation of these design concerns.
- *verifiable intermediate design results*: each design step results in a design at a lower abstraction level, which can be verified with respect to the higher level design based on the defined relationship between the abstraction levels (the lower level design should conform to the higher level design with respect to the aspects considered at the higher level; this is indicated by the conformance relation in Figure 5.6). Inconsistencies between user requirements and design decisions can thus be detected and corrected at the earliest possible stage in the design process.
- *design methods and tools*: the defined relationship between abstraction levels facilitates the development of design methods, possibly supported by design tools (e.g., for automated verification of intermediate design results).

5.2.2 Distributed system perspective

A commonly used abstraction of a distributed system is one that represents the system as a composition of interconnected functional entities. Since the system is embedded in an environment, some or all of the component entities will also be connected to the environment. This abstraction will be called the *distributed system perspective*. Figure 5.7 depicts a system represented according to the distributed system perspective.

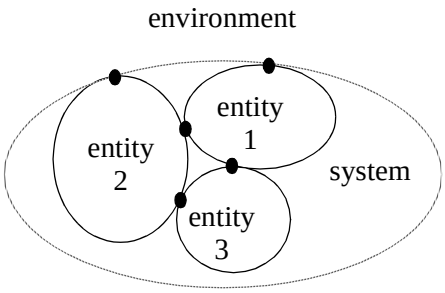


Figure 5.7: Example of distributed system perspective

The figure shows the representation of the system in the entity domain, which also most clearly illustrates the purpose of the distributed system perspective: to define the composition of a system from subsystems. Interconnections of functional entities are represented by interaction points and indicate that these entities are capable of interacting. Also the system and its environment can be considered as (composite) functional entities. The functional entities from which the system is composed can be viewed at different abstraction levels. However, the most abstract view according to the distributed system perspective is one that only defines the external behaviours of the identified entities (that are used to compose the system). At lower abstraction levels, the internal behaviours of these entities can be considered, or a further entity decomposition.

5.2.3 Integrated system perspective

The distributed system perspective defines an internal organization of a system, and thus a possible implementation. We can abstract from this specific internal organization and the many possible alternative structures of the system by not considering its composition from functional entities. This abstraction, a higher abstraction level compared to the distributed system perspective, will be called the *integrated system perspective*.

A system is represented by a single functional entity when using the integrated system perspective. It shares interaction points with its environment, indicating that at these points interactions between the system and its environment are possible. Figure 5.8 depicts a system represented according to the integrated system perspective.

The most abstract view according to the integrated system perspective is one that defines the external behaviour of the system. Note, however, that the definition of internal behaviour of the system does not necessarily represent a decomposition of the system, and can therefore also be considered in the integrated system perspective (see Figure 5.4, for example). Even structured internal behaviour can be considered in the integrated system perspective as long as the component behaviours are not assigned to component entities. Consequently, the integrated system perspective can be viewed at different abstraction

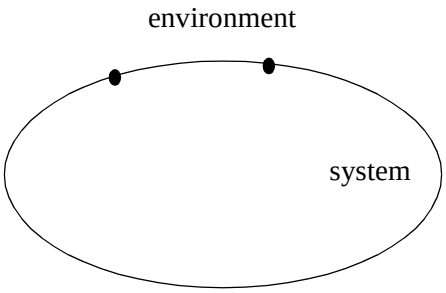


Figure 5.8: Example of integrated system perspective

levels, corresponding to different behaviour views. Each of these levels is a higher abstraction level than the distributed system perspective.

The behaviour of the system as defined using the distributed system perspective should conform to the behaviour of the integrated system perspective. For example, the composition of the behaviours of component entities has to correspond to the external behaviour of the system represented according to the integrated system perspective. The requirements that apply to correct behaviour refinement have to be defined in the behaviour domain.

5.2.4 Interaction system perspective

The integrated system perspective defines the behaviour assigned to a system and does not consider the behaviour assigned to the environment of the system. The separation of system behaviour and environment behaviour implies a possible distribution of responsibility with respect to their joint behaviour. We can abstract from this specific distribution and the many possible alternative distributions over the system and its environment by not considering the system and the environment as separate entities. This implies that the distribution of responsibilities in interactions between the system and its environment is no longer considered, nor who is responsible for imposing constraints on the relationships between the interactions. This abstraction, a higher abstraction level compared to the integrated system perspective, will be called the *interaction system perspective* of the system and its environment.

The interaction system perspective does not include interaction points, but only action points, since no separate functional entities are distinguished. The actions that can occur at these action points are in fact *integrated* interactions between the system and its environment. Figure 5.9 depicts a system and its environment represented according to the interaction system perspective.

The definition of a system according to the integrated system perspective can be derived from the interaction system perspective of the system and its environment by identifying and defining individual responsibilities of the system and its environment. It is possible to determine a *partial distribution of responsibilities*. For example, a partial

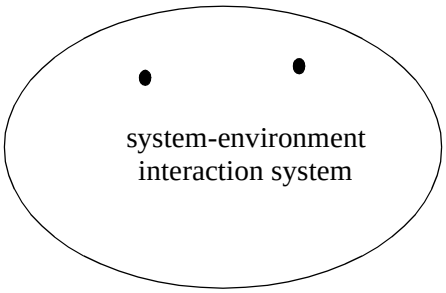


Figure 5.9: Example of interaction system perspective

distribution of responsibilities over the system and its environment is accomplished if it is established what role the system plays in constraining the relationship between interactions at different interaction points. In this case, it is not yet determined who imposes what constraints on the relationship between interactions at the same interaction point. The definition of such a partial distribution already constitutes an *integrated system perspective*. (An intermediate step is to define first a behaviour structure in the interaction system perspective.)

The definition of a partial distribution of responsibilities plays an important role in protocol standards. Here, interactions at different interaction points fall under different implementation authorities, while interactions at the same interaction point *may* fall under a single implementation authority. For this reason, constraints on the relationship between interactions (service primitives) at the same interaction point (service access point) are not distributed over the system (service provider) and its environment (service user). An integrated view of these constraints is instead defined by an abstract interface (see also Chapter 4, OSI Upper Layer Architecture and Model: Evaluation).

5.2.5 Integrated and distributed interaction system perspective

The distributed system perspective, discussed in section 5.2.2, considers interactions between functional entities which take place at shared interaction points. However, in the design of interaction systems, of protocol systems in particular, we are interested in interactions between functional entities that are geographically distributed. Protocol entities are not directly interconnected, but indirectly, via another entity, the lower level service provider. Consequently, they do not share interaction points and their interactions are distributed. Interactions between protocol entities are defined by the individual contributions (protocol actions) made by the protocol entities, and the relationship to interactions (service primitives) between the protocol entities and the lower level service provider.

To represent protocol actions, the internal behaviour of protocol entities has to be considered. We will denote the distributed system perspective in which protocol actions are considered with the term *distributed interaction system perspective*. The interaction system perspective will be referred to as the *integrated* interaction system perspective if

we want to distinguish it from the distributed interaction system perspective. Figure 5.10 depicts a system represented according to the distributed interaction system perspective.

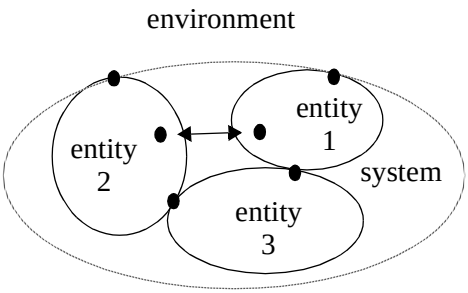


Figure 5.10: Example of distributed interaction system perspective

The most abstract view according to the distributed interaction system perspective is one that defines actions corresponding to protocol actions, i.e. when possible implementations of protocol actions are not considered. This view corresponds to a protocol architecture.

Figure 5.11 shows the correspondence between the identified abstraction levels and the OSI concepts of service, service provider and protocol (see also Chapter 4, OSI Upper Layer Architecture and Model: Evaluation).

Abstraction level	*OSI concept*
integrated interaction system perspective	service
integrated system perspective	service provider
distributed interaction system perspective	protocol (protocol entities and lower level service provider)

Figure 5.11: Correspondence between identified abstraction levels and OSI concepts

5.3 Application protocol design trajectory

A design process is not normally carried out as a linear sequence of design steps. A typical design process involves both top-down and bottom-up design decisions, and involves many design cycles ([Boehm88], [Bogaards90]) in order to correct errors, to improve design results, or to add functionality. Nevertheless, a design trajectory is useful as a simplified model of the design process for the reasons mentioned in subsection 5.2.1. Intermediate design results identified in the design trajectory as designs at specific abstraction levels can be used as milestones in the design process. Milestones play an important role as synchronization points in the design process, even if the design is not ‘linear’ or strictly top-down. Dependent on the complexity of the system under design,

and on the starting point that is taken for the design process, a subset of the potential milestones will be selected.

This section presents an application protocol design trajectory based on the abstraction levels identified in the previous section. The design trajectory covers a wide abstraction spectrum, so as to allow application protocol design to start from requirements generated by a computer system environment, and to establish a relationship between distributed processing and application protocols. First different areas of concern are identified in the application protocol design trajectory, followed by a presentation of the milestones in each of these areas.

5.3.1 Milestones in different areas of concern

Any system can be represented at the abstraction levels presented in section 5.2. Hence, the functional entities defined in the distributed system perspective (or the lower level service provider in the distributed interaction system perspective) of an arbitrary system can again be represented at each of these abstraction levels. The abstraction levels can thus be used repeatedly in a design process.

Here, we will consider the application of the abstraction levels in three related areas: (1) to application protocols, where the distributed interaction system perspective is used to represent an application protocol architecture; (2) to distributed processing applications, where the distributed system perspective is used to represent a composition of information processing entities; and (3) to computer system environments, where the distributed system perspective is used to represent a composition of enterprise entities and computer system applications. These areas are related by the fact that computer systems applications may require distributed processing or information transfer, while information processing entities may be implemented using application protocols.

5.3.2 Application protocol architecture

An application protocol architecture is the lowest abstraction level we want to consider. Hence, this abstraction level coincides with the distributed interaction system perspective of an application service provider. The following milestones are distinguished:

- *required application service*: this is the integrated interaction system perspective of the application service provider and its user environment.
- *application service provider*: three behaviour views are distinguished of the integrated system perspective. Each of these views preserves the *abstract interfaces* (i.e., the abstract interfaces remain a shared responsibility of the provider and its environment) defined in the required service:
  - *external behaviour*;

  - *internal behaviour*, with actions representing a combination of a protocol action and a lower level service primitive (i.e., geographical distribution is considered, but not hierarchical decomposition); and
  - *internal behaviour*, with actions representing protocol actions and lower level service primitives.
- *application protocol architecture*: the distributed interaction system perspective of the application service provider identifies a data transfer service provider and application protocol entities on top of this provider. The application protocol entities share *abstract interfaces* with their environment, i.e. with the required service users and with the data transfer service provider.

Figure 5.12 depicts the resulting design trajectory. The abstraction levels can be repeatedly used if, instead of a data transfer service provider, a lower level application service provider is defined in the distributed interaction system perspective. In this way, the design method for application layer protocol design discussed in Chapter 4 (OSI Upper Layer Architecture and Model: Evaluation) can also be supported. (In Chapter 6, Design Model, behaviour refinements will be presented that can be used in the design steps of this trajectory. Repeated use of the abstraction levels, and repeated application of the behaviour refinement methods will be discussed in Chapter 7, Application Protocol Reference Architecture).

5.3.3 Distributed processing architecture

In Chapter 1 (Introduction), we mentioned that there are two extreme approaches to the design of distributed systems: either the design focuses on system parts or it focuses on interaction systems. The lower part of the application protocol design trajectory is concerned with interaction system design. We now present the middle part, which is concerned with the design of system parts. The following milestones are distinguished:

- *required application service*: this is the interaction system perspective of an application service provider and its user environment.
- *application service provider*: this is the integrated system perspective of the application service provider, of which two behaviour views are distinguished. Each of these views preserves the *abstract interfaces* defined in the required service:
  - *external behaviour*; and
  - *internal behaviour*, with actions that are integrated interactions between information processing entities.
- *distributed processing architecture*: this is the distributed system perspective of the application service provider, which represents a composition of information processing entities and possibly information transfer entities. These entities share abstract interfaces with their environment, i.e. with the required service users and/or with some of the other entities.

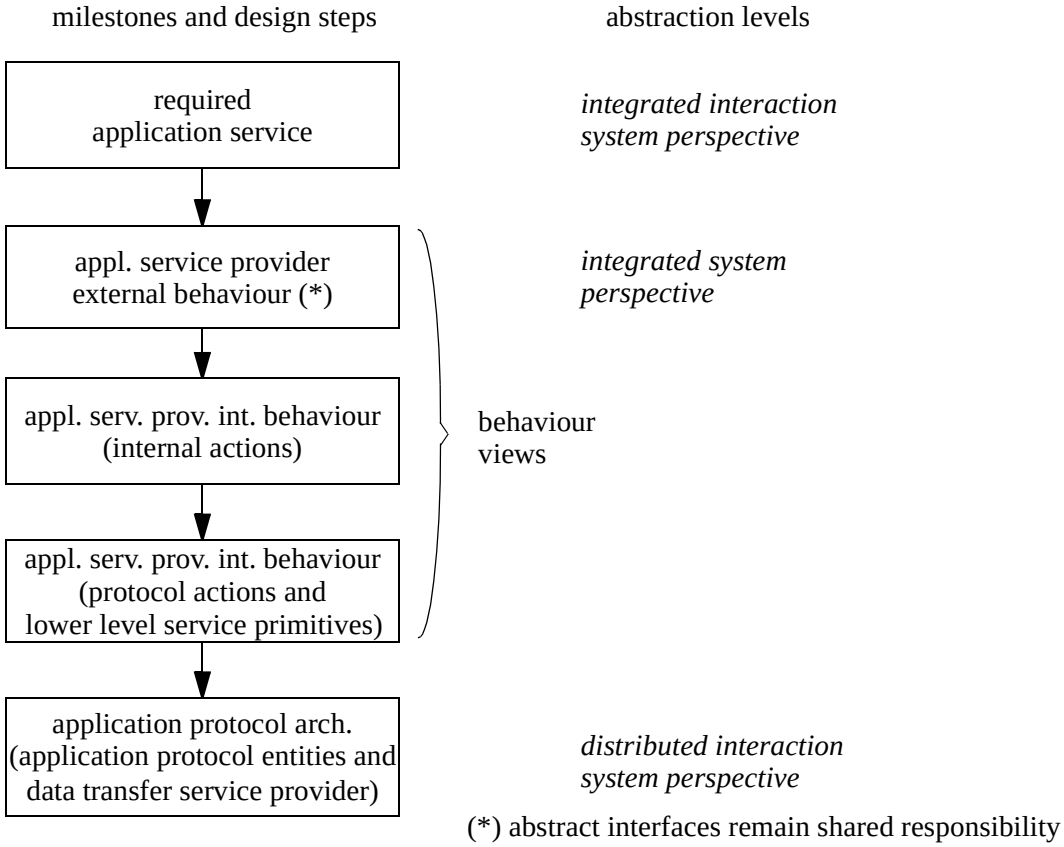


Figure 5.12: Lower part of application protocol design trajectory

Figure 5.13 depicts the resulting design trajectory. The difference with the lower part of the application protocol design trajectory lies in the second and third design step, leading to the application service provider internal behaviour and the distributed processing architecture, respectively. The actions introduced in the second step are integrated interactions. In the third step it is determined which entities are involved in these integrated interactions. The integrated interactions are then grouped in abstract interfaces that are shared by specific entities.

The lower and middle part of the application protocol design trajectory can be roughly characterized by saying that the lower part supports the development of OSI standards and the middle part supports the development of ODP standards[1] (see [Bowen91], for

1. This is also testified by the fact that ‘openness’ in OSI refers to the use of common protocol standards, and ‘openness’ in ODP refers to the use of common interface (operational, stream, etc.) standards. Further support of the design trajectory to the development of ODP standards would require an additional step during which abstract interfaces are refined into more concrete interfaces.

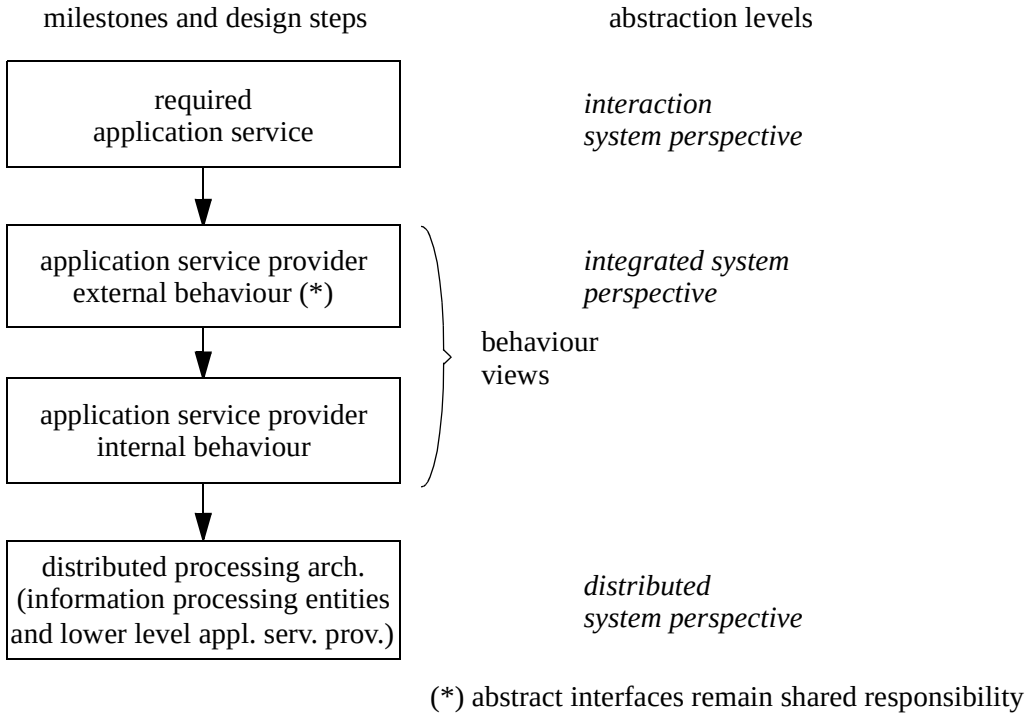


Figure 5.13: Middle part of application protocol design trajectory

example). The middle part of the application protocol design trajectory can therefore also be seen as an alternative to the viewpoints approach adopted by ODP ([Sinderen95]).

The abstraction levels of the middle part of the design trajectory can be repeatedly used if (some of) the information processing entities defined in the distributed system perspective constitute lower level application service providers that will again be implemented by a distributed processing architecture. Here, we assume that information processing entities either fall under a single implementation authority or, if this is not the case, constitute an application service provider that is refined following the design steps of the lower part of the application protocol design trajectory. Information transfer entities are always refined following the design steps of the lower part of the design trajectory.

5.3.4 Distributed enterprise architecture

In general, the design of a computer system application cannot be carried out without also considering the user environment of the application. The interaction system perspective of the computer system application and its environment is therefore at least necessary to satisfy this requirement. Often, however, even this perspective cannot be defined without taking initial design decisions that also affect other parts of the computer system environment. We introduce here the term *enterprise system* to denote a system that integrates a (distributed) computer system and part of its environment[1]. For example, an enterprise

system can correspond to a business organization. The task of the application designer is to introduce computer system applications that supports some of the activities of the enterprise system.

A naive view on the design process is that an existing internal structure of the enterprise system can be taken as the starting point for design, and that some of the enterprise activities in this structure can be partially automatized, independently of other activities. This view is naive because the existing structure is based on the performance of present enterprise activities. Partially automatizing some activities, while basically preserving the existing work organization, may introduce a mismatch of performances and thus may not lead to expected performance improvements of the enterprise system as a whole. Nonetheless, insufficient insight in or account of the relationship between enterprise activities and the impact of the computer system applications on these relationships has often led to ineffective integration of these applications in enterprise systems. Many computer supported cooperative work (CSCW) applications have failed for this reason (see [Bannon91] and [Markus90], for example).

The ‘safe’ starting point for the design of computer systems applications that have to support activities of an enterprise system is therefore an enterprise service. An *enterprise service* is an interaction system perspective of an enterprise system (enterprise service provider) and its environment. A decomposition of the enterprise service allows a restructuring of the enterprise system (for business organizations this can be considered a method for business process redesign) and the identification of automated support by computer system applications. The proposed abstraction levels can be used as milestones in this design process. The milestones are, apart from the enterprise service: *enterprise service provider* (integrated system perspective), *enterprise service provider internal behaviour*, and *distributed enterprise architecture* (distributed system architecture).

The distributed enterprise architecture represents a composition of enterprise entities and computer system applications. Here, we assume that the computer system applications are distributed processing applications. The complete application protocol design trajectory then consists of the milestones as depicted in Figure 5.14. The application service (provider) in the last part of the design trajectory is denoted in the figure as *component* application service (provider) to distinguish it from the application service provider in the middle part. (The middle part of the design trajectory can be skipped if the distributed enterprise architecture identifies information transfer applications.)

1. Actually, an enterprise system can be any undertaking involving human beings and does not necessarily have to include computer systems. The assumption that computer systems will be used is only made to establish the relation with application protocol design (cf. the enterprise viewpoint of ODP).

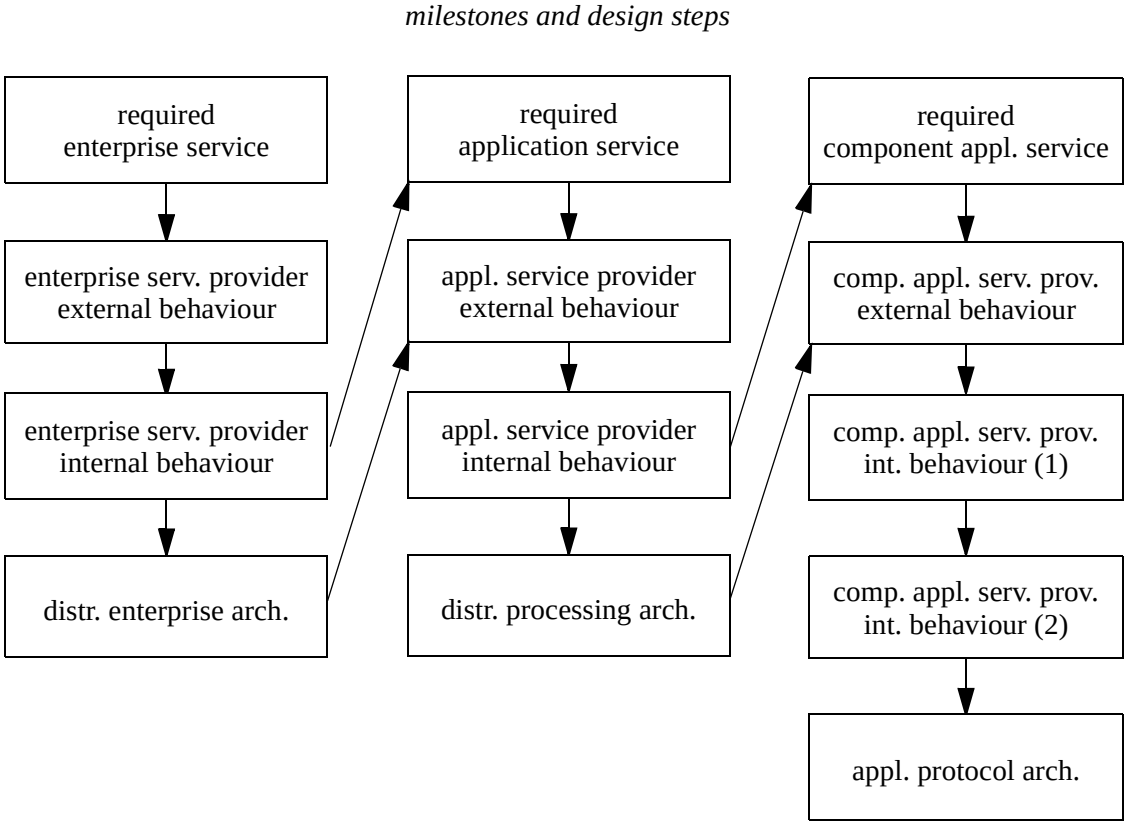


Figure 5.14: Complete application protocol design trajectory

5.4 Example

This section illustrates the use of the lower part of the application protocol design trajectory with a design example, based on the question-answer service that was first presented in [Vissers88].

5.4.1 Required question-answer service

The purpose of an instance of the question-answer application is to establish a question and a corresponding answer. If we do not want to consider the possible geographical distribution of the system and its environment, this can be represented according to the interaction system perspective by a single action. On the other hand, if we anticipate that the question and the corresponding answer cannot be established at the same time, because of the geographical distribution of either the system or its environment, it is better to represent the question and the answer by separate actions.

Both views are depicted in Figure 5.15. In this figure, actions are represented by circles and relationships between actions by arrows (the arrow connects two actions if one action is a condition for the occurrence of another action; the arrow points to the result action). The interaction system perspective consisting of a single action (*qareq*) is called the abstract question-answer interface. Both the value of the question (*q*) and of the answer (*a*) are established in this action. The interaction system perspective consisting of two actions is called the question-answer service. In one of the actions (*qreq*) the value of the question (*q*) is established, while in the other action (*qind*) the value of the corresponding answer (*a*) is established.

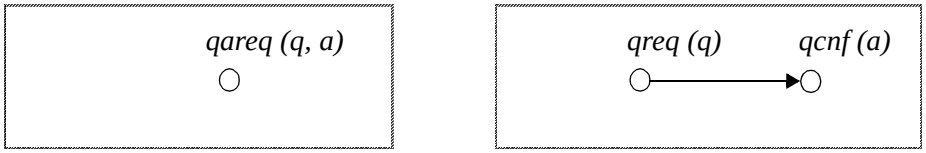


Figure 5.15: Abstract question-answer interface and question-answer service

5.4.2 Question-answer service provider

External behaviour

We can distribute the responsibility for the question-answer service behaviour such that the question-answer service provider imposes the constraints on the relationship between *qreq* and *qcnf*. This decomposition is depicted in Figure 5.16. In this figure, partial responsibility for an action is represented by a circle segment. Figure 5.16 also shows the composition of the entities to which behaviour responsibility is assigned: the question-answer service provider (*QASP*) and the question-answer service user (*QASU*). *QASP* and *QASU* share a single interaction point, if we assume that *qreq* and *qcnf* occur at the same logical location (hence, we anticipate the geographical distribution of *QASP*)

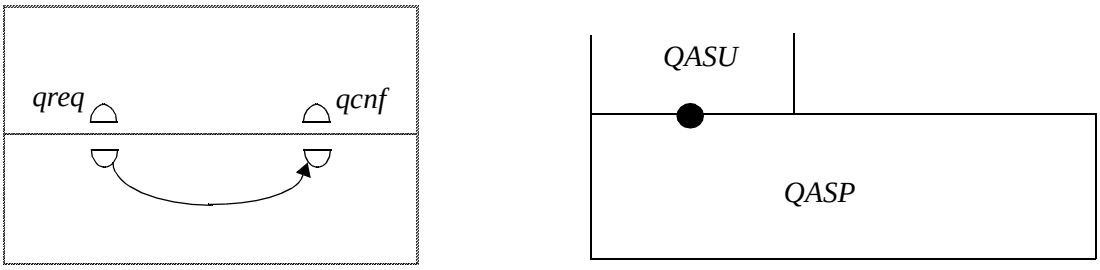


Figure 5.16: Question-answer service provider external behaviour

The establishment of values in *qreq* and *qcnf* may involve contributions from both *QASU* and *QASP*. For example, the value of the question in *qreq* is normally determined

by *QASU*, but may be constrained by the *QASP* (i.e., the service user cannot establish any value because the service provider only accepts values in a certain range). As discussed in subsection 5.2.4, not all constraints on service primitives need to be distributed. In particular, 'local' constraints are normally not distributed until protocol implementation time.

Internal behaviour

Figure 5.17 depicts an internal behaviour of *QASP* that reflects its geographical distribution. The service request (*qreq*) leads to an internal action (*send1*) in which the question value is represented according to a format that can be understood by any subsystem of the *QASP*. The question value can then be received (*rec1*) by the subsystem that determines the corresponding answer value (*send2*). The answer value is also represented in some agreed format so that it can be received (*rec2*) by the subsystem local to the service user. The service confirmation (*qcnf*) with the answer value (in any local format) can then be executed.

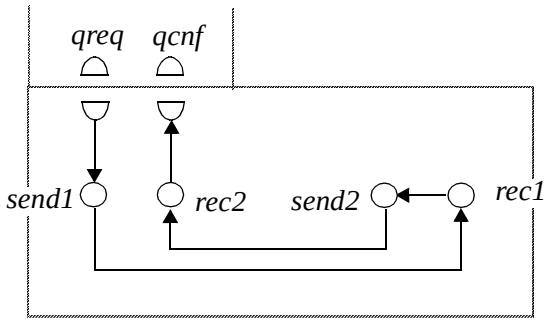


Figure 5.17: Question-answer service provider internal behaviour

Figure 5.18 depicts a more refined internal behaviour of *QASP* that reflects also a possible hierarchical decomposition of the provider functionality. Here, protocol actions (*send-q*, *rec-q*, *send-a*, and *rec-a*) are distinguished from lower level service primitives (*qreq'*, *qind'*, *qrsp'*, and *qcnf'*). The lower level service primitives together represent transparent data transfer. They do not constrain question and answer values or their representation (except that a maximum length constraint may be imposed on the concrete representation of values that need to be transferred).

5.4.3 Question-answer protocol architecture

The lower level service primitives and their relationships form a lower level service. This service is concerned with (one instance of) confirmed data transfer. We will call this service the question-answer transfer service. The relationship between *qind'* and *qrsp'* is achieved in the internal behaviour via the occurrence of *rec-q* and *send-a*. A general-

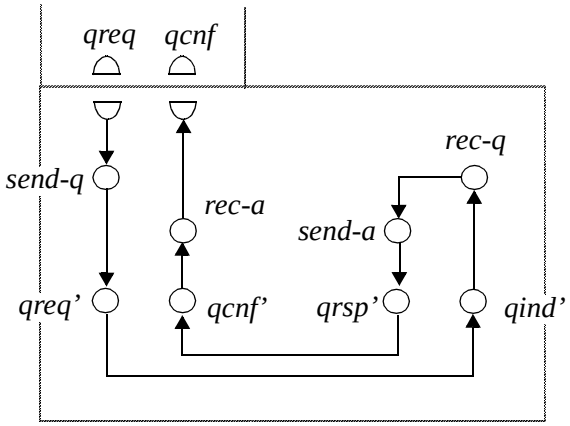


Figure 5.18: Question-answer service provider internal behaviour with protocol actions and lower level service primitives

purpose question-answer transfer service would directly represent the ordering relationship between *qind'* and *qrsp'*.

We can distribute the responsibility for the question-answer transfer service behaviour such that the question-answer transfer service provider imposes the constraints on the relationship between *qreq'* and *qind'*, and between *qrsp'* and *qcnf'*. The ordering constraint between *qind'* and *qrsp'*, mentioned above, is generally not distributed and assigned until protocol implementation time. (Only at that time it becomes clear whether *qind'* and *qrsp'* will be implemented under one or several implementation authorities. If they are implemented under one implementation authority, the ordering constraint can even be eliminated, as was illustrated by the internal behaviour; see Figure 5.18.) Here, we assume, however, that this constraint is assigned to the question-answer service provider. This behaviour decomposition is depicted in Figure 5.19.

Figure 5.19 also shows the composition of the entities to which behaviour responsibility is assigned: the question-answer transfer service provider (*QATSP*), the question protocol entity (*QPE*), the answer protocol entity (*APE*), and the question-answer service user (*QASU*). *APE* only shares an interaction point with *QATSP*, whereas *QPE* shares interaction points with *QASU* and *QATSP*. Together, *QPE* and *APE* form the question-answer layer protocol that uses the question-answer transfer service and provides the question-answer service.

5.5 Conclusion

We argued that it is useful to distinguish two domains in distributed system specification, viz. an entity domain and a behaviour domain. Each of these domains addresses different design concerns and design objectives. Discussion of the domains led to the identification of a number of elementary architectural concepts, viz. action, interaction, action point,

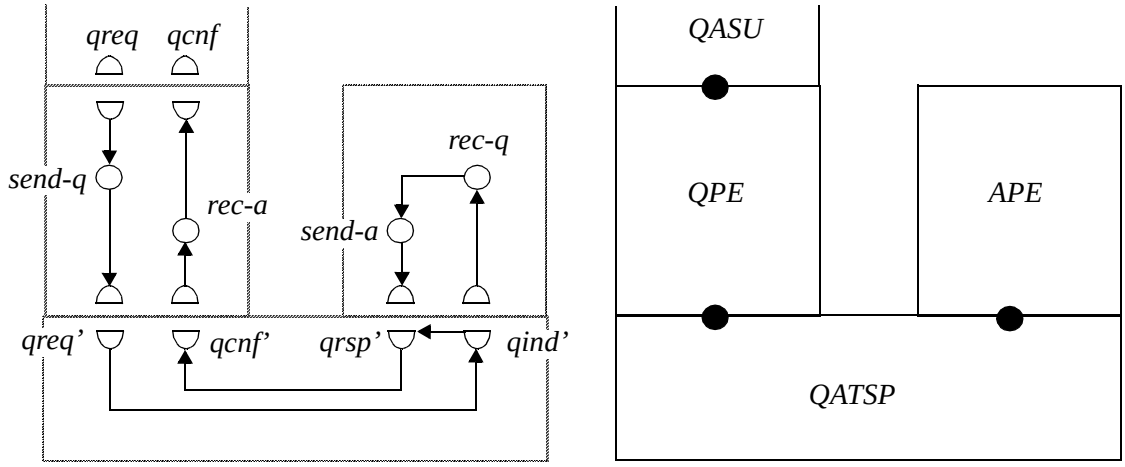


Figure 5.19: Question-answer protocol architecture

and interaction point. In addition, it must be possible to represent the relationship between (inter)actions (to define behaviour) and the assignment of behaviour to functional entities. The behaviour of a composite functional entity should correspond to a composition of the behaviours of the component functional entities. The representation of behaviour, in terms of the above mentioned elementary concepts, will be addressed in chapter 6 (Design Model).

