JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Specifying and Composing Layered Architectures

Manfred Broy* **and Bran Selić**†‡
*Technische Universität München, Germany
†Malina Software Corp., Canada
‡Monash University, Australia

ABSTRACT We define the layered architectural style as an architectural pattern and a structuring method in terms of precise specifications of interface behavior and modular composition, based on a formal model. Interface predicates and interface assertions are used to specify the interface behavior of systems, which is a description of the services that they require and provide. We give both a syntactic and a semantic description of the design pattern of layered system architectures. Moreover, we prove that the composition of multiple layers also generates layers, subject to the condition that the services provided by lower layers are refinements of the required services of upper layers. We demonstrate the approach by examples. We seek two goals: (a) to give a precise semantic characterization of the concept of layered architectures, and (b) to provide a method for specifying and for structuring layered system architectures. We show how to decompose services as offered and used by layered architectures into encapsulated subservices called functional features. A related issue is the identification of feature interactions between these subservices. Feature interactions between functional features can be identified by studying their interface behavior. A number of small examples is given to illustrate the introduced notions and concept.

KEYWORDS Architecture Specification, Verification, Layered Architecture, Feature Interaction.

1. Introduction

Layered structures are an essential way of designing the architecture of software systems, communication systems, and cyber-physical systems, among others. The big advantage of this approach is that it gives a very clear structuring style that supports modularity, encapsulation, information hiding, maintainability, changeability, and portability. At the same time, it also allows a crisp and incremental separation of concerns. One notable advantage is that layered architectures can be flexibly deployed onto distributed platforms.

These clear and undisputed advantages of layered architectures have led to its almost ubiquitous deployment in software engineering practice. It is fair to say that, at the highest level, most practical software systems exhibit some form of a layered architecture. Yet, despite this widespread adoption, there is still some lack of clarity and consensus regarding what constitutes the essential nature of layers and layered systems (Selić 2021).

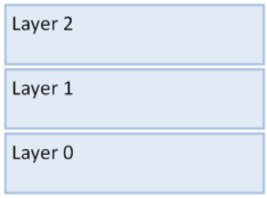

Figure 1 A common graphical approach to depicting layered system architectures

Consider, for example, the most common graphical representation of layers involved in a layering relationship, which is typically rendered as a vertical stack, as depicted in Figure 1. This representation is intended to convey the essential features of the layering relationship. The vertical arrangement suggests a hierarchical ordering, whereby layers lower in the stack "support" higher layers in some way.

Thus, layering is sometimes referred to as constituting "*vertical*" composition of components, in contrast to "*horizontal*" (or

JOT reference format:
Manfred Broy and Bran Selić. *Specifying and Composing Layered Architectures*. Journal of Object Technology. Vol. 23, No. 1, 2024.
http://dx.doi.org/10.5381/jot.2024.23.1.a2

"peer") composition (see (Mietzner et al. 2010)). Despite the intuitive appeal of this simple representation and terminology, it does not answer some important questions about this architectural design pattern. Moreover, although the concept of layering and its use in software architecture have been discussed extensively in literature (e.g., (Schmidt et al. 1996; Clements et al. 2003; Dijkstra 1983; Hofmeister et al. 2000; Selic 2020; Sarkar et al. 2009; Mary & David 1996; Zdun & Avgeriou 2005)) its precise semantics still remain elusive. For example, there is still an ongoing debate as to whether a given layer, such as Layer 2 in Fig. 1, can directly access only the layer immediately below, or can it also access layers further down in the hierarchy (e.g., Layer 0)[1]? More generally, what exactly is the nature of the dependencies between layers (i.e., what does it mean for one layer to "access" another)?

Without consensus on the precise nature and characterization of layers and layering it becomes difficult to analyze and assess the key properties or suitability of a particular layered design. Therefore, a precise and clear definition of the layering design pattern would provide a foundation for a formal specification, implementation, and verification of layered architectures. To that end, in this paper we propose a formally precise definition of layering and related concepts. The main result of this study is a discussion and a formal definition of the concepts of layer and platform, as well as a specification method for defining layers and a method for composing them to provide layered system architectures.

In what follows, we first provide an informal description of the key properties that uniquely characterize the layering design pattern. This is intended as background to help explain the rationale behind layering as well as for the formal model proposed in this work. Section 3 provides an informal overview of that approach, while more technical details are covered in the Appendix. This section also provides a formal definition of its two key concepts: *interface specifications* and *services*, as formally defined in (Broy et al. 2007) and explained in section 3. The *layer* concept itself is then defined formally in section 4, which also covers methods for composing layers to provide layered system architectures. In particular, the applied specification technique also introduces a calculus for proving properties of layers. To support more complex analyses, a more refined data flow in layers is studied in section 5. Section 6 discusses the decomposition of services into subservices called *features*, their relationship in terms of feature interactions, and the relationships between the services within a layer. Section 7 provides a more extensive illustrative example of a layered architecture. Finally, the summary in section 8 briefly discusses possible perspectives of this work, such as implementation issues, and other potential future research areas.

