Service-Oriented Systems Engineering: Modeling Services and Layered Architectures

Manfred Broy

Institut für Informatik, Technische Universität München
D-80290 München Germany
broy@in.tum.de
http://wwwbroy.informatik.tu-muenchen.de

Abstract. Based on the FOCUS theory of distributed systems (see [Broy, Stølen 01]) that are composed of interacting components we introduce a formal model of services and layered architectures. In FOCUS a component is a *total* behavior. In contrast, a service is a *partial* behavior. A layer in a layered architecture is a service with two service interfaces, an import and an export interface. A layered architecture is a stack of several layers. For this model of services and service layers we work out specification and design techniques for layers and layered architectures. Finally we discuss more specific aspects of layered architectures such as refinement and layer models in telecommunication.

Keywords: Service Engineering, Software Architecture, Layered Architecture

1 Introduction

Software development is today one of the most complex and powerful tasks in engineering. Modern software systems typically are *embedded* in technical or organizational processes and support those. They are *deployed* and *distributed* over large networks; they are *dynamic*, and accessed *concurrently* via a couple of independent user interfaces. They are based on software infrastructure such as operating systems and middleware and use the service of object request brokers.

Large software systems are typically built in a modular fashion and structured into components. These components are grouped together in software architectures. Software architectures are typically structured in layers. It is well known that hierarchies of layered architectures provide useful structuring principles for software systems. These ideas go back to "structured programming" according to Dijkstra and to Parnas (see [Parnas 72]).

The purpose of this paper is to present a comprehensive theory that captures the notions of services and those of layers and layered architectures in terms of services. It is aiming at a basis for a more practical engineering approach to services and the design of layered architectures, which is not within the scope of this paper.

H. König, M. Heiner, and A. Wolisz (Eds.): FORTE 2003, LNCS 2767, pp. 48–61, 2003.

In this paper we study semantic models of services and layered architectures. We introduce a mathematical model of layers. The purpose of this theory is to provide a basis for an engineering method for the design and specification of layered architectures.

The paper is organized as follows: in chapter 2 we define the notion of a service and in chapter 3 layered architectures in terms of services. We introduce the notion of a service and that of a component. We give an abstract semantic model of software component interfaces and of service interfaces. On this basis we define a model for layered architectures. According to this model we introduce and discuss specification techniques for layered architectures. Finally, we study specific aspects of service layers and layered architectures such as refinement, the extension of the layered architecture, and the application of the idea of layered architectures in telecommunication.

2 Components and Services

In this section we introduce the syntactic and semantic notion of a *component interface* and that of a *service*. Since services are partial functions a suggestive way to describe them are assumption/commitment specifications. We closely follow the FOCUS approach explained in all its details in [Broy, Stølen 01]. It provides a flexible modular notion of a component and of a service, too.

2.1 Interfaces, Components, and Services

In this section we define the concepts of a component, an interface, and a service. These three concepts are closely related. All three are based on the idea of a data stream and a behavior as a relation on data streams.

2.1.1 Streams

We introduce the notion of a component based on the idea of a data stream. Throughout this paper we work with only a few simple notations for data streams.

Streams are used to represent histories of communications of data messages in a time frame. Given a message set M we define a timed stream by a function

$$s: \mathbb{N} \to M^*$$

where M^* is the set of sequences over M. For each time t the sequence s(t) denotes the sequence of messages communicated at time t in the stream s.

We use channels as identifiers for streams in systems. Let I be the set of input channels and O be the set of output channels. With every channel c in the channel set $I \cup O$ we associate a data type Type(c) indicating the type of messages sent along that channel. A data type is in our context simply a data set. Let C be a set of channels with types assigned by the function

$$\text{Type: } C \to \text{TYPE}$$

Here TYPE is a set of types $\tau \in$ TYPE, which are carrier sets of data elements. Let M be the universe of all messages. This means

$$M = \bigcup \{\tau: \tau \in \text{TYPE}\}$$

The concept of a stream is used to define the concept of a channel history. A channel history is given by the messages communicated over a channel.