Furthermore, we advocated the use of a set of related abstraction levels to support the design of distributed systems. Three abstraction levels are identified which can be used for the representation of any functional entity: the distributed system perspective, the integrated system perspective, and the interaction system perspective. Each of these abstraction levels can be combined with several behaviour views, which results in several more abstraction levels. In this way the distributed interaction system perspective can be distinguished as a special case of the distributed system perspective. The distributed interaction system perspective is used to represent protocol architectures. The (integrated) interaction system perspective is used to represent services, and the integrated system perspective is used to represent service providers.

The abstraction levels are used to define an application protocol design trajectory. The abstraction protocol design trajectory covers a wide abstraction spectrum. Three related areas of concerns are covered: the design trajectory from enterprise service to distributed enterprise architecture, the design trajectory from application (processing) service to distributed processing architecture, and the design trajectory from application service to application protocol architecture. In each of these areas, possible design milestones are identified. All design milestones are related by design steps in the application protocol design trajectory. In this way, the design trajectory permits the systematic consideration of

design concerns and design objectives in application protocol design starting from a convenient (high) abstraction level.

References

[Bannon91] Bannon, L.J. and Schmidt, K., CSCW: four characters in search of a context, In: *Studies in Computer Supported Cooperative Work - Theory, Practice and Design*, Bowers, J.M., and Benford, S.D. (editors), North-Holland, 1991, 3-17.

[Boehm88] Boehm, B.W., A spiral model of software development and enhancement, *IEEE Computer*, Vol. 21, No. 5, May 1988, 61-72.

[Bogaards90] Bogaards, K., *A methodology for the architectural design of open distributed systems*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1990.

[Bowen91] Bowen, D., Open distributed processing, *Computer Networks and ISDN Systems*, 23 (1991) 195-201.

[Ferreira93] Ferreira Pires, L., Vissers, C.A., and Sinderen, M. van, Advances in architectural concepts to support distributed systems design, In: *Proceedings of the 11th Brazilian Computer Network Symposium*, 9-13 May 1993, São Paulo, Brazil.

[Gotzhein93] Gotzhein, R., *Open distributed systems: on concepts, methods, and design from a logical point of view*, Vieweg, 1993.

[Kramer90] Kramer, J., Magee, J., and Finkelstein, A., A constructive approach to the design of distributed systems, In: *10th IEEE International Conference on Distributed Computing Systems*, IEEE Computer Society Press, 1990, 580-587.

[Markus90] Markus, M.L. and Connolly,T., Why CSCW applications fail: problems in the adoption of interdependent work tools, In: *Proceedings of the Conference on Computer-Supported Work*, 7-10 October 1990, Los Angeles, CA, USA, 371-380.

[Sinderen95] Sinderen, M., Ferreira Pires, L., Vissers, C.A., and Katoen, J.-P., A design model for open distributed systems, To appear in: *Computer Networks and ISDN Systems*, special issue on Open Distributed Processing. (Also available as: *Memorandum Informatica 94-27*, University of Twente, Enschede, The Netherlands, 1994.)

[Vissers88] Vissers, C.A., Scollo, G., and Sinderen, M. van, Architecture and specification style in formal descriptions of distributed systems, In: *Protocol Specification, Testing, and Verification, VII*, Aggerwal, G. and Sabnani, K. (editors), Elsevier Science Publishers B.V. (North-Holland), 1988, 189-204.

[Vissers93] Vissers, C.A., Sinderen, M. van, Ferreira Pires, L., What makes industries believe in formal methods, In: *Protocol Specification, Testing, and Verification, XIII*, Danthine, A., Leduc, G., and Wolper, P. (editors), Elsevier Science Publishers B.V. (North-Holland), 1993, 3-26.

Chapter 6

Design Model

This chapter presents a design model that allows the definition of behaviour and the assignment of behaviour to functional entities. It is based on the concepts of action, interaction and causality relation. Different ways to compose behaviours from instances of these concepts are discussed, including composition techniques that allow the representation of structured behaviours. Also behaviour refinement (including behaviour decomposition) is discussed. Different behaviour refinement types are identified, and their relevance to design steps in the application protocol design trajectory is indicated. Finally, some requirements for correct behaviour refinement are presented.

Action, interaction and causality relation are the elementary design, or architectural, concepts of our design model. Hence, instances of these concepts are the basic building blocks available to the designer. The ability to represent different milestones of the design trajectory and to carry out design steps depends ultimately on the choice and definition of these concepts. The purpose of this chapter is to identify and define elementary concepts, composition rules and refinement requirements that allow proper support of the application protocol design trajectory.

The structure of this chapter is as follows: section 1 defines the concepts of action and interaction; section 2 defines the concept of causality relation; section 3 discusses the composition of monolithic, i.e. unstructured, behaviour; section 4 and section 5 present two behaviour composition techniques that allow the representation of structured behaviours; section 6 investigates the ‘power of expression’ of our design model by considering the representation of general behaviour patterns; section 7 identifies and characterizes some refinement types that are particularly useful in the application protocol design trajectory; and section 8 presents the conclusions of this chapter.

6.1 Action and interaction

Actions and interactions are used as abstractions of activities in the real world. These abstractions should be able to capture all aspects of activities that are essential at the considered abstraction level, allowing us to reason about activities without the burden of

their details. Since we like to use the concepts of action and interaction throughout the design process of application protocols, the definition of the concepts should be independent of the granularity at which activities are considered (since the level of granularity varies per abstraction level). At each level of abstraction, actions and interactions are used as basic (i.e., the smallest possible) building blocks, abstracting from their possible implementations represented at lower abstraction levels.

This section defines action and interaction properties, and introduces a textual notation which facilitates reasoning about action and interaction properties. (This notation will be extended in later sections, such that also compositions of actions and interactions can be represented.)

6.1.1 General properties

An *action* represents a unit of activity that is assigned to a single functional entity. An *interaction* represents a unit of activity that is common to two or more functional entities. An *interaction contribution* represents a distinct responsibility of one of the entities with respect to the activity. The qualification 'unit of activity' indicates that no sub-activities are distinguished at the considered abstraction level. Consequently, only overall characteristics of the activity are represented, i.e. results of the activity which can be observed after its successful completion and properties that apply to the activity as a whole.

We say that an (inter)action *occurs* if we want to model that the activity which is represented is successfully completed. Furthermore, we say that an (inter)action is *enabled* if all conditions for this (inter)action to occur are satisfied. Such conditions state the occurrence or non-occurrence of other (inter)actions.

Since a designer generally wants to be able to refer to individual actions and interactions, we assume that each (inter)action introduced in the design can be distinguished from others. Any identification method can be used for this purpose. For convenience we adopt the solution that each (inter)action is assigned a single *identifier* that is unique in the context of the behaviour that contains the action or contributes to the interaction.

An interaction can be considered as a decomposition, or distribution, of a corresponding action. The action represents the same activity, but at a higher abstraction level, where the involvement of different entities is not considered. From a designer's perspective, this action is an integrated interaction, i.e. an interaction viewed in such a way that the distribution of responsibilities over different entities is ignored. Note that although all interactions can be represented by actions at higher abstraction levels, not all actions represented at some abstraction level are necessarily decomposed into interactions at lower abstraction levels. All integrated interactions are therefore actions, but not all actions are integrated interactions. As we have seen in Chapter 5 (Design Framework), decomposing an action into an interaction may involve an intermediate step where the

action is represented in a distributed form but the assignment of responsibilities to different entities is ignored. It is even possible to have distributed actions which remain assigned to a single entity (namely if the distribution serves to represent a convenient structure of the behaviour, and not to represent an entity structure). Thus, all interactions are distributed actions, but not all distributed actions are interactions. Figure 6.1 illustrates the possible views on the representation of activities in the entity domain and the behaviour domain.

assignment	*entity domain*	*behaviour domain*
one entity, monolithic behaviour	action point	action
one entity, structured behaviour	action point	distributed action
multiple entities, multiple behaviours	interaction point	interaction

Figure 6.1: Related views on the representation of activities

In the sequel, we will use the term action to denote both actions and interaction contributions, unless the properties discussed are specific to either actions or interaction contributions. This will be made clear from the context. In addition, since the distinction between distributed actions and interactions is only relevant in the entity domain, we will use the term interaction to denote both distributed actions and interactions, unless properties of the entity domain are discussed.

6.1.2 Attributes

We identify, based on available design models and our own design experience (see [Ferreira93], [Vissers93], [Sinderen95], for example), a number of overall characteristics of activities which are relevant in the design process. We model these characteristics by action *attributes*:

- *Location* attribute: this attribute defines *where* an action occurs. A value of the location attribute of an action can be considered as an identification of the action point with which the action is associated. With this attribute it is thus possible to relate the behaviour domain, in which the action is defined, and the entity domain, in which the action point is defined.
- *Time* attribute: this attribute defines *when* an action occurs. Consequently, it also determines when (attribute values of) the action can be referred to by other actions. We allow that an action refers to the time attribute of other, previous actions, thus supporting the definition of relative and absolute time.

- *Information* attribute: this attribute defines *what* information is established in an action. A single action can establish values of many information elements, each of arbitrary complexity. Only those characteristics of an information element should be represented which are relevant at the considered abstraction level.
- *Functionality* attribute: this attribute defines what attribute values of previous actions are retained by an action (and thus can be referred to by subsequent actions). The functionality attribute can be seen as an extension of the 'local results' of an action. It can be roughly compared with the application of scope rules, as defined in specification and programming languages. The functionality attribute allows, however, the explicit definition of the scope of attribute values of actions.
- *Probability* attribute: this attribute defines the probability that an action occurs once all conditions for this action to occur are satisfied, i.e. once it is enabled. A value of the probability attribute may be used to model the reliability of the activity represented by the action.

Not all action attributes are always used. They may be omitted, and replaced by default interpretations, if they are not relevant at the considered level of abstraction. Or, conversely, action attributes may be introduced as soon as they become relevant in the design process. In this thesis, we will not consider reliability aspects of activities during design, hence we will not use the probability attribute.

6.1.3 Constraints

During design, the characteristics of activities must somehow be quantified. This is achieved in our basic design model with action *constraints*, constraints that apply to the values of action attributes. Action constraints define what values are permitted for the attributes of an action.

Sometimes the characteristics of an activity are completely determined at some abstraction level, in which case the activity can be represented by an action with constraints that permit only a single value for each attribute or attribute element. It is also possible that, at some point in the design process, different characteristics of an activity must be represented without being completely deterministic about each of them. In such a case, the activity can be represented by an action with constraints that permit more than one value for those attributes and attribute elements that are not yet completely determined. This type of non-determinism may be (partially) resolved in later design steps, or may be intentionally preserved to serve as a specification of freedom left to the implementor. For example, if we want to consider an activity which establishes an integer result between 5 and 10, a proper representation would be an action with an information element *integer* and a constraint $5 < integer < 10$.

Constraints are in principle applicable to any attribute and attribute element of an action, except the functionality attribute (the functionality attribute merely states which attribute values of a previous action are retained). Constraints on the location attribute determine the alternative locations where the action can occur; constraints on the time attribute determine the alternative moments in time when the action can occur; and constraints on information elements of the information attribute determine what alternative information values can be established in the action.

So far, we made no distinction between actions and interactions. This was possible since the discussion above was independent of the distribution of responsibilities for an activity over multiple behaviours or functional entities. The concept of constraints allows us to be more explicit on the representation of responsibility distribution. We choose to represent the distribution of responsibilities for a unit of activity by a decomposition of constraints. This means that contributions to an interaction are indicated by distinct constraints on the values of interaction attributes.

It should be possible to determine the consistency of an action and a corresponding interaction at a lower abstraction level. If constraints are expressed as boolean expressions to be satisfied by attribute values, consistency requires that the logical conjunction of the constraints of interaction contributions corresponds to the action constraints. Constraints of different interaction contributions should not be mutually inconsistent (otherwise the interaction can never occur). This requirement is always satisfied if the constraints of interaction contributions are derived through decomposition of the action constraints.

6.1.4 Interpretation of non-deterministic constraints

To illustrate the interpretation of non-deterministic action and interaction constraints, we consider the constraint *5 < integer < 10*. If this constraint is taken as the constraint of an action, the interpretation by the designer should be that any activity that generates an integer value between 5 and 10 is a valid implementation of this action. The designer will choose between different implementation options on basis of, for example, performance and cost considerations.

If we consider the same constraint for an interaction, this interpretation is still valid at an integrated interaction level. However, if we decompose the constraint into constraints of interaction contributions, this interpretation cannot be applied to each constraint in isolation. For example, consider two interaction contributions with constraints *integer > 5* and *integer < 10*. Applying the above interpretation to each of these constraints would most probably lead to an implementation of the interaction that deadlocks.

The following cases can be distinguished with respect to interaction constraints:

- if the constraints of all interaction contributions on some information value are non-deterministic, then the designer should consider the interaction at an integrated interaction level. In this way, he is able to design a proper *value generation* mechanism.
- if only the constraints of some interaction contributions on some information values are non-deterministic, then the interaction defines *value passing*. Valid implementations of an interaction contribution with non-deterministic constraints should be able to accept any value that is possible according to the constraints.
- if the constraints of all interaction contributions on some information values are deterministic, then the interaction defines *value matching*. The implementation of each interaction contribution should be able to synchronize on the value determined by the constraint.

Figure 6.2 shows what are the possible combinations of deterministic and non-deterministic constraints when an interaction is composed of two interaction contributions, and what are the resulting interpretations.

integrated interaction	*interaction*		*interpretation*
	contribution 1	contribution 2	
non-deterministic constraints (?)	?	?	value generation
deterministic constraints (!)	!	?	value passing
	!	!	value matching

Figure 6.2: Interpretation of interaction constraints

One can argue that the transformation of an action with non-deterministic constraints into an interaction is not very effective. This is the case since the constraints of all interaction contributions will be non-deterministic so that implementation decisions with respect to the interaction can only be taken at the integrated interaction level. Hence, non-determinism should first be resolved at an integrated level. If an action with non-deterministic constraints is introduced at some level of abstraction, the designer is free to introduce a mechanism at lower abstraction levels that (partially) resolves the non-determinism, or he may preserve the non-determinism and leave its resolution to the implementor.

Similar interpretations can be adopted with respect to constraints on location values and time values. Consider, for example, an action with constraint $t + 5 < time < t + 10$, where t is some reference time and 5 and 10 are time periods of 5 and 10 time units, respectively. The designer is then free to choose at what time his implementation of the action must complete, provided that this happens between $t + 5$ and $t + 10$. Which choice

is preferred by the designer depends, among others, on the performance and cost considerations that are taken into account. Transforming the action into an interaction, for example composed of two interaction contributions with constraints *time > t + 5* and *time < t + 10*, is not effective, as explained above. On the other hand, if the original constraint was deterministic, for example *time = t + 6*, an interaction can be derived. For example, the interaction can be composed of two interaction contributions, such that the constraint of one of the contributions is a duplication of the original constraint, and the constraint of the other interaction contribution is non-deterministic, but implied by the original constraint, e.g. *time > t + 5*. The implementation of the interaction contribution to which the non-deterministic constraint applies is such that participation in the interaction is offered at any point in time later than *t + 5*.

6.1.5 Textual notation

In order to facilitate reasoning about action and interaction properties, and properties of compositions of actions and interactions, we introduce an ad hoc notation. This notation is defined using Backus-Naur Form in an Appendix of this thesis. Here follows an explanation of the notation as far as it concerns the concepts defined until now:

- an action is textually represented by its identifier, possibly followed by a list of attributes between round brackets, and a list of constraints between square brackets;
- the order of attributes in the attribute list is: time attribute, location attribute, information attribute, and functionality attribute. The attributes (attribute elements) are separated by a comma, except the information and functionality attribute, which are separated by a vertical bar. There is no prescribed ordering of constraints in the constraints list;
- the location and time attribute, and each information element of the information attribute, are represented by a value identifier, followed by a colon and an indication of the type of the attribute (element). The type of the location attribute is *Location*, the type of the time attribute is *Time*, and the type of an information element is arbitrary, except that it may not be equal to *Location* or *Time*;
- the functionality attribute is represented as a list of location, time and information attributes.

  Examples of actions:
  a
  b (t: Time, p: Location, v: Integer)
  c (t1: Time, p: Location, v1: Integer, v2: Boolean | t0: Time, v0: Integer)
  d (t: Time, v: Integer) [t < t0 + 10, v = v0 + 5]

- an interaction contribution is represented in the same way as an action, except that interaction contribution identifiers are underlined. In order to be able to associate the

contributions to an interaction, the same identifier is used for each contribution (it is also possible to define a renaming of identifiers to accomplish this).

- contributions to an interaction have matching attribute lists, i.e. attributes (attribute elements) of the same type must be present in the same order in all contributions.

Examples of interactions (contributions are placed in different columns):

a	*a*
b (t: Time, p: Location, v: Integer)	*b (t: Time, p: Location, v: Integer)*
c (t1: Time, p: Location, v1: Integer, v2: Boolean | t0: Time, v0: Integer)	*c (t1: Time, p: Location, v1: Integer, v2: Boolean | t0: Time, v0: Integer)*
d (t:Time, v:Integer) [t < t0+10]	*d (t:Time, v:Integer) [t < t0+10, v = v0+5]*

6.2 Causality relation

The occurrence of an action generally depends on the occurrence or non-occurrence of other actions. In order to represent this dependency, we introduce the concept of *causality relation*. A causality relation of a specific action defines the enabling condition for that action. An enabling *condition* is expressed in terms of actions that must have occurred, the *enabling* actions, and actions that must not have occurred, the *disabling* actions. The action that is enabled if the condition is satisfied, is referred to as the *result* action.

This section discusses causality relations in order of increasing complexity. It also introduces an ad hoc graphical notation, which helps to understand the relationships between actions. (This notation will be extended in later sections, such that also relationships between interactions can be represented.)

6.2.1 Basic conditions

Enabling condition consisting of a single enabling action

We define the causality relation $a1 \rightarrow a2$ as: the occurrence of action *a1* is a condition for the occurrence of action *a2*.

This causality relation has the following properties:

- if *a2* occurs, *a1* must have occurred earlier. This timing constraint is not specified by a constraint on the time attribute *a2*, but is implicitly represented by the causality relation.
- *a2* can refer to attributes of *a1*. These references occur in the constraints of *a2* (if they are necessary to constrain attribute values of *a2*), or in the functionality attribute of a2 (if values of *a1* are retained).

Examples:
a (v0: Integer) → b (v1: Integer) [v1 = v0 +1]
c (t0: Time, v0: Integer) → d (t1: Time, v1: Integer | v0: Integer) [t1 < t0 + 5]

Enabling condition consisting of a single disabling action

We define the causality relation *¬a1 → a2* as: the non-occurrence of action *a1* is a condition for the occurrence of action *a2*.

This causality relation has the following properties:

- if *a2* occurs, *a1* may not have occurred earlier, nor may it occur at the same time. Or, in other words: *a2* can only occur before *a1* occurs. This timing constraint is implicitly represented by the causality relation. (It would not have been possible to represent this timing constraint with a constraint on the time attribute of *a2* anyway, since there is no time to refer to, assuming that references to the future are not possible.)
- the occurrence of *a1* implies that *a2* can no longer occur. The occurrence of *a2* does not imply that *a1* may no longer occur; whether *a1* actually occurs in this case depends on the causality relation of *a1*.

This causality relation characterizes a special type of causality, called *conflict*. Actually, the conflict described is a one-way conflict: the occurrence of *a1* inhibits the occurrence of *a2*, but the occurrence of *a2* does not necessarily inhibit the occurrence of *a1*. Later on we will see that conflicts play an important role in defining certain behaviour patterns. The implied property that two actions related by a conflict may not occur at the same time is essential for these patterns.

Note that in this causality relation, *a2* may not refer to attributes of *a1*, simply because *a1* has not occurred if *a2* occurs. Disabling actions will therefore only be represented by their identifier (unless the constraints of these actions are used; see subsection 6.2.3).

Example:
¬a → b (v: Integer) [v > 1]

6.2.2 Composite conditions

Conjunction of basic conditions

An enabling condition may be formed by the conjunction of basic conditions, each stating the occurrence or non-occurrence of an action. Such conditions are combined using the *and* ("$\wedge$") logical operator. Consider for example the causality relation:

$$a1 \wedge a2 \rightarrow a3$$

This causality relation states that the occurrence of both *a1* and *a2* is a condition for the occurrence of *a3*. Consequently, *a3* can refer to attributes of both *a1* and *a2*. In a similar way we can define the conjunction of non-occurrence conditions and of a mix of occurrence conditions and non-occurrence conditions.

Disjunction of basic conditions

An enabling condition may also consist of the disjunction of basic conditions. Such conditions are combined using the *or* ("$\vee$") logical operator. Consider the example:

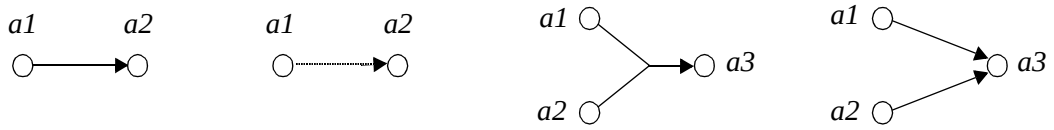
$$a1 \vee a2 \rightarrow a3$$

This causality relation states that the occurrence of *a1* or *a2* is a condition for the occurrence of *a3*. It is possible that *a1* and *a2* both happen, but the occurrence of one of them is sufficient for the occurrence of *a3*. Action *a3* can refer to attributes of either *a1* or *a2*, depending on which action has caused *a3*. If *a1* and *a2* both happen before a3, there is a choice with respect to the action that has caused *a3*, and consequently with respect to the attributes that are used by *a3*. This choice is non-deterministic, and should be resolved at lower abstraction levels.

In a similar way we can define the disjunction of non-occurrence conditions and of a mix of occurrence conditions and non-occurrence conditions.

Graphical notation

It is often convenient to have a graphical notation that provides a comprehensive representation of the relationships between actions, i.e. of the causality relations. Figure 6.3 depicts the building blocks of an ad hoc graphical notation that will be used in this and subsequent chapters. The graphical notation does not allow the representation of attributes and constraints (graphical representations may be combined, however, with textual representations).



(a) $a1 \rightarrow a2$ (b) $\neg a1 \rightarrow a2$ (c) $a1 \wedge a2 \rightarrow a3$ (d) $a1 \vee a2 \rightarrow a3$

Figure 6.3: Building blocks of graphical notation

6.2.3 Constraints of enabling actions and disabling actions

The constraints of a result action are used to define what attribute values are permitted for the result action. Constraints can also be applied to actions in enabling conditions. These constraints are not used to define what attribute values are permitted for an action (since this is defined in the causality relation where this action is the result action), but to define the range of attribute values that qualify the action as an enabling or disabling action.

The constraints of an enabling action are only effective if the constraints are not implied by the constraints of the corresponding result action (i.e., the result action, defined in another causality relation, with the same identifier as the enabling action). The same applies for the constraints of a disabling action.

Examples:
a (v0: Integer) [v0 < 10] → b (v1: Integer) [v1 = v0 +1]
¬a (v0: Integer) [v0 < 10] → b (v1: Integer) [v1 > 1]

6.3 Monolithic behaviour

A causality relation can be considered as an elementary form of behaviour. It uniquely defines the conditions for an action to occur and the constraints on the values established by that action. A finite behaviour can thus be represented by a set of causality relations, one relation for each action in this behaviour.

An important advantage of using causality relations as the basis for defining behaviour is that causality relations enable the definition of required relationships between actions, without imposing additional constraints for reasons of simplifying model manipulation. For example, there is no implicit global ordering or interleaving of interactions; actions are not related, unless their relationship is defined by causality relations.

A basis for defining a formal semantics for causality-based behaviour definitions can be found in [Gunawardena92]. A mapping of causality-based behaviour definitions onto place/transition Petri-nets is discussed in [Sinderen95]. Both papers, however, do not

completely cover the concepts introduced in this chapter. The definition of a complete formal semantics is outside the scope of this thesis. Instead, we will informally define the semantics of the concepts in design engineering terms.

This section introduces some additional concepts needed for behaviour definition, and discusses the composition of monolithic (i.e., unstructured) behaviour.

6.3.1 Initial actions and terminal actions

Start condition for initial actions

In any behaviour there must be at least one initial action which does not depend on any other actions defined by the behaviour. Hence, we need a special enabling condition for initial actions. If the behaviour is also independent of other behaviours, this condition is called the start condition, indicated by *start* in the textual notation.

A start condition is spontaneously true since it is independent of the occurrence or non-occurrence of actions. A start may be specified with attributes values, like actions stated in 'normal' enabling conditions, which can be referred to by the result action. A start can be interpreted as the initialization of a defined behaviour.

Terminal actions

In any finite behaviour there must be at least one terminal action. A terminal action is an action whose occurrence or non-occurrence is not required as a condition for any other action of the behaviour. It is not necessary to explicitly define which actions are terminal actions, since this can be determined by checking all enabling conditions. However, since readability is often improved if terminal actions are explicitly indicated, our textual notation supports the definition of terminal actions with a special keyword *stop*. An action is a terminal action if it is the only enabling action in a causality relation with *stop* instead of a result action.

6.3.2 Monolithic behaviour composition

Any finite behaviour can be represented by a set of causality relations. If the behaviour does not depend on other behaviours, at least one of the causality relations defines a start condition. The textual notation is extended as follows:

- a behaviour definition consists of a behaviour identifier, followed by a definition symbol (":="), followed by the actual definition of the behaviour;
- the behaviour is represented by a list of causality relations between curly brackets. Causality relations are separated by a comma.

Furthermore, comments can be indicated with the textual notation by placing text between percent symbols.

Examples:

B1 := % abstract question-answer interface %
{ start → qareq (q: Question, a: Answer) [aRq] }

B2 := % question-answer service %
{ start → qreq (q: Question), qreq (q: Question) → qcnf (a: Answer) [aRq] }

The graphical representation of these examples is depicted in Figure 6.4.

Figure 6.4: Abstract question-answer interface and question-answer service

Some care should be taken when defining the causality relations of a behaviour. We illustrate this with an example. Consider the following behaviour:

{ start → a (v1: Integer, v2: Integer),
a (v1: Integer, v2: Integer) → b (v3: Integer) [v3 = v1 +1],
a (v1: Integer, v2: Integer) → c (v4: Integer) [v4 = v2 + 1] }

In this behaviour action *a* establishes two information values, *v1* and *v2*. Action *a* is the enabling action for action *b* and action *c*. Action *b* depends on *a* since it needs *v1*, and action *c* depends on *a* since it needs *v2*. Action *a* may be implemented by an activity that first produces *v1* and at a later point in time *v2*. Furthermore, action *b* may be implemented by an activity that starts immediately after *v1* has been produced. As a consequence, it is possible that the activity that implements *b* is successfully completed before the activity that implements *a*. This would however contradict with the interpretation of the causality relation of *a*: action *a* must have occurred before action *b*, or, in terms of their implementations, the activity that implements *a* must complete before the activity that implements *b* can complete.

Unless it is explicitly the intention of the designer to let action *a* be implemented by an activity that makes *v1* and *v2* available to other activities at the same time (for example, if *a* is implemented by a transaction), one can conclude that the granularity of the actions in

the behaviour above do not match. Either *a* should be replaced by two separate actions, one establishing *v1* and one establishing *v2*, or *b* and *c* should be replaced by a single action that refers to both *v1* and *v2*.

alternative 1 (higher abstraction level):
{ start → a (v1: Integer, v2: Integer),
a (v1: Integer, v2: Integer) →
bc (v3: Integer, v4: Integer) [v3 = v1 +1, v4 = v2 + 1] }

alternative 2 (lower abstraction level):
{ start → a1 (v1: Integer)
start → a2 (v2: Integer),
a1 (v1: Integer) → b (v3: Integer) [v3 = v1 +1],
a2 (v2: Integer) → c (v4: Integer) [v4 = v2 + 1] }

Note that the second alternative can be considered as a decomposition, or refinement, of the first alternative.

Causality relations, as presented so far, support a parsimonious representation of relationships between actions of finite, monolithic behaviours. Complex behaviours can, however, generally not be easily represented in a monolithic fashion. Indeed, in Chapter 5 (Design Framework), we defined milestones in the design trajectory of application protocols which are based on the representation of structured behaviours. Composition techniques that support the representation of structured and repetitive (infinite) behaviours are therefore needed. These will be discussed in the next two sections.

6.4 Causality-oriented behaviour composition

Just as we can define causality relations between actions, we would like to be able to define causal relationships between behaviours. Such relationships are characterized by the fact that enabling conditions determined inside one (instance of) behaviour enable actions in other (instances of) behaviours. This type of distribution of causality relations may be very useful to represent the composition of ASEs, since it is more direct than the use of constraints (see Chapter 4, OSI Upper Layer Architecture and Model: State of the Art).

This section discusses *causality-oriented* behaviour composition, a technique for composing behaviours by defining causal relationships between them. It first introduces some additional concepts, and then presents generalized causality relations for causality-oriented behaviour composition.

6.4.1 Entry point and exit point

We introduce the concepts of *entry point* and *exit point* in our design model to support the definition of causal relationships between behaviours:

- *entry point*: a behaviour contains a causality relation with an entry point instead of an enabling condition if the result action depends on actions defined in other behaviours. An entry point does not specify enabling or disabling actions, since these actions are not part of the behaviour. On the other hand, an entry point may be specified with attributes, which can be referred to by the result action. The values of these attributes are established by actions defined in other behaviours.
- *exit point*: a behaviour contains a causality relation with an exit point instead of a result action if the enabling condition enables actions defined in other behaviours. An exit point does not specify which actions are enabled, since these actions are not part of the behaviour. It may, however, be specified with attributes of enabling actions, which can be referred to by actions of other behaviours.

Our textual notation uses the keywords *entry* and *exit* to denote an entry point and an exit point, respectively:

Example:

B := % question-answer service function %
{ entry → qreq (q: Question),
qreq (q: Question) → qcnf (a: Answer) [aRq],
qcnf (a: Answer) → exit }

A behaviour may have multiple entry and exit points. In order to distinguish between these points, they should have unique identifiers. Unique identifiers are constructed with the textual notation with numerical suffixes to the keywords *entry* and *exit* (e.g., *entry1*, *entry2*, etc.).

6.4.2 Exit/entry construct

A relationship between two (instances of) behaviours is defined by the coupling of an entry point of one behaviour to an entry point of the other behaviour. Such a coupling is called an *exit/entry construct*. It requires that external references can be made to the exit points and entry points of behaviours. An external reference to an entry point is called a behaviour entry; an external reference to an exit point is called a behaviour exit. We specify such references with our textual notation as follows:

- a behaviour entry consists of the behaviour identifier of the behaviour that contains the referenced entry point followed by the entry point identifier between round brackets;

- a behaviour exit consists of the behaviour identifier of the behaviour that contains the referenced exit point followed by the exit point identifier between round brackets.

An exit/entry construct is defined in the same way as causality relations between actions. A behaviour exit is used instead of an enabling condition with enabling and disabling actions, and a behaviour entry is used instead of a result action. This is illustrated by the following example:

B := % composition of two question-answer service functions in sequence %
{ start → B1 (entry),
B1 (exit) → B2 (entry)
where
B1 :=
{ entry → qreq (q: Question),
qreq (q: Question) → qcnf (a: Answer) [aRq],
qcnf (a: Answer) → exit }
B2 :=
{ entry → qreq (q: Question),
qreq (q: Question) → qcnf (a:Answer) [aRq],
qcnf (a: Answer) → stop }
}

Here, two behaviours, *B1* and *B2*, are composed. Both behaviours define a sequence of two actions, *qreq* followed by *qcnf* (representing a question-answer service function, i.e. a question followed by an answer). However, *qreq* of *B1* is enabled by a start condition, whereas *qreq* of *B2* is enabled by the occurrence of *qcnf* of *B1*. The textual representation also shows the use of *start* to enable a behaviour. The keyword *where* is used to separate a top (or intermediate) level behaviour from its component behaviours.

Figure 6.5 depicts the graphical representation of this example.

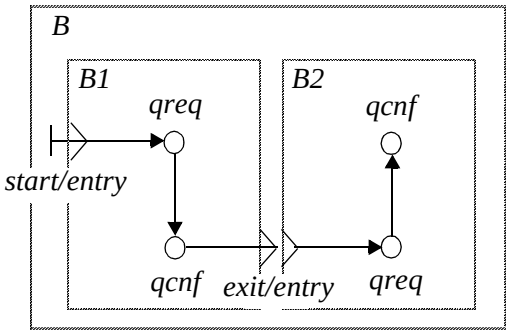


Figure 6.5: Composition of two question-answer service functions in sequence

An example of composing behaviours with multiple entry and exit points is:

B := % composition of two question-answer service functions in parallel %
{ start → B1 (entry1),
B1 (exit1) → B2 (entry1),
B1 (exit2) → B2 (entry2)
where
B1 :=
{ entry → qreq (q: Question),
qreq (q: Question) → qcnf (a: Answer) [aRq],
qreq (q: Question) → exit1,
qcnf (a: Answer) → exit2 }
B2 :=
{ entry1 → qreq (q: Question),
entry2 ∧ qreq (q: Question) → qcnf (a: Answer) [aRq],
qcnf (a: Answer) → stop }
}

Figure 6.6 shows the corresponding graphical representation. This example illustrates how causality-oriented composition can be used to define relationships between partially concurrent behaviours: *qreq* of *B2* is enabled by *qreq* of *B1*, and *qcnf* of *B2* is enabled by *qcnf* of *B1*; however, *qreq* of *B2* may occur before *qcnf* of *B1*.

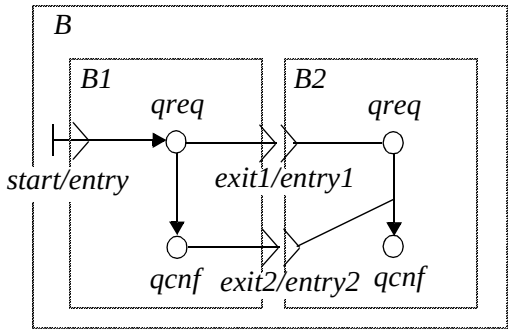


Figure 6.6: Composition of two question-answer functions in parallel

6.4.3 Other uses of behaviour entries

It is also possible to use behaviour entries directly in a behaviour to define the relationship to other behaviours. This is illustrated by the following example:

B := % nested composition of two question-answer service functions %
{ start → qreq (q: Question),
qreq (q: Question) → qcnf (a: Answer) [aRq],

qreq (q: Question) → B1 (entry1)
qcnf (a: Answer) → B1 (entry2)
where
B1 :=
{ entry1 → qreq (q: Question),
entry2 ∧ qreq (q: Question) → qcnf (a: Answer) [aRq],
qcnf (a: Answer) → stop }
}

This example represents the same behaviour as the second example in section 6.4.2, however with a nested composition instead of a composition where the component behaviours are defined at the same level. Figure 6.7 depicts the graphical representation of this example.