2. About the Layering Pattern

As commonly understood in the software engineering domain, *layering* is a structural and operational relationship between at least two functional modecifules called *layers*. It is a hierarchical and asymmetric relationship in the sense that there is a distinct dependency of the upper layer on its lower layers but not the other way around. More precisely, the correct operation of the higher layer is dependent on the correct operation of the lower layer, whereas the correct operation of the lower layer is independent of the upper layer.

2.1. Layers, Platforms, and Services

In general, the model of layers and layering adopted in this work is that lower layers provide *a set* of functional *capabilities* in terms of *services*, which can be utilized by the upper layer to realize its specified functionality. From that viewpoint, lower layers can be thought of as providing a *platform* for the upper layer. Perhaps the most obvious example of layering is the relationship between computer hardware – the lower layer – and the software that it executes – the upper layer.

As noted, a common architectural approach consists of stacking two or more layers, one on top of another, thereby allowing for a gradual realization of some desired complex functionality. For example, a firmware layer may first be placed "on top of" computer hardware. Its purpose is both to hide implementation details of the hardware that are irrelevant to higher layers, as well as to provide a conceptually simpler and, typically, semantically more suitable representation of the hardware to those layers. Next, an operating system layer of software is usually placed over the firmware. Operating systems generally provide certain widely useful functional capabilities (i.e., services), such as memory management, concurrency support, and interprocess communication, which are useful for realizing applications in the next layer above. When required, an application will select and invoke (e.g., using an API call) the appropriate operating system services. From that perspective, supporting (i.e., lower) layers can be viewed as providing an abstract execution environment, or "virtual machine", to upper layers.

It is crucial to note that, in the case of inter-layer interactions, platform *service requests are made directly by the implementation elements of an application*. This is because the selection of which platform services are to be used (and when) is strictly an application implementation decision, which, according to time-proven principles of modularity and information hiding, should be of no concern to external entities. From that perspective, a platform has a kind of "privileged" direct connection to the (otherwise private) implementations of the applications that it supports. In a sense, a *platform layer can be viewed as an external extension of the implementations of its applications*. Nevertheless, a platform remains fully encapsulated with respect to applications, which access its services strictly through the platform's provided interfaces. Moreover, due to the asymmetric dependency relationship between layers noted above, a platform layer is independent of its application layer and can exist and function regardless of whether its services are being used by applications. This particular arrangement is unique to layering and distinguishes it from any other types of client/server relationships[2].

[1] This issue of "layer jumping" is often considered to be a case of poor design. However, as argued in (Mietzner et al. 2010) the issue can be much more complicated than it appears.

[2] For a more detailed discussion of this topic, refer to (Hofmeister et al. 2000).

With regards to *how and why* an application uses the services of its platform, it is worth distinguishing two different uses case. In one of these – which we provisionally call a *semantically transparent* usage – the interaction between the upper and lower layers is exclusively a discretionary application design issue. That is, the selection of which platform services are used is driven purely by implementation concerns, and will have *no observable consequences* on the behavior that the application displays to external entities. In contrast, in case of a *semantically observable* usage, the invocation of a lower layer service *will* have an effect on an application's externally observable behavior. The most common example of this occurs in cyber-physical systems, whose function involves monitoring and control external physical devices. Such devices are necessarily connected to a computer via the computing hardware (e.g., through an interrupt mechanism), which is the bottommost layer in a system architecture. To provide access to these devices, they have to be "propagated upwards" to the appropriate application layer. This is generally done by means of a suitable platform service (this case is discussed in greater detail in sect. 5.3). For example, to control the brake of a vehicle, an automotive driving control application may need to invoke the "apply brake" service, which provides a simplified abstract view of what is likely a very complex hydro-mechanical mechanism. That type of service is often provided by a specialized application-specific framework platform layer, such as AUTOSAR[3]. Clearly, the ability to control the state of the brake is a first-order concern for a driving control application, and, consequently, invoking this service will have observable effects at the application level.

2.2. Platform Sharing and Feature Interactions

An often overlooked consequence of sharing a single platform between multiple (possibly independent) applications, is that undesirable "hidden" interactions between such applications can arise. This can occur when a given platform service is used concurrently by two or more applications, leading to concurrency conflicts. These effects, which are sometimes referred to as *feature interactions*, may take place directly in services of the layer immediately below the application. However, they are even more problematic if the interactions occur in layers deeper down in the layer stack, which makes them more difficult to trace and identify. In these types of situations, it is necessary to specify how data flows through the layers, so that the connection between the invocation of a required service and its effect can be traced. This issue is discussed in greater detail in sect. 6.1.

2.3. Capturing the Semantics of Layering

In what follows, we take advantage of the FOCUS approach (Broy & Stølen 2012) to capture the semantics of layering in a precise formal way. The essential nature of this approach is briefly introduced in the following section, and, in somewhat more detail, in the Appendix. However, in the most abstract sense, it can be characterized as a "black box" approach, whereby a functional module (or, *component*), is specified in purely in terms of its input-output behavior, without delving into its internal implementation details. In the specific case of layering, this means treating layers as modules characterized by their *required* and *provided* interfaces. The required interfaces of a layer represent the services that it needs of its supporting layers, whereas its provided interfaces capture its desired observable behavior.