Definition. Channel history
Let C be a set of typed channels; a channel history is a mapping

$$x : C \to (\mathbb{N} \to M^*)$$

such that x.c is a stream of type Type(c) for each $c \in C$. Both by $\mathbb{H}(C)$ as well as by $\dot{C}$ the set of channel histories for the channel set C is denoted. ❑

We use, in particular, the following notations for a timed stream s:

z^s	concatenation of a sequence or stream z to a stream s,
s.k	k-th sequence in the stream s,
s↓k	prefix of the first k sequences in the timed stream s,
$\bar{s}$	finite or infinite (nontimed) stream that is the result of concatenating all sequences in s.

Note that $\bar{s}$ defines a time abstraction for the timed stream s. Similarly we denote for a channel valuation $x \in \dot{C}$ by $\bar{x}$ its time abstraction, defined for each channel $c \in C$ by the equation

$$\bar{x}.c = \overline{x.c}$$

The operators easily generalize to sets of streams and sets of valuations by element wise application.

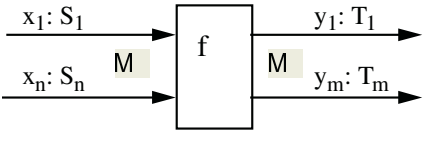

Fig. 1. Graphical Representation of a Component as a Data Flow Node with Input Channels x1, ..., xn and Output Channels y1, ... , ym and their Types

Given two disjoint sets C and C' of channels with $C \cap C' = \emptyset$ and histories $z \in \mathbb{H}(C)$ and $z' \in \mathbb{H}(C')$ we define the *direct sum* of the histories z and z' by $(z \oplus z') \in \mathbb{H}(C \cup C')$. It is specified as follows:

$$(z \oplus z').c = z.c \Leftarrow c \in C \qquad (z \oplus z').c = z'.c \Leftarrow c \in C'$$

The notion of a stream is essential for defining the behavior of components.

2.1.2 Components

Components have interfaces determined by their sets and types of channels. We describe the black box behavior of components by their interfaces.

An interface provides both a syntactic and semantic notion. We use the concept of a channel, a data type and a data stream to describe interfaces.

The syntactic interfaces define a kind of type for a component. The semantic interfaces characterize the observable behavior.

Definition. Syntactic interface
Let I be a set of typed input channels and O be the set of typed output channels. The pair (I, O) characterizes the syntactic interface of a component. By (I▶O) this *syntactic interface* is denoted. ❑

A component is connected to its environment exclusively by its channels. The syntactic interface indicates which types of messages can be exchanged but it tells nothing particular about the interface behavior.

For each interface (I▶O) we call (O▶I) the converse interface.

Definition. Interface of a Semantic component
A component interface (behavior) with the syntactic interface (I▶O) is given by a function

$$F : \dot{I} \to \wp(\dot{O})$$

that fulfills the following *timing property,* which axiomatizes the time flow. By F.x we denote the set of output histories of the component described by F. The timing property reads as follows (let $x, z \in \dot{I}$, $y \in \dot{O}$, $t \in \mathbb{N}$):

$$x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t+1 : y \in F(x)\} = \{y{\downarrow}t+1 : y \in F(z)\}$$

Here $x{\downarrow}t$ denotes the stream that is the prefix of the stream x and contains the first t finite sequences. In other words, $x{\downarrow}t$ denotes the communication histories in the channel valuation x until time interval t. ❑

The timing property expresses that the set of possible output histories for the first t+1 time intervals only depends on the input histories for the first t time intervals. In other words, the processing of messages within a component takes at least one time tick. We call functions with this property *time-guarded* or *strictly causal*.

As we will see in the following the notion of causality is essential and has strong consequences. We give a first simple example of these consequences of the causality assumption.

Let us consider the question whether we can have $F.x = \varnothing$ for a component with behavior F for some input history x. In this case, since $x{\downarrow}0 = \langle\rangle$ for all streams x, we get $x{\downarrow}0 = z{\downarrow}0$ for all streams z and by causality

$$\{y{\downarrow}1 : y \in F(x)\} = \{y{\downarrow}1 : y \in F(z)\} = \varnothing$$

for all streams x. Therefore the result of the application of a strictly causal function is either empty for all its input histories or F is "total", in other words $F.x \neq \varnothing$ for all x. In the first case we call the interface function *paradoxical*. In the later case we call the interface function *total*.