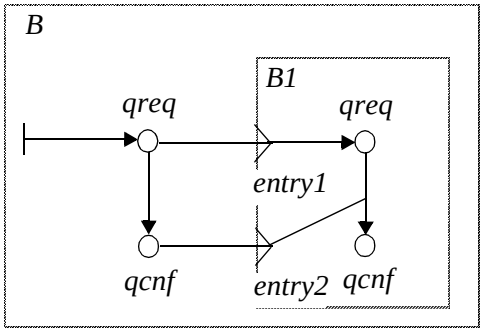


Figure 6.7: Nested composition of two question-answer service functions

A repetitive (infinite) behaviour can be defined with infinite nesting, as illustrated by the following example (see Figure 6.8):

B := % repetitive question-answer service behaviour %
{ start → B1 (entry1),
start → B1 (entry2)
where
B1 :=
{ entry1 → qreq (q: Question),
entry2 ∧ qreq (q: Question) → qcnf (a: Answer),
qcnf (q: Question) → B1 (entry) }
}

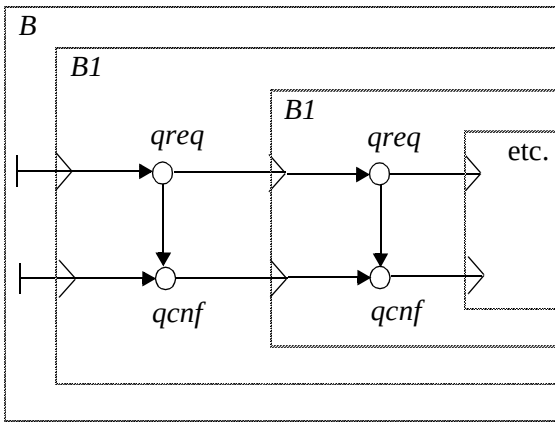


Figure 6.8: Repetitive question-answer service behaviour

6.4.4 Other uses of behaviour exits

Just like behaviour entries, behaviour exits can be used directly in a behaviour to define the relationship to other behaviours. This is illustrated by the following example (see Figure 6.9):

B := % alternative nested composition of two question-answer service functions %
{ start → B1 (entry),
B1 (exit1) → qreq (q: Question),
B1 (exit2) ∧ qreq (q: Question) → qcnf (a: Answer) [aRq],
qcnf (a: Answer) → stop
where
B1 :=
{ entry → qreq (q: Question),
qreq (q: Question) → qcnf (a: Answer) [aRq],
qreq (q: Question) → exit1,
qcnf (a: Answer) → exit2 }
}

6.4.5 Requirements for causality-oriented composition and decomposition

Composition requirements

Entry points and exit points can only be coupled if they have matching attributes (or no attributes). When using the textual notation, the attribute lists of coupled exit and entry

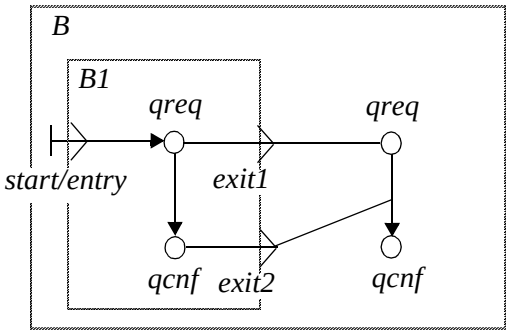


Figure 6.9: Alternative nested composition of two question-answer service functions

points (and related behaviour exits and entries) should have attribute lists with attributes of the same type and presented in the same order.

Example:

B :=
{ start → B1 (entry),
B1 (exit (v1: Integer, v2: Boolean)) → B2 (entry (v1: Integer, v2: Boolean))
where
B1 := { entry → ... , ... , ... → exit (v1: Integer, v2: Boolean) }
B2 := { entry (v1: Integer, v2: Boolean) → ..., ... }
}

Decomposition requirements

In addition to the above mentioned requirement, a causality-oriented composition should represent the behaviour intended by the designer. The interpretation of causality-oriented composition, i.e. the relationship between a causality-oriented behaviour composition and a monolithic behaviour composition, is expressed by (correct) decomposition requirements. We consider a causality relation of a monolithic behaviour and the requirements that apply to the causality-oriented decomposition of this causality relation:

- the enabling condition of the original causality relation is equivalent to enabling condition of the exit point;
- the result action of the original causality relation is equivalent to the result action that is enabled by the entry point; and
- the exit point and the entry point must match, and their attributes should be the attributes referred to by the result action.

6.5 Constraint-oriented behaviour composition

Causality-oriented behaviour composition allows us to represent a distribution of causality relations where enabling conditions and result actions are assigned to different component behaviours. The distribution of action constraints is not considered in this composition technique. Structures based on distributed actions and interactions are, however, highly relevant in application protocol design. The representation of the behaviour of a distributed system (e.g., a protocol), that is composed of interacting system parts (e.g., protocol entities and a lower level service provider), requires such structures.

This section discusses *constraint-oriented* behaviour composition, a technique for composing behaviours that provide contributions (impose partial constraints) on common interactions (or distributed actions). It first introduces an additional concept, and then presents requirements that apply to constraint-oriented behaviour composition and decomposition.

6.5.1 Synchronization requirements

In a constraint-oriented behaviour composition, component behaviours are related by interactions. It is therefore necessary to indicate what behaviour components are related by what interactions. This is defined by a set of *synchronization requirements*, where each requirement indicates (1) a collection of behaviour components that together define interactions and, (2) the interactions they define.

With our textual notation, synchronization requirements are represented by a list of synchronization requirements between round brackets, where:

- synchronization requirements in the list are separated by a semicolon;
- each synchronization requirement consists of a list of behaviour identifiers and a list of interaction identifiers, separated by a colon;
- behaviour identifiers as well as interaction identifiers are separated by a comma.

A synchronization requirements list is inserted before the causality relations, separated from them by a comma.

We consider as example the constraint-oriented composition of the question-answer service, such that constraints imposed by the service user and the service provider are separated:

B := % question-answer service %
{ (B1, B2: qreq, qcnf),
start → B1 (entry),
start → B2 (entry)

where
B1 := % question-answer service user constraints %
{ entry → qreq (q: Question), entry → qcnf (a: Answer) }
B2 := % question-answer service provider constraints %
{ entry → qreq (q: Question), qreq (q: Question) → qcnf (a: Answer) [aRq] }
}

With the graphical notation, synchronization requirements follow from the representation of interactions: circle segments that represent contributions to the same interaction are placed together, while dotted lines enclose circle segments to make clear to what behaviours the contributions belong. Figure 6.10 depicts the graphical representation of the previous example.

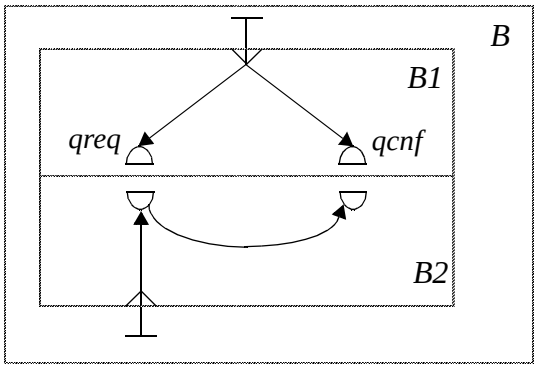


Figure 6.10: Constraint-oriented composition of question-answer service separating constraints of the user and the provider

The previous example illustrates a behaviour structure that corresponds to an entity structure (the behaviour structure can also be considered as the result of an intermediate design step in the design of an entity structure; the next design step is then concerned with the assignment of behaviour components to functional entities). The constraint-oriented behaviour composition technique is also used to represent behaviours that are difficult to represent with a monolithic style. In this case the behaviour structure does not necessarily correspond to an intended entity structure, but is used for the separation of behaviour concerns (constraints, in this case). The following example illustrates a constraint-oriented composition that is often used to represent behaviours involving related actions at different locations. The same structure is also used to prepare the assignment of component behaviours to functional entities (see Figure 6.11):

B := % constraint-oriented question-answer transfer service %
{ (BQ, BT: qreq, qcnf; BA, BT: qind, qrsp),
start → BQ (entry), start → BA (entry), start → BT (entry)
where
BQ := % constraints local to "a" %

{ entry → qreq (a: Location, b: Destination, d: Data),
qreq (a: Location, b: Location, d: Data) →
qcnf (a1: Location, d1: Data) [a1 = a] }
BA := % constraints local to "b" %
(entry → qind (b: Location, a: Source, d: Data),
qind (b: Location, a: Source, d: Data) →
qrsp (b1: Location, a1: Source, d1: Data) [b1 = b, a = a1] }
BT := % constraints on the relationship between actions occurring at different locations %
{ entry → qreq (a: Location, b: Destination, d: Data),
qreq (a: Location, b: Destination, d: Data) →
qind (b1: Location, a1: Source, d1: Data) [b1 = LocDst (b), a1 = Src (a), d1 = d],
qind (b1: Location, a1: Source, d1: Data) → qrsp (b: Location, a: Source, d: Data),
qrsp (b: Location, a: Source, d: Data) →
qcnf (a1: Location, d1: Data) [a1 = LocSrc (a), d1 = d }
}

(*LocDst*, *Src*, and *LocSrc* are operations that interpret destination values as locations, locations as source values, and source values as locations, respectively.)

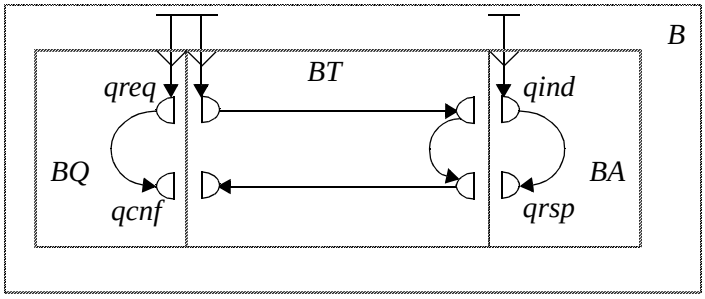


Figure 6.11: Constraint-oriented question-answer transfer service

Constraint-oriented composition and causality-oriented composition can also be combined. The following example illustrates this with the question-answer transfer service that provides any number of confirmed transfers in sequence (see Figure 6.12):

B := % constraint-oriented, repetitive question-answer transfer service %
{ (BQ, BT: qreq, qcnf; BA, BT: qind, qrsp),
start → BQ (entry), start → BA (entry), start → BT (entry)
where
BQ := % constraints local to "a" %
{ entry → qreq (a: Location, b: Destination, d: Data),
qreq (a: Location, b: Location, d: Data) → qcnf (a1: Location, d1: Data) [a1 = a],

qcnf (a1: Location, d1: Data) → BQ (entry) }
BA := % constraints local to "b" %
(entry → qind (b: Location, a: Source, d: Data),
qind (b: Location, a: Source, d: Data) →
qrsp (b1: Location, a1: Source, d1: Data) [b1 = b, a1 = a],
qrsp (b1: Location, a1: Source, d1: Data) → BA (entry) }
BT := % constraints on relationship between interactions at different locations %
{ entry → qreq (a: Location, b: Destination, d: Data),
qreq (a: Location, b: Destination, d: Data) →
qind (b1: Location, a1: Source, d1: Data) [b1 = LocDst (b), a1 = Src (a),
d1 = d],
qind (b1: Location, a1: Source, d1: Data) → qrsp (b: Location, a: Source, d: Data),
qrsp (b: Location, a: Source, d: Data) → BT (entry)
qcnf (a1: Location, d1: Data) [a1 = LocSrc (a), d1 = d] }
}

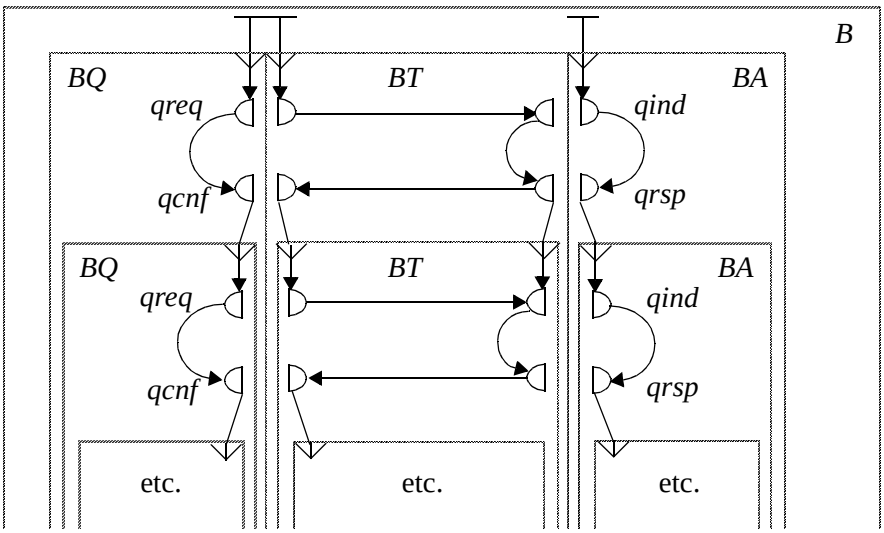


Figure 6.12: Constraint-oriented, repetitive question-answer transfer service

6.5.2 Requirements for constraint-oriented composition and decomposition

Composition requirements

The requirements that apply to constraint-oriented composition are more complex than those applicable to causality-oriented composition ('matching exit and entry points'). This is the case since requirements apply not only to the composition of interaction contributions, but also to the composition of the enabling conditions of the interaction contribu-

tions. Interaction contributions can be composed if their constraints are not mutual exclusive. Enabling conditions of interaction contributions can be composed if they do not imply conflicting ordering relations. Conflicting ordering relations exist if the component behaviours impose different ordering constraints on interactions. Examples of *violations* of these requirements are:

B1 := { ..., $\underline{a} \rightarrow \underline{b}$ (v: Integer) [v < 5], ... }
B2 := { ..., $\underline{a} \rightarrow \underline{b}$ (v: Integer) [v > 10], ... }

B1 := { ..., $\underline{b} \rightarrow c$, $c \wedge d \rightarrow \underline{a}$, ...}
B2 := { ..., $e \rightarrow \underline{a}$, $\underline{a} \rightarrow \underline{b}$, ...}

Conflicting ordering relations are visible in graphical representations as cyclic graphs that cross behaviour boundaries. Figure 6.13 depicts an example.

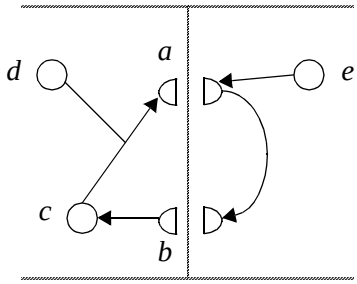


Figure 6.13: Example of constraint-oriented behaviour composition with conflicting ordering relations

In addition to the above mentioned requirements, a constraint-oriented composition should represent the behaviour intended by the designer. The interpretation of constraint-oriented composition, i.e. the relationship between a constraint-oriented behaviour composition and a monolithic behaviour composition, is expressed by (correct) decomposition requirements.

Decomposition requirements

We consider a causality relation of a monolithic behaviour and the requirements that apply to the constraint-oriented decomposition of this causality relation. We will refer to the causality relation of the monolithic behaviour as the *original* causality relation, and to the causality relations of the component behaviours that result from the decomposition as the *result* causality relations. Decomposition requirements with respect the following three cases can be distinguished:

- only (some) enabling/disabling actions in the enabling condition of the original causality relation are decomposed.

This means that the result action is not decomposed, and one of the component behaviours will contain this action. There are two requirements:

- if the enabling/disabling interaction contributions for the result action are replaced by the corresponding enabling/disabling actions, then the enabling condition of the result causality relation must be equivalent to the enabling condition of the original causality relation.

  Example:
  $a \vee b \rightarrow c$ can be decomposed as
  $\underline{a} \vee b \rightarrow c$ (component behaviour 1) and ... $\rightarrow \underline{a}$ (component behaviour 2).

- the conjunction of the constraints of enabling interaction contributions must be equivalent to the constraints of the corresponding enabling actions, and the disjunction of the constraints of disabling interaction contributions must be equivalent to the constraints of the corresponding disabling actions.

  Examples:
  a (v: Integer) [5 < v < 10] $\vee b \rightarrow c$ can be decomposed as
  $\underline{a}$ *(v: Integer) [v >5]* $\vee b \rightarrow c$ (component behaviour 1) and
  ... $\rightarrow \underline{a}$ *(v: Integer) [v < 10]* (component behaviour 2).

  $\neg$*a (v: Integer) [(v < 5)* $\vee$ *(v > 10)]* $\wedge b \rightarrow c$ can be decomposed as
  $\neg\underline{a}$ *(v: Integer) [v <5]* $\wedge b \rightarrow c$ (component behaviour 1) and
  ... $\rightarrow \neg\underline{a}$ *(v: Integer) [v > 10]* (component behaviour 2).

- only the result action is decomposed.

  Hence, the enabling/disabling actions are not decomposed, and each of them will be contained in one of the component behaviours. There are two requirements:

  - the conjunction of the enabling conditions of the result causality relations must be equivalent to the enabling condition of the original causality relation.

    Example:
    $a \wedge b \rightarrow c$ can be decomposed as
    $a \rightarrow \underline{c}$ (component behaviour 1) and $b \rightarrow \underline{c}$ (component behaviour 2).

  - the conjunction of the constraints of the result interaction contributions must be equivalent to the constraints of the result action of the original causality relation.

    Example:
    $a \wedge b \rightarrow c$ *(v: Integer) [5 < v < 10]* can be decomposed as
    $a \rightarrow \underline{c}$ *(v: Integer) [v > 5]* (component behaviour 1) and
    $b \rightarrow \underline{c}$ *(v: Integer) [v < 10]* (component behaviour 2).

- both (some of) the enabling/disabling actions and the result action are decomposed.

  In this case, all the above requirements apply.

Example:
a (v1: Integer) [v1 = 7] → b (v2: Integer) [5 < v2 < 10] can be decomposed as
a (v1: Integer) [v1 < 8] → b (v2: Integer) [v2 > 5] (component behaviour 1) and
a (v1: Integer) [v1 > 6] → b (v2: Integer) [v2 < 10] (component behaviour 2).

Figure 6.14 depicts the graphical representation of the examples.

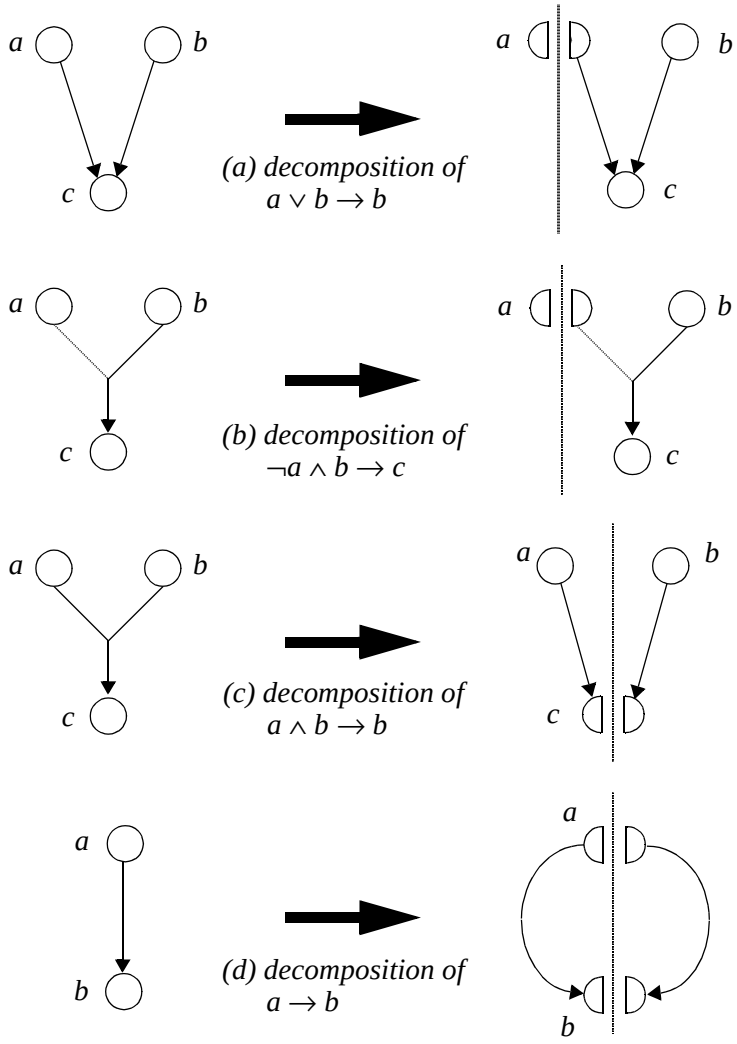


Figure 6.14: Examples of constraint-oriented decomposition

Decomposition freedom

Given a monolithic behaviour, the designer has considerable freedom when decomposing this behaviour into a constraint-oriented behaviour. Instead of systematically considering all possibilities of decomposition for different types of causality relations. Figure 6.15 illustrates some further decomposition possibilities of the causality relation $a \wedge b \rightarrow c$ (see

also Figure 6.14(c); these figures do not represent all possible valid decompositions). An exhaustive discussion of the decomposition possibilities of basic causality relations can be found in [Ferreira94].

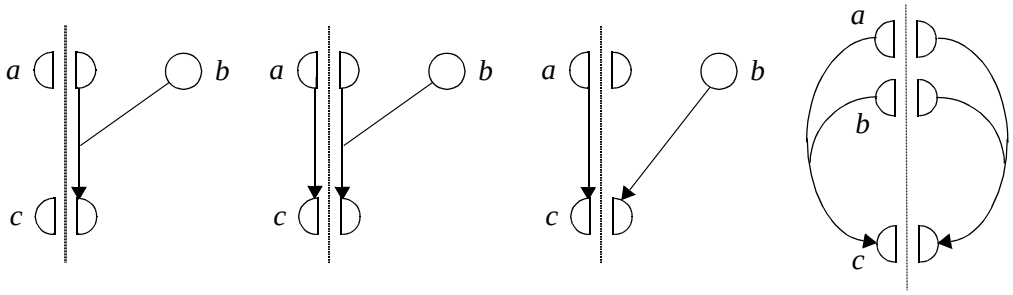


Figure 6.15: Four possible decompositions of $a \wedge b \rightarrow c$

6.6 Power of expression

This section investigates the power of expression of our design model. We will consider the representation of general behaviour patterns and the representation of behaviour according to recognized specification styles. General behaviour patterns are patterns of behaviour that appear very often in distributed system behaviours. Specification styles have been developed to help the designer with the task of producing high quality designs.

6.6.1 General behaviour patterns

Specification languages (e.g., LOTOS, Estelle, and SDL) often define separate operators to support different kinds of general behaviour patterns (in this way, more concise specifications can be produced). We will investigate the representation of the action and behaviour patterns that were considered important and general enough to deserve special support in most languages: sequence (LOTOS operator: ”;” or ”>>”), concurrency (synchronized or independence; ‘modelled’ with LOTOS operator: "||" or "|||"), disabling (LOTOS operator: "[>"), choice (LOTOS operator: "[]"), and interleaving (LOTOS operator: "|||").

Sequence

Two actions are composed in sequence, or sequentially ordered, if one action can only occur after the other. This implies that a causality relation exists between these two actions. For example, the sequence of actions *a1* and *a2* is represented by:

{ ..., start → a1, a1 → a2, ... }

Two behaviours are composed in sequence if the first action of one behaviour can only occur if the last action of the other behaviour has occurred. Hence an appropriate causal relationship must exist between these two behaviours. This can be expressed with a causality relation that defines an exit point of one behaviour as a condition for an entry point for the other behaviour. For example, the sequence of two behaviours *B1* and *B2* is represented by:

{ ..., start → B1 (entry), B1 (exit) → B2 (entry), ... }

where the exit point of *B1* is enabled by the last action of *B1*, and the entry point of *B2* is the enabling condition for the first action of *B2*.

Figure 6.16 depicts the representation of sequential ordering.

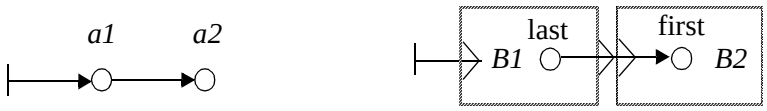


Figure 6.16: Sequence of two actions a1 and a2, and of two behaviours B1 and B2

Concurrency

Two interaction contributions are synchronized if they can only occur together, forming an interaction. Two behaviours are synchronized if they have common interactions.

Two actions are independent of each other if no relationship exists between them. Thus these actions may not be related through a (sequence of) causality relation(s). The independency of *a1* and *a2* is represented by:

{ ..., start → a1, start → a2, ... }

Two behaviours are independent if they are not synchronized (i.e., they have no common interactions) and no relationships exists between their actions. The latter implies that no entry/exit constructs are defined involving these behaviours. The independency of *B1* and *B2* is represented by:

{ ..., start → B1 (entry), start → B2 (entry), ...}

where *B2* has no further entry points that depend on exit points of *B1*, and conversely.

Figure 6.17 depicts the representation of independency.

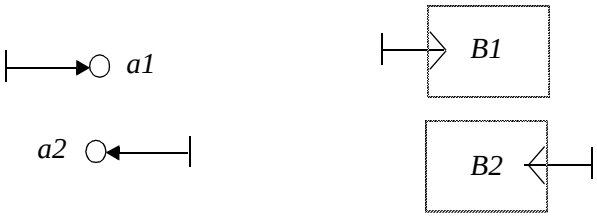


Figure 6.17: Independency of two actions a1 and a2, and of two behaviours B1 and B2

Disabling

An action is disabled by another action if the occurrence of the latter action, the disabling action, prevents the occurrence of the former action, the disabled action. A conflict relation must thus exist between these actions, such that the non-occurrence of the disabling action is a condition for the occurrence of the action that can be disabled. The possible disabling of *a2* by *a1* is represented by:

$\{ ..., start \rightarrow a1, start \wedge \neg a1 \rightarrow a2, ... \}$

One form of disabling applied to behaviours is that the completion of a disabling behaviour prevents another behaviour to start, i.e. the occurrence of the last action of a disabling behaviour prevents the occurrence of the first action of the disabled behaviour. The possible disabling of *B2* by *B1* according to this form is represented by:

$\{ ..., start \rightarrow B1\ (entry), start \wedge B1\ (exit) \rightarrow B2\ (entry), ... \}$

where the exit point of *B1* is disabled by the last action of *B1* and the entry point of *B2* enables the first action of *B2*.

We may also consider another form of disabling where the start of a disabling behaviour prevents the start of another behaviour or, in case this behaviour is already in progress, interrupts this other behaviour[1]. In other words, the non-occurrence of a first action of the disabling behaviour is a condition for the occurrence of any action of the behaviour that can be disabled. The possible disabling of *B2* by *B1* in this case is represented in the same way as above, however the exit point of *B1* is enabled after the first action of *B1* and the entry point of *B2* is a necessary enabling for any action of *B2*.

Figure 6.18 depicts the representation of these disabling forms.

1. This is the semantics of the LOTOS disabling operator.

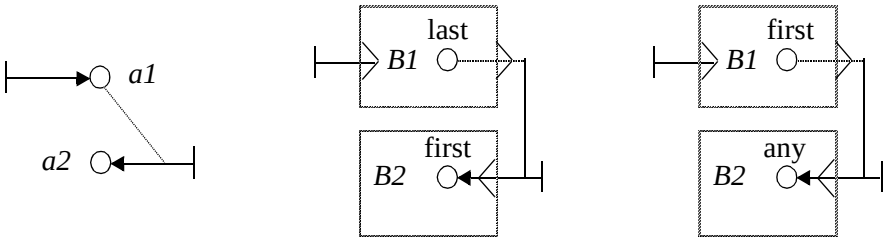


Figure 6.18: Disabling of action a2 by action a1, and of behaviour B1 by behaviour B2 (two forms)

Choice

A choice between two actions means that either of the two actions may occur and that occurrence of one of the actions prevents the occurrence of the other action. This implies the existence of a symmetric conflict between these actions, represented by two conflict relations. A choice between *a1* and *a2* is represented by:

{ ..., start $\wedge \neg$*a2* $\rightarrow$ *a1, start* $\wedge \neg$*a1* $\rightarrow$ *a1, ... }*

A choice between two behaviours means that only one of these behaviours may start and that the start of one of the behaviours prevents the start of the other behaviour. Consequently, the non-occurrence of a first action of one behaviour is a condition for the occurrence of a first action of the other behaviour, and conversely. The choice between *B1* and *B2* is then represented by:

{ ..., start $\wedge$ *B2 (exit)* $\rightarrow$ *B1 (entry), start* $\wedge$ *B1 (exit*) $\rightarrow$ *B2 (entry), ... }*

where the exit point of *B2* is disabled by the first action of *B2* and the entry point of *B2* enables the first action of *B2*. Similar requirements apply to the exit point and entry point of *B1*.

Figure 6.19 depicts the representation of choice.

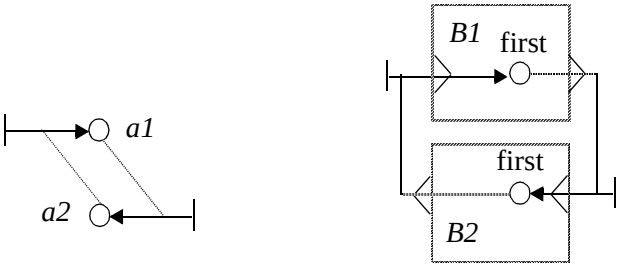


Figure 6.19: Choice between actions a1 and a2, and between behaviours B1 and B2

Interleaving

Two actions are interleaved if they can occur in any order, but not at the same time. The interleaving of two actions can thus be expressed as a choice between two different sequences of these actions. By defining the different sequences as behaviours, interleaving of two actions can be represented as the choice between two behaviours (see above). A direct representation of the interleaving of two actions *a1* and *a2* is:

$\{ ..., start \wedge (a2 \vee \neg a2) \rightarrow a1, start \wedge (a1 \vee \neg a1) \rightarrow a2, ... \}$

Two behaviours are interleaved if actions of different behaviours cannot occur at the same time. A general representation in terms of exit/entry constructs is however not straightforward and will therefore not be considered here. If this pattern will be frequently used, it is probably necessary to introduce an additional concept that enables its direct representation.

Figure 6.20 depicts the representation of interleaving of two actions and of two behaviours, each consisting of a single action.

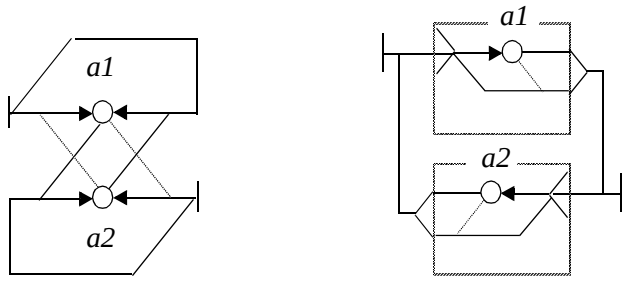


Figure 6.20: Interleaving of actions a1 and a2, and of two behaviours, each consisting of a single action (a1 and a2, respectively)

Remark on interleaving

Interleaving is much more difficult to represent with our model (and notation) than independency or synchronization. This may be justified from an implementation point of view: interleaving, as a requirement, is difficult to implement if the behaviours run on different machines (a common situation in a distributed environment), while independency comes for free in that case. If interleaving is locally required, we believe that this requirement will more likely apply to some specific actions than to complex behaviours.

6.6.2 Specification styles

The advantages of adhering to well defined specification styles during design was first discussed in [Vissers88]. Four different specification styles are proposed in [Vissers88]: the monolithic, constraint-oriented, resource-oriented, and state-oriented style. In retro-

spect we can say that one of the important achievements of these styles is that they define a mapping between the entity domain and the behaviour domain at different abstraction levels. Applications of these styles in various specification projects have proved their value in practice (see [Sinderen89], [Sinderen91], [Widya93], and [Turner95], for example). Later on, also other styles of specification were proposed, but these styles appear less fundamental than the above mentioned. Because of this we only investigate the support of the afore mentioned styles.

Monolithic style

This style is used to directly express the actions (interaction contributions) of a system and their causal relationships, and to ignore possible structures of (the behaviour of) the system. In our model, this style is enforced if a behaviour is specified in a single, monolithic behaviour definition.

Constraint-oriented style

This style is used to express the external behaviour of a system as a composition of component behaviours, each of which defines a distinguished set of constraints on the external behaviour. Constraints are represented as constraints on attribute values of interaction contributions and conditions for their causal relationships. The constraint-oriented style is directly supported by the constraint-oriented composition technique of our model.

Resource-oriented style

This style is used to express internal behaviour of a system as a composition of component behaviours, each of which defines a distinguished function of the system. Functions are represented as external behaviours of entities (resources) from which the system is composed. The resource-oriented style is again directly supported by the constraint-oriented composition technique of our model. Actually, we are able to apply this style in both the entity domain and the behaviour domain, resulting in a process-oriented style (logical functions in the behaviour domain) next to a resource-oriented style (assigned functions in the entity domain).

State-oriented style

This style is used to express the state space of a system. In most cases the state-oriented style is used in the late phases of the design process, when no further structuring of the system is required[1], and a representation is needed that best suites implementation practices. Since a state is determined by the past behaviour of a system, and in turn determines

1. A system that is specified in the state-oriented style therefore usually corresponds to a resource identified in a resource-oriented style specification (produced in an earlier phase of the design process).

possible future behaviour of the system, states are represented in our design model by enabling conditions with disabling actions removed. Thus, the collection of states of a system corresponds to the collection of different enabling conditions that result from removing the disabling actions from enabling conditions. It is also possible to define 'macro' states (collections of related states) with the causality-oriented composition technique of our model. Each exit/entry construct then corresponds to a possible transition from one macro state to another.

6.7 Behaviour refinement

The application protocol design trajectory, presented in Chapter 5 (Design Framework), consists of a sequence of design steps. Each design step is concerned with transforming a given design (constituting a milestone in the design trajectory) into a more concrete design (constituting another milestone, corresponding to a lower abstraction level). Design steps must preserve the characteristics of the system captured by the given design, and consider the refinement of these characteristics, or the inclusion of additional characteristics.