Specifically, we do not only give a formal definition of layered architectures, but we also describe the decomposition of layers into a family of subservices called *functional features*. For these features, a critical question is about potential feature interactions between them, how to find and to model them, and how these can affect the services in higher layers. In the following section, we first introduce the Focus-based approach to specifying services by their behaviors as observed via their interfaces and also how systems defined in this way can be composed into larger systems. This provides the foundation for a formal specification of layers and layered architectures in sect. 4.

An important aspect here is the difference between *syntactic* characterizations of layered architecture ("who is allowed to call whom") and the *semantic* characterizations of layered architectures ("what are the required properties of the data flow between the interfaces of adjacent layers"). This is related to the question of how encapsulation of layers and their connections are defined and what semantic properties are required for layering. We define a layer as a system that can be used as a building block in a layered architecture. Its key property is that it *provides* services to its "higher" layers subject to the condition that it gets access to required services offered by layers directly below. This definition, of course, needs a precise definition of the notion of service (see (Broy et al. 2007) and sect. 3.2 below).

3. System Interface Behavior: Specifying Services and Systems

We specify systems and their services in terms of their *interface behavior*. Interface behavior is modelled by relations between input and output streams of data. Interface behavior is specified by interface predicates and interface assertions (for details, see the Appendix). Thus, a system or a service specification is given by a syntactic interface with a corresponding interface assertion. Interface behaviors are a formal model both of system interface behaviors and of services. In fact, a system shows an interface behavior that is called the *service(s) offered by the system*.

Let T_i and S_i be data types. Let $\text{TSTR}\ T_i$ and $\text{TSTR}\ S_i$ denote the data type of timed streams of elements of type T_i and S_i respectively (for a formal definition, see the Appendix). Given two sets of typed channel names:

$$X = \{x_1 : \text{TSTR}\ T_1, \ldots, x_n : \text{TSTR}\ T_n\}$$
$$Y = \{y_1 : \text{TSTR}\ S_1, \ldots, y_m : \text{TSTR}\ S_m\}$$

we denote a syntactic interface of a system by $(X \blacktriangleright Y)$ where X denotes the set of input channels and Y denotes the set of output channels of the system interface of a system or of a service. We call X and Y *signatures*. This way, interfaces consist

[3] https://www.autosar.org/

syntactically of channels where each channel is identified by an identifier with a specified data type indicating which types of data are communicated.

Given a signature X, we denote by $\vec{X}$ the valuation of the channels in X by the streams of the respective type. The elements of $\vec{X}$ are called *histories*. Formally, a history $x \in \vec{X}$ is a mapping from the set of channels X onto streams. A history is therefore just a family of streams with streams named by the corresponding channel identifiers. Given history $x \in \vec{X}$ and channel $c \in X$ we denote the stream associated in history x with channel c by $x(c)$.

Given $x \in \vec{X}$ and a subset $C \subseteq X$ we denote by $x \mid C$ the history which is the restriction of x to the channels in C. For $x \in \vec{X}$ and $y \in \vec{Y}$, where $X \cap Y = \emptyset$ the history $x \oplus y \in \vec{Z}$ where $Z = X \cup Y$ is defined by:

$$(x \oplus y)(c) = x(c) \Leftarrow c \in X$$
$$(x \oplus y)(c) = y(c) \Leftarrow c \in Y$$

For many applications this needs to be refined to support a more refined hierarchy, whereby channels can be grouped into channel groups, such that a particular channel group can contain a subset of X and Y. This allows modeling of engineering concepts such as bidirectional virtual (or physical) connections, which is how channels are typically realized in practice (see sect. 6.1). The reason why this might be useful is that such "grouped channels" share certain properties, such as quality of service (QoS) or failure modes, and also represent administrative units (e.g., enabling, disabling, etc.).

We specify the behavior of a system or of a service with the syntactic interface $(X \blacktriangleright Y)$ by a Boolean expression $Q(x, y)$ where x is an input history (or in simpler cases an input stream) and y is an output history (or in simpler cases an output stream) and Q is a predicate:

$$Q : \vec{X} \times \vec{Y} \to \mathbb{B}$$

on the channel histories over X and Y.

In system specifications, for better readability, we write the Boolean expression $Q(x, y)$ with histories $x \in \vec{X}$ and $y \in \vec{Y}$ by logical assertions in higher-order predicate logic with free variables from X and Y where each of these variables stands for a timed stream of the respective type. The logical formula is called the interface assertion over the syntactic interface $(X \blacktriangleright Y)$. In the following sections, we introduce interface specifications for services written in the form of templates. We introduce composition and refinement, and illustrate these concepts with examples.