2.1.3 Services

A service has a syntactic interface like a component. Its behavior, however, is "partial" in contrast to the totality of a component interface. Partiality here means that a service is defined only for a subset of its input histories. This subset is called the service domain.

Definition. Service interface
A service interface with the syntactic interface (I ▶ O) is given by a function

$$F : \dot{I} \to \wp(\dot{O})$$

that fulfills the *timing property* only for the input histories with nonempty output set (let $x, z \in \dot{I}$, $y \in \dot{O}$, $t \in \mathbb{N}$):

$$F.x \neq \emptyset \neq F.z \wedge x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t+1 : y \in F(x)\} = \{y{\downarrow}t+1 : y \in F(z)\}$$

The set

$$Dom(F) = \{x : F.x \neq \emptyset\}$$

is called the *service domain*. The set

$$Ran(F) = \{y \in F.x : x \in Dom(F)\}$$

is called the *service range*. By

$$\mathbf{F}[I \blacktriangleright O]$$

we denote the set of all service interfaces with input channels I and output channels O. By $\mathbf{F}$ we denote the set of all interfaces for arbitrary channel sets I and O. ❑

In contrast to a component, where the causality requirement implies that for a component F either all output sets F.x are empty for all x or none, a service may be a partial function.

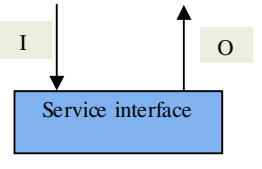

Fig. 2. Service Interface

To get access to a service, in general, certain access conventions have to be valid. We speak of a *service protocol*. Input histories x that are not in the service domain do not fulfill the service access assumptions. This gives a clear view: a nonparadoxical component is total, while a service may be partial. In other words a nonparadoxical component is a total service. For a component there are nonempty sets of behaviors for every input history. A service is close to the idea of a use case in object oriented analysis. It can be seen as the formalization of this idea. A service provides a partial view onto a component.

2.1.4 Composition of Components and Services

In this section we study the composition of components. Services and components are composed by parallel composition with feedback along the lines of [Broy, Stølen 01].

Definition. Composition of Components and Services
Given two service interfaces $F_1 \in \mathbb{F}[I_1 \blacktriangleright O_1]$ and $F_2 \in \mathbb{F}[I_2 \blacktriangleright O_2]$, we define a composition for the feedback channels $C_1 \subseteq O_1 \cap I_2$ and $C_2 \subseteq O_2 \cap I_1$ by

$$F_1[C_1 \leftrightarrow C_2]F_2$$

The component $F_1[C_1 \leftrightarrow C_2]F_2$ is defined as follows (where $z \in \mathbb{H}[I_1 \cup O_1 \cup I_2 \cup O_2]$, $x \in \mathbb{H}[I]$ where $I = I_1 \backslash C_2 \cup I_2 \backslash C_1$):

$$(F_1[C_1 \leftrightarrow C_2]F_2).x = \{z|(O_1 \backslash C_1) \cup (O_2 \backslash C_2): x = z|I \wedge z|O_1 \in F_1(z|I_1) \wedge z|O_2 \in F_1(z|I_2)\}$$

The channels in $C_1 \cup C_2$ are called *internal* for the composed system $F_1[C_1 \leftrightarrow C_2]F_2$.❑

The idea of the composition of components and services as defined above is shown in Fig. 3.

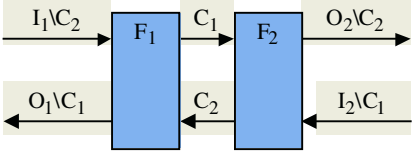

Fig. 3. Composition $F_1[C_1 \leftrightarrow C_2]F_2$ of Services or Components

In a composed component $F_1[C_1 \leftrightarrow C_2]F_2$ the channels in the channel sets C_1 and C_2 are used for internal communication.

Parallel composition of independent sets of internal channels is associative. If

$$(I \cup O) \cap (I' \cup O') = \varnothing$$

then

$$(F_1[I \leftrightarrow O]F_2)[I' \leftrightarrow O']F_3 = F_1[I \leftrightarrow O](F_2[I' \leftrightarrow O']F_3)$$

The proof of this equation is straightforward.