This section discusses design steps in the behaviour domain. We will call such design steps behaviour refinements. First some different types of behaviour refinements are identified which are considered useful in the application protocol design trajectory. Subsequently, requirements for correct refinement are presented for two such refinement types.

6.7.1 Identification of refinement types

We consider the design steps with the following transformation aims:

- from application *service* to application *service provider* external behaviour.

  In this step, the service behaviour is decomposed, such that the responsibility of service users at one hand and the responsibility of a service provider at the other hand are separated. This separation can be accomplished by *constraint-oriented decomposition* of the service behaviour (see Figure 6.21). The resulting behaviour should be a composition of component behaviours that reflect service user constraints, service provider constraints, and shared constraints (i.e., local constraints not distributed over service users and service provider, corresponding to shared abstract interfaces) on the service behaviour.

- from application service provider *external* behaviour to application service provider *internal* behaviour.

  In this step, internal actions are introduced that implement the causal relationship between interactions of the service provider with the service users. The aim of this step is to determine a *distributed* processing activity of the service provider at the highest

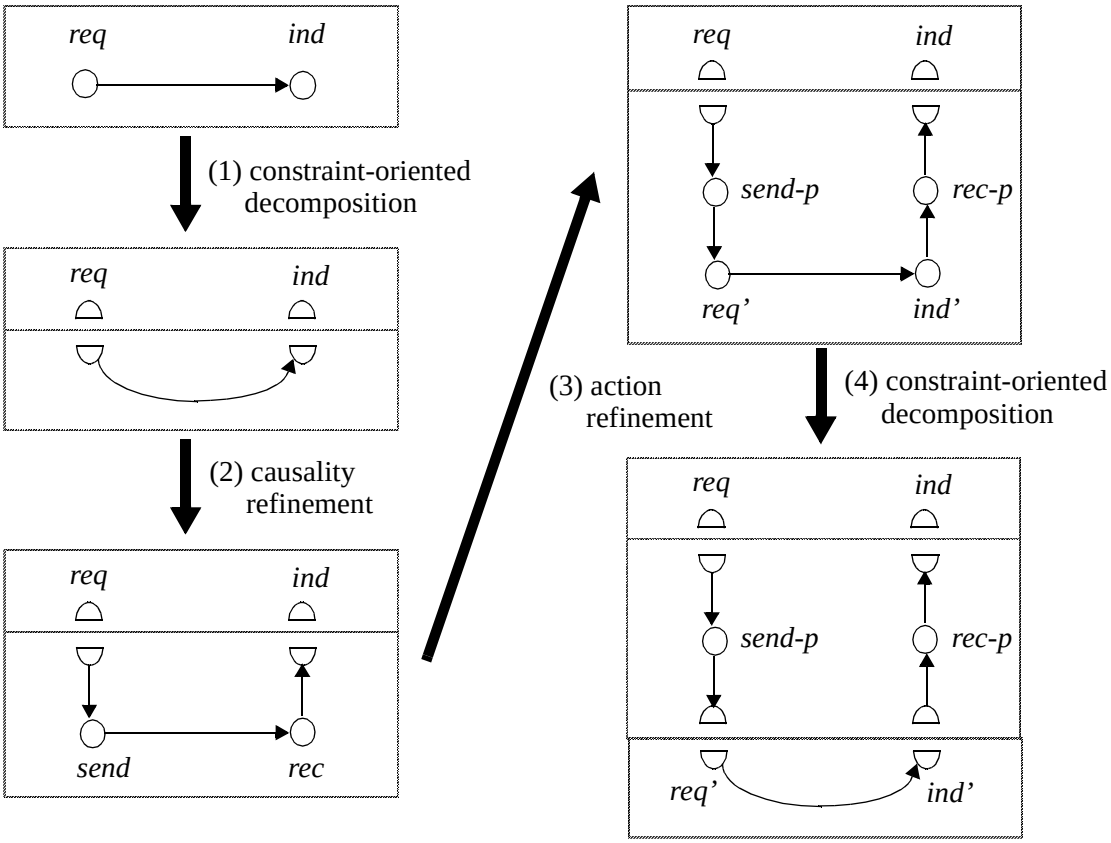


Figure 6.21: Illustration of refinement types

possible level of abstraction. Each internal action represents a localized contribution to this distributed processing activity. Furthermore, internal actions are not directly related to other, collocated internal actions (since this would constitute a possible implementation of a localized contribution), but only to internal actions that occur at remote locations. Since the interactions (interaction contributions) of the given behaviour are preserved in this step, while their relationships are refined, we will refer to this type of refinement as *causality refinement* (see Figure 6.21).

- from application service provider *internal behaviour* to a refined internal behaviour, here referred to as the application *protocol behaviour.*

  In this step, the internal actions in the given behaviour are decomposed into protocol actions and lower level service primitives. Protocol actions are actions that can be assigned to local protocol processing behaviours, while lower level service primitives are actions, or rather integrated interactions, that can be assigned to a lower level service. Since actions are replaced by compositions of actions, we will refer to this type of refinement as *action refinement* (see Figure 6.21). (The protocol behaviour can be structured to reflect the assignment of protocol actions and lower level service

primitives to local protocol processing behaviours and a lower level service, respectively. This composition can be derived, after the application of action refinement, with *causality-oriented decomposition*. More practical, however, is to apply constraint-oriented decomposition instead, as is discussed next. The practicality of this refinement is demonstrated in the Chapter 7, Application Protocol Reference Architecture.)

- from application *protocol behaviour* to an application *protocol architecture*.

  In this step, the protocol behaviour is decomposed, such that the responsibility of protocol entities at one hand and the responsibility of a lower level service provider at the other hand are separated. As with the first mentioned design step, *constraint-oriented decomposition* is applied to accomplish this separation (see Figure 6.21). The constraints separated are protocol entity constraints, lower level service provider constraints, and shared constraints (i.e., local constraints not distributed over protocol entities and lower level service provider, corresponding to shared abstract interfaces). The possibility to assign (some) constraints to entities means that a distributed service provider can be defined.

Figure 6.22 illustrates the relation between these behaviour refinement types and design steps in the lower part of the application protocol design trajectory. It also lists the changes in the entity domain that are related by the behaviour refinements.

It is not always possible to relate specific refinement types to particular design steps of the application protocol design trajectory. The above refinement types, and additional ones, will also be used to support more general design concerns, not related to specific design steps. For example:

- to structure complex behaviours. This may be done in any design step. Since a structured behaviour is, in this case, usually not derived from a monolithic behaviour (the behaviour is too complex to be represented in this way), it is composed rather than produced through refinement. Causality-oriented (de)composition (separation of relatively self-contained units of behaviour) or constrain-oriented (de)composition (separation of distinct constraints on a behaviour) can be used for this purpose.
- to introduce the representation of characteristics of a behaviour that were previously not considered. This requires the introduction of additional attributes of actions of the behaviour. We will call this type of refinement *action enrichment*. Action enrichment may be used in early phases of the design process, e.g. to introduce ‘location’, or in subsequent cycles, e.g. to introduce ‘absolute time’.
- to introduce additional constraints on actions. This may be done in design steps where technical constraints of systems that have to implement some behaviour are considered. We call this type of refinement *action specialization*. Action specialization may be applied before, or in combination with constraint-oriented decomposition, when a

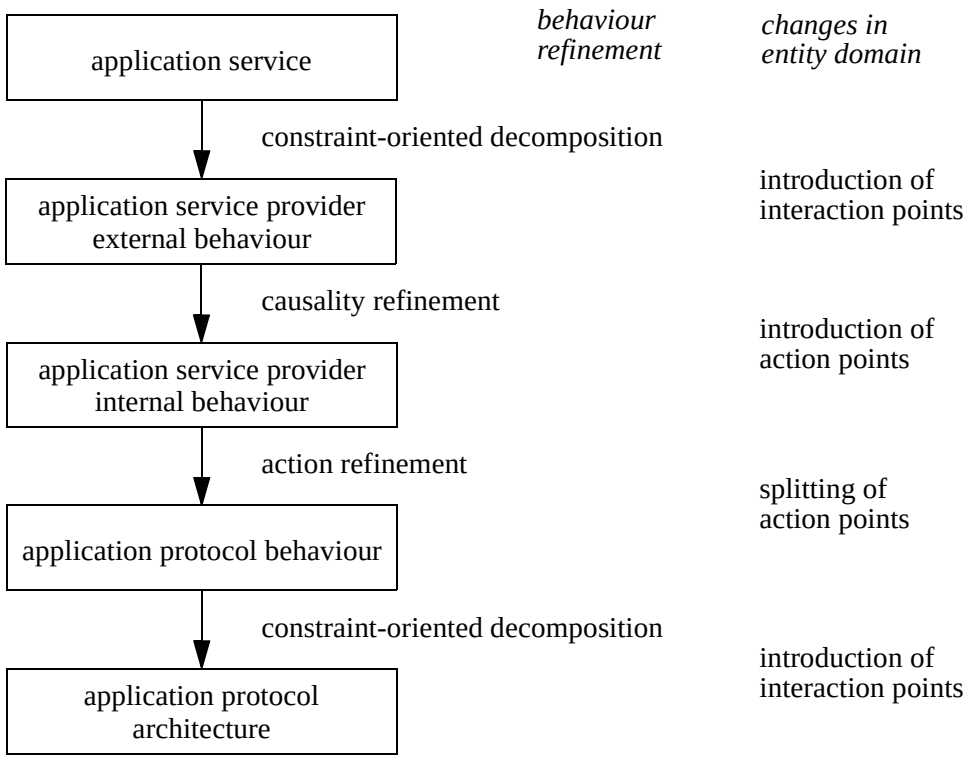


Figure 6.22: Relation between refinement types and design steps in the lower part of the application protocol design trajectory

separation of the responsibilities, hence also of the capabilities, of functional entities is considered.

We will not further consider action enrichment and action specialization in this thesis.

6.7.2 Refinement requirements

Causality-oriented (de)composition and constraint-oriented (de)composition have been discussed in section 6.4 and section 6.5, respectively. Requirements for correct causality refinement and action refinement have been developed in [Ferreira94] (causality refinement is termed behaviour refinement in [Ferreira94]). We will summarize these requirements below, in order to allow their application in the next chapter (Chapter 7, Application Protocol Reference Architecture).

Requirements apply to the relationship between a given, abstract behaviour and the refined behaviour that is obtained by behaviour refinement. One should be able to deduce the abstract behaviour by abstracting from the refined behaviour according to some defined method.

Causality refinement

The following steps define a method to determine the correctness of an application of causality refinement:

- abstract from references to inserted actions and their attribute values that appear in the enabling conditions of other actions of the refined behaviour;
- (possibly) simplify the enabling conditions of causality relations obtained; and
- repeat these steps, unless a behaviour without inserted actions has been obtained.

The behaviour that is finally determined by this method must be equal to the original abstract behaviour.

Action refinement

Two types of requirements exist with respect to correct action refinement:

- conformance between the activity obtained by action refinement and the corresponding abstract action; and
- proper embedding of the activity in the context of the abstract action.

The first requirement is verified as follows:

- the abstract action establishes the same information values as the activity;
- the abstract action has the same functionality values as the activity;
- the location of the abstract action must be an abstraction of the locations of actions of the activity, or all actions of the activity must have the same location as the abstract action.

(Information and functionality) values of an activity are values of actions of the activity that are referenced by actions outside the activity.

Proper embedding of the activity is verified by determining the condition for the completion of the activity, in terms of actions outside the activity. This condition must correspond to the enabling condition of the abstract action. The following steps define a method to determine this condition (here, we do not consider disabling actions):

- identify the final actions of the activity (i.e., actions referenced by actions outside the activity) and determine the conjunction of the enabling conditions for these actions;
- (possibly) simplify the resulting condition;
- if the resulting condition contains enabling actions that are part of the activity, then replace these actions by their enabling conditions;

- repeat the second and third step, unless a condition is obtained with enabling actions that are all part of the context of the abstract action.

6.8 Conclusion

We have defined the concepts of action, interaction and causality relation, which are the elementary architectural concepts on which our basic design model is based.

A unit of activity is represented by an interaction or action, depending on whether or not we want to consider the distribution of this activity over different functional entities. Different relevant characteristics of the activity are represented by attributes of the (inter)action. Identified attributes include the location attribute, the time attribute, the information attribute, and the functionality attribute. Possible attribute values are determined by constraints of the (inter)action. In the case of an interaction, constraints are distributed over interaction contributions.

The relationships, or dependencies, between (inter)actions are represented by causality relations. A causality relation defines what (inter)actions must have occurred and which must not have occurred in order to enable another (inter)action. Monolithic behaviour can be represented by a collection of causality relations.

We have identified two techniques for composing structured behaviour. The causality-oriented behaviour composition technique allows to compose behaviours by defining causality relations between component behaviours. The constraint-oriented behaviour composition technique allows to compose behaviours by defining shared (inter)actions between component behaviours.

Considering the fact that the definition of concepts has been based on the requirements that emerged from the design framework discussed in the previous chapter (Chapter 5, Design Framework), we believe that our design model allows the representation of all milestones in the application protocol design trajectory. We investigated the ability of the model to represent general behaviour patterns and specification styles. This investigation showed that general behaviour patterns can be represented, although in particular ‘disabling’ may require the introduction of some shorthand notation. ‘Independency’, on the other hand, is particularly easy to represent with our model. Furthermore, the monolithic, causality-oriented, and constraint-oriented composition techniques are well suited to support recognized specification styles.

Furthermore, the design steps of the application protocol design trajectory have been investigated to identify different types of useful behaviour refinements. Three important refinement types are identified: constraint-oriented decomposition, causality refinement and action refinement. It is in general not possible to provide algorithms for behaviour refinement (that can be automated). However, it is possible to define requirements on the

relationship between an abstract behaviour and correct refinement of that behaviour. Such requirements are presented for the three refinement types.

References

[Ferreira93] Ferreira Pires, L., Sinderen, M. van, and Vissers, C.A., Advanced design concepts for open distributed systems development, In: *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, IEEE Computer Society Press, 1993, 419-425.

[Ferreira94] Ferreira Pires, L., *Architectural notes: a framework for distributed systems development*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 1994.

[Gunawardena92] Gunawardena, J., Causal automata, *Theoretical Computer Science*, 101:265-288, 1992.

[Sinderen89] Sinderen, M. van, Ajubi, I., and Caneschi, F., The application of LOTOS for the formal description of the ISO session layer, In: *Formal Description Techniques, I*, Turner, K.J. (editor), Elsevier Science Publishers B.V. (North-Holland), 1989, 263-277.

[Sinderen91] Sinderen, M. van, and Widya, I., On the design and formal specification of a transaction processing protocol, In: *Formal Description Techniques, III*, Quemada, J., Mañas, J., and Vázques, E. (editors), Elsevier Science Publishers B.V. (North-Holland), 1991, 411-426.

[Sinderen95] Sinderen, M. van, Ferreira Pires, L., Vissers, C.A., and Katoen, J.-P., A design model for open distributed processing systems, To appear in: *Computer Networks and ISDN Systems*, special issue on Open Distributed Processing. (Also available as: *Memorandum Informatica 94-27*, University of Twente, Enschede, The Netherlands, 1994.)

[Turner95] Turner, K.J., and Sinderen, M. van, LOTOS specification style for OSI, In: *LOTOSphere: software development with LOTOS*, Bolognesi, T., Lagemaat, J. van de, and Vissers, C.A. (editors), Kluwer Academic Publishers, 1995, 137-160.

[Vissers88] Vissers, C.A., Scollo, G., and Sinderen, M. van, Architecture and specification style in formal descriptions of distributed systems, In: *Protocol Specification, Testing, and Verification, VIII*, Aggerwal, G. and Sabnani (editors), Elsevier Science Publishers B.V. (North-Holland), 1988, 189-204.

[Vissers93] Vissers, C.A., Sinderen, M. van, and Ferreira Pires, L., What makes industries believe in formal methods, In: *Protocol Specification, Testing, and Verification, XIII*, Danthine, A., Leduc, G., and Wolper, P. (editors), Elsevier Science Publishers B.V. (North-Holland), 1993, 3-26.

[Widya93] Widya, I., and Heijden, G.-J. van der, Towards an implementation-oriented specification of the TP protocol in LOTOS, In: *Formal Methods Europe '93*, Woodcock, J.C.P., and Larson, P.G. (editors), Springer-Verlag, Lecture Notes in Computer Science 670, 1993, 93-109.

Chapter 7

Application Protocol Reference Architecture

This chapter proposes an alternative reference architecture for application protocols. The proposed reference architecture consists of the set of possible architectures for application protocols. This set is implicitly defined by defining the different types of application protocol components and their relationships. The component types and their relationships are derived by application of the design trajectory, design methods and structuring techniques presented in the previous chapters. This chapter also identifies and characterizes some generic application protocol building blocks that can be used as components in many different application protocol architectures.

The main purpose of a reference architecture is to help the designer in choosing a suitable architecture for his specific design, and to incorporate pre-defined, generic building blocks. A reference architecture should therefore either define a single architecture that is generally applicable in the design domain at hand, or a set of architectures, each appropriate in a specific sub-domain. Because of the diversity of interaction requirements of different classes of distributed applications, we do *not* believe that it is possible to define a *single* useful architecture for application protocols. Such a reference architecture would unnecessarily constrain the designer in many specific design instances, or it would be too coarse to be of use in composing pre-defined building blocks. A flexible reference architecture for application protocols should define *all* possible architectures at the level of abstraction that allows the positioning of pre-defined building blocks.

The structure of this chapter is as follows: section 1 discusses the purpose of the proposed reference architecture; section 2 presents the top level structure of the reference architecture. This structure is based on the principal structuring decisions incorporated by the application protocol design trajectory; section 3 through section 5 discuss the design of an application protocol in general terms, such that the structures developed are generic structures and the design options indicated correspond to alternative generic structures. The alternative generic structures that result from this design process represent the alternative architectures of the reference architecture; section 3 discusses the design of an application service; section 4 discusses the design of an integrated application service provider; and section 5 discusses the design of a distributed application service provider;

section 6 investigates a generic function for flexible PDU coding, and presents general purpose data transfer functions; section 7 presents some application protocol building blocks; section 8 compares the proposed application protocol reference architecture with the OSI-ULA; and finally, section 9 presents the conclusions of this chapter.

7.1 Purpose of the reference architecture

An application protocol reference architecture should help the designer in choosing a suitable architecture for his specific design. It should be *flexible* enough to be of use independently of the varying requirements of different classes of distributed system applications. Furthermore, it should support optimal *re-use* of already defined (e.g., standardized) application protocol building blocks, as well as support the *extension* of the set of predefined building blocks.

The OSI-ULA is basically a static reference architecture because it enforces the use of three application protocol layers. We believe that no static architecture can appropriately support all current distributed applications, let alone future applications with their continuously expanding and evolving requirements (similar arguments have been used by several authors that propose flexible protocol architectures; see [Tschudin91], [O'Malley92], [Solvie92], and [Box93], for example). The flexibility offered by the dynamic Application Layer Structure (ALS) is not sufficient, since it still enforces the use of the Presentation and Session Layer. We do not propose a single-layer architecture. Any fixed number of layers, with fixed relationships, will be appropriate to some classes of distributed applications, but inappropriate to others. There is no right number of layers in the general case (see Figure 7.1). Any static architecture for application protocols will be a compromise solution that is not optimal in most specific situations. An application protocol architecture should therefore provide the flexibility to compose layers and layer hierarchies dependent on the application service that is required (i.e., the class of distributed applications that must be supported).

The inappropriateness of the static three-layer approach of the OSI-ULA was demonstrated in Chapter 4 (OSI Upper Layer Architecture and Model: Evaluation). Also alternative architectures were presented in this chapter, based on the application of the dynamic layer approach, as used in the Application Layer, to all three application protocol layers. This discussion was, however, limited to a subset of the current application protocol standards. In order to develop the application protocol reference architecture in this chapter, we take another approach. We study the design of an application protocol architecture in general terms, starting from an (arbitrary) application service, and using:

- the design methods presented in Chapter 4 (OSI Upper Layer Architecture and Model: Evaluation);
- the structuring techniques and refinement types presented in Chapter 6 (Design Model);

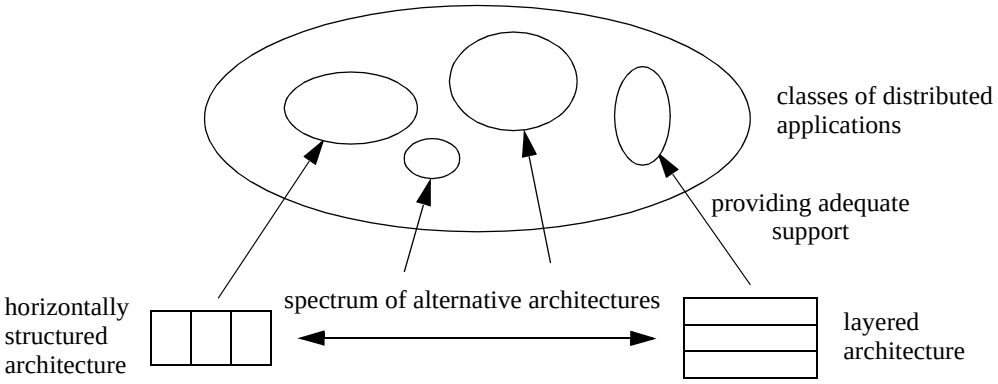


Figure 7.1: Alternative application protocol architectures satisfying different distributed application requirements

- within (the lower part of) the design trajectory presented in Chapter 5 (Design Framework).

This approach allows the identification of (alternative) structures characterized by different types of application protocol components and their relationships, which implicitly defines the set of possible application protocol architectures.

Figure 7.2 illustrates the use of an application protocol reference architecture consisting of a collection of possible architectures. A designer can choose a suitable architecture for his specific design, use the set of elementary design concepts of the design model to define a specific instance of this architecture, and incorporate pre-defined building blocks.

7.2 Top level structure of distributed application

The application protocol design trajectory identifies a number of milestones in the application protocol design process. As argued in Chapter 5 (Design Framework), the starting point for the design process can be chosen at higher or lower levels of abstraction, depending on the objective of the designer and the design boundary conditions. For example, if the introduction of distributed computer system support in an organization may impact the organizational structure of that organization, the starting point for design may be an enterprise service.

We assume a special purpose (to the needs of a specific user environment) application service, to be provided by a distributed computer system, as the starting point for design. The nature of the special purpose application service may be such that first the middle part of the application protocol design trajectory is followed (see Figure 7.3). This means that

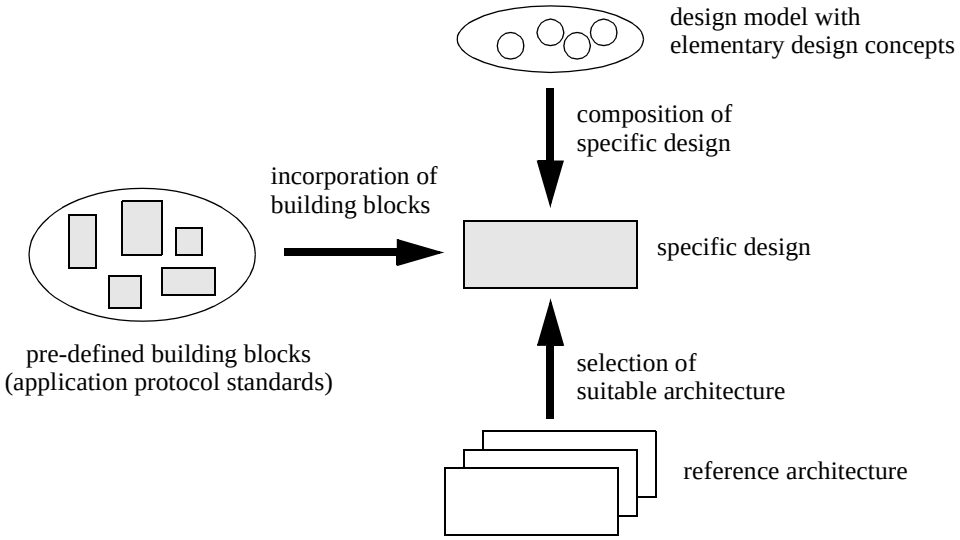


Figure 7.2: Use of a flexible application protocol reference architecture

first system parts are designed corresponding to information processing entities in a distributed processing architecture. Some of the information processing entities constitute distributed systems which are best implemented by a protocol architecture. These entities are again considered as application service providers, but their implementation, in terms of application protocols, is derived by following the lower part of the design trajectory.

Consequently, an important decision in the design trajectory is choice of the application service (provider) which forms the initial milestone of the lower part of the application protocol design trajectory. This milestones constitutes a common concern of two design approaches used in the design trajectory, viz. system part design (the middle part) and interaction system design (the lower part).

Application services that are considered for standardization must be generic, i.e. they must support a sufficiently wide class of distributed applications. Another important design decision is therefore the choice of a *gener*ic application service. The level of generality should justify standardization. Identification of generic application services corresponding to already standardized application protocols is important in any application protocol design (not limited to the development of new application protocol standards), since these protocols can be incorporated as building blocks.

Yet another choice is that of a suitable data transfer service. This choice is made in the last step of the lower part of the application protocol design trajectory. It constitutes a milestone that separates application protocol design and data transfer protocol design.

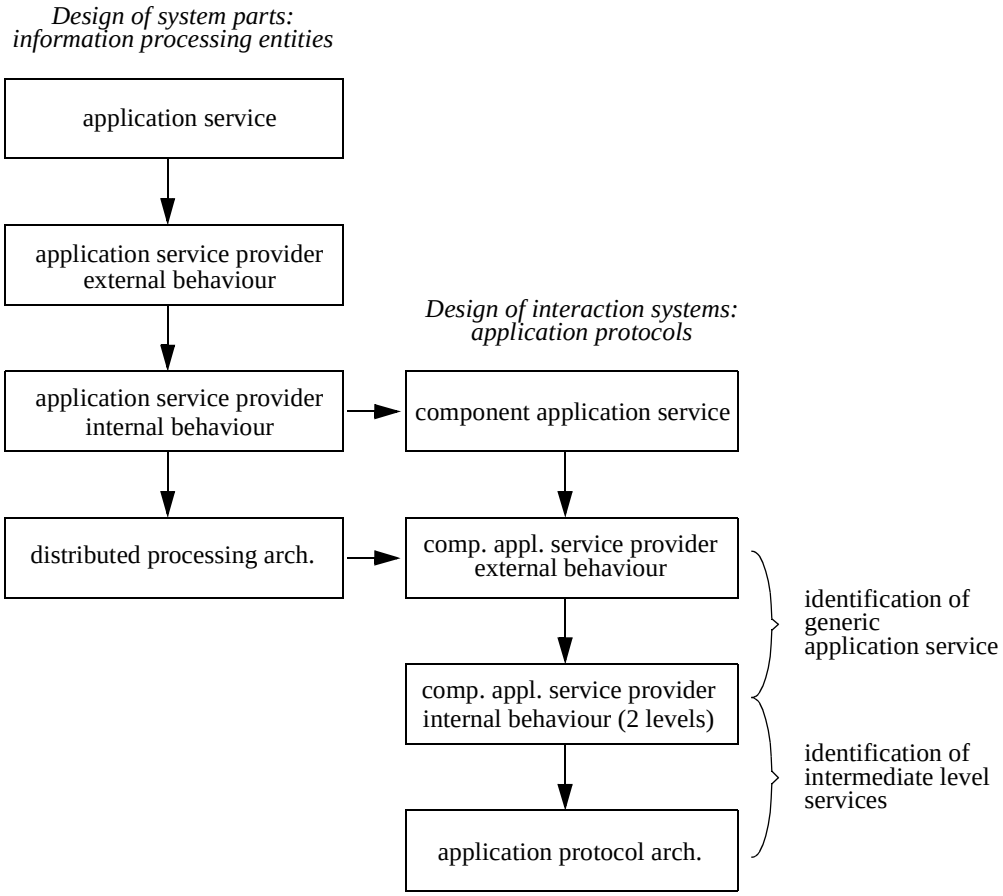


Figure 7.3: Milestones in the middle and lower part of the application protocol design trajectory

These three choices lead to the top level structure of a distributed application service provider, as shown in Figure 7.4. The structure shown can be varied in a number of ways:

- a protocol, whether user-defined or standardized, can be multi-peer (i.e., defining distributed interactions involving three or more protocol entities) instead of peer-to-peer;
- a protocol entity does not always interact with a higher level entity (protocol entity or information processing entity). However, at least one protocol entity involved in a protocol should interact with a higher level entity (otherwise the protocol would not play any useful role in the application service provider). In general, both protocol entities involved in a standardized peer-to-peer application protocol interact with a user-defined protocol entity (because the interactions pass user data).

- an information processing entity does not necessarily interact with another information processing entity, or with a human user. Also, an information processing entity does not necessarily interact with a user-defined protocol entity. It should, however, interact with at least one other entity, of any type (otherwise it would not play any useful role in the application service provider).

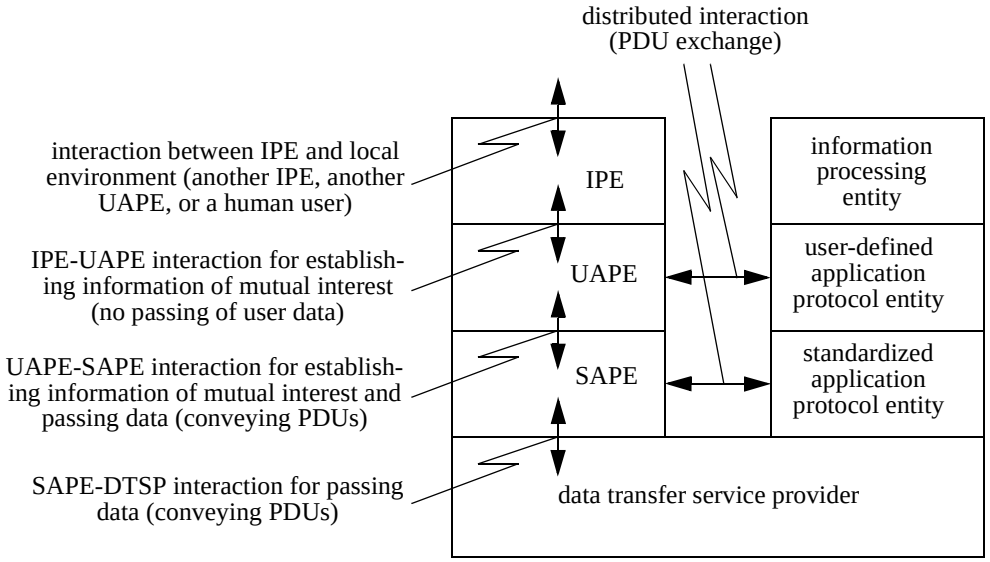


Figure 7.4: Top level structure of a distributed application service provider

An application protocol can be further structured using the design methods presented in Chapter 4 (OSI Upper Layer Architecture and Model: Evaluation): layer protocol design, and ASE protocol design. Layer protocol design requires the choice of intermediate level application services (intermediate with respect to the required application service and the data transfer service). Figure 7.5 depicts the method for developing layered application protocols, in terms of steps of the lower part of the application protocol design trajectory. This is further discussed in section 7.5. ASE protocol design requires the choice of subsets of service functions, corresponding to ASE service at each service level. Because we want to use this design method at arbitrary levels, we will use the term *section* instead of ASE in the following, to avoid confusion with OSI Application Layer terminology. Thus, a *service section* denotes a part (subset) of a service, and a *protocol section* denotes a part of a (horizontally structured) protocol. Application protocol section design is further discussed in section 7.4.

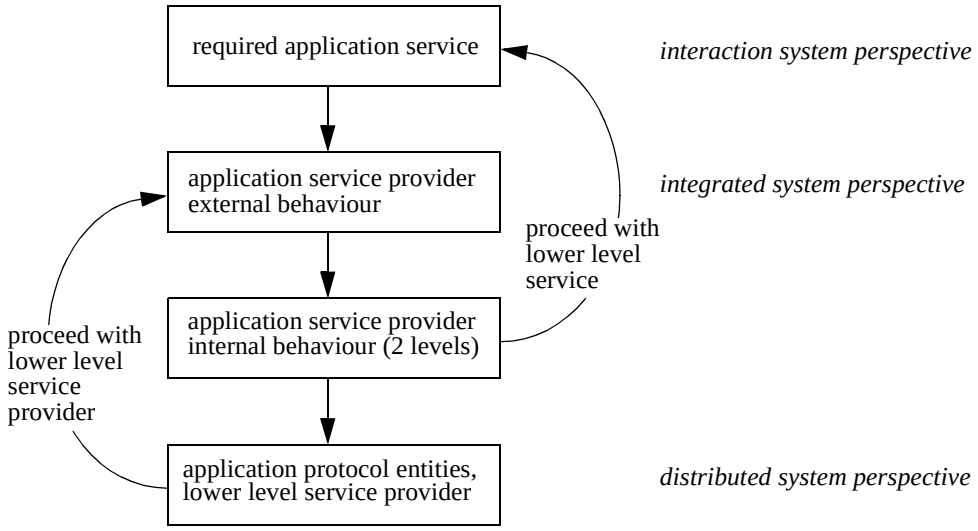


Figure 7.5: Recursive application of design steps for the design of layered application protocols

7.3 Design of an application service

A generic application service must define a behaviour which is useful in different specific application service environments. In other words, a generic application service must be able to support a *class* of distributed applications. This section discusses the implications of generality for application service definitions. It also presents two ways of structuring service definitions, both of which can be used as a preparation for protocol structuring. Finally, this section discusses the use of quality criteria for determining the level of generality and the choice of structure components.

7.3.1 Generality

One way to advance generality is by the inclusion of a *user data* parameter in service primitives. User data represents information defined at a higher protocol level, which is *transparently transferred* between the locations bound by the service. Transparent transfer means that the information is not interpreted or changed by the protocol supporting the service. Different higher level protocols can thus use user data parameters in different ways to support different classes of distributed applications.

Another way of increasing generality is by inclusion of *options*. Options represent aspects of the service behaviour, either functional or non-functional (e.g., optional QoS

parameters), that can be selected. A service comprising functional options can be defined in terms of a kernel functionality and a set of optional functional units. The combination of functional options may be constrained by the service. Different (combinations of) options support different distributed application classes, whereas the kernel is the greatest common divisor of these classes. (When a service is decomposed into a service user environment and a service provider, it is possible to distinguish between user options and provider options. User options are agreed upon by the users and must always be supported by the provider, if requested by the users. Provider options must be agreed upon by the users and the provider. The provider may refuse or alter the options proposed by a user. Since it is not required to implement provider options, the definition of such options supports the development of standard protocol products with different degrees of functionality and with different costs to meet the needs of different environments.)