3.1. Syntax

We use the following form of templates for concrete interface specifications of systems or services as demonstrated by the following brief example (see Fig. 2 below):

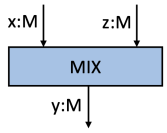

Figure 2 System or service MIX as a data flow node

MIX
in x, z: TSTR M
out y: TSTR M
$\forall d \in M : d\#x + d\#z = d\#y$

The above template gives a name to a system or to a service (in our example, MIX). It describes its syntactic interface (in our example $(x, z : \text{TSTR } M \blacktriangleright y : \text{TSTR } M)$) and an interface assertion (in our example $\forall d \in M : d\#x + d\#z = d\#y$) that specifies the system or the service. We derive the proposition:

$$MIX \vdash (x, z : \text{TSTR } M \blacktriangleright y : \text{TSTR } M) :$$
$$\forall d \in M : d\#x + d\#z = d\#y$$

from the template which states that the system (or the service) MIX fulfills ("implies" and therefore "is required to be a refinement of") the specification:

$$(x, z : \text{TSTR } M \blacktriangleright y : \text{TSTR } M) :$$
$$\forall d \in M : d\#x + d\#z = d\#y$$

Using (Broy 2023a) we may deduce further properties from the specifying assertion (see also the Appendix).

In the interface assertion:

$$\forall d \in M : d\#x + d\#z = d\#y$$

x, z, and y denote timed streams. MIX is the name of the system, x and z are input channels carrying messages of type M, and y is an output channel of type M. Figure 2 depicts MIX as a data flow node.

System specifications of this form are called *untimed*, since according to the interface assertion the output data stream y does not depend on the timing of the input streams x and z. As a result, MIX is called *non-time-critical*. In the following, we specify interface behavior by such interface assertions.

3.2. Services

A service is an interface behavior specified by a syntactic interface and an interface predicate. A service specification coincides with a system interface specification. A system typically offers ("provides") a family of services while it may need ("require") a number of services. In the simplest case, a system just offers one service without requiring any services. Thus, interface specifications support the specification of services while services are used to specify the functional behavior of systems. This explains in turn how interface specifications specify the functional behavior of systems.

Key issues are the relationship between system specifications in terms of services and modularity concepts. The specification

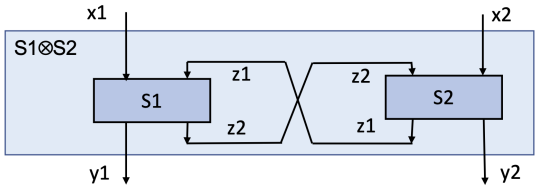

Figure 3 Composition of systems S1 and S2 (hiding feedback channels)

of systems in terms of their services by syntactic interfaces and predicates specifying interface behavior defines systems that can be used as components in architectures (see the composition operator in the next section). The specification method achieves encapsulation including information hiding as discussed in more detail in sect. 6. It guarantees modularity in the sense that the specification of a composed system can be deduced from the specifications of its components.

3.3. Concurrent Composition with Interaction via Feedback Loops

To define concurrent composition of systems we use the idea that systems are composed by putting them side by side introducing feedback loops for mutual communication via their channels that fit together.

Systems are composed as shown in Fig. 3. There the channels z1 and z2 serve as feedback channels carrying streams for the communication between the two systems S1 and S2. These streams are defined by fixpoints.

Systems S_k with syntactic interfaces $(X_k \blacktriangleright Y_k)$ with $k = 1,2$ are called *composable*, if their channel types are consistent and $Y_1 \cap Y_2 = \emptyset$. Moreover, we assume that for $k = 1,2$ the channel sets X_k and Y_k are disjoint $X_k \cap Y_k = \emptyset$. We define concurrent composition with feedback of system specified by interface predicates by logical conjunction of their interface predicates. Feedback channels become hidden by existential quantification.

Consider systems S_k with given specifications $S_k \vdash (x_k : X_k \blacktriangleright y_k : Y_k) : Q_k(x_k, y_k)$ for $k = 1,2$, that are composable. Define:

$X = (X_1 \cup X_2) \backslash Z$ *input channels for the composed system*
$Y = Y_1 \cup Y_2$ *output channels for the composed system*
$Z = Y \cap (X_1 \cup X_2)$ *feedback channels – internal channels*

The channel set Z denotes the set of feedback channels in the composition. We specify the composite system $(S1 \otimes S2)$ by:

$$
\begin{aligned}
&(S1 \otimes S2) \vdash (x : X \blacktriangleright y : Y \backslash Z) : \exists\, y' \in \overrightarrow{Y} : \\
&\quad (Q_1(x \oplus y' \mid X_1, y' \mid Y_1) \\
&\quad \wedge\, Q_2(x \oplus y' \mid X_2, y' \mid Y_2) \\
&\quad \wedge\, y = y' \mid (Y \backslash Z)
\end{aligned}
$$

hiding feedback channels as output. Here $x \mid X'$ denotes the restriction of the history $x \in \overrightarrow{X}$ to the history in $\overrightarrow{X'}$ where for the channels $c \in X' \backslash X$ we get $x'(c) = x(c)$.