The set of services and the set of components form together with compositions an algebra. The composition of components (strictly causal stream functions) yields components and the composition of services yields services.

Composition is a partial function on the set of all components and the set of all services. It is only defined if the syntactic interfaces fit together.

3 Layers and Layered Architectures

In this section we introduce the notion of a *service layer* and that of a *layered architecture* based on the idea of a component interface and that of a service. Roughly speaking a layered software architecture is a family of components forming

layers in a component hierarchy. Each layer defines an upper interface called the *export interface* and makes use of a lower interface called the *import interface*.

3.1 Service Layers

In this section we introduce the notion of a service layer. A service layer is a service with a syntactic interface decomposed into two complementary subinterfaces. Of course, one might consider not only two but many separate interfaces for a system building block – however, considering two interfaces is enough to discuss most of the interesting issues of layers.

3.1.1 Service Layers

A layer is a service with (at least) two syntactic interfaces. Therefore all the notions introduced for services apply also for service layers.

Definition. Service Layer
Given two syntactic service interfaces $(I \blacktriangleright O)$ and $(O' \blacktriangleright I')$ where we assume $I \cap O' = \emptyset$ and $O \cap I' = \emptyset$; the behavior of a service layer L is represented by a service interface

$$L \in \mathbb{F}[I \cup O' \blacktriangleright O \cup I']$$

For the service layer the first syntactic service interface is called the syntactic *upward interface* and the second one is called the syntactic *downward interface*. The syntactic service layer interface is denoted by $(I \blacktriangleright O/O' \blacktriangleright I')$. We denote the set of layers by $\mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$. ❑

The idea of a service layer interface is well illustrated by Fig. 4.

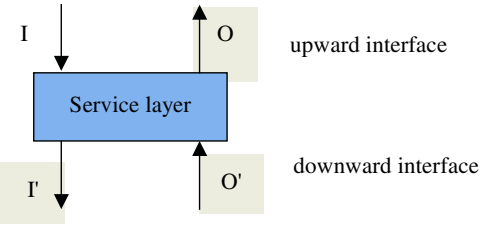

Fig. 4. Service Layer

From a behavioral view a service layer is itself nothing but a service, with its syntactic interface divided into an upper and a lower part.

3.1.2 Composition of Service Layers

A service layer can be composed with a given service to provide an upper service. Given a service interface $F' \in \mathbb{F}[I' \blacktriangleright O']$ called the *import service* and a service layer $L \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ we define its composition by the term

$$L[I' \leftrightarrow O']F'$$

This term corresponds to the small system architecture shown in Fig. 6. We call the layered architecture correct with respect to the *export service* $F \in \mathbf{F}[I \blacktriangleright O]$ for a provided import service F' if the following equation holds:

$$F = L[I' \leftrightarrow O']F'$$

The idea of the composition of layers with services is illustrated in Fig. 5. This is the parallel composition introduced before. But now we work with a structured view on the two interfaces.

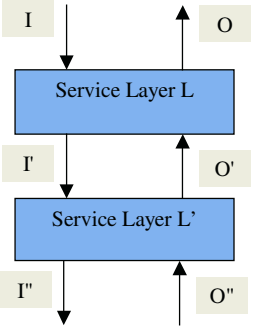

Fig. 5. Service Layer Composed of Two Service Layers

We may also compose two given service layers $L \in \mathbf{L}[I \blacktriangleright O/O' \blacktriangleright I']$ and $L' \in \mathbf{L}[O' \blacktriangleright I'/O'' \blacktriangleright I'']$ into the layer

$$L[I' \leftrightarrow O']L'$$

This term denotes a layer in $\mathbf{L}[I \blacktriangleright O/O'' \blacktriangleright I'']$. If we iterate the idea of service layers, we get hierarchies of layers also called *layered architectures* as shown in Fig. 7.

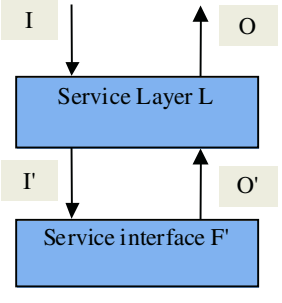

Fig. 6. Layered Architecture Formed of a Service and Service Layer

As Fig. 6 shows there are three services involved in a layer pattern for the service layer L:

- The *import* service $F' \in \mathbf{F}[I' \blacktriangleright O']$.