A generic service should include a negotiation phase in which two or more locations are bound by the service and options are selected. The agreement of a set of options between an application service provider and two or more application service users at different locations constitutes a relationship which is called an application association. If the negotiation phase must be (locally) completed before any of the other service functions can be initiated, the application service can be divided into an application association establishment phase and a subsequent application association utilization phase.

7.3.2 Service structure

Structuring of a service is considered useful (1) to manage the complexity of the service, (2) to prepare the structuring of supporting protocols, and (3) to incorporate pre-defined general purpose service constructs. In the case of developing a reference architecture, the latter objective should be reversed: to distinguish service components which can be used to support several classes of distributed applications (i.e., which are common to several distinct services).

A service can be structured in terms of service *constraint components*. A service constraint component partially constrains the occurrence and the argument values of service primitives. It may apply to all or some subset of the service primitives. Two generic types of service constraints can be distinguished, viz. local constraints and remote constraints:

- *local constraints* (LC): service primitives are constrained by the occurrence of previous service primitives at the same location. These constraints are defined by causality relations between service primitives with the same location argument.
- *remote constraints* (RC): service primitives are constrained by the occurrence of previous service primitives at other, remote locations. These constraints are defined by causality relations between service primitives with different location arguments.

Figure 7.6 illustrates a service behaviour with local and remote constraint components. The number of local constraint components depends on the number of locations bound by the service. (The figure shows a point-to-point service with two local constraint components; the parts denoted by SAP represents collections of service primitives at the same location, or service access point). Both local and remote constraint components can be further subdivided into smaller constraint components. The constraint-oriented composition technique can be used to compose service constraints. The proper composition of all constraints yields the complete service behaviour.

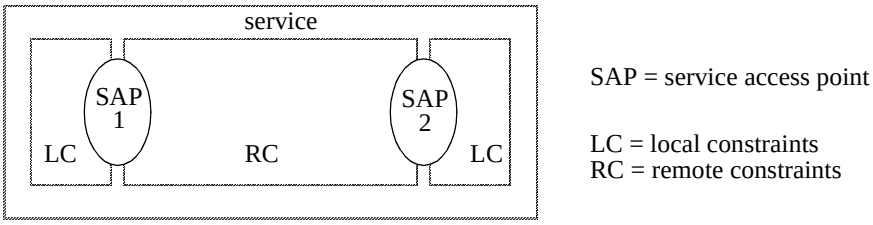


Figure 7.6: Local and remote constraint components in a service behaviour

Another way of structuring is in terms of service *sub-behaviour components* which are mutually related through causal relationships. We refer to these components as *service sections*. Important types of service sections are:

- *phases*: a phase is a sub-behaviour which does not (locally) overlap with other sub-behaviours. This means that at each location, a phase can only start if the previous phase has completed. Examples of this type of service section are the application association establishment phase and the application association utilization phase.
- *functional units*: a functional unit corresponds to a sub-behaviour consisting of a set of strongly related (tightly coupled) service functions. Functional units may overlap in time. This means that at a certain location, service primitives pertaining to different functional units may interleave in time. The interleaving is not arbitrary, but determined by the causality relations defined between the functional units.

Figure 7.7 illustrates a service behaviour with two service section components. Service sections must be defined with ‘handles’ for their composition. Such handles can be entry and exit points, since the most natural choice for service segment composition seems to be the use of the causality-oriented composition technique. Causality relationships between service sections can also be defined with local constraints: these constraints are then used to limit the possible orderings of service primitives pertaining to different service sections. Action identifiers can be seen as handles for composition in this case. (The figure only shows the use of entry and exit points).

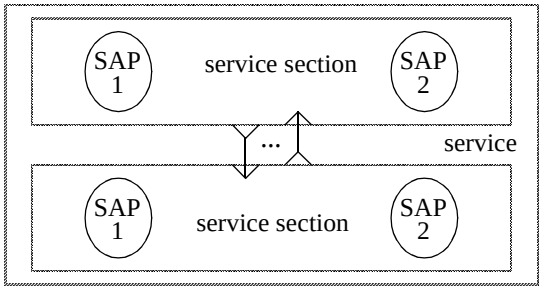


Figure 7.7: Service section components in a service behaviour

Service constraints and service sections can also be combined: constraints may be structured in terms of sections, and sections in terms of constraints. Sections, in particular functional units, may correspond to functional options of a service.

7.3.3 Quality criteria

The development of a generic application service starting from a special purpose application service permits to check the appropriateness of functions defined by the generic service.

The generalization of application services by the inclusion of user data parameters (or by the replacement of specific information parameters by user data) should be balanced with propriety. A data transfer service is the most general service for the support of distributed applications, but at the same time it does not include any functions which are tailored for the support of some distinct class of distributed applications.

The inclusion of functional options in application services supports both parsimony and completeness: for each instance of service use, only those options that are needed can be selected. The number of options should not be too great, for two reasons. First, because the negotiation mechanism becomes more complex, and consequently more time consuming and costly, when more options are included. Second, because user options must be implemented by the service provider. If many options are included, only a fraction of these options may be frequently used by individual users. Protocol standard products are then less cost-effective.

Restricting the number of options that can be included in a single service leads to the definition of multiple services, or service types, each supporting a different class of distributed applications. The definition of different service types may also support orthogonality: if the service functions required for the support of some class of distributed applications are very different from the service functions required for the support of another

class of distributed applications, then these functions should not be combined in a single service. Parsimony calls for service types which support classes of distributed applications which do not overlap. That is, different service types should not permit the selection of the same functionality, e.g. by different combinations of options. A similar reasoning applies to non-functional properties.

The determination of service sections should be based on a balance between generality and propriety. Some sections are necessarily service type specific, other sections are more general purpose, i.e. they should be useful as building blocks in different service types. Functional options may be included in a service section in order to make its applicability wider; by doing this, however, its appropriateness in some service types which do not need these options decreases. The re-use of service sections improves consistency among service types.

7.4 Design of an integrated application service provider

An application service is implemented by an application protocol. The design of an application protocol can be carried out by application of a sequence of three steps. These steps are: (1) determination of the user environment responsibility and the service provider responsibility in the service; (2) the design of an internal behaviour of the service provider, such that an application protocol behaviour is represented; (3) determination of the responsibility of the application protocol entities and the lower level service provider responsibility in the protocol behaviour (see Figure 7.5). The second step can be divided into two sub-steps, corresponding to two different views of the internal behaviour.

This section discusses the first two steps in more detail. The structure of the required application service is explicitly considered in these steps, resulting in a corresponding structure of the application protocol.

7.4.1 External provider behaviour

An application service is an interaction system of an application service user environment and an application service provider. The integrated system perspective of the application service provider can be derived from a constraint-oriented behaviour definition of the application service. The top level structure of the service behaviour should have two constraint components, one which corresponds to the ‘responsibility’ of the application service user environment and one which corresponds to the ‘responsibility’ of application service provider. The application service provider is then defined by assignment of the corresponding constraint component.

A service behaviour structure in terms of local and remote constraints is close to the desired constraint-oriented behaviour structure. Since the application service users are geographically separated, *remote constraints* are necessarily the responsibility of the

application service provider. A set of *local constraints* is the common responsibility of one user and the provider, and therefore should be further subdivided. However, this subdivision should be based on technical criteria which are case specific, prohibiting a general discussion, and often rather implementation oriented. The last fact is responsible for deferring all or part of the distribution and assignment of local constraints to a later design phase, e.g. the protocol implementation phase (this implementation aspect is often referred to as the interface refinement).

Figure 7.8 shows the development of the integrated perspective of the application service provider from the application service behaviour with local and remote constraint components.

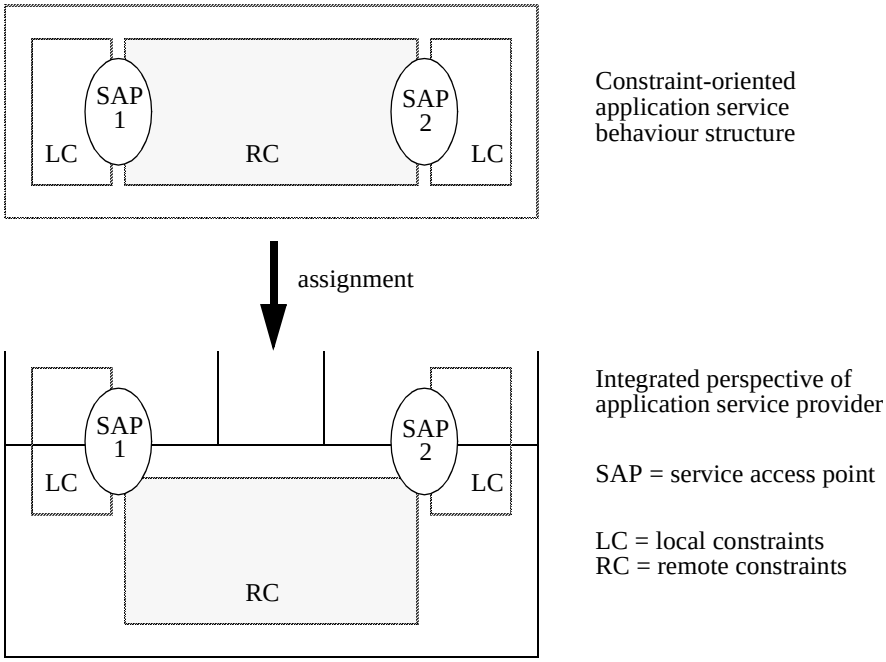


Figure 7.8: Development of integrated perspective of application service provider

7.4.2 Internal behaviour of a single provider function

An internal behaviour of the application service provider is developed through causality refinement, or through the successive application of causality refinement and action refinement (see Chapter 6, Design Model). *Causality refinement* consists of the replacement of a given behaviour by a refined behaviour, such that the actions (interaction contributions) in the given behaviour are preserved in the refined behaviour. *Action refinement*

consists of the replacement of actions by activities, where activities are compositions of actions.

Both causality refinement and action refinement must be guided by the objective of the designer, namely to prepare the assignment of behaviour components to entities in an application protocol architecture. Thus, actions introduced by these refinements should not anticipate application protocol implementation decisions. We will consider the application of causality refinement to determine *basic application protocol actions*, and the application of action refinement to decompose basic application protocol actions into *application protocol actions* and *lower level service primitives*.

Basic protocol actions

Figure 7.9 illustrates the application of causality refinement. The given abstract behaviour represents a service provider function, consisting of a request primitive, *req*, and an indication primitive, *ind*, with *req* being a condition for *ind*. This behaviour is refined by the insertion of two actions, viz. the basic protocol actions *send* and *receive*.

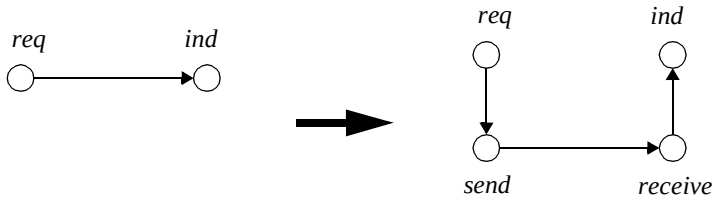


Figure 7.9: Causality refinement applied to a service provider function

The causality relation in the abstract behaviour is preserved in the refined behaviour since the latter imposes that *req* is a condition for *send*, *send* is a condition for *receive*, and *receive* is a condition for *ind*. Suppose that the service provider function performs some information transformation, represented by the constraints of *ind*. The introduction of *send* and *receive* should represent a distribution of part of the processing required to perform this function. Hence, in the refined behaviour, the constraints of *send* represent the protocol processing local to *req* (*send* references values of *req*), while the constraints of *ind* represent the protocol processing local to (of course) *ind* (*ind* references values of *receive*). Because *receive* references values of *send*, the constraints of *receive* represent processing whose distribution is still undecided. For example, the abstract and refined behaviour, with value attributes (but omitting types), could be as follows:

SPF := % service provider function, external behaviour %
{ start → req (a1, b1), req (a1, b1) → ind (a2, b2) [a2 = Fa (a1), b2 = Fb (b1)] }

SPF' := % service provider function, internal behaviour %
{ start → req (a1, b1),
req (a1, b1) → send (a2 | b1) [a2 = Fa1 (a1)],
send (a2 | b1) → receive (b2 | a2) [b2 = Fb (b1)],
receive (b2 | a2) → ind (a3, b2) [a3 = Fa2 (a2)] }

The service provider function in this example performs an information transformation, which is represented by two operations, *Fa* and *Fb*, for the sake of simplifying the definition of the internal behaviour. Only *Fa* is replaced in the refined behaviour by distributed protocol processing activities, represented by *Fa1* and *Fa2*.

If *Fa = Fa2.Fa1*, then the attribute values of the abstract behaviour are preserved in the refined behaviour. This can be validated by substituting references to values of *send* and *receive* by their values or constraints. By applying this rule to the values of *ind* we find that: *a3 = Fa2 (a2)* can be substituted by *a3 = Fa2 (Fa1 (a1))*, and *b2* by *b2 = Fb1 (b1)*.

Protocol actions and lower level service primitives

Figure 7.10 illustrates the application of action refinement based on the previous example. Both *send* and *receive* are replaced in the refined behaviour by a composition of two actions: *send* is replaced by a send activity composed of *send-pdu* and *req'*, and *receive* is replaced by a receive activity composed of *ind'* and *receive-pdu*. The objective of the

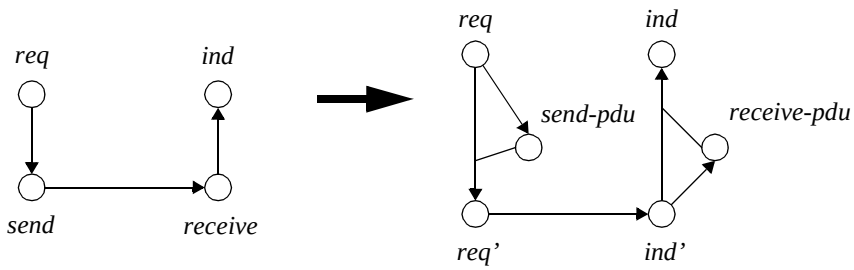


Figure 7.10: Action refinement applied to internal actions of a service provider function

action refinement here is to separate local protocol processing from behaviour that represents a lower level service. In the refined behaviour, *send-pdu* and *receive-pdu* represent protocol actions which can be assigned to different protocol entities. The other introduced actions, *req'* and *ind'*, represent lower level service primitives. The composition of *req'* and *ind'* can be regarded as a lower level service function, whose implementation is not yet considered. The separation also implies that values established by the protocol actions (only *send-pdu*, in this case) are not interpreted or changed by the lower level service function.

A possible refined behaviour, with action attributes, is the following:

SPF" := % service provider function, internal behaviour with transparent transfer %
{ start → req (a1, b1),
req (a1, b1) → send-pdu (a2) [a2 = Fa1 (a1)],
req (a1, b1) ∧ send-pdu (a2) → req' (d, b1) [d = Encode (a2)],
req' (d, b1) → ind' (d, b2) [b2 = Fb (b1)],
ind' (d, b2) → receive-pdu (a3) [a3 = Decode (d)],
ind' (d, b2) ∧ receive-pdu (a3) → ind (a4, b2) [a4 = Fa2 (a3)] }

In this example, *Encode* denotes an operation which yields a concrete (e.g., binary encoded) representation of a PDU value (i.e., a value produced by a protocol action) and *Decode* denotes an operation which yields the PDU value given its concrete representation. These operations are necessary to represent transparent transfer of user data.

Values established by the send activity which are referenced from outside the activity should correspond with values of *send* in the abstract behaviour. The referenced values of the send activity are: *d* and *b1*, where *d = Encode (a2)*. The values of the *send* are: *a2* and *b1*. Consequently, if we consider *d* as a concrete representation of *a2*, this requirement is satisfied. Similarly, values established by the receive activity and referenced from outside the activity must correspond with values of *receive*. The values of the receive activity are: *a3* and *b2*, where a*3 = Decode (d) = Decode (Encode (a2))*. The values of *receive* are: *a2* and *b2*. If the *Decode* operation is the inverse of the *Encode* operation, *a2 = a3*, and this second requirement is also satisfied. Hence, attribute values in the abstract behaviour are preserved in the refined behaviour.

Another requirement for correct refinement is that the send activity must have a condition for its completion that corresponds with the condition for *send*; similarly, the receive activity must have a condition for its completion that corresponds with the condition for *receive*. The completion condition for the send activity can be derived by backtracking the condition for the final action of this activity, *req'*. This yields: *req ∧ send-pdu*, where the condition for *send-pdu* is *req*. Thus *req* is the completion condition for the send activity, which corresponds with the condition for *send*. The receive activity has two final actions, *receive-pdu* and *ind'*; their conditions are: *ind'* and *req'*, respectively. The condition for *req'* is *req*, which is thus the completion condition for the receive activity. This condition corresponds with the condition for *receive* (the condition for *receive* is *send*, and for *send* is *req*).

Multi-peer protocol actions

The previous examples considered a peer-to-peer protocol function. If the required application service function is a multipoint service function, then the internal behaviour will represent a multi-peer protocol function. Even if the required service function is point-to-

point, an internal behaviour with a multi-peer protocol function may be designed. Figure 7.11 illustrates these cases (the intermediate internal behaviour representation, with basic protocol actions, is not shown).

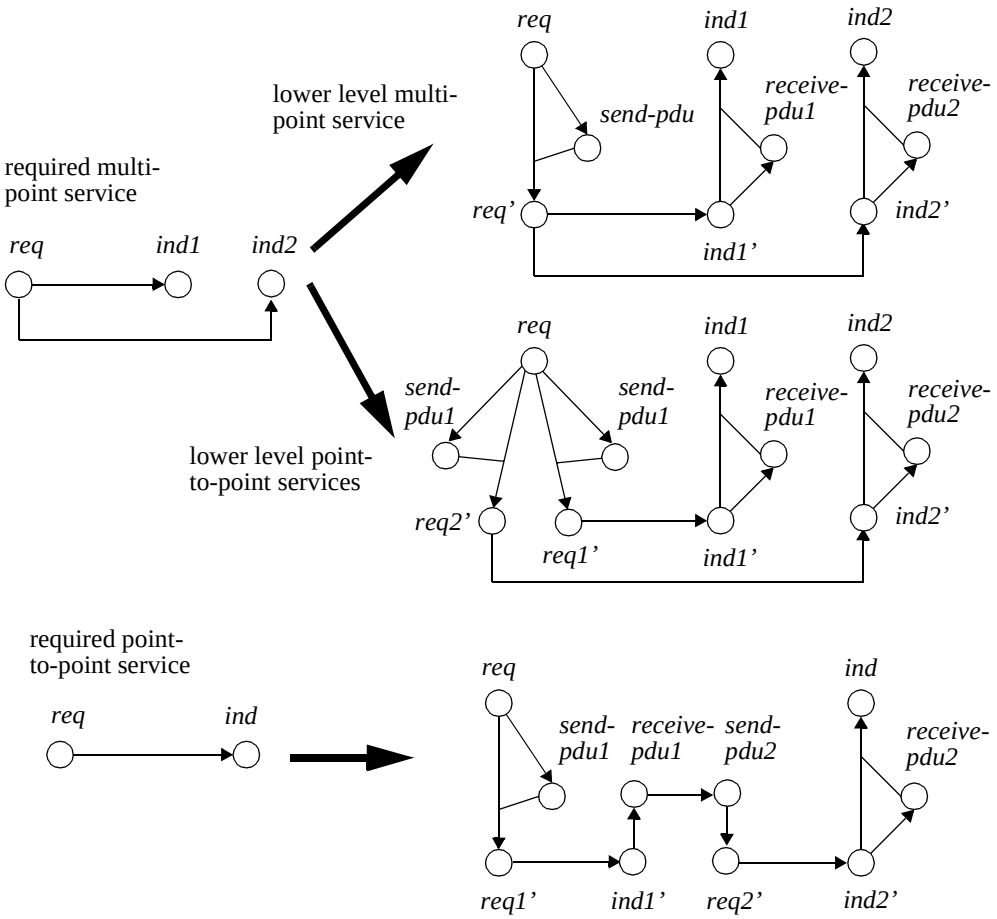


Figure 7.11: Examples of multi-peer protocol functions

7.4.3 Internal behaviour of composed provider functions

In the previous examples, we considered an application service function in isolation, i.e. independently of other functions. If we want to design an internal behaviour of the complete application service provider, we have to consider the relationship between service functions. Two types of relationships are distinguished: local constraints in the required application service are transformed into protocol state constraints in the internal

behaviour, while remote constraints in the required service are transformed into sequencing constraints in the internal behaviour.

Protocol state

Figure 7.12 illustrates the transformation of local constraints. Two related functions of the required application service are shown. One service function is defined by a request primitive *req1* and a corresponding indication primitive *ind1*, the other function is defined by a request primitive *req2* and a corresponding indication primitive *ind2*. The occurrence of *ind1* enables *req2* at the same location. The internal behaviour retains this condition for *req2*. In addition, conditions are introduced for *ind2'* and *receive-pdu2*: *ind2'* can only occur if *req1'* has occurred previously, and *receive-pdu2* can only occur if *send-pdu1* has occurred previously. Although these conditions are implied by the condition for *req2*, *ind2'* and *receive-pdu2* must be defined independently of this condition since they fall in another sphere of control (another implementation domain) than *req2*. In other words, the

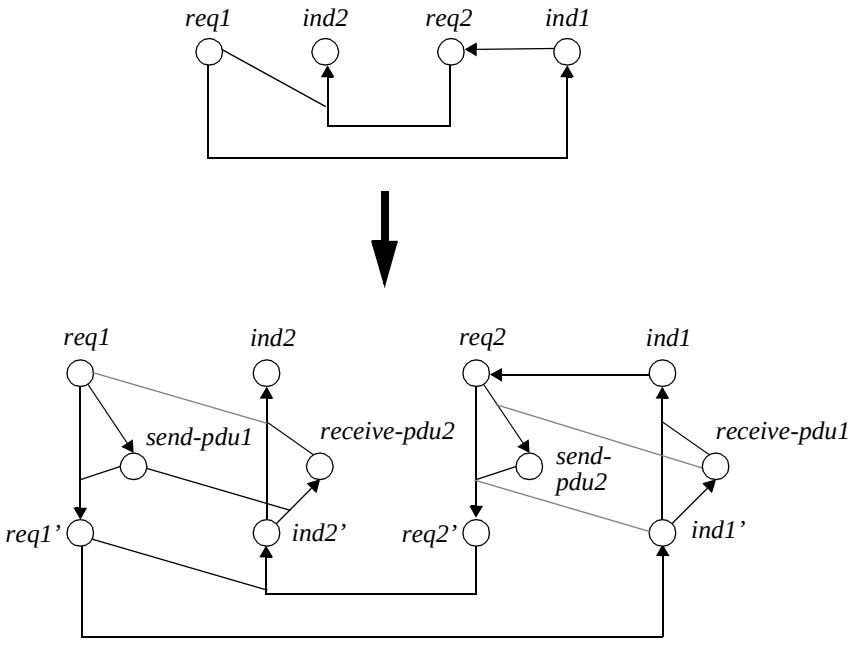


Figure 7.12: Protocol state relationship between different service functions in the internal behaviour of the provider

conditions for *ind2'* and *receive-pdu2* have to be imposed locally in order to cope with the possibility of a misbehaving remote entity. (For the same reason, *req1* is stated as a condition for *ind2* in the local constraints of the required application service behaviour.)

The condition *req1* for *ind2* appears to become redundant in the internal behaviour because of the introduction of conditions for *ind2'* and *receive-pdu*. However, local constraints on service primitives are not (completely) assigned, but remain shared until they are reconsidered during a later implementation phase. During this later phase, they will be distributed and assigned to different protocol entities, or integrated in the behaviour of a single protocol entity (in case of multi-layer implementation). In order to support the first case, the original local constraints, as defined in the required application service behaviour, are necessary and thus must also be retained in the internal behaviour. Any redundancy can be removed in the protocol implementation phase. For the same reason, conditions may be introduced for *send-pdu2* and *req2'*. The causality relations defining these potentially redundant conditions are indicated in the figure by grey lines.

Each sphere of control identified in the internal behaviour has an associated set of protocol states which controls the local actions that may occur, dependent on previous local actions. The relationship between different protocol functions in the internal behaviour (corresponding to service functions in the external behaviour) can be defined by local constraints on the required application service primitives and local constraints on the application protocol actions. The local constraints on the required application service primitives defined in the external behaviour can be retained in the internal behaviour. The local constraints on (some of) the lower level service primitives may be similar to those on the required application service primitives because of the mapping of parameters between these primitives.

Sequencing

Remote constraints in the required application service behaviour define (1) for each service function the causal relationship between a request primitive and one or more indication primitives, and (2) the order of indications of different service functions constrained by the order of the corresponding request primitives, where the indications occur at the same location and the requests occur at the same location. The transformation of the first aspect of remote constraints is already discussed in subsection 7.4.2. The second aspect is called the *sequencing relationship*, and will be discussed here.

Figure 7.13 illustrates the transformation of the sequencing relationship. Two related functions of the required application service are shown. One service function is defined by a request primitive *req1* and a related indication primitive *ind1*, the other function is defined by a request primitive *req2* and a related indication primitive *ind2*. If *req1* occurs before *req2*, then *ind1* must occur before *ind2* (assuming that a 'normal' sequencing relationship exists between these functions). The internal behaviour of the provider should preserve this relationship. *Often this can be enforced* by ordering constraints implied by the local constraints of the service, which are retained in the internal behaviour, or by ordering constraints implied by the protocol states. If this is the case, no additional

causality relations have to be defined in the internal behaviour. The following discussion is concerned with the case that the sequencing relationship cannot be enforced by the local constraints or protocol states.

One solution is to define the local protocol processing of each service function as an atomic activity, and by defining a lower level service that imposes sequencing relations between lower level service functions in a similar way as the required application service (see Figure 7.13(a)). Defining *atomic* local processing activities means that these activities are completely interleaved with respect to each other. A disadvantage of this solution is that it restricts implementation freedom with respect to *parallel local protocol processing*[1]. For example, an implementation must define that the construction of a next PDU (e.g., an activity corresponding to *send-pdu2*) can only be started after the previous PDU has been sent (e.g., an activity corresponding to *req1'*). (These restrictions exist assuming that the implementor interprets the protocol architecture as an architecture, i.e a prescription for implementation.)

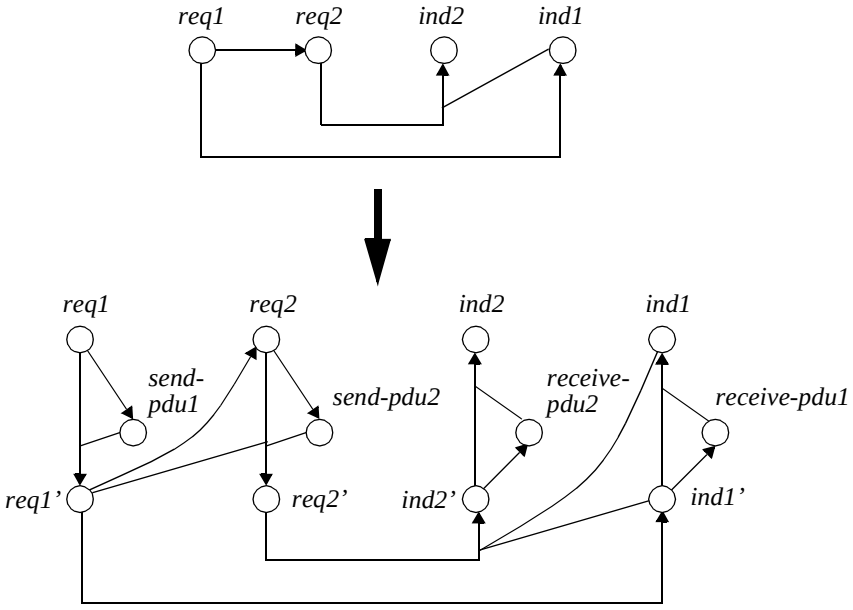
Another solution is to define a sequencing relation between action pairs in the internal behaviour which is similar to the sequencing relation between pairs of request primitives and pairs of indication primitives (as in the remote constraints of the required application service). Thus, as shown in Figure 7.13(b): if *req1* occurs before *req2*, then *send-pdu1* must occur before *send-pdu2*; if *send-pdu1* occurs before *send-pdu2*, then *req1'* must occur before *req2'*; etc. This solution does not impose restrictions on the ordering of actions which unnecessarily constrain parallel implementations.

Both solutions assume a lower level service which can at least support the 'normal' sequencing requirements of the protocol (in-sequence transfer of PDUs). This is the case with the connection-oriented data transfer service type (see subsection 7.6.2). Connection-oriented transfer may also support expedited transfer (as a provider option). Special sequencing requirements of the required application service can be supported by the application protocol on top of a lower level service supporting in-sequence or expedited transfer of PDUs. For example, local protocol processing activities may reverse the order of certain PDUs (expedited transfer) or they may discard certain PDUs if followed by certain other PDUs (disruptive transfer).

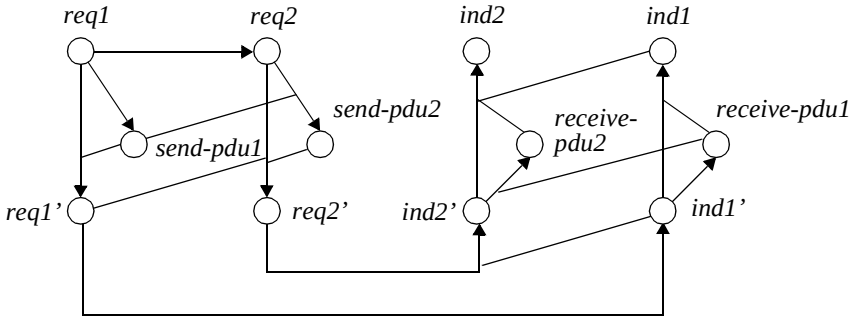
7.4.4 Structured internal behaviour

The internal behaviour of the provider can be structured in order to simplify its definition, to define re-usable behaviour components, and to prepare the intended assignment to entities. As with service behaviour, we distinguished between structuring in terms of constraints and structuring in terms of sections.

1. This approach is adopted in state table descriptions of OSI protocols.



(a) parallel processing restrictions



(b) no parallel processing restrictions

Figure 7.13: Sequencing relationship between different service functions in the internal behaviour of the provider

Protocol constraints

The top level structure in terms of constraint components is one that prepares the assignment to functional entities: application protocol entities and a lower level service provider. These constraints are:

- *upper local constraints* (ULC): required application service primitives are constrained by the occurrence of previous service primitives at the same location. These constraints are the same as the local constraints in the required application service. A number of ULC components are distinguished, corresponding to the number of locations, or upper service access points, bound by the required application service.
- *lower local constraints* (LLC): lower level service primitives are constrained by the occurrence of previous service primitives at the same location. These constraints may be a subset of the local constraints in the required application service, namely when they apply to lower level service primitives whose non-data parameters correspond to a subset of the parameters of required application service primitives. A number of LLC components are distinguished, corresponding to the number of locations, or lower service access points, bound by the lower level service. This number may not be the same as the number of ULC components, e.g. when multiple point-to-point lower level services are used by a multi-peer application protocol.
- *lower remote constraints* (LRC): lower level service primitives are constrained by the occurrence of previous lower level service primitives at other, remote locations. Multiple LRC components exist if multiple lower level services are distinguished in the internal behaviour.
- *protocol constraints* (PC): required application indication primitives are constrained by the occurrence of previous lower level indication primitives and/or ‘receive PDU’ protocol actions, and lower level request primitives are constrained by the occurrence of previous required application request primitives and/or ‘send PDU’ protocol actions. Application service primitives and lower level service primitives related by a PC component occur at different locations, or service access points, but these locations are assumed to be associated with the same local system, i.e. they are in the same sphere of control. Thus a PC component does not comprise a LRC component (which would be necessary if service primitives occurring at locations which are remote to each other are related). The number of PC components corresponds to the number of spheres of control that are distinguished in the internal behaviour. This number may not be the same as the number of ULC components (e.g., in case a multi-peer application protocol supports a point-to-point application service) or the number of LLC components (e.g., in case a multi-peer application protocol uses multiple point-to-point lower level services).

A LRC component and the LLC components associated with the locations bound by the LRC component together form a lower level service behaviour. The PC, LLC and LRC components in the internal behaviour replace (implement) the remote constraints in the application service provider external behaviour. Figure 7.14 illustrates the composition of the distinguished constraints if in the internal behaviour in case two upper service access points (USAPs) and two lower service access points (LSAPs) are distinguished. The lower level service constraints are represented in the figure by shaded boxes.