Whenever for the considered subsystems S_k and the specifying predicates P_k the specification

$$S_k \vdash (x_k : X_k \blacktriangleright y_k : Y_k) : P_k(x_k, y_k)$$

is derived or stated, for the composed system $S1 \otimes S2$ the specification:

$$
\begin{aligned}
&(S1 \otimes S2) \vdash (x : X \blacktriangleright y : Y \backslash Z) : \exists\, y' \in \overrightarrow{Y} : \\
&\quad (P_1(x \mid X_1, y' \mid Y_1) \\
&\quad \wedge\, P_2(x \mid X_2, y' \mid Y_2) \\
&\quad \wedge\, y = y' \mid (Y \backslash Z)
\end{aligned}
$$

is concluded. This holds, in particular, if for the considered subsystems S_k the laws of strong causality or full realizability are applied (see the Appendix). Note that this form of defining composition works both for specifications and data flow programs and both for service and system specification.

3.4. Modularity and Refinement

We define modularity as the following property of an approach for specifying systems:

- systems are specified by their interface behavior while the details of their implementation are hidden, following the principles of information hiding,
- the composition of systems from subsystems leads to a composite system with a specification of its interface behavior that can be derived by composing the specifications of the interface behavior of its subsystems.

A key issue is the relationship between layered architectures and key modularity concepts such as components, encapsulation, and information hiding as discussed in sect. 6.

For a system or service specification $(x : X \blacktriangleright y : Y) : Q(x,y)$ a specification $(x : X \blacktriangleright y : Y) : Q'(x,y)$ is called a refinement if $Q' \Rightarrow Q$.

4. Layered Architectures

A layered system architecture consists of one or more layer "stacks". Unfortunately, many people interpret the term "stack" to refer exclusively to a single linear hierarchy of layers . But, as noted in [18], in practice, a layered architecture is much more often realized as a more complex directed acyclic graph. Therefore, it is more precise to say that a layered system architecture consists of a "non-cyclical hierarchy of layers".

4.1. Layers

As noted, layers are systems which *provide* one or more specialized services, which they often realize by relying on services (required services) provided by lower layers. Both provided and required services are described by *interface specifications*. The union of the individual syntactic interfaces of these services defines the *syntactic interface of the layer*. The interface predicate of the layer is composed of the interface predicates associated with both the required and the provided services.

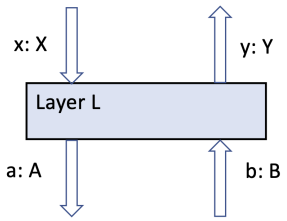

Figure 4 Figure 4. Layer L with its input and output channels

We start with a simple form of layers: Let X, Y, A, and B be pairwise disjoint sets of channels, and P be an interface predicate for the syntactic interface $(X \blacktriangleright Y)$ and be an interface predicate for the syntactic interface $(A \blacktriangleright B)$. Let the interface behavior:

$$S \vdash (x : X \blacktriangleright y : Y) : P(x, y)$$

denote the *provided service* and:

$$W \vdash (a : A \blacktriangleright b : B) : R(a, b)$$

denote the *required service*. As we show in sect. 6, both S and W may describe families of subservices represented here respectively as a single compound service.

We then formulate the following specification of layer L (see Fig. 4:

$$L \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : R(a, b) \Rightarrow P(x, y)$$

It specifies that, if the required service W is provided via the channels in a and b, then the provided service S is offered via the channels x and y. It is an instance of an assumption/commitment specification (see (Broy 2018)).

In this essential description of a layer, we do not demand a specific data flow between the provided and the required service. (Moreover, we do not even specify whether a required service is actually used by a layer for realizing its own provided service.) The specification simply states that the provided service is offered if the required service is available. In this case we speak of a basic layer. We may add to the specification, as a detail, that the provided service is offered as long as the required service is available (see sect. 5.1).

If layer L is composed with a system U for which the interface predicate G holds and that shares with L exactly the syntactic sub-interface $(A \blacktriangleright B)$, then if $G \Rightarrow R$ then $L \otimes U$ implies P. In other words, if U offers the required service W as a subservice, then the composite system $L \otimes U$ offers the provided service S.

As noted above, a further interesting aspect has to do with adding specifications of an explicit data flow between the provided and required service. We may add – as a refinement – conditions which relate the input to the provided service explicitly to the output to the required service. Such additional data dependencies allow a more explicit relationship between the streams of the provided and the required service. That way we can model some data flow relationships between the two services (see sect. 5).

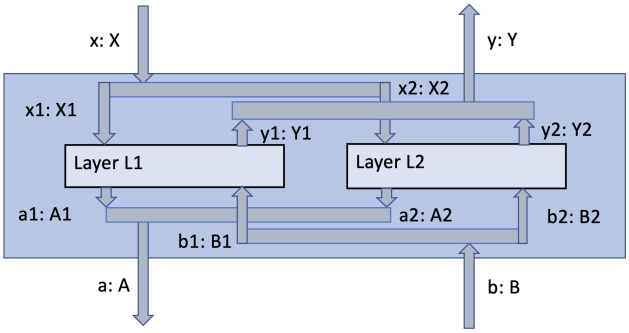

Figure 5 Parallel composition of two layers L1 and L2

As explained in the Appendix, a fully realizable service is assumed to be *strongly causal*. This means that the messages in the output streams of a service till some time $t+1$ depend at most on the input streams till time t. There is no instantaneous reaction by a service, output is delayed at least one time step. This assumption has strong consequences since it allows the deduction of additional properties in terms of refined specifications given an initial specification.