- The *export* service $F \in \mathbb{F}[I \blacktriangleright O]$ with $F = L[I' \leftrightarrow O']F'$.
- The *downward* service $G \in \mathbb{F}[O' \blacktriangleright I']$ which is the restriction of L to $(O' \blacktriangleright I')$.

The downward service G is the service "offered" by L to the downward layer; it uses the import service F'. We assume that all inputs to the downward service are within its service domain. Thus the proposition

$$\mathrm{Ran}(G) \subseteq \mathrm{Dom}(F') \quad (*)$$

is required. Vice versa all the output produced by F' on input from G is required to be in the domain of G:

$$\{y \in F'.x: x \in \mathrm{Ran}(F')\} \subseteq \mathrm{Dom}(G)$$

Actually the requirement (*) is stronger than needed, in general! If G does not use its whole range due to the fact, that F' does not use the whole domain of G then we can weaken the requirement $\mathrm{Ran}(G) \subseteq \mathrm{Dom}(F')$.

In fact, we may use a kind of invariant that describes the interactions between F' and G. However in top down system design it is more convenient to work with (*). This introduces a methodologically remarkable asymmetry between the services downward service G and the import service F'.

The idea of a layered architecture is illustrated in Fig. 7. It is characterized best by the family of export services $F_j \in \mathbb{F}[I_j \blacktriangleright O_j]$ for $0 \leq j \leq n$.

3.2 Specifying Service Layers

In this section we discuss how to characterize and to specify service layers. As we have shown, one way to specify layers is the assumption/commitment style.

We concentrate here on the specification of layers in terms of services.

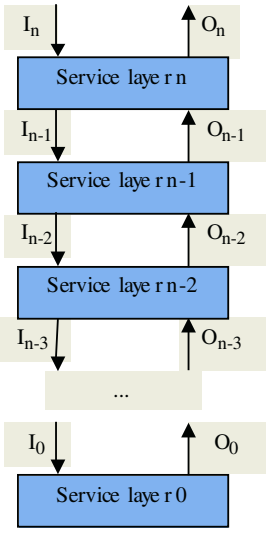

Fig. 7. Layered Architecture

3.2.1 Characterizing Layers by Their Import and Export Services

The idea of a layer is characterized best as follows: a service layer $L \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ offers an export service $F = L[O' \leftrightarrow I']F'$ provided an adequate import service $F' \in \mathbb{F}[I' \blacktriangleright O']$ is available. In general, a layer shows only a sensible behavior for a small set of import services F'. Therefore the idea of a layer is best communicated by the characterization and the specification of its required import and its provided export services.

Note, however, that a layer $L \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ is not uniquely characterized by a specification of its import and export service. In fact, given two services, an import service $F' \in \mathbb{F}[I' \blacktriangleright O']$ and an export service $F \in \mathbb{F}[I \blacktriangleright O]$ there exist, in general, many layers $L \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ such that the following equation holds

$$F = L[I' \leftrightarrow O']F'$$

In the extreme, the layer L is never forced to actually make use of its import service. It may never send any messages to but F' but realize this service by itself internally. This freedom to use an import service or not changes for two or multi-SAP layers (SAP = service access point) that support communication. We come back to this issue.

3.3 Export/Import Specifications of Layers

Typically not all input histories are good for an access to a service. Only those that are within the service domain and thus fulfill certain service assumptions lead to a well controlled behavior. This suggests the usage of assumption/commitment specifications for services as introduced above. The specification of layers is based on the specification of services.

A layer is a bridge between two services. In a layered architecture a layer exhibits several interfaces:

- the upward interface, also called the *export* service interface,
- the downward interface, the converse of which is also called the *import* service interface.

In a requirement specification of a layer we do not want to describe all behaviors of a layer and thus see the layer as a component, but only those that fit into the specific scheme of interactions. We are, in particular, interested in the specification of the behavioral relationship between the layer and its downward layer. There are three principle techniques to specify these aspects of a layer:

- We specify the interaction interface $S \subseteq \mathbb{H}(I' \cup O')$ between the layer and its downward service.
- We specify the layer $L \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ indirectly by specifying the export service $F \in \mathbb{F}[I \blacktriangleright O]$ and the import service $F' \in \mathbb{F}[I' \blacktriangleright O']$ such that F is a refinement of $L[I' \leftrightarrow O']F'$.
- We specify the layer $L \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ as a service $F_L \in \mathbb{F}[I \cup O' \blacktriangleright O \cup I']$.