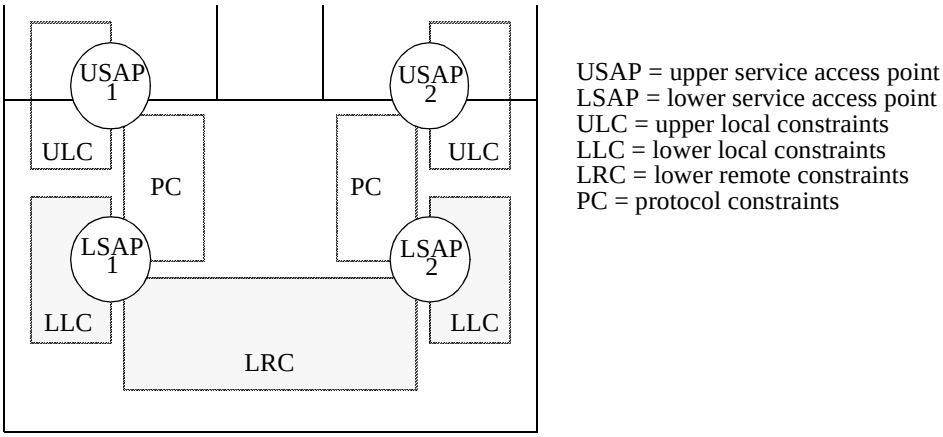
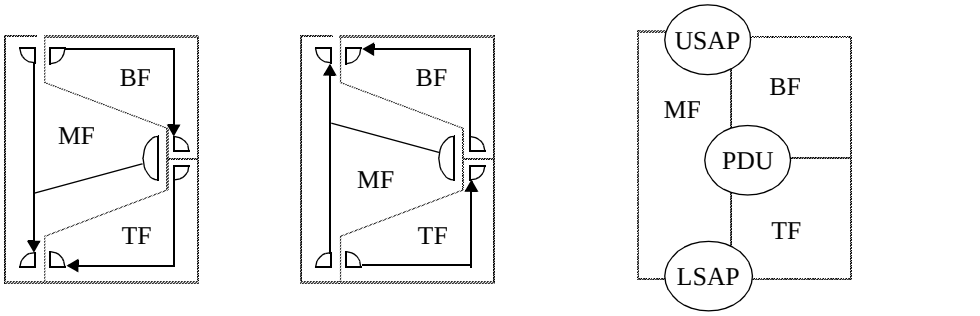


Figure 7.14: Constraint-oriented structure of application service provider internal behaviour

The discussion of the application of causality refinement and action refinement to design an internal behaviour for service functions (see subsection 7.4.2) suggests a further decomposition of the protocol constraints:

- *basic protocol function* (BF): required application indication primitives are (usually) constrained by the occurrence of previous 'receive PDU' protocol actions, and 'send PDU' protocol actions are constrained by the occurrence of previous application request primitives and/or 'receive PDU' protocol actions. The constraints defined by the BF concern the basic local protocol processing activities, which are separated from the higher level and lower level protocol processing activities by the definition of service boundaries.
- *protocol transfer function* (TF): 'receive PDU' protocol actions are constrained by the occurrence of previous lower level indication primitives, and lower level request primitives are (usually) constrained by the occurrence of previous 'send PDU' protocol actions. The constraints defined by the TF concern the representation and transfer of PDUs as user data of lower level service primitives.
- *protocol mapping function* (MF): required application indication primitives are constrained by the occurrence of previous lower level indication primitives, and lower level request primitives are (usually) constrained by the occurrence of previous required application request primitives. The constraints defined by the MF concern the mapping of non-data parameters of required application request service primitives onto non-data parameters of lower level request service primitives and reversely. This mapping may be influenced by the occurrence of protocol actions (e.g., parameter values of a lower level request primitive may depend on the value of the PDU conveyed as user data in the primitive).

Figure 7.15 shows the constraint-oriented composition of a single send and receive protocol activity (see subsection 7.4.2), in addition to a generic constraint-oriented structure for the protocol constraints. Another orthogonal decomposition of the protocol constraints is possible by distinguishing constraints related to request primitives and sending PDUs, and constraints related to indication primitives and receiving PDUs. The constraints resulting from this decomposition are not considered here.



(a) send activity (b) receive activity (c) generic structure

BF = basic protocol function
TF = protocol transfer function
MF = protocol mapping function

USAP = upper service access point
LSAP = lower service access point

Figure 7.15: Further decomposition of protocol constraints

Protocol sections

If the required application service is structured in terms of service sections, the internal behaviour of the application service provider may be structured in terms of protocol sections and possibly also in terms of lower level service sections. We will only consider here the structuring in terms of protocol sections and assume that these sections all use the same lower level service. The structuring of the lower level service in terms of service sections is discussed in subsection 7.4.5. If different protocol sections require the use of different lower level services, the composition of protocol sections (i.e., the definition of their relationship) should be delayed until the internal behaviours of the lower level service providers have been designed. This is also discussed in subsection 7.4.5.

Protocol sections should correspond with required application service sections: a protocol section should implement the corresponding service section if it is composed with the lower level service. A protocol section therefore consists of the local constraints of the service section, and the protocol constraints that are derived with behaviour refinement from the service section.

The relationship between protocol sections is defined with causality relations, using entry-exit constructs for synchronization (enabling, disabling) and for passing information. These causality relations are defined at a higher structural level, i.e. outside the protocol sections. In this way, concerns pertaining to different protocol sections are separated: details of the actions of one protocol section are not visible for another protocol section, except the information that has to be passed from one section to another. The relationship between protocol sections is similar to the relationship between protocol functions, discussed in subsection 7.4.3. This is the case since the protocol sections can be derived from the integrated protocol behaviour by cutting the causality relations between protocol actions assigned to different protocol sections, and replacing the cut causality relations by corresponding exit/entry constructs.

Figure 7.16 illustrates the same composition of service functions as in subsection 7.4.3 (cf. Figure 7.12 and Figure 7.13(b)), but now with these functions assigned to different sections. The local constraints of the required application service are retained in the internal behaviour, including the structure in terms of service sections. The structure of the protocol constraints corresponds to that of the upper local constraints. A single lower level service is used by the protocol sections. It is assumed that this service supports the sequencing requirements of the composite application protocol.

As mentioned in Chapter 4 (OSI Upper Layer Architecture and Model: Evaluation), in section 4.5, the relationship between protocol sections (ASEs, in Chapter 4) *can often be enforced* by the local constraints of the required service. This is also the case in the examples presented in Figure 7.16, *provided* that protocol sections do not pass protocol state information to each other.

7.4.5 Recursive refinement of internal behaviour

If the required application service is structured in terms of service sections, an internal behaviour will be designed for each of these sections separately. If in each of the internal behaviours the same lower level service is identified, the protocol sections will be composed. The composition must be such that it implements the required application service (i.e., the composition of the service sections). If the lower level services identified in the internal behaviours are different, the composition is delayed until the internal behaviours of the lower level service providers have been designed and in these internal behaviours the same lower level service has been identified (the reason for this is explained below). In this case, the internal behaviour of the application service provider is further refined by application of causality refinement and action refinement, as discussed in subsection 7.4.2, but now applied to the lower level services.

A lower level service may again be structured in terms of service sections, in the same way as the required application service. The same reasoning now applies to the design of an internal behaviour for each of the lower level service providers: an internal behaviour

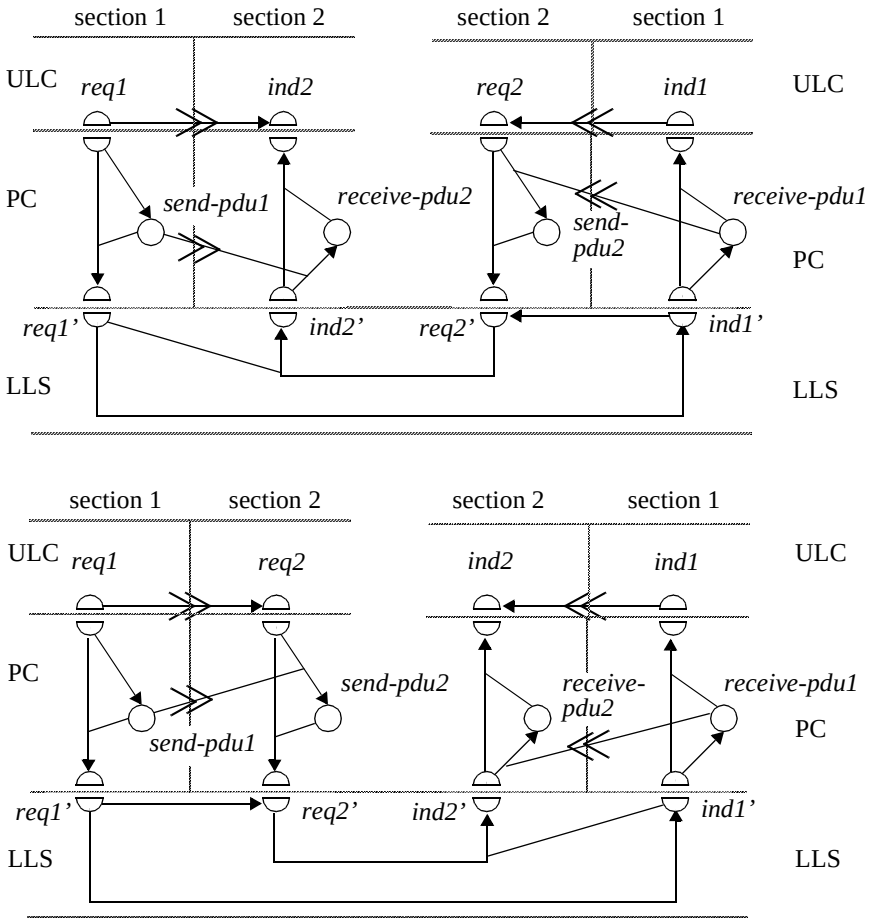


Figure 7.16: Examples of the composition of protocol sections

will be designed for each of the service sections separately; only if a common lower level service can be identified for all protocol sections in the internal behaviours, a composition of protocol sections will be designed; otherwise, a further causality/action refinement applied to the lower level services in these internal behaviours is carried out; etc. If the internal behaviour of an application service provider can be composed, identical protocol sections, identified when designing the internal behaviour of different (lower level) service providers, should be combined. (Note that a lower level service which is shared by multiple protocol sections may be structured in terms of service sections, where the service sections do not correspond with the (higher level) protocol sections.) Figure 7.17 summarizes the recursive application of causality/action refinement.

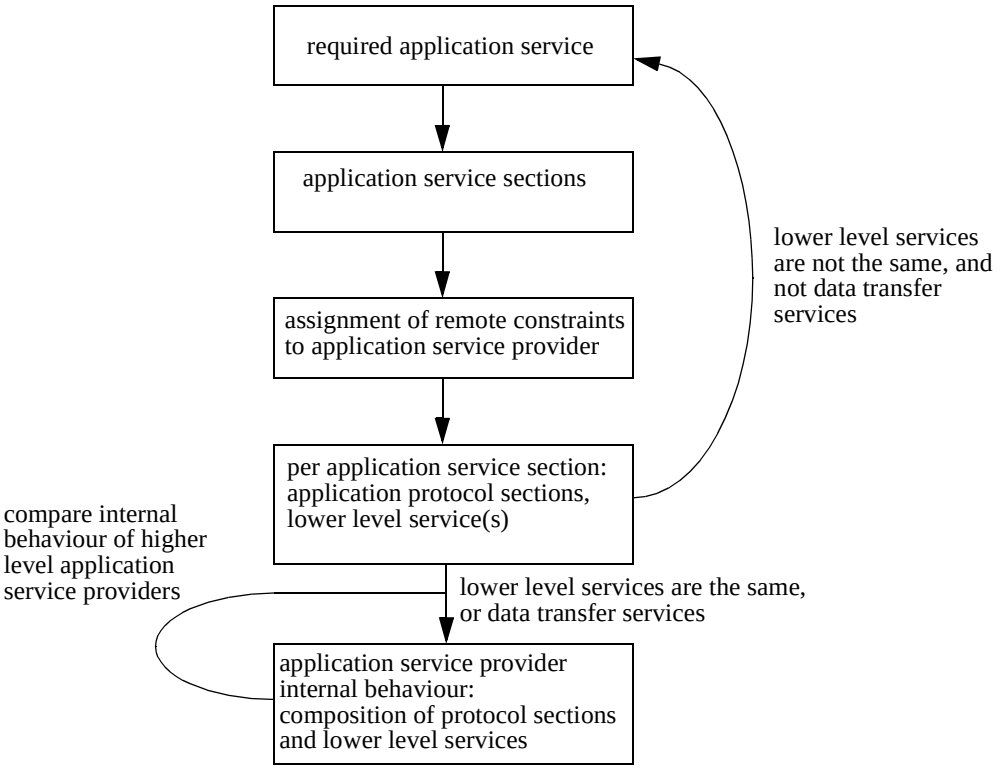


Figure 7.17: Recursive application of causality/action refinement to design application provider internal behaviour

The recursive application of causality/action refinement may seem strange at first sight. Why are the lower level services identified in the internal behaviours of each of the required application service sections not composed, as service sections, to form a single lower level service? This would enable the definition of a composition of the protocol sections, and thus the definition of a complete internal behaviour of the application service provider. There are, however, two reasons for using recursive causality/action refinement:

- the differences between lower level services identified in the internal behaviours of the required application service sections may be such that the design of each supporting protocol is not achieved in the same number of steps. In other words, the supporting protocols will have different layer structures. This implies that causality relations between protocol sections cannot be fixed until a shared lower level service has been established.
- the lower level service identified in the internal behaviour of one required application service section may also be identified in the internal behaviour of another required application service, not during the first causality/action refinement step but during later

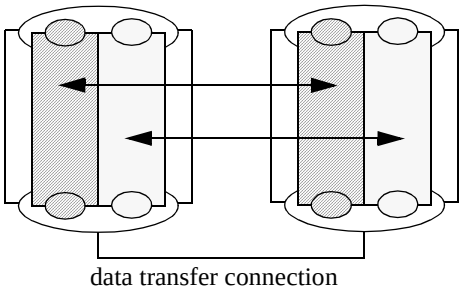
refinement steps. Again, this implies that causality relations between protocol sections cannot be fixed until a shared lower level service has been established.

The stop criterion for recursive causality/action refinement is the identification of a lower level service shared by all protocol sections. In order to share a lower level service, both functional and non-functional requirements of the protocol sections with respect to this service must be the same. For example, if two protocol sections have the same functional requirements, but different data transfer requirements with respect to the quality of data transfer, they cannot share the same instance of a lower level service. (It is assumed that data transfer requirements of different protocol sections are determined during application association establishment. Instead of distinguishing between different sets of data transfer requirements, the most demanding set may be selected for all protocol sections. This choice prevents the exploitation of the differences between the actual requirements of protocol sections. The differences between data transfer requirements for the exchange of different types of information may sometimes be the main reason for distinguishing protocol sections, e.g. in the case of distributed multimedia applications.)

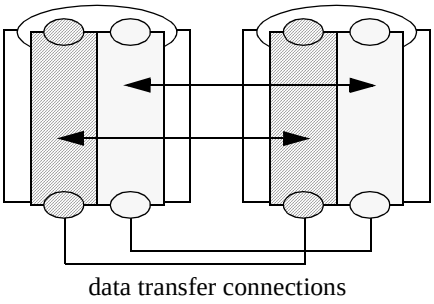
The lowest level service that can be identified in the application protocol design trajectory is a data transfer service. Even if protocol sections cannot share the same data transfer service type, because they have different data transfer requirements, the recursive application of causality/action refinement is stopped. In the case of different data transfer requirements, a ‘bundle’ of data transfer connections (assuming connection-oriented data transfer) will be used between the same locations, where each connection has different data transfer characteristics. In order to satisfy the causality (including timing) relations defined in the required application service between service sections, it may be necessary to design a special protocol that is able to restore synchronization of PDUs conveyed as data parameters over different data transfer connections.

So far, we did not distinguish between peer-to-peer and multi-peer application protocol sections. The rules mentioned above apply to both types of protocols. In the case of multi-peer protocol sections which each use multiple point-to-point lower level services, the composition criterion is extended: these protocol sections are only composed if each protocol section uses the same (in terms of functional and non-functional characteristics, and locations bound) point-to-point lower level services. (A single protocol section may use point-to-point lower level services with different functional and non-functional characteristics.)

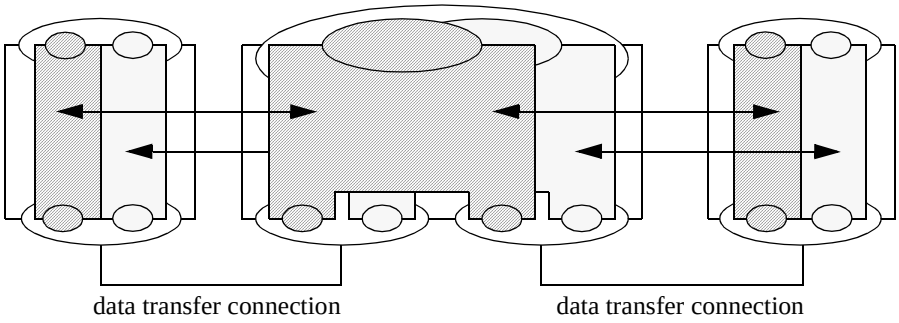
Figure 7.18 shows the basic composition possibilities of application protocol sections. Note that each of the application protocol sections can be composite, where its composition is according to one of the basic composition possibilities. The data transfer connections shown in the figure may also be replaced by arbitrary (lower level) application services.



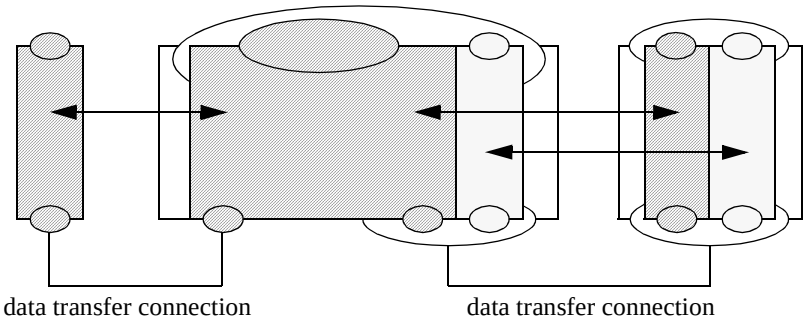
(a) composition of peer-to-peer protocol sections sharing a data transfer connection



(b) composition of peer-to-peer protocol sections using different data transfer connections



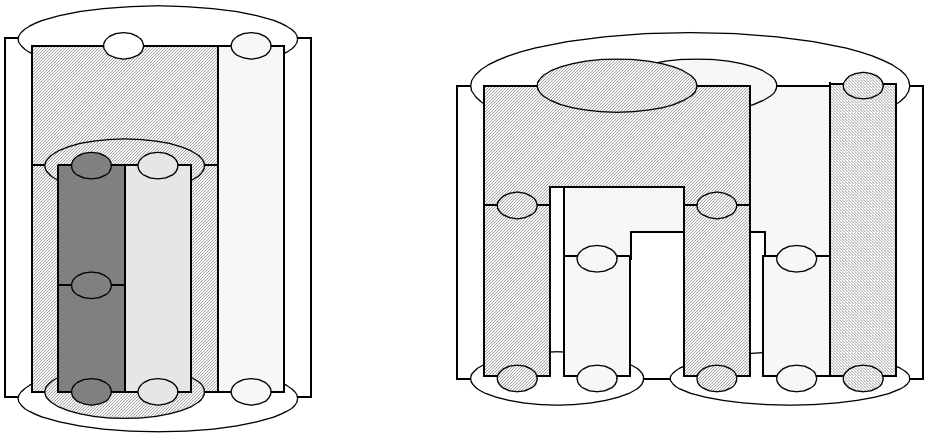
(c) composition of multi-peer protocol sections



(d) composition of multi-peer and peer-to-peer protocol sections

Figure 7.18: Basic composition possibilities of protocol sections

Figure 7.19 shows two examples of composite application protocol sections which could result from the recursive application of causality/action refinement.



(a) peer-to-peer protocol entity *(b) multi-peer protocol entity*

Figure 7.19: Examples of composite application protocol sections

7.5 Design of a distributed application service provider

A distributed perspective of the application service provider can be derived from the internal behaviour by assigning behaviour components to entities. As argued in subsection 7.4.1, this requires a constraint-oriented behaviour composition, where constraints correspond to the responsibilities of entities. Only local constraints on service primitives may not be (completely) assigned, but become a (partially) shared responsibility of entities. Their distribution and assignment is then deferred to a later design phase.

Following the same reasoning as for the determination of an application service provider that supports the required application service (see subsection 7.4.1), a lower level service provider can be determined by assigning the remote constraints of the lower level service to the lower level service provider. Protocol constraints are assigned to application protocol entities. The local constraints of the lower level service become a shared responsibility of the lower level service provider and the application protocol entities.

Figure 7.20 shows the assignment of constraints in a structured internal behaviour of the application service provider. Each protocol constraint component may be further structured in terms of constraints and/or sections (see subsections 7.4.4 and 7.4.5).

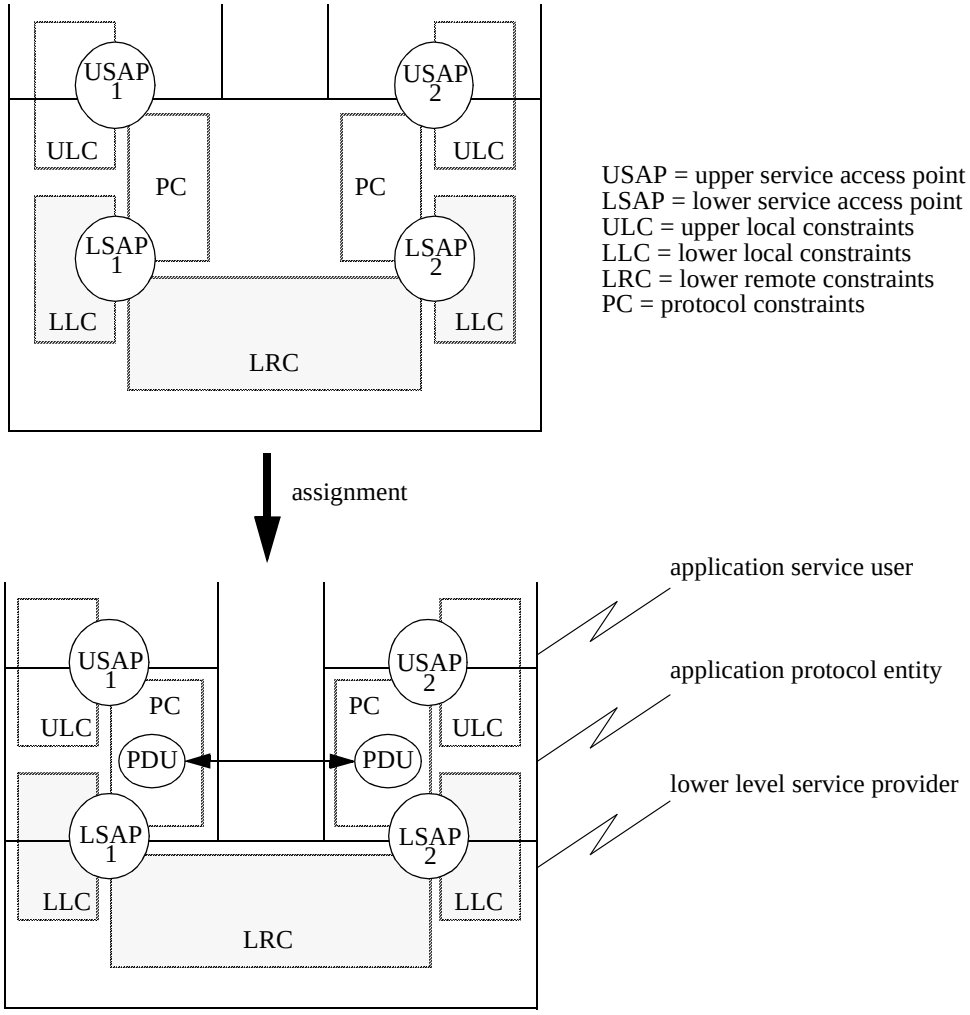


Figure 7.20: Distributed perspective of application service provider with (one) layer of application protocol entities and a lower level service provider

If the lower level service is again an application service, the same sequence of design steps can be carried out. The result of this sequence of steps is a distributed perspective of a lower level application service provider. This process can be continued until the lower level service provider is a data transfer service provider. Note that the same quality arguments apply to the lower level application service as to the required application service. Thus, the lower level service may be structured in terms of sections, where for each service section an internal behaviour is designed. Furthermore, a recursive application of

causality/action refinement may take place before a complete internal behaviour of the lower level application service can be composed.

Recursive application of the identified sequence of design steps yields a layered application protocol architecture. Each application protocol layer may be structured in terms of application protocol sections. The number of application protocol layers is not fixed, but depends on the required application service that must be supported. Also, no application protocol layers with a general purpose functionality can be identified that can be used to support any distributed application, only application protocols for the support of some class of distributed applications and generic application protocol sections for the support of a number of different classes of distributed applications. Some very generic application protocol sections are discussed in section 7.7.

A layered application protocol architecture consists of a hierarchy of layers. Such a hierarchy suggests that the (in particular, basic protocol) functions assigned to a layer depend on the functions assigned to the underlying layer. If this is not the case, it would have been possible to reverse the order of these two layers, i.e. to define an inverse hierarchy. It can be shown that if for two adjacent layers holds that the protocol mapping function is independent of the basic protocol function, then the order of these layers can be reversed without changing the basic protocol functions (of course, the mapping functions must be changed, since they determine the hierarchy). Figure 7.21 illustrates this with a simple example. The protocol mapping functions in this example are independent of the basic protocol functions: the parameters mapped are not changed by the mapping function. For example, at the sending side, a higher level basic protocol function transforms a parameter *a1*, and a lower level basic protocol function transforms a parameter *b1*. Furthermore, a higher level protocol mapping function maps the parameters *b1* and *c1*, and a lower level protocol mapping function maps the parameter *c1*. If we interchange these basic protocol functions and adapt the protocol mapping functions (at the sending side, the higher level protocol mapping function now maps parameters *a1* and *c1*, while the lower level protocol mapping function is not changed) at both sides, the overall provider functionality is not changed.

Application protocols with this property thus seem to be assigned to hierarchical layers by a somewhat arbitrary choice of the designer. The hierarchy of layers in this case is based on the degree of generality of the basic protocol functions assigned to these layers: more general basic protocol functions will be assigned to lower level protocol layers.

7.6 Information coding and data transfer

PDUs are transparently transferred as data by a lower level service. This implies that application protocol entities must agree upon a common concrete representation of the PDUs. The ‘best’ common representation is not only determined by efficient data transfer requirements (‘packed’ encoding), but also by application requirements. The latter

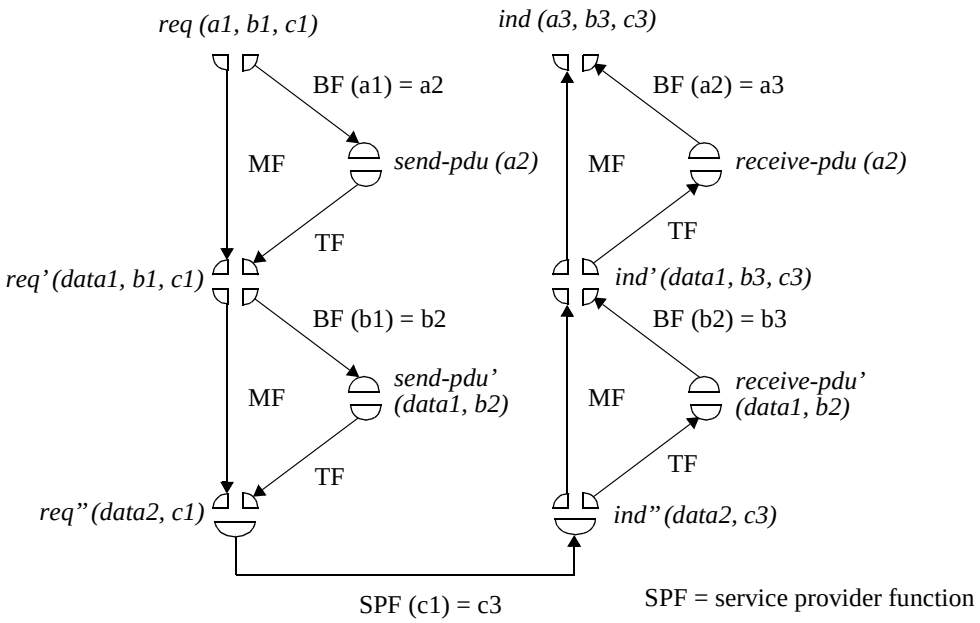


Figure 7.21: Hierarchy with protocol mapping functions which are independent of basic protocol functions

requirements are related to processing (fast conversion to and from a concrete representation used for local processing) and security (encrypted coding of information). Since different application protocol entities will have different application requirements, the 'best' common representation of PDUs depends on the application protocol entities that want to exchange PDUs. Hence, a flexible common representation is desirable.

Each generic application service will provide transparent transfer of (higher level) PDUs, namely by conveying PDUs in user data parameters. A general purpose data transfer service is a service which provides transparent transfer of PDUs, but does not comprise information transformation functions.

This section discusses the possibility of a flexible common representation of PDUs, i.e. a representation which can be selected by the application protocol entities that want to exchange PDUs. It also discusses different types of general purpose data transfer and the management of data transfer connections by application protocol entities.

7.6.1 Coding flexibility

We have assumed that coding constraints are imposed by the protocol transfer function(s) within each application protocol entity or protocol section. That is, each protocol

constraint component is responsible for the correct encoding of its 'own' PDUs (if a protocol is structured in terms of protocol sections, there may be multiple protocol constraint components, and thus multiple transfer functions, in an application protocol entity). Basic protocol functions do not impose constraints on the concrete representation of PDUs. This enables the implementor to choose a local concrete representation depending on the (real) computer system that will be used to realize an application protocol entity or protocol section.

If we want to allow flexible common representations, transfer functions should be parameterized with an identifier that indicates the coding to be used. For example, a transfer function could be defined as follows:

TF :=
{ entry (s: Id) → send (entry (s)), entry (s: Id) → receive (entry (s))
where
send :=
{ entry (s: Id) → send-pdu (p: PDU | s),
send-pdu (p: PDU | s: Id) → req' (sp: Req | s) [Data (sp) = Encode (p, s)],
req' (sp: Req | s: Id) → send (entry (s)) }
receive :=
{ entry (s: Id) → ind' (sp: Ind | s),
ind' (sp: Ind | s: Id) → receive-pdu (p: PDU | s) [p = Decode (Data (sp), s)],
receive-pdu (p: PDU | s: Id) → receive (entry (s)) }
}

In this definition, the parameter *s* symbolizes the identification of the selected common representation. The operations *Encode* and *Decode* (see also subsection 7.4.2) depend on *s*. The operation *Data* represents the extraction of the user data parameter from a service primitive.

Common representations must be selected before any PDUs that depend on these representations can be exchanged. The selection of common representations can take place during the application association establishment phase. PDUs exchanged during this phase should have a fixed common representation or they should be represented with one or more of the proposed common representations (in the latter case, the receiving entity must be able to determine what common representations are proposed, thus still requiring a fixed common representation for at least a part of each PDU).

For each protocol section that will be used during the application association utilization phase, a common representation for its PDUs can be selected through negotiation. This negotiation process can be similar to the negotiation process defined by the OSI Presentation Protocol, and can be defined by a separate protocol section of the application association establishment phase. Thus, using the terminology of OSI, for each abstract syntax (associated with an application protocol section) a transfer syntax (common concrete

representation) must be selected. The relation between an abstract syntax and a transfer syntax that is so established constitutes a presentation context. The transfer syntax of each presentation context must be passed, as a parameter, to the transfer function of the protocol section whose PDUs are identified by the abstract syntax of the presentation context. Note that, as opposed to OSI, no presentation context identification is needed during the application association utilization phase. The reason for this is that each protocol section has its own protocol transfer function, whereas in the OSI-RM the Presentation Service is the protocol transfer function that is shared by all Application Layer protocols.

7.6.2 General purpose data transfer

A general purpose data transfer service should provide data transfer between any user locations and with characteristics that are acceptable to its users. The quality of service characteristics of data transfer (e.g., transfer delay, throughput, and error rate), are limited by the capabilities of the data transfer service provider. There are two basic types of data transfer services: connectionless and connection-oriented.

Connectionless transfer

Connectionless data transfer is characterized by the fact that each ‘data unit’ transfer (a data unit corresponds to a data parameter of a service primitive) is self-contained. This means that the quality of data transfer must by requested by the sending user for each individual data unit that is sent. Also, the destination location must be indicated by the sending user for each individual data unit. For example, a data unit transfer can be defined as follows:

CL-transfer :=
{ entry → dreq (a: Location, sp1: DReq) [a = Source (sp1)],
dreq (a: Location, sp1: Dreq) → dind (b: Location, sp2: DInd)
[b = Destination (sp2), Source (sp2) = Source (sp1), Data (sp2) = Data (sp1)]
}

Quality aspects are not represented in this definition. Because each data unit transfer is self-contained, data units may not be received in the same sequence as they are sent. This depends on the implementation of the data transfer service provider. Several variations of the connectionless transfer type are possible, e.g. with confirmation of successful data transfer from the receiving user, or with confirmation of successful data transfer from the service provider. Multipoint connectionless data transfer is possible by using a group destination parameter instead of an individual destination parameter. (The OSI connectionless services are point-to-point connectionless data transfer services without confirmation; such services are for example defined at Transport Service, Session Service, and Presentation Service level.)

Connection-oriented transfer

Connection-oriented data transfer is characterized by the fact that data transfer characteristics are selected for (potentially) a sequence of data transfers between certain locations. The selection of the data transfer characteristics is accomplished in a separate phase, called the connection establishment phase. The relationship between service users and data transfer service provider that results from this phase is called a (data transfer) connection. The definition of point-to-point connection-oriented data transfer can be structured as follows:

CO-transfer :=
{ entry → establishment (entry),
establishment (exit1 (sap1: Location, cep1: CEP)) → transfer (entry1 (sap1, cep1)),
establishment (exit2 (sap2: Location, cep2: CEP)) → transfer (entry2 (sap2, cep2))
where
establishment :=
{ (LC1, RC: creq, ccnf; LC2, RC: cind, crsp),
entry → LC1 (entry),
entry → LC2 (entry),
entry → RC (entry),
LC1 (exit (sap1: Location, cep1: CEP)) → exit1 (sap1, cep1),
LC2 (exit (sap2: Location, cep2: CEP)) → exit2 (sap2, cep2) }
transfer :=
{ entry1 (sap1: Location, cep1: CEP) → one-way (entry2 (sap1, cep1)),
entry2 (sap2: Location, cep2: CEP) → one-way (entry1 (sap2, cep1)),
entry1 (sap1: Location, cep1: CEP) ∧ entry2 (sap2: Location, cep2: CEP) →
one-way (entry1 (sap1, cep1), entry2 (sap2, cep2)) }
...
}

In this definition, *sap1* and *sap2* represent different service access points, and *cep1* and *cep2* represent connection endpoint identifiers (local to *sap1* and *sap2*, respectively). The two phases distinguished in the definition can be considered as service sections. The connection establishment phase is further structured in terms of local and remote constraints. The data transfer phase (utilization of the connection) is composed of two components, each one representing the transfer of data in a single direction. The instantiation of these components is such that they concern different directions of concern. (It is assumed that at the responder side data transfer can start immediately after a positive connection response has occurred.)

Connection-oriented data transfer may provide different kinds of data transfer: ‘normal’ data transfer and expedited data transfer. Expedited data transfer is a provider option which enables the expedited delivery of expedited data units. Expedited data units may overtake normal data units. Normal data units are transferred in sequence with

respect to each other; also expedited data units are transferred in sequence with respect to each other.