Moreover, since we assume that both the required service and the provided service are strongly causal (see the Appendix for definitions) we deduce additional properties. Since in most practical cases, strongly causal specifications are fully realizable, we consider only the case where both the required service and the provided service are strongly causal. Then the key correctness assumption of a layer is as follows: *As long as the required service is guaranteed, the provided service is guaranteed*. If the required service is delivered correctly all the time, the provided service is delivered correctly all the time. This is worked out in detail in sect. 4.7.

4.2. Composing Layers

In this section we introduce two ways of composing layers to produce more complex layers: (1) *parallel composition* that puts two layers side by side, and (2) *layer stacking*, that is, putting one layer "on top of" another layer.

For parallel composition (Fig. 5), let as assume that we have two layers:

$$\begin{aligned} L_1 \vdash & (x_1 : X_1, b_1 : B_1 \blacktriangleright y_1 : Y_1, a_1 : A_1) : \\ & R_1(a_1, b_1) \Rightarrow P_1(x_1, y_1) \\ L_2 \vdash & (x_2 : X_2, b_2 : B_2 \blacktriangleright y_2 : Y_2, a_2 : A_2) : \\ & R_2(a_2, b_2) \Rightarrow P_2(x_2, y_2) \end{aligned}$$

where all channel sets are pairwise disjoint. We can combine these into a single unifying layer, LP, by parallel composition so that: $X = X_1 \cup X_2, Y = Y_1 \cup Y_2, A = A_1 \cup A_2, B = B_1 \cup B_2$:

$$\begin{aligned} LP \vdash & (x : X, b : B \blacktriangleright y : Y, a : A) : \\ & (R_1(a \mid A_1, b \mid B_1) \wedge R_2(a \mid A_2, b \mid B_2)) \Rightarrow \\ & (P_1(x \mid X_1, y \mid Y_1) \wedge P_2(x \mid X_2, y \mid Y_2)) \end{aligned}$$

Here $a \mid A_1$ denotes the restriction of the history a (which assigns streams to the channels in signature A) to the channels in

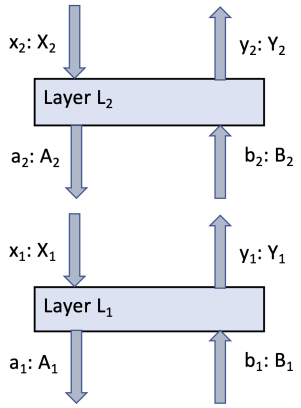

Figure 6 Two layers, where the provided service of layer L_1 is a refinement of the required service of layer L_2

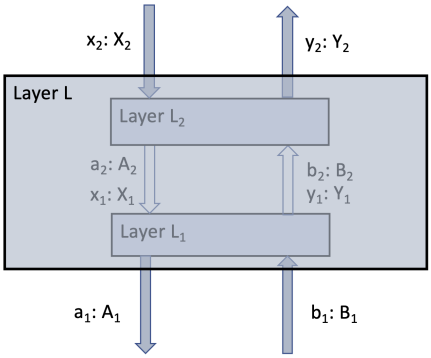

Figure 7 Stacked layer composition $L = L_1 \otimes L_2$ of the two composable layers L_1 and L_2

signature $A_1 \subseteq A$. The history $a \mid A_1 \in \vec{A}_1$ then assigns streams to the channels in signature A_1 such that for each channel $c \in A_1$ we have $(a \mid A_1)(c) = a(c)$.

Now, for layer stacking, let us assume that we have two layers $(k = 1, 2)$:

$$L_k \vdash (x_k : X_k, b_k : B_k \blacktriangleright y_k : Y_k, a_k : A_k) : \\ R_k(a_k, b_k) \Rightarrow P_k(x_k, y_k)$$

The two layers are shown in Fig. 6 where we assume that the sets of channels $X_1 \cup Y_1$ and $X_2 \cup Y_2$ are disjoint.

Next, we compose the two layers (Fig. 7), one on top of the other, to form composite layer L. Recall that the proposition "$R \Rightarrow P$" expresses that the described system S offers an interface behavior that is a refinement of the specified service P.

These two layers fit syntactically together, if $X_1 = A_2$ and $Y_1 = B_2$, and semantically if the provided service:

$$S_1 \vdash (x_1 : X_1 \blacktriangleright y_1 : Y_1) : P_1(x_1, y_1)$$

of the lower layer L_1 is a refinement of the required service:

$$W_2 \vdash (a_2 : A_2 \blacktriangleright b_2 : B_2) : R_2(a_2, b_2)$$

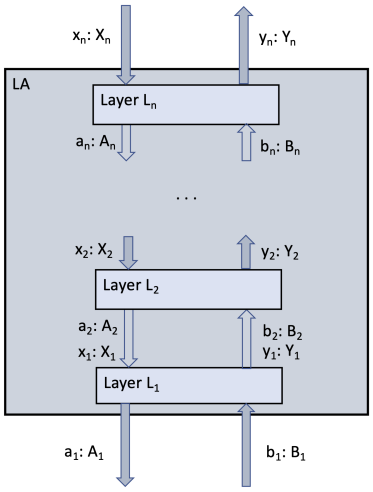

Figure 8 Figure 8. Layered architecture LA

of the upper layer L_2 which is expressed by (note that $X_1 = B_2$ and $Y_1 = A_2$):