All three techniques work in principle and are related. However, the second one seems from a methodological point of view most promising. In particular, to specify a layered architecture, we only have to specify for each layer the export service.

An interesting and critical question is the methodological difference we make between the two services associated with a layer, the export service and downward service.

4 More on Layered Architectures

In this section apply our approach of services and layered architectures to telecommunication applications. We deal with two classes of layered architectures. A in telecommunication service interface of a service, a component or a system is called a *service access point* (SAP). Note that our layers so far had only one SAP (the export service).

4.1 Double Layered Architectures

In telecommunication also layered architectures are used. The ISO/OSI layered protocols provide typical examples. But there are at least two (or actually many) service interfaces, for instance one of a sender and one of a receiver. We speak of *double layered architectures*.

4.1.1 Double SAP Services

The idea of a double service interface is well illustrated by Fig. 9. It shows a unit with two separated service interfaces (SAP). Formally it is again a layer. But in contrast to layers, the two service interfaces have the same role. There is no distinction of the two SAPs into an import and an export interface. Rather we have two *simultaneous* interfaces.

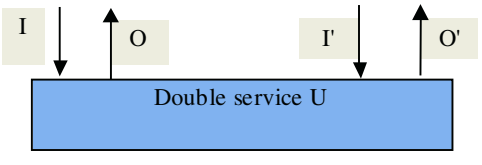

Fig. 8. Double SAP Service

From a behavioral view a double service $D \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ is formally a service layer, where its syntactic interface is divided instead of an upper and a lower part into a left and a right part.

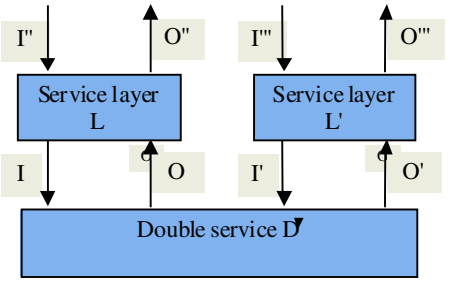

Fig. 9. Doubled Layered Architecture and Service Layer

In contrast to layers, which are best specified by their import and their export services we describe both SAPs by their offered services. So we describe the communication component by two export services or – to express how they relate – more precisely as one joint service.

4.1.2 Layers of Double Services

In fact, we now can associate a stack of layers with each of the service interfaces. This leads to a doubled layered architecture.

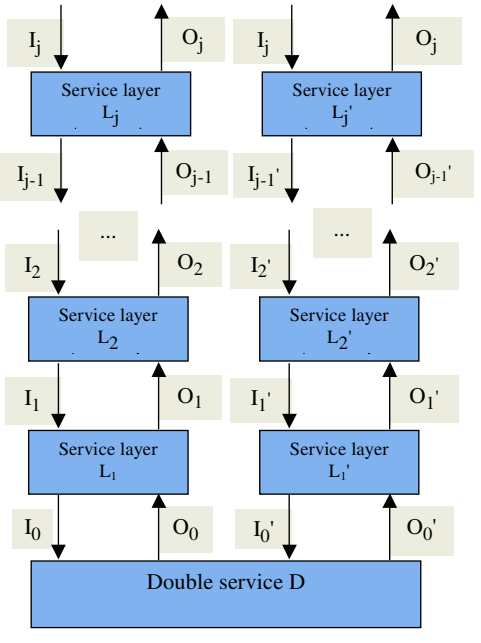

Fig. 10. Doubled Layered Architecture and Service Layer

A service layer can be composed with a service to provide an upper service. Given two service layers $L \in \mathbb{L}[I'' \blacktriangleright O''/O \blacktriangleright I]$ and $L' \in \mathbb{L}[I''' \blacktriangleright O'''/O' \blacktriangleright I']$ and a double layered architecture $D \in \mathbb{L}[I \blacktriangleright O/I' \blacktriangleright O']$ we construct a composed system called *layered communication architecture*

$$L[I \leftrightarrow O]D[I' \leftrightarrow O']L'$$

It is illustrated in Fig. 9.