Connection-oriented data transfer can be multipoint if connections are multipoint, i.e. if connections bind more than two locations. (The OSI connection-oriented Transport Service is a point-to-point data transfer service with the expedited data transfer option.)

Application protocols with a separate application association establishment phase are most naturally supported by a connection-oriented data transfer service. The same is true for application protocols that define specific sequences of PDUs (i.e., where PDUs are related to each other).

Connection management

A connection must be established before it can be used for PDU transfer, unless the connection establishment service primitives have a user data parameter. If a user data parameter is available, the application association and the data transfer connection may be established in ‘one shot’.

The use of connections must be managed. The following choices (among others) exist with respect to connection management:

- one shot establishment of an application association and a connection, or application association establishment following the assignment of a connection;
- terminating the connection after the application association has been terminated, or retaining the connection for future use;
- establishing and assigning a new connection to support an application association, or assigning a retained connection.

Establishment of data transfer connections and assignment of (new or retained) connections to application associations forms part of the application association establishment phase.

7.7 Application protocol building blocks

As mentioned before, no general purpose layers can be identified that provide application protocol support for all classes of distributed applications. Such ‘layers’ exist only for specific classes of distributed applications. Examples are the OSI Application Layer standards that were formerly called specific application service elements (SASEs): File Transfer, Access and Management, Job Transfer and Manipulation, Virtual Terminal, etc. On the other hand, it is possible to identify application protocol sections that can be used to support many different classes of distributed applications. Such application protocol

sections can be considered as general application protocol building blocks. The main reason for their separate definition is re-usability in different contexts.

Application protocol sections can be considered as pre-defined implementation constructs ([Ferreira92]) for application services. Standardized application protocol sections will, however, still be fairly complex protocols. Some authors have proposed very simple protocol functions (e.g., micro-protocols in [O'Malley91], and functional units in [Steinmetz92]) to support the construction of flexible protocol architectures.

We identify a number of general application protocol building blocks that could be considered for standardization (in most cases, they are already standardized, but as part of more comprehensive functionality):

- *application association establishment*

  This component provides service functions for the establishment of application associations. It can be structured in terms of two hierarchical protocol sections, that can be characterized as application context establishment and presentation context establishment. In addition, this component performs the establishment and/or assignment of data transfer connections. Application context establishment performs the negotiation of functional options that, if selected, become available in the application association utilization phase. For example, application protocol sections and their composition may be negotiated by this component. Presentation context establishment performs the negotiation of transfer syntaxes to support abstract syntaxes that are proposed for use during the application association utilization phase (see subsection 7.6.1).

- *inter-stream synchronization*

  If multiple connections are used by an application protocol between the same locations ('bundled' connections), then the application protocol must ensure that the PDUs received over different connections remain properly synchronized. The synchronization of different PDU 'streams' may be based on relative ordering or on absolute time. Synchronization based on relative ordering effects that indications occur in the same order as their corresponding requests. Synchronization based on absolute time effects that (approximately) the same time intervals exist between indications as between their corresponding requests. Whereas inter-stream synchronization based on relative ordering may be rather application class specific, inter-stream synchronization based on absolute time may be provided with an application protocol building block. For example, such a building block may exchange at regular intervals synchronization marks over all connections.

- *intra-stream synchronization*

  Also if PDUs are exchanged over one connection, varying time delays can be introduced. Consequently, if the required application service defines that the time intervals between indications must be (approximately) the same as the time intervals

between the corresponding requests, the application protocol must ensure proper synchronization of the PDU stream. This is called intra-stream synchronization. Intra-stream synchronization can be provided with an application protocol building block. For example, it may exchange synchronization marks at fixed, previously agreed, time intervals.

- *activity synchronization*

  Distributed information processing that is complex and time consuming, or requires the exchange of large amounts of information, may be decomposed into phases with checkpoints between successive phases. Information processing entities then can inform each other about their progress. Instead of exchanging predefined checkpoints, a checkpoint request can be transferred to a remote service user, requesting confirmation of the progress of the local processing of the remote user up to the point that all information received before the checkpoint request has been considered. The remote user returns a checkpoint confirmation if this point has been reached or if the user can guarantee that this point will be reached. The latter kind of activity synchronization can be supported with an application protocol building block. (The functionality of this building block corresponds to the session synchronization and activity management functions.) Activity synchronization may also include functions for resynchronizing out-of-sync activities.

- *commitment control*

  It may be necessary to coordinate distributed information processing activities such that either all activities successfully complete or all activities terminate without introducing state changes (e.g., changing stored information). This type of coordination is called commitment control. Commitment control can be supported by an application protocol building block. Such a building block provides functions to request confirmation of readiness (to commit), commitment, and rollback, and to indicate or confirm readiness (to commit), rollback, and commitment. (The functionality of this building block corresponds to a subset of the functions of the Commitment, Concurrency and Recovery ASE. It should, however, not be limited to peer-to-peer coordination, but also include multi-peer coordination)

- *dialogue control*

  Sometimes it is useful or necessary to structure the exchange of information by assigning exclusive ‘transfer rights’ to service users. That is, a user must be in send mode if it wants to send information (of some type, corresponding to the type of send mode), or in receive mode if it wants to receive information. Furthermore, a user must be able to request the transfer of rights from a remote user, and to surrender its rights to a remote user. This type of functionality, called dialogue control, can be supported by an application protocol building block. (The functionality of this building block corresponds to the session token management functions.) It is possible to extend dialogue control to application service functions in general: not only exchange of

information can be controlled by the assignment of rights, but also the transformation of information.

7.8 Comparison with the OSI-ULA

The application protocol reference architecture developed in the previous sections, combined with the design quality criteria of Chapter 2, the design framework of Chapter 5, and the design model of Chapter 6, forms a complete application protocol design methodology. This methodology differs from the OSI-ULA and the OSI-ULM in a number of ways. We discuss below some differences with respect to methodology, architecture and definition.

- *methodology*

The proposed methodology includes design methods that permit the systematic design of application protocols, using the milestones of the application protocol design trajectory. In particular, it clearly indicates the relation between distributed information processing system design and application protocol design, and the role of application services in a distributed information processing system. Furthermore, it indicates how services can be systematically transformed into protocols, and how a layered application protocol architecture results from the recursive application of this approach. Another structuring approach clarified by the proposed methodology is that of separating protocol sections. It is indicated what kind of relationships must be defined between service sections and between protocol sections in order to compose a complete service and a complete protocol, respectively. The systematic transformation of service sections into protocol sections is also pointed out, including its recursive application which results in a hierarchical protocol section composition (within an application protocol layer).

The OSI-ULA and OSI-ULM do not comprise design methods, nor do they clearly define the relation between services and protocols, and between ASEs and Application Layer entities.

- *architecture*

In the OSI-ULA, a static three-layer application protocol architecture is identified. In the proposed application protocol reference architecture, no static layers are identified. Protocol layers can be selected for specific classes of distributed applications (some layers may be used in the support of several classes, but no fixed protocol stack is prescribed), and protocol sections may be used to compose protocol layers. A correspondence between architectural concepts identified in the proposed application protocol reference architecture and concepts identified in the OSI-ALS can be recognized:

- the concept of ASE (application service element) corresponds to that of peer-to-peer protocol section, and the concept of SAO (single association object) corresponds to that of peer-to-peer composite protocol section. Note that both ASE and SAO are

limited to peer-to-peer relationships, whereas protocol sections may also concern multi-peer relationships. (In this respect, the concept of protocol section is more similar to the concept of ASO, or application service object, which is defined in the OSI-XALS, a revision of the OSI-ALS).

- the concept of SACF (single association control function) corresponds to the definition of causality relations between protocol sections.

On the other hand, the concept of MACF (multiple association control function) is not recognized as a distinct concept in the proposed application protocol reference architecture. A MACF is either used for local coordination, in which case its definition is incorporated in the definition of a multi-peer application protocol (and possibly represented by a separate behaviour component, e.g. a set of constraints in a constraint-oriented definition), or it is used for distributed coordination, in which case it corresponds to a separate multi-peer protocol.

Another important distinction between the OSI-ULA and the proposed application protocol reference architecture is that the former identifies a common protocol transfer function, namely the Presentation Layer, whereas the latter includes a protocol transfer function in each protocol entity or protocol section. The solution adopted in the proposed application protocol reference architecture avoids the definition of a complex service due on the use of presentation context identifications, and permits the development of appropriate transfer syntaxes in connection with a basic protocol function. Also a ‘central’ code conversion function is not prescribed (the OSI-RM suggests that the conversion between local concrete representations and common concrete representations is performed in the Presentation Layer).

Session Layer functionality is defined in the proposed application protocol reference architecture by a set of protocol sections. Concatenation and separation, and segmentation and reassembly are not included since these functions are considered the responsibility of the data transfer service provider (i.e., the Transport Service provider, according to the OSI-RM[1]).

- *definition*

In the proposed application protocol reference architecture, a distinction is made between behaviour structure and entity structure. (Adjacent) protocol entities are related by the definition of interactions (service primitives); protocol sections that pass control to one another are related by the definition of causality relations. The use of causality relations between protocol sections is more abstract than the definition of interactions between

1. It should be noted, however, that the current OSI Transport Protocol does not include a concatenation/separation function, nor a blocking/deblocking function.

protocol sections. The latter imposes implementation decisions on the implementor and therefore should be avoided in a protocol architecture.

The proposed application protocol architecture also indicates the use of constraints to structure a behaviour definition, and identifies some generic constraint components (or constraint types). Constraint components may be very useful to simplify a definition, and the identification of constraint types is valuable to promote definition styles (and specification styles). The consistent use of proper specification styles facilitates design definition and comprehensibility of designs.

The OSI-ULA does not identify structuring techniques besides layering and ASE composition.

7.9 Conclusion

A flexible application protocol reference architecture should not prescribe a single, static architecture for all application protocols. A static architecture does not allow the design of optimal application protocol support for different classes of distributed applications with diverse interaction requirements. A flexible reference architecture should therefore define the set of possible architectures, at the right level of abstraction. The level of abstraction is determined by the need of designers to incorporate pre-defined building blocks in their specific designs. Such a reference architecture allows the application protocol designer to choose a suitable architecture for his specific design, use the set of elementary design concepts of the design model to define a specific instance of this architecture, and incorporate pre-defined building blocks.

In this chapter, we have implicitly defined a flexible application protocol reference architecture by considering the design decisions concerning the structure of a design in the application protocol design trajectory. The top level structure of a distributed application consists of a layer of local processing functions (information processing entities), a layer of user-defined application protocols, and a layer of standardized application protocols. User-defined application protocols can use standardized application protocols as pre-defined building blocks.

Application protocols can be further structured in terms of protocol layers, and protocol layers in terms of protocol sections. The number of protocol layers, and the choice and composition of protocol sections depends on the class of distributed applications that must be supported. In particular, it may be possible to compose a single application protocol layer from protocol sections that provides adequate support for a specific class of distributed applications. The concept of protocol sections also *allows to integrate the layer of user-defined application protocols and the layer of standardized application protocols*: a single application protocol layer can be composed from user-defined protocol sections and standardized protocols sections.

Application services (and service sections) and application protocols (and protocol sections) can be structured using a constraint-oriented behaviour definition. Some useful generic constraint components are identified for service and protocol definition.

Some application protocol sections that can be used as very general application protocol building blocks have been identified and characterized: association establishment (with application context establishment and presentation context establishment as components; these can also be considered as separate building blocks), inter-stream synchronization, intra-stream synchronization, activity synchronization, commitment control, and dialogue control.

The development of protocol layers and protocol sections is supported by the recursive use of two design methods. These methods start from an application service, and transform this service into a protocol layer/section structure, as follows (in a simplified form):

1. designing a hierarchical protocol layer structure
   (a) determine external behaviour of the application service provider that supports the required application service;
   (b) design an internal behaviour of the application service provider in which protocol constraints and a lower level service are recognized, and structure the behaviour in terms of constraints accordingly;
   (c) assign the constraint components to functional entities, thus establishing an entity structure with application protocol entities and a lower level service provider;
   (d) continue with the lower level service provider as in 1(a), unless this is a data transfer service provider.
2. designing a protocol section composition
   (a) design a structure of the required application service in terms of service sections;
   (b) design an internal behaviour for each service section as in 1(b), resulting in protocol sections and a lower level service;
   (c) if all internal behaviours include the same lower level service, then compose the protocol sections and combine the lower level services. Continue with 1(c);
   (d) if the internal behaviours include different lower level services, then design an internal behaviour for each of the lower level service providers as in 1(b) or 2(a, b). If the same lower level service (not necessarily the lowest level) is included in all refined internal behaviours for the service sections of 2(a), then compose the (if possible combined) protocol sections and combine the lower level services. Continue with 1(c).

References

[Box92] Box, D.F., Schmidt, D.C., and Suda, T., ADAPTIVE - an object-oriented framework for flexible and adaptive communication protocols, In: *High Performance Networking, IV*, Danthine, A., and Spaniol, O. (editors), Elsevier Science Publishers B.V. (North-Holland), 1993, 367-382.

[Ferreira92] Ferreira Pires, L., Sinderen, M. van, and Vissers, C.A., On the use of pre-defined implementation constructs in distributed systems design, In: *3rd IEEE Workshop on Future Trends of Distributed Computing Systems in the 1990's*, IEEE Computer Society Press, 1992, 114-120.

[O'Malley91] O'Malley, S.W., and Peterson, L.L., A highly layered architecture for high-speed networks, In: *Protocols for High-Speed Networks, II*, Johnson, M.J. (editor), Elsevier Science Publishers B.V. (North-Holland), 1991, 141-156.

[O'Malley92] O'Malley, S.W., and Peterson, L.L., A dynamic network architecture, *ACM Transactions on Computer Systems*, Vol. 10, No. 2, May 1992, 110-143.

[Solvie92] Solvie, G., A flexible open systems architecture satisfying modern communication requirements, In: *International Workshop on Advanced Communications and Applications for High Speed Networks*, Munich, Germany, March 16-19, 1992, 383-392.

[Steinmetz92] Steinmetz, R., and Meyer, T., Modelling distributed multimedia applications, In: *International Workshop on Advanced Communications and Applications for High Speed Networks*, Munich, Germany, March 16-19, 1992, 337-349.

[Tschudin91] Tschudin, C., Flexible protocol stacks, *Computer Communication Review*, Vol. 21, No. 4, September 1991, 197-205.

Chapter 8

Suggestions for Further Work

This chapter presents some suggestions for further work: section 1 suggests an approach for testing the proposed application protocol reference architecture in practice; section 2 discusses the notion of application service engineering and presents some items for further work in this area; section 3 presents an alternative graphical notation which should be further investigated; section 4 indicates areas where the application protocol design methodology may be useful; section 5 mentions the need to investigate the combined use of object-oriented approaches and the application protocol design methods; section 6 presents possibilities for generalized constraint-oriented composition; and section 7 mentions the need for further work in the area of specification language support.

8.1 Elaboration and application of the reference architecture

The application protocol reference architecture proposed in Chapter 7 (Application Protocol Reference Architecture) should be tested by applying it to realistic examples. One way to do this is the following:

- define the building blocks characterized in Chapter 7 with the design model presented in Chapter 6 (Design Model); possibly, characterize further useful building blocks and develop their definitions as well. The definitions should be at two abstraction levels, viz. at a service level and at a protocol level. The protocols (protocol sections) should be designed on basis of the required services (service sections).
- take an existing Application Layer service standard and design the corresponding protocol using the application protocol reference architecture and the pre-defined building blocks. The design methods for protocol layer design and protocol section design should be used in this exercise.

Figure 8.1 depicts this approach. The application protocol reference architecture comprises the characterization of building blocks.

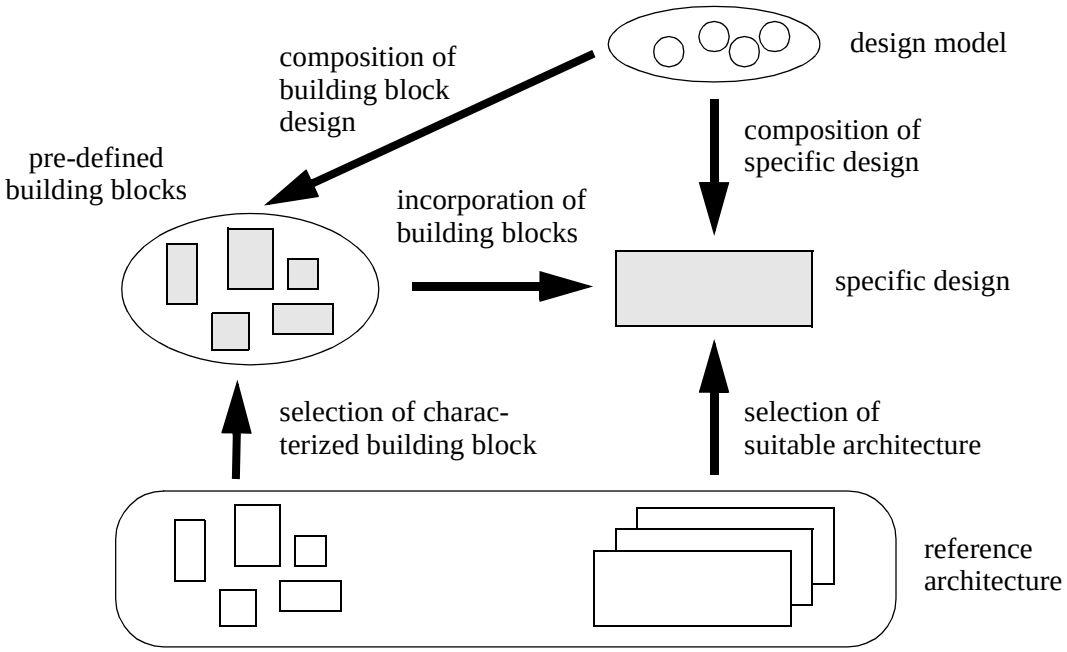


Figure 8.1: Application of the application protocol reference architecture

8.2 Application service engineering

Service engineering, also called integrated service engineering (ISE), is a term that emerged from the work on Intelligent Networks (IN) and that thereafter has been adopted in several projects, including in many RACE projects. Currently, there is no single interpretation of the concept of service engineering. However, a common characteristic is that *service engineering* aims at the composition of ‘services’ from ‘service elements’ by various ‘actors’. Differences exist with respect to the definition of service, service element, composition, and the set of actors.

In this section we propose the term application service engineering to denote a method of composing application services from application service sections under control of the end user, the user organization, or the service provider organization. That is, the composition of an application service type is done with pre-defined building blocks by actors that were not involved in the design of these building blocks. In the following, if no distinction needs to be made between the different categories of actors, an actor will be referred to with the term *service engineer.*

Before proposing a direction of work for application service engineering, this section first presents a brief overview of the background of service engineering and its interpretation in different contexts.

8.2.1 Background and interpretations

Increased competitiveness in the telecommunications market and changing user needs towards more sophisticated and specialized telecommunications services are the main reasons for the current research on service engineering. Conventional telecommunications systems are generally composed of manufacturers/platform-dependent components for which no standard interfaces have been defined. For example, the switches in conventional telecommunications systems integrate service logic, switching control, and switching functions in manufacturers-specific components. As a consequence, if one of these functions must be changed or extended, all switches need considerable re-design, involving all original switch manufacturers.

The introduction of new services in conventional telecommunications systems is therefore both time consuming and costly. Intelligent network architectures are being developed that are more suitable to cope with the challenge of faster and cheaper service introduction. The basic idea behind intelligent network architectures is the separation of service logic and switching functions, based on the definition of standard interfaces ([Brégant92], [Kung92]). This separation enables provider organizations to introduce new services by implementing new service logic procedures, independently of the switch manufacturers. In the context of IN, service engineering is the activity of creating new service logic procedures, or service logic programs, from pre-defined ‘service independent building blocks’ (SIBs).

SIBs are currently low level constructs, which roughly correspond to basic conditions and basic constraints with respect to service primitives. These constraints and conditions can be used to compose different types of causality relations between service primitives. Compositions of SIBs can be automatically translated into switching control procedures.

Many RACE projects have adopted an object oriented modelling approach, and service engineering in this context is the composition of ‘integrated’ services from pre-defined ‘service objects’ ([Key90], [Insulander93]). Object oriented modelling approaches are based on the basic concept of object, which is an entity represented by its external interfaces, where each external interface consists of a set of basic operations. Thus, an object can be considered as the architecture of an entity. Specific object oriented approaches define specific ways to classify objects and to manipulate object classes. One important form of manipulation, called inheritance, is the re-use of object class properties in newly defined object classes. An object oriented design is a composition of objects, where composition is accomplished by the binding of external object interfaces.

Service objects are objects with external interfaces that are accessible to service users. So far, little work has been done on the specific composition constraints that should be imposed on service engineers when creating their own service. In addition, the relationship between service objects and ‘protocol objects’ is not clear in these projects. Protocol objects, i.e. objects with external interfaces where service operations (service primitives)

are distinguished from protocol operations (sending and receiving PDUs), are not identified.

8.2.2 Direction of work

The general aim of service engineering is to ease the introduction of new services and to facilitate the tailoring of services to specific user needs. The general approach adopted is to provide the service engineer with pre-defined building blocks which can be composed subject to certain constraints. These constraints should be incorporated in a special support environment for the service engineer. The purpose of the support environment, usually called the service creation environment, is:

- to provide the service engineer with a 'service composition editor', and possibly other tools that support the service engineer in composing services that satisfy user requirements;
- to prohibit the composition of services that may harm the functioning of the distributed system. This implies that the tools provided to the service engineer should have appropriate built-in constraints;
- to translate composed services into protocol realizations.

We propose to use application service sections as the building blocks for composing application services. The composition should be defined, by service engineers, in terms of causality relations between service sections, where service sections should have corresponding (composite) protocol sections that operate on top of a common (set of) lower level service(s).

If the service user is also service engineer, he is involved in three successive phases when he wants to use a (newly created) service, viz. (1) service type creation phase, (2) application association (service instance) establishment phase, and (3) application association (service instance) utilization phase. The first phase is a consequence of service engineering, the latter two are the same as in the 'normal' case, discussed in Chapter 7 (Application Protocol Reference Architecture).

During the service type creation phase, the service engineer composes an application service type that satisfies the requirements of a particular (group of) service user(s). The composition constraints imposed on the service engineer should at least include the following:

- *well-defined* exit/entry constructs: the values expected by an entry must match the values defined by the exit with which the entry is related;
- *complete* set of exit/entry constructs: all entries of a building block must be considered during composition. An entry of a building block is enabled by either an exit of another building block, a spontaneous start, or the environment of the required service.

A service engineer may define a composition in terms of local constraints sections only, if the composition of remote constraints is implied by the composition of local constraints. Consider, for example, the following composition of a service type *S*:

```
S := % composite application service %
{   ...
    SX (exit1 (a: Location)) → SY (entry1 (a)),
    SX (exit2 (b: Location)) → SY (entry2 (b)),
    SY (exit1 (a: Location)) → SZ (entry1 (a)),
    SY (exit2 (b: Location)) → SZ (entry2 (b)),
    SZ (exit1 (a: Location)) → ... ,
    SZ (exit2 (b: Location)) → ... ,
    ...
where
    ...
    SY := % application service section "Y" %
    {   (LCY1, RCY: yreq, yind; LCY2, RCY: yreq, yind),
        entry1 (a: Location) → LCY1 (entry (a)),
            % enable "yreq" and "yind" primitives at location "a" %
        entry2 (b: Location) → LCY2 (entry (b)),
            % enable "yreq" and "yind" primitives at location "b" %
        entry1 (a: Location) → RCY (entry1 (a)),
            % enable transfer from and to location "a" %
        entry2 (b: Location) → RCY (entry2 (b))
            % enable transfer from and to location "b" %
        LCY1 (exit (a: Location)) → exit1 (a),
        LCY2 (exit (b: Location)) → exit2 (b)   }

    SZ := % application service section "Z" %
    {   (LCZ1, RCZ: zreq, zind; LCZ2, RCZ: zreq, zind),
        entry1 (a: Location) → LCZ1 (entry (a)),
            % enable "zreq" and "zind" primitives at location "a" %
        entry2 (b: Location) → LCZ2 (entry (b)),
            % enable "zreq" and "zind" primitives at location "b" %
        entry1 (a: Location) → RCZ (entry1 (a)),
            % enable transfer from and to location "a" %
        entry2 (b: Location) → RCZ (entry2 (b))
            % enable transfer from and to location "b" %
        LCZ1 (exit (a: Location)) → exit1 (a),
        LCZ2 (exit (b: Location)) → exit2 (b)   }
    ...
}
```

In this composition, exit conditions of service sections only depend on the local constraints of these sections. The service engineer may therefore concentrate on the composition of the local constraints of the required service (this composition has the same structure as the composition of the service sections in the above definition).

In order to be able to transform a required service into a protocol realization, the distributed system must contain protocol section realizations corresponding to the service sections used in the composition of the required service. The causality relations that define the composition of the required application service must be transformed into an application protocol realization composed of protocol section realizations. Furthermore, since the application protocol realization consists of real application protocol entities, the composition of the application protocol realization must be distributed. The distribution is (at least, in part) determined by the service engineer, since he defines the locations bound by the required application service. In the local systems associated with these locations, a real application protocol entity is composed. If the protocol section realizations required for the composition of a real application protocol entity are contained by the local system, then no additional requirements need to be considered during the standardization of application protocol sections. If, however, protocol section realizations are not locally available, they must be *imported*, which implies that *concrete interfaces* of application protocol sections should be standardized.

In a distributed environment where new functionalities (provided by application protocol section realizations) are frequently added by different service provider organizations, and old functionalities are removed, it may be convenient to the service engineer if he can inform about the availability of certain types of functionality. This becomes possible with a trader component, a component which is currently being defined by ISO and ITU-TSS in the ODP project[1] ([Bearman94]). A trader is used by service provider organizations to advertise functionalities and by service users to inform about available functionalities. The trader can also support the service user in finding an optimal match between its requirements and a (combination of) advertised functionalities.

Important issues of further work in relation to application service engineering include:

- investigation of composition possibilities and constraints;
- investigation of transforming service composition constructs into protocol realizations;
- investigation of importing protocol section realizations from remote systems; and
- investigation of proper use of trader-like components.

1. Similar components are being defined by other projects.

8.3 Graphical notation for composite enabling conditions

In Chapter 6 (Design Model), we introduced a graphical notation for representing compositions of design concepts. This notation includes a representation of the conjunction and disjunction of conditions. Figure 8.2(a) depicts the basic causality relations comprising conjunction and disjunction of conditions. The choice of this notation is rather arbitrary, and an alternative notation, as depicted in Figure 8.2(b), could have been chosen instead.

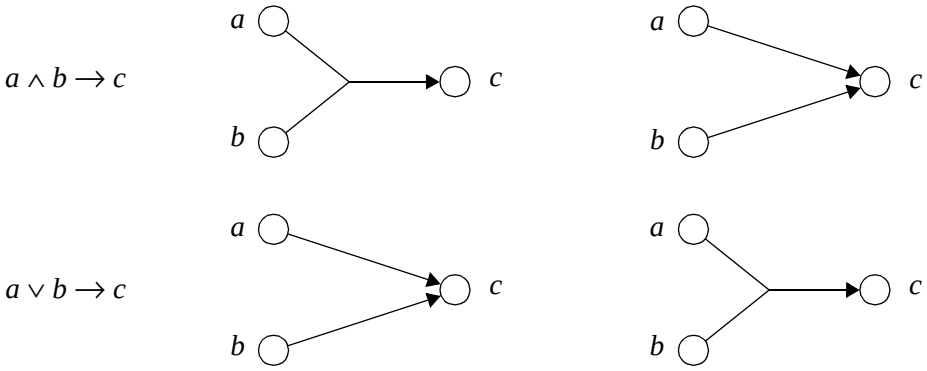


Figure 8.2: Two notations for conjunction and disjunction of conditions

The intuitive argument in favour of the original notation is that *combining* the shafts of two arrows more naturally represents the notion of *conjunction*. The alternative notation, however, appears to have a number of practical advantages:

1. it facilitates the constraint-oriented *decomposition* of behaviours

   The representation of a constraint-oriented behaviour composition is derived from an integrated behaviour representation by cutting (some of) the action representations in the latter. When using the alternative notation, this cutting does not interfere with the representation of the causality relations. If constraint-oriented decomposition is allowed, the arrows representing causality relations can be copied to the representation of the constraint-oriented behaviour composition. And reversely, if constraint-oriented decomposition is not allowed, the alternative notation indicates this since it would necessitate to cut an arrow head, and part of a shaft, lengthwise. Figure 8.3 illustrates this property.

2. it supports *symmetry* in behaviour representations

   Protocol behaviours often possess a symmetry with respect to ordering relations at the sending and receiving side. This symmetry should also be apparent from the graphical representation. Since the original notation sometimes requires the redrawing of arrows

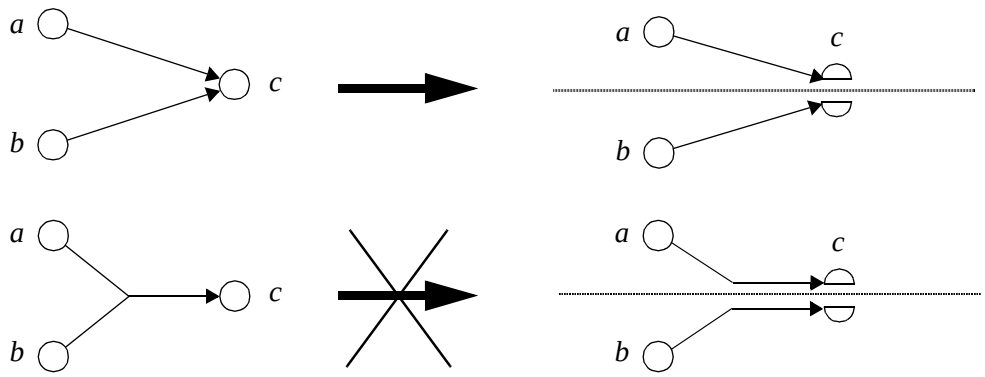
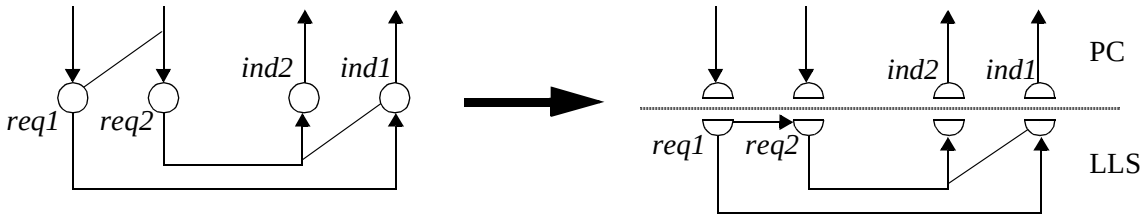
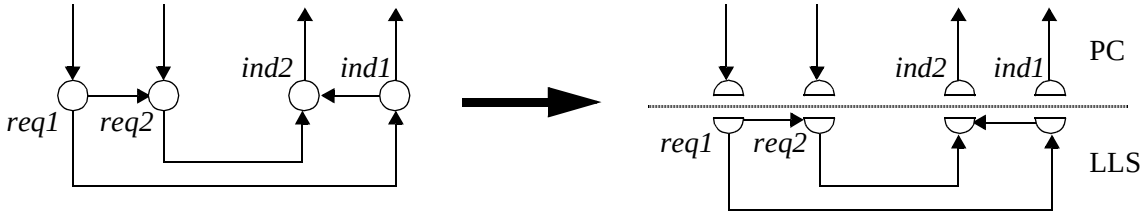


Figure 8.3: Constraint-oriented decomposition, represented with the alternative notation

that represent causality relations if a constraint-oriented decomposition is introduced, symmetry may not be preserved depending on the nature of the decomposition. This is not the case with the alternative notation. Figure 8.4 illustrates this with a constraint-oriented decomposition of a protocol behaviour involving two successively invoked lower level service functions, each consisting of a request primitive and a corresponding indication primitive. The constraint components in the decomposed behaviour are two protocol constraints and a lower level service, where the lower level service defines the ordering relations. With the original notation, the symmetry with respect to ordering relations is not represented, whereas with the alternative notation it is.



(a) original notation

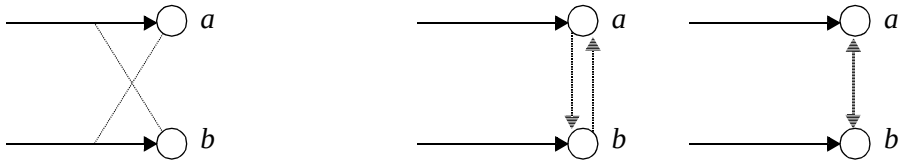


(b) alternative notation

Figure 8.4: Representation of symmetry with the original and with the alternative notation

3. it enables a simpler representation of (mutual) *conflict*.

   Disabling or mutual disabling (mutual conflict, or mutual exclusion) of actions is expressed with non-occurrence conditions which are composed in conjunction with one or more occurrence conditions. The original notation unjustly suggests that a conflict of two actions is somehow related to the occurrence of other actions, as the graphical convention requires the combination of part of the shafts of two arrows, representing the causality relation of the result action with the enabling action(s) and with the disabling action. This is not the case with the alternative notation. Furthermore, the alternative notation allows a straightforward shorthand for a mutual conflict relation, namely a double-headed arrow. Figure 8.5 illustrates the different representations resulting from the use of the two notations.



(a) original notation (b) alternative notation, without and with shorthand

Figure 8.5: Representation of mutual conflict with the original and with the alternative notation

Note that the original and the alternative notation should be based on the application of two different canonical forms for representing conjunction and disjunction of conditions. With the original graphical notation, a complex condition is written as a disjunction of component conditions, where each component condition is a conjunction of elementary conditions. With the alternative graphical notation, this is the way around: a complex condition is written as a conjunction of component conditions, where each component condition is a disjunction of elementary conditions. For example, the condition in the causality relation $a \wedge (b \vee c) \rightarrow d$ is in canonical form if the alternative notation is used, and can immediately be represented using this notation, whereas it should be rewritten as $(a \wedge b) \vee (a \wedge c)$ if the original notation is used. The reverse situation exists for the condition in the causality relation $a \vee (b \wedge c) \rightarrow d$: this condition is in canonical form if the original notation is used, but should be rewritten as $(a \vee b) \wedge (a \vee c)$ if the alternative notation is used. Figure 8.6 illustrates these examples.

8.4 Enterprise protocols

In Chapter 5 (Design Framework), we presented an application protocol design trajectory whose starting point can be chosen at very high abstraction levels. For example, the initial milestone of the design trajectory may be an enterprise service, i.e. the interaction system of an enterprise system (e.g., a business organization) and its environment. Although we

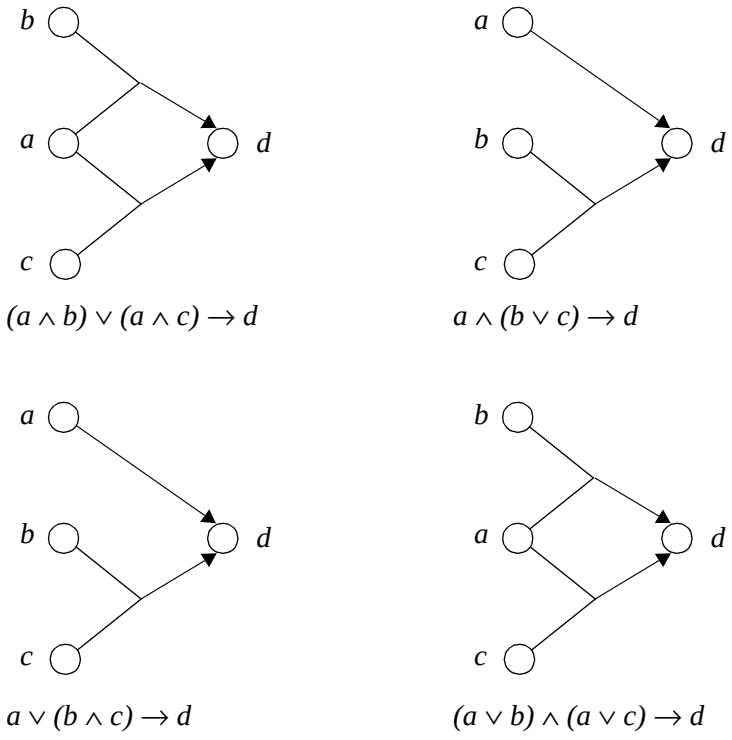


Figure 8.6: Canonical form of the original notation and of the alternative notation

briefly outlined the design trajectory from an enterprise service to an application protocol architecture, we mainly focused on the part of the design trajectory starting from the milestone of an application service provided by a distributed computer system.

There are, however, a number of situations and application domains, where the consideration of earlier milestones, or higher abstraction levels, is particularly relevant. Examples of such application domains are:

- *Computer Supported Cooperative Work* (CSCW): CSCW is concerned with the design of work procedures involving two or more human beings, including the design of computer support for these work procedures (see [Bannon91], for example). The objective of CSCW is to improve the quality and performance of work that requires work efforts from more than one person (the term ‘cooperative work’ is used to denote that persons are aware of each other’s involvement in the work process).
- *Work Flow Management* (WFM): WFM is concerned with the design of procedures for managing the flow of work in business environments, including the computer support for these procedures. The objective of WFM is to achieve a better distribution of work load among workers (both human beings and automated systems).

- *Business Process Re-design* (BPR): BPR is concerned with the design of business activities, composed of interacting business processes, that are intended to replace existing business activities (see [Davenport90], for example). The design essentially preserves the services provided by existing business activities, but changes internal structures and mechanisms such that a more efficient and effective performance of work is accomplished. BPR may also involve the (re-) design of computer support for business processes or business process interactions.

Each of these application domains may require distributed computer support. If this is the case, application protocol design may constitute a separate phase in the design trajectories used in these domains. Computer system application protocols may only cover part of the distributed interactions required for the support of enterprise services. Thus, 'informal' protocols may be defined at higher levels between enterprise processes (e.g., organizational units). Such protocols can be referred to as enterprise (application) protocols. Enterprise protocols may also be used to represent the internal behaviour of enterprise systems, independently of the use of computer systems within enterprise systems. The design of enterprise protocols may be based on similar design methods as used for the design of (computer system) application protocols. An enterprise protocol may be designed on basis of an enterprise service, and intermediate lower level enterprise services may be identified, resulting in a layered enterprise protocol architecture. Also enterprise protocol sections may be used as a way of structuring enterprise protocols. Furthermore, enterprise protocols may be related to local processing activities in the enterprise system in the same way as application protocols are related to local information processing activities performed by the distributed computer system. However, the extent to which this is possible and the need for additional methods and structures should be further investigated.

8.5 Relation with object oriented approaches

Object oriented (OO) approaches are currently widely applied or investigated in projects concerned with distributed systems development, including ROSA [(Key90]) and CASSIOPEIA ([Insulander92]) in the RACE programme, and ODP ([Raymond94)] of ISO and ITU-TSS. OO approaches are not restricted to modelling, but may also support particular phases of design (e.g., OO analysis, OO design, and OO implementation).

The main attractions of OO approaches, when applied in early design phases, are the availability of a uniform conceptual framework and the support for re-usability of design results. All OO approaches are based on the basic concept of *object*. An object is similar to an entity architecture: it is represented by its external interfaces, where each external interface consists of a set of basic operations. At a lower level of abstraction, an object is replaced by an object implementation. An object implementation is similar to an entity implementation: it defines how the operations of external interfaces are performed and interrelated. Most OO approaches support the decomposition of objects into compositions of interacting 'smaller' objects.

Re-usability is based on the definition of object classes and the support of methods to construct new classes by *incremental modification* of existing classes (inheritance). Unfortunately, OO approaches do not consistently define concepts and methods related to the support of re-usability. Often definitions depend on the specification language adopted or defined by these OO approaches.

The operations of an object can be invoked by *any* other object. This implies that no distinction is made between local (direct) interactions (cf. service primitives), and distributed (indirect) interactions accomplished via local interactions (cf. PDU exchange, where PDUs are conveyed as user data of service primitives). Indeed, the concepts of SDU and PDU (or concepts with a similar meaning) are not identified in OO approaches. These concepts, and the distinction mentioned above, are the basis for protocol design. OO approaches therefore seem to focus on system part design, instead of interaction system design. Consequently, application protocols, and their design, are not considered by OO approaches, either intentionally or unintentionally (data transfer protocols are usually not considered since a particular data transfer infrastructure is assumed).

Since we believe that OO approaches can play an important role in distributed systems (distributed applications), the relationship between OO approaches and application protocol design approaches should be further investigated. In particular, it is worthwhile to study the possible combination of the proposed application protocol design trajectory and reference architecture, and methods that support re-useability as proposed by OO approaches. Examples of studies of the application of OO approaches to OSI application protocol standards development can be found in [Bochmann92] and [Feldhoffer92].

8.6 Generalized constraint-oriented composition

The cooperating main service approach discussed in Chapter 3 (OSI Upper Layer Architecture and Model: State of the Art) can be supported with the constraint-oriented composition technique. A disadvantage of structures derived with this approach and defined with constraint-oriented composition is that the cooperating main service and the auxiliary service cannot be defined independently of each other. Consider, for example, a request primitive *mreq* with parameters *p1* and *p2*, where parameter *p1* is defined and constrained by a cooperating main service, and parameter *p2* is considered a ‘dummy’ parameter, to be defined and constrained by an auxiliary service. In order to allow constraint-oriented composition, the main service must, however, indicate the presence and type of *p2*, and the auxiliary service must do the same with respect to *p1*. (This example and the following examples only consider service primitives. A similar reasoning can be applied to PDUs.) The following (incomplete) definition illustrates the constraint-oriented composition:

S := % service composed of cooperating main service and auxiliary service %
{ (MS, AS: ... , mreq, ...),
entry → MS (entry),
entry → AS (entry)
where
MS := % coordinating main service %
{ ...
... → mreq (p1: P1, p2: P2) [p1 = Fm (...)],
... }
AS := % auxiliary service %
{ ...
... → mreq (p1: P1, p2: P2) [p2 = Fa (...)],
... }
}

(*Fm* and *Fa* are operations with arguments determined by the condition for *mreq* in *MS* and *AS*, respectively.)

The dependency is a consequence of our definition of constraint-oriented composition, where interaction contributions must have matching arguments. It is possible, of course, to replace the parameters by their representation (encoding), in which case the matching requirements can be satisfied without duplicating type information:

for the cooperating main service: *mreq (d1: Data, d2: Data) [d1 = E (Fm (...))]*, and
for the auxiliary service: *mreq (d1: Data, d2: Data) [d2 = E (Fa (...))]*.

(*E* is an encoding operation; *Fm* and *Fa* implicitly define the types of the parameters represented by the encodings.)

A serious objection against this solution is that it mixes levels of abstraction: the choice and definition of an appropriate encoding is a protocol concern. (Even if we consider the application to PDUs, this solution is not desirable, since it does not permit the separation of basic protocol and protocol transfer concerns; see Chapter 7, Application Protocol Reference Architecture.) Another solution is to define a generic parameter type. All specific parameters of service primitives would be subtypes of this generic type. If a generic type *P* can match any subtype, including *P1* and *P2*, the following definition is possible:

for the cooperating main service: *mreq (p1: P1, p2: P) [p1 = Fm (...)]*, and
for the auxiliary service: *mreq (p1: P, p2: P2) [p2 = Fa (...)]*.

A disadvantage of this solution is that the auxiliary service must still indicate the number of parameters defined by the cooperating main service, as well as the position of the parameters that itself defines relative to those defined by the cooperating main service.

Hence, with this solution it is still difficult to design auxiliary services that can be used as building blocks, i.e. that can be composed with various cooperating main services. Yet another solution is to extend the notion of constraint-oriented composition by allowing certain arguments in interaction contributions *not* to match; these arguments extend the list of arguments proposed in the other interaction contributions. Assuming that we may indicate arguments that are not subject to matching by underlining, we can define:

for the cooperating main service: *mreq (p1: P1, p2: P) [p1 = Fm (...)]*, and
for the auxiliary service: *mreq (p2: P2) [p2 = Fa (...)]*.

A possible objection against this solution is that, without further constraints, the order of arguments in the interaction is not uniquely determined in general. For example, the interaction contributions *req (p1: P1, p3: P3)* and *req (p2: P2, p3: P3)* may result in two possible interactions *req (p1: P1, p2: P2, p3: P3)* and *req (p2: P2, p1: P1, p3: P3)*.

It should be further investigated whether a generalization of the notion of constraint-oriented composition is possible and desirable. Besides supporting the cooperating main service approach, such a generalization may also be useful to simplify the definition of generic structures for application protocols (e.g., the composition of the basic protocol function, the protocol transfer function, and the protocol mapping function; see Chapter 7, Application Protocol Reference Architecture).

8.7 Specification language support

The design framework, design model and reference architecture proposed in this thesis have been presented independently of any specification language or formal semantics. Instead, ad-hoc textual and graphical notations have been used. This approach was intentional since it enabled us to concentrate on concepts, techniques and methods necessary to support application protocol design, without being forced to consider at the same time possible limitations of specification languages. The ad-hoc notation allowed us to express the concepts that were introduced, and to present relatively simple examples, but no (serious) attempts have been made to make it applicable to realistic designs.

Further work is therefore necessary to select or develop appropriate specification languages, language-based design methods, tools, etc. that can support the concepts, techniques and methods for application protocol design proposed in this thesis.

References

[Bannon91] Bannon, L.J. and Schmidt, K., CSCW: four characters in search of a context, In: *Studies in Computer Supported Cooperative Work - Theory, Practice and Design*, Bowers, J.M., and Benford, S.D. (editors), North-Holland, 1991, 3-17.

[Bearman94] Bearman, M.Y., ODP-trader, In: *Open Distributed Processing, II*, Meer, J. de, Mahr, B., and Storp, S. (editors), Elsevier Science Publishers B.V. (North-Holland), 1994, 37-51.

[Bochmann92] Bochmann, G. von, Poitier, S., and Mondain-Monval, P., Object-oriented design for distributed systems and OSI standards, In: *Upper Layer Protocols, Architectures and Applications*, Neufeld, G., and Plattner, B. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 265-280.

[Brégant92] Brégant, G., Towards a convergence between telecommunication services architectures and open distributed processing, In: *Open Distributed Processing*, Meer, J. de, Heymer, V., and Roth, R. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 155-166.

[Davenport90] Davenport, T.H., and Short, J.E., The new industrial engineering: information technology and business process redesign, *Sloan Management Review*, Summer 1990, 11-27.

[Feldhoffer92] Feldhoffer, M., Object-oriented modelling of the application layer structure, In: *Upper Layer Protocols, Architectures and Applications*, Neufeld, G., and Plattner, B. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 235-247.

[Insulander93] Insulander, J., Schoo, P., Tönnby, I., and Trigila, S., An architectural approach to integrated service engineering for an open telecommunication service market, In: *Proceedings of the RACE International Conference on Intelligence in Broadband Services and Networks*, November 1993, Paris, France.

[Key90] Key, M., Leask, S., and Oshisanwo, A., ROSA: an object-oriented architecture for open services, *BT Technology Journal*, Vol. 8, No. 4, October 1990, 41-49.

[Kung92] Kung, R., Rationale for intelligent networks, In: *Open Distributed Processing*, Meer, J. de, Heymer, V., and Roth, R. (editors), Elsevier Science Publishers B.V. (North-Holland), 1992, 69-79.

[Raymond94] Raymond, K.A., Reference model of open distributed processing: a tutorial, In: *Open Distributed Processing, II*, Meer, J. de, Mahr, B., and Storp, S. (editors), Elsevier Science Publishers B.V. (North-Holland), 1994, 3-14.

Chapter 9

Summary of Conclusions

This chapter summarizes the conclusions of this thesis. We categorize the conclusions by their relation to design quality criteria, state-of-the-art description and evaluation, the design framework, the design model, and the reference architecture.

Design quality criteria

The design of application protocols should be guided by design quality criteria. Although effectiveness and efficiency (cost/performance ratio) are important desirable qualities of application protocol realizations, no criteria related to these qualities can be formulated at an architectural level. Criteria related to ‘ease of use’, however, can be formulated at this level. Application of these criteria leads to ‘clean’ designs, which are found to be the right basis for deriving effective and efficient realizations. The criteria have been characterized by the particular aspects of ease of use they pursue: consistency, orthogonality, propriety, and generality.

State-of-the-art description and evaluation: OSI upper layer architecture and model

The OSI reference model identifies three layers of application protocol functionality. The two lower layers, viz. the Session and the Presentation Layer, are involved in the support of all distributed applications. The highest layer, the Application Layer, consists of a collection of Application Service Elements (ASEs) and Control Functions (CFs). Depending on the distributed application that must be supported, different compositions of ASEs and CFs can be selected with the Association Control Service Element (ACSE). The ACSE is a special ASE that itself is always involved in the support of distributed applications.

This structure and the assignment of functions to entities in this structure have been criticized on basis of design quality criteria. The most important criticisms are:

- the OSI upper layer architecture (OSI-ULA) is a static architecture which is not suitable for all distributed applications. The flexibility of the Application Layer should

apply to the OSI-ULA as a whole, i.e. no static protocol layer hierarchy should be enforced;

- transfer syntax constraints (enforcing a common encoding for PDUs) should not be assigned to a 'central' entity, such as the Presentation Layer in the OSI-ULA. Each distinct application protocol should define abstract syntax constraints (enforcing valid PDU formats) and the corresponding transfer syntax constraints.
- segmentation/reassembly and concatenation/separation should be performed by the Transport Layer and not, as is currently the case, by the Session Layer;
- the (re)synchronization functions of the Session Layer are fine examples of political compromises that lead to poor technical design solutions.

Some of these deficiencies are due to the poor definition of service concepts and of Application Layer concepts. As a consequence, service decomposition methods are not defined, and protocols are not designed on basis of required services, with proper consideration of quality design criteria. Adequate design methods and design concepts can help to reduce the, currently long, development times of application protocol standards.

Despite the quality deficiencies, derived application protocol realizations are not necessarily inefficient. If an implementation is made of the combined functionality of the Session, Presentation, and Application Layer protocol entities, the implementor has considerable freedom to restructure functions, and to optimize local protocol processing. On the other hand, the process of restructuring adds complexity to the implementation phase.

Design framework

A design can be structured with different intentions or for different reasons. Two important structuring domains have been distinguished:

- behaviour domain: in this domain, behaviour and behaviour structure is defined. Relevant primitive concepts in this domain are action, interaction, and causality relation; and
- entity domain: in this domain, entities are defined with associated behaviour. Relevant primitive concepts in this domain are action point and interaction point.

Furthermore, three related abstraction levels have been identified that support distributed system design:

- interaction system perspective: abstracts form the individual responsibility of interacting entities;
- integrated system perspective: abstracts from the internal behaviour (implementation) of an entity;

- distributed system perspective: defines the internal behaviour of an entity in terms of interacting sub-entities.

Based on these abstraction levels, an application protocol design trajectory has been defined. This design trajectory is a simplified model of the application protocol design process, with design milestones that run from an enterprise service to an application protocol architecture.

Design model

Compositions of the primitive concepts of action, interaction and causality relation can be used to represent arbitrarily complex behaviours. Two behaviour structuring techniques have been presented based on composition rules with respect to these concepts:

- causality-oriented composition: behaviour components are related through the coupling of entry and exit points, forming causality relations between the components;
- constraint-oriented composition: behaviour components are related through contributions to shared interactions.

Different types of behaviour decomposition and refinement have been identified that support the design steps in the application protocol design trajectory:

- causality-oriented decomposition: decomposes an integrated behaviour such that a causality-oriented composition of that behaviour results;
- constraint-oriented decomposition: decomposes an integrated behaviour such that a constraint-oriented composition of that behaviour results;
- causality refinement: introduces new actions, but preserves the actions of the given behaviour; and
- action refinement: replaces (some) actions of the given behaviour by activities, where activities are compositions of actions;

These manipulations can in general not be automated, because of the design freedom involved. However, it is possible to verify whether a resulting behaviour is a correct decomposition/refinement of the given behaviour.

Reference architecture

The main purpose of a reference architecture is to help the designer in choosing a suitable architecture for his specific design, and to incorporate pre-defined building blocks. An application protocol reference architecture must be flexible enough to cope with the diverse requirements of distributed applications. It should therefore not define a single, static architecture, but it should allow the designer to select a suitable architecture from a set of possible architectures.

A flexible application protocol reference architecture has been derived by considering the design decisions concerning the structure of a design in the application protocol design trajectory. The top level structure of a distributed application consists of a layer of local processing functions (information processing entities), a layer of user-defined application protocols, and a layer of standardized application protocols. User-defined application protocols can use standardized application protocols as pre-defined building blocks.

Application protocols can be further structured in terms of protocol layers, and protocol layers in terms of protocol sections. The structuring techniques used here are constraint-oriented composition and causality-oriented composition. The number of protocol layers, and the choice and composition of protocol sections depends on the class of distributed applications that must be supported, and should therefore be decided by the designer. The concept of protocol sections also allows to integrate the layer of user-defined application protocols and the layer of standardized application protocols: a single application protocol layer can in principle be composed from user-defined protocol sections and standardized protocols sections.

Two design methods have been presented that support the development of protocol layers and protocol sections.

Some application protocol sections that can be used as very general application protocol building blocks have been identified and characterized: association establishment, inter-stream synchronization, intra-stream synchronization, activity synchronization, commitment control, and dialogue control.

Appendix

Textual Notation Syntax Definition

This appendix defines the syntax of the textual notation used in this thesis to represent behaviour. The definition is expressed in Backus-Naur Form.

A.1 Behaviour definition

behaviour-definition = behaviour-identifier "*:=*" definition-block.
definition-block = "*{*" [synchronization-requirements-list] causality-relation
{"," causality-relation} [local-definitions] "*}*".
local-definitions = "*where*" behaviour-definition {behaviour-definition}.

A.2 Synchronization requirements

synchronization-requirements-list =
"(" synchronization-requirement {*";"* synchronization-requirement} *")"*.
synchronization-requirement =
behaviour-identifier *","* behaviour-identifier {*","* behaviour-identifier} *":"*
(action-identifier | interaction-identifier) {*","* (action-identifier | interaction-identifier)}.

A.3 Causality relation

causality-relation = condition "→" result.
condition = simple-condition | composite-condition.
simple-condition =
start | entry-point | occurrence-condition | non-occurrence-condition | behaviour-exit.
start = *"start"* [attributes-list [constraints-list]].
entry-point = entry-identifier [attributes-list [constraints-list]].
occurrence-condition = enabling-action | enabling-interaction-contribution.
non-occurrence-condition =
("¬" disabling-action) | ("¬" disabling-interaction-contribution).
behaviour-exit = behaviour-identifier *"("* exit-point {*","* exit-point} *")"*.

result = result-action | result-interaction-contribution | exit-point | behaviour-entry.
exit-point = exit-identifier [attributes-list [constraints-list]].
composite-condition =
    (simple-condition ("∧" | "∨") (simple-condition | composite-condition)) |
    "(" (simple-condition ("∧" | "∨") (simple-condition | composite-condition)) *")"*.

A.4 Action and interaction

enabling-action = action.
disabling-action = action.
result-action = action.
action = action-identifier [attributes-list [constraints-list]].
enabling-interaction-contribution = interaction-contribution.
disabling-interaction-contribution = interaction-contribution.
result-interaction-contribution = interaction-contribution.
interaction-contribution = interaction-identifier [attributes-list [constraints-list]].

A.5 Attributes

attributes-list = "*(*" location-time-information-list | (*"|"* functionality-attribute) |
    (location-time-information-list "|" functionality-attribute) "*)*".
location-time-information-list =
    (location-attribute ["," time-information-list]) | time-information-list.
time-information-list = (time-attribute ["," information-attribute]) | information-attribute.
location-attribute = location-value-identifier "*:*" "*Location*".
time-attribute = time-value-identifier "*:*" "*Time*".
information-attribute = information-element {"," information-element}.
information-element = information-element-value-identifier "*:*"
    information-type-identifier.
functionality-attribute = extended-location-time-information-list.
extended-location-time-information-list =
    (extended-location-attribute ["," extended-time-information-list]) |
    extended-time-information-list.
extended-time-information-list =
    (extended-time-attribute ["," information-attribute]) | information-attribute.
extended-location-attribute = location-attribute {"," location-attribute}.
extended-time-attribute = time-attribute {"," time-attribute}.

A.6 Constraints

constraints-list = "*[*" constraint {"," constraint} "*]*".
constraint = simple-constraint | composite-constraint.

simple-constraint =
    (attribute-value-identifier ”=” term-expression) |
    (term-expression operation-identifier term-expression).
term-expression = simple-term | composite-term.
simple-term = attribute-value-identifier | attribute-value.
composite-term =
    (simple-term operation-identifier (simple-term | composite-term)) |
    ”(” (simple-term operation-identifier (simple-term | composite-term)) ”)”.
composite-constraint =
    (simple-constraint ("∧" | "∨") (simple-constraint | composite-constraint)) |
    ”(” (simple-constraint ("∧" | "∨") (simple-constraint | composite-constraint)) ”)”.

A.7 Identifiers

behaviour-identifier = identifier.
entry-identifier = "*entry*" {digit}.
exit-identifier = "*exit*" {digit}.
action-identifier = identifier.
interaction-contribution-identifier = underlined-identifier.
attribute-value-identifier =
    location-value-identifier | time-value-identifier | information-element-value-identifier.
location-value-identifier = identifier.
time-value-identifier = identifier.
information-element-value-identifier = identifier.
information-type-identifier = identifier.
attribute-value = identifier.
operation-identifier = identifier | special-character.
identifier = letter [{normal-character}].
underlined-identifier = underlined-letter [{underlined-normal-character}].

A.8 Characters

digit = "*0*" | "*1*" | ... | "*9*".
underlined-digit = "*0*" | "*1*" | ... | "*9*".
letter = "*a*" | "*b*" | ... | "*z*" | "*A*" | "*B*" | ... | "*Z*".
underlined-letter = "*a*" | "*b*" | ... | "*z*" | "*A*" | "*B*" | ... | "*Z*".
normal-character = letter | digit.
underlined-normal-character = underlined-letter | underlined-digit.
special-character = "+" | "-" | "×" | "÷" |"=" | "<" | ">" | "≤" | "≥".

Samenvatting

Dit proefschrift behandelt het systematisch ontwerpen van applicatieprotocollen. Het laagste niveau van abstractie waarop applicatieprotocollen worden beschouwd in dit proefschrift is dat van protocolarchitectuur. Een protocolarchitectuur is een voorschrift voor het implementeren van een protocol en omvat de condities die noodzakelijk maar voldoende zijn om protocolentiteiten onafhankelijk van elkaar te kunnen implementeren. Protocolarchitecturen worden gestandaardiseerd door internationale organisaties, zoals ISO en ITU-TSS, teneinde eindgebruikers onafhankelijk te maken van fabrikant- en leverancierspecifieke protocolproducten.

Een belangrijke referentie-architectuur voor het ontwikkelen van protocolstandaarden, inclusief applicatieprotocolstandaarden, is het ‘Reference Model for Open Systems Interconnection’ (OSI-RM). Het OSI-RM beschrijft een gelaagde applicatieprotocolarchitectuur, bestaande uit drie lagen: de Applicatielaag, de Presentatielaag en de Sessielaag. De Presentatielaag en de Sessielaag zijn ‘vaste’ lagen: alle gedistribueerde computertoepassingen moeten van deze lagen gebruik maken. De Applicatielaag daarentegen omvat ‘Application service elements’ (ASEs) die elk een bepaalde samenwerking voorschrijven en die geselecteerd worden afhankelijk van de behoefte aan een dergelijke samenwerking. Het is dus mogelijk dat gedurende bepaalde fases in een gedistribueerde samenwerking geen enkel ASE betrokken is.

De applicatieprotocolarchitectuur beschreven door het OSI-RM heeft een aantal belangrijke nadelen: (1) de ‘vaste’ lagen in de applicatieprotocolarchitectuur maken dat deze architectuur niet geschikt is voor het ondersteunen van alle mogelijke gedistribueerde computertoepassingen. Bovendien leidt het gebruik van ’vaste’ applicatieprotocollagen tot omvangrijke, onderling afhankelijke en ingewikkelde standaarden, die daardoor moeilijk te gebruiken en te onderhouden zijn; (2) de concepten waarop de ‘flexibele’ Applicatielaag berust zijn niet duidelijk beschreven. De potentiële voordelen van deze laag, zoals een betere scheiding van ontwerpzorgen, kunnen hierdoor niet worden benut; en (3) de ontwikkeling van applicatieprotocolstandaarden kost in het algemeen erg veel tijd.

Deze problemen zijn voor een belangrijk deel te wijten aan het ontbreken van ontwerpmethoden in het OSI-RM. Concepten die een belangrijke rol kunnen spelen in ontwerpmethoden, zoals het concept ‘service’, zijn niet duidelijk gedefinieerd en worden (hierdoor) niet optimaal benut. Methoden voor service-decompositie komen daardoor als vanzelfsprekend niet aan de orde. Een ander probleem is de onduidelijke relatie met andere ontwerpdisciplines voor gedistribueerde systemen, met name het ontwerpen van

systemen voor gedistribueerde verwerking. Als gevolg hiervan is het moeilijker om de behoefte aan applicatieprotocollen te identificeren tijdens het ontwerpen van complexe gedistribueerde systemen waarbij meerdere ontwerpdisciplines betrokken zijn. De integratie van gedistribueerde verwerking en applicatieprotocollen wordt beschouwd in het 'Reference Model of Open Distributed Processing' (ODP-RM), maar de genoemde relatie is ook hier niet duidelijk beschreven.

Bij het ontwerpen van applicatieprotocollen is het nuttig om een onderscheid te maken tussen gedrag en functionele entiteiten waaraan gedrag wordt toegewezen. Het structureren van gedrag wordt hierdoor mogelijk zonder de samenstelling van een systeem uit deelsystemen te impliceren. Het onderscheiden van een gedragsdomein geeft aanleiding tot het identificeren van de elementaire concepten 'actie', 'interactie', en 'causale relatie'. Het onderscheiden van een entiteitendomein leidt tot de identificatie van de elementaire concepten 'actiepunt' en 'interactiepunt'.

Het onderscheiden van gerelateerde abstractieniveaus vormt een goed uitgangspunt voor het definiëren van ontwerpmethoden. In dit proefschrift worden abstractieniveaus gedefinieerd waarbij: (1) een systeem beschouwd wordt als een samenstelling van deelsystemen, (2) het systeem beschouwd wordt onafhankelijk van zijn mogelijke samenstellingen, en (3) de interacties tussen het systeem en zijn omgeving beschouwd worden onafhankelijk van de rol van het systeem (en de omgeving) in deze interacties. Deze abstractieniveaus kunnen voor de beschrijving van willekeurig systemen worden gebruikt, dus ook voor deelsystemen binnen systemen. Op deze manier is het mogelijk om een ontwerptraject voor applicatieprotocollen te definiëren, waarin relevante ontwerpresultaten in het ontwerpproces worden geordend op basis van de corresponderende abstractieniveaus. Deze ontwerpresultaten worden 'ontwerpmijlpalen' genoemd. Opeenvolgende mijlpalen bakenen ontwerpstappen af waarin specifieke ontwerpdoelen worden nagestreefd.

Mijlpalen en ontwerpstappen vereisen dat gedrag kan worden gerepresenteerd en gemanipuleerd. Gedrag kan worden gedefinieerd in termen van acties, interacties en hun causale relaties. Om de rol van een functionele entiteit in interacties met andere entiteiten te kunnen definiëren is een structureringstechniek nodig die we 'constraint-oriented' gedragscompositie hebben genoemd. Deze techniek kan ook gebruikt worden om gedrag te structureren, onafhankelijk van een toewijzing aan functionele entiteiten. Daarnaast is een structureringstechniek nodig die het mogelijk maakt deelgedragingen te relateren, zodanig dat (inter)acties van een deelgedrag afhankelijk gemaakt worden van (inter)acties van andere deelgedragingen. Deze techniek hebben we de 'causality-oriented' gedragscompositie genoemd. Het manipuleren van gedrag in ontwerpstappen is gericht op het decomponeren of het verfijnen van een gegeven gedrag. Gedragsmanipulaties die een rol spelen in het applicatieprotocol ontwerptraject zijn geïdentificeerd en correctheidseisen aan deze manipulaties zijn geformuleerd.

Met behulp van de geïdentificeerde abstractieniveaus, gedragsmanipulaties en structureringstechnieken is het mogelijk specifieke applicatieprotocol ontwerpmethoden te definiëren. Twee methoden worden gepresenteerd die naast elkaar gebruikt kunnen worden. Beide methoden hebben als uitgangspunt een vereiste applicatieservice. Een van de ontwerpmethoden leidt tot een gelaagde structuur van applicatieprotocollen. Een applicatieprotocol samengesteld uit protocollagen wordt beschreven met ‘constraint-oriented’ gedragscompositie.

Zoals eerder opgemerkt hebben ‘vaste’ lagen belangrijke nadelen. Het is beter de ontwerper van een applicatieprotocol het aantal en de keuze van lagen te laten bepalen, en geen ‘vaste’ lagen voor te schrijven. Omdat hergebruik van complete lagen beperkt is, is nog een andere, horizontale structurering nodig die hergebruik mogelijk maakt van delen van een laag. De tweede ontwerpmethode leidt tot een structurering van applicatieprotocollen in deelgedragingen. Een deelgedrag van een protocol hebben we ‘protocolsectie’ genoemd. Het concept ‘protocolsectie’ is vergelijkbaar met dat van ASE, behalve dat protocolsecties gedefinieerd kunnen worden in willekeurige lagen. Een applicatieprotocol samengesteld uit protocolsecties wordt beschreven met ‘causality-oriented’ gedragscompositie. De compositie van applicatieprotocollen met behulp van protocollagen en protocolsecties vormt de basis voor een flexibele referentie-architectuur voor applicatieprotocollen.

Het proefschrift is als volgt gestructureerd:

- *Hoofdstuk 1* presenteert een globale probleembeschrijving voor het onderzoek, en de afbakening, de doelstellingen en de methode van aanpak van het onderzoek. Verder worden een aantal concepten geintroduceerd die van algemeen belang zijn bij het ontwerpen van gedistribueerde systemen.
- *Hoofdstuk 2* bespreekt kwaliteitscriteria die gebruikt kunnen worden om het ontwerpen van applicatieprotocollen te sturen en om reeds ontworpen applicatieprotocollen te evalueren.
- *Hoofdstuk 3* bespreekt de OSI applicatieprotocolarchitectuur en de hiermee gerelateerde concepten. Het bevat ook een korte beschrijving van de belangrijkste applicatieprotocolstandaarden die ontwikkeld zijn in het kader van het OSI-RM.
- *Hoofdstuk 4* evalueert de OSI applicatieprotocolarchitectuur en de hiermee gerelateerde concepten. Het bespreekt tevens de relatie tussen de kwaliteit van applicatieprotocolstandaarden en de aard van standaardisatie, en de implementatievrijheid die geboden wordt door protocolstandaarden.
- *Hoofdstuk 5* presenteert een raamwerk voor het ontwerpen van applicatieprotocollen. Het identificeert elementaire concepten voor gedrag en voor functionele entiteiten, en verschillende abstractieniveaus waarop willekeurige systemen kunnen worden gerepresenteerd. Op basis hiervan wordt een applicatieprotocol ontwerptraject beschreven

dat bestaat uit een aantal achtereenvolgende ontwerpstappen tussen mijlpalen in het applicatieprotocol ontwerpproces.

- *Hoofdstuk 6* bespreekt een model dat gebruikt kan worden in het applicatieprotocol ontwerptraject voor het representeren en het manipuleren van gedrag. Het model bevat, naast de eerder genoemde elementaire ontwerpconcepten, concepten en regels voor het samenstellen van gedrag. Twee technieken voor het structureren van gedrag worden beschreven en een aantal vormen van gedragsdecompositie en -verfijning worden geïdentificeerd.
- *Hoofdstuk 7* presenteert een flexibele referentie-architectuur voor applicatieprotocollen. Deze referentie-architectuur is gebaseerd op de compositie van protocollagen en van protocolsecties, waarbij gebruik wordt gemaakt van respectievelijk de ‘constraint-oriented’ en de ‘causality-oriented’ structureringstechniek. De architectuur omvat ook de karaterisering van een aantal applicatieprotocolsecties die als bouwstenen in veel verschillende toepassingen gebruikt kunnen worden.
- *Hoofdstuk 8* beschrijft enige suggesties voor verder werk.
- *Hoofdstuk 9* presenteert een samenvatting van de conclusies uit de vorige hoofdstukken.