$$P_1(x_1, y_1) \Rightarrow R_2(x_1, y_1)$$

We compose the two layers to create composite system L:

$$L = L1 \otimes L2 \vdash \\ (x_2 : X_2, b_1 : B_1 \blacktriangleright y_2 : Y_2, a_1 : A_1) : \\ \exists x_1 \in \vec{X}_1, y_1 \in \vec{Y}_1 : \\ (R_1(a_1, b_1) \Rightarrow P_1(x_1, y_1)) \wedge \\ (R_2(x_1, y_1) \Rightarrow P_2(x_2, y_2))$$

Since $P_1(x_1, y_1) \Rightarrow R_2(x_1, y_1)$ holds, we conclude:

$$L \vdash (x_2 : X_2, b_1 : B_1 \blacktriangleright y_2 : Y_2, a_1 : A_1)) : \\ R_1(a_1, b_1) \Rightarrow P_2(x_2, y_2)$$

This shows that system L which is the result of composing the two layers is yet another layer with a provided service S_2 that is the provided service of Layer L_2, while its required service W_1 is the required service of layer L_1.

4.3. Layered Architectures: Stacks of Layers

The composition of layers, as introduced, can be iterated for n given layers, $n \in \mathbb{N}$. Given a family of n layers L_k for $k \in \mathbb{N}$, $1 \leq k \leq n$, with specifications:

$$L_k \vdash (x_k : X_k, b_k : B_k \blacktriangleright y_k : Y_k, a_k : A_k) : \\ R_k(a_k, b_k) \Rightarrow P_k(x_k, y_k)$$

where $X_k = A_{k+1}, Y_k = B_{k+1}$, and where all other channel sets are disjoint, and where:

$$P_k(x_k, y_k) \Rightarrow R_{k+1}(x_k, y_k)$$

for all k, $1 \leq k < n$, we get a "stacked" layered architecture:

$$LA = L_n \otimes L_{n-1} \otimes \cdots \otimes L_1$$

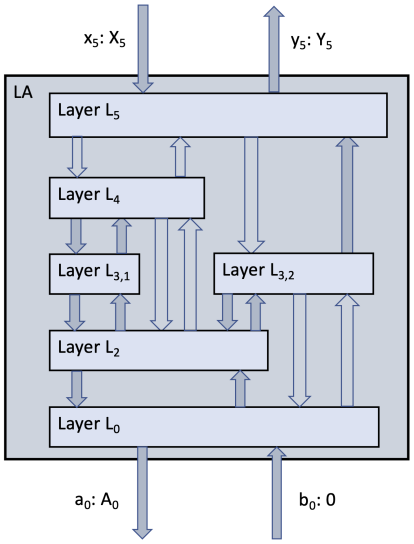

Figure 9 A DAG-structured layered architecture with partially overlapping layers

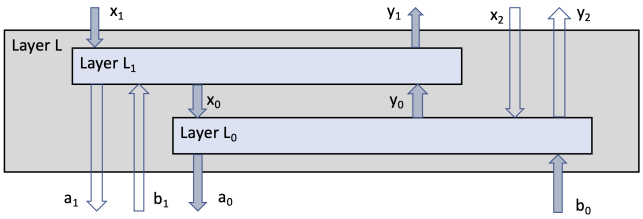

Figure 10 Composing two layers which only partially overlap

The layered architecture *LA* is depicted in Fig. 8.

Applying the rule of composing layers iteratively to form *LA* we conclude:

$$LA \vdash (x_n : X_n, b_1 : B_1 \blacktriangleright y_n : Y_n, a_1 : A_1) : \\ R_1(a_1, b_1) \Rightarrow P_n(x_n, y_n)$$

Of course, we can get a large variety of different layered architectures, through different combinations of these two layer composition patterns (parallel or stacked).

4.4. Layers with only Partial Overlap

As noted earlier, it is not uncommon to have a layered architecture that is not a strict layer stack, but one whose topological configuration is represented by a directed acyclic graph (DAG), as illustrated by the example in Fig. 9. Here, both layer L_2 and $L_{3,2}$ require some services of layer L_0, but layer $L_{3,2}$ also depends on the serves of layer L_2.

Accordingly we examine the case of just two partially overlapping layers, shown in Fig. 10. Let X_k, Y_k for $k = 0, 1, 2$, and A_k, B_k for $k = 0, 1$ be channel sets and x_k, y_k, for $k = 0, 1, 2$ as well as a_k, b_k, for $k = 0, 1$ be histories. We assume that the channel sets are pairwise disjoint.

For layer stacking, let us assume that we have two layers (see Fig. 10) which only partially fit:

$$L_0 \vdash (x_0 : X_0, x_2 : Y_2, b_0 : B_0 \blacktriangleright y_0 : Y_0, a_0 : A_0, y_2 : Y_2) : \\ R_0(a_0, b_0) \Rightarrow P_0(x_0, x_2, y_0, y_2)$$
$$L_1 \vdash (x_1 : X_1, b_1 : B_1, y_0 : B_0 \blacktriangleright y_1 : Y_1, a_1 : A_1, x_0 : X_0) : \\ R_1(a_1, x_0, b_1, y_0) \Rightarrow P_1(x_1, y_1)$$

The two layers fit together syntactically (see Fig. 10) and semantically, with respect to the required service for layer L_1, if the following assertion holds:

$$P_0(x_0, x_2, y_0, y_2) \Rightarrow \forall\, a_1 \in \vec{A_1} : \exists\, b_1 \in \vec{B_1} : R_1(a_1, x_0, b_1, y_0)$$

Then, we construct the specification layer $L = (L_1 \otimes L_0)$ by composition of the specifications of layer L_1 and layer L_0:

$$L \vdash (x_1 : X_1, x_2 : X_2, b_0 : B_0, b_1 : B_1 \blacktriangleright \\ y_1 : Y_1, y_2 : Y_2, a_0 : A_0, a_1 : A_1) : \exists\, y_0 \in \vec{Y_0}, x_0 \in \vec{X_0} : \\ (R_0(a_0, b_0) \Rightarrow P_0(x_0, x_2, y_0, y_2)) \\ \wedge\, (R_1(a_1, x_0, b_1, y_0) \Rightarrow P_1(x_1, y_1))$$

and get by simple propositional logic a layer specification:

$$L \vdash (x_1 : X_1, x_2 : X_2, b_0 : B_0, b_1 : B_1 \blacktriangleright \\ y_1 : Y_1, y_2 : Y_2, a_0 : A_0, a_1 : A_1) : \exists\, y_0 \in \vec{Y_0}, x_0 \in \vec{X_0} : \\ (R_0(a_0, b_0) \wedge R_1(a_1, x_0, b_1, y : 0)) \Rightarrow \\ (P_0(x_0, x_2, y_0, y_2) \wedge P_1(x_1, y_1))$$

Thus:

$$(a_0 : A_0, a_1 : A_1 \blacktriangleright b_0 : B_0, b_1 : B_1) : \\ \forall\, x_0 \in \vec{X_0} : \exists\, y_0 \in \vec{Y_0} : R_0(a_0, b_0) \wedge R_1(a_1, x_0, b_1, y_0)$$

specifies the required service while:

$$(x_1 : X_1, x_2 : X_2 \blacktriangleright y_1 : Y_1, y_2 : Y_2) : \forall\, x_0 : \vec{X_0} : \exists\, y_0 : \vec{Y_0} : \\ (\exists\, a_0 \in \vec{A_0}, b_0 \in \vec{B_0}, a_1 \in \vec{A_1}, b_1 \in \vec{B_1} : \\ R_0(a_0, b_0) \wedge R1(a1, x0, b1, y0)) \Rightarrow \\ (P_1(x_1, y_1) \wedge P_0(x_0, x_2, y_0, y_2)))$$

specifies the provided service.

4.5. Platforms

If for a layer the required service $W \vdash (a : A \blacktriangleright b : B) : R(a, b)$ is empty, for precisely if no service is required and thus $A = \emptyset, B = \emptyset$, and $R = true$, then the layer L degenerates to:

$$L \vdash (x : X \blacktriangleright y : Y) : P(x, y)$$

which is the provided service which is offered with any need of a required service.

We get a layered architecture LAP as a stack of layers on top of a platform L0 which is the bottom layer that does not require a "required" service.

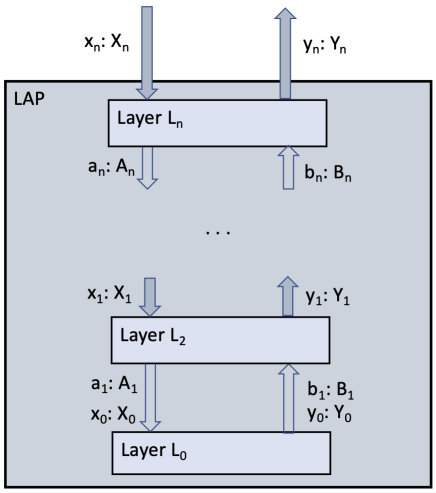

Figure 11 Stack of layers LAP on top of a bottom layer without required service.

4.6. Refining Layers

Of course, it may also be useful to apply the concept of refinement to layers. Recall, a specification $(X \blacktriangleright Y) : P'$ is a refinement of a specification $(X \blacktriangleright Y) : P$ if $P' \Rightarrow P$. For a layer L:

$$L \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : R(a,b) \Rightarrow P(x,y)$$

we obtain a refinement:

$$(x : X, b : B \blacktriangleright y : Y, a : A) : Q(a,b,x,y)$$

provided that $Q(a,b,x,y) \Rightarrow (R(a,b) \Rightarrow P