As before we can iterate the layering as illustrated in Fig. 10. We obtain a diagram very similar to the ISO/OSI protocol hierarchy.

In principle, we can use all our techniques for describing and specifying the layers. In principle, there is no difference between the layers in layered architectures and those in communication architectures.

4.2 Layers as Refinement

In each syntactic interface a special layer is the identity. For each syntactic interface of a layer where the syntactic interfaces of the export and the import services coincide we get an identity.

Definition. Identity Layer
Given the syntactic service interface $(I \blacktriangleright O)$ the syntactic service layer interface is denoted by $(I \blacktriangleright O/I \blacktriangleright O)$; it is represented by a service interface

$$\mathbb{F}[I \cup O \blacktriangleright O \cup I]$$

$Id(I \cup O \blacktriangleright I \cup O) \in \mathbb{L}[I \blacktriangleright O/O \blacktriangleright I]$ is the service with $Id(x \oplus y) = \{x \oplus y\}$ for all $x \in \hat{I}$, $y \in \hat{O}$. ❑

For each service $F \in \mathbb{F}[I \blacktriangleright O]$ we get the equation

$$Id(I \blacktriangleright O)[I \leftrightarrow O]F' = F'$$

and for any layer $L \in \mathbb{L}(I \blacktriangleright O/O' \blacktriangleright I')$ we get the equation

$$Id(I \blacktriangleright O)\ [I \leftrightarrow O]\ L = L$$

These rules are quite straightforward. The more significant issue for identity is the definition of refinement pairs.

Definition. Refinement Pairs
Two layers $L \in \mathbb{L}[I \blacktriangleright O/O' \blacktriangleright I']$ and $L' \in \mathbb{L}[I' \blacktriangleright O'/O \blacktriangleright I]$ are called a *refinement pair* for $(I \blacktriangleright O / O \blacktriangleright I)$ if

$$L[I' \leftrightarrow O']L' = Id(I \blacktriangleright O)$$

In this case both L and L' do only change the representation of their input and output histories, but let all the information through. ❑

5 Summary and Outlook

Why did we present this quite theoretical setting of mathematical models of services, layers, layered architectures and relations between them? First of all, we want to show how rich and flexible the tool kit of mathematical models is and how far we are in integrating and relating them within the context of software design questions. In our case the usage of streams and stream processing functions is the reason for the remarkable flexibility of our model toolkit and the simplicity of the integration.

Second we are interested in a simple and basic model of a service and a layer just strong and rich enough to capture all relevant notions. Software development is a difficult and complex engineering task. It would be very surprising if such a task could be carried out properly without a proper theoretical framework. It would at the same time be quite surprising if a purely scientifically theoretical framework would be the right approach for the practical engineer. The result has to be a compromise as we have argued between formal techniques and theory on one side and intuitive notations based on diagrams. Work is needed along those lines including experiments and feedback from practical applications. But as already our example and experiment show a lot is to be gained that way.

Acknowledgements

It is a pleasure to thank Andreas Rausch and Bernhard Rumpe for stimulating discussions and helpful remarks on draft versions of the manuscript.

References

[Broy 91] M. Broy: Towards a formal foundation of the specification and description language SDL. Formal Aspects of Computing 3, 1991, 21-57

[Broy 97] M. Broy: Refinement of Time. M. Bertran, Th. Rus (eds.): Transformation-Based Reactive System Development. ARTS'97, Mallorca 1997. Lecture Notes in Computer Science 1231, 1997, 44-63

[Broy, Stølen 01] M. Broy, K. Stølen: Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement. Springer 2001

[Herzberg, Broy 03] D. Herzberg, M. Broy: Modelling Layered Distributed Communication Systems. To appear

[Parnas 72] D. Parnas: On the criteria to be used to decompose systems into modules. Comm. ACM 15, 1972, 1053-1058

[Room 94] B. Selic, G. Gullekson. P.T. Ward: Real-time Objectoriented Modeling. Wiley, New York 1994

[Zave, Jackson 97] P. Zave, M. Jackson: Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, January 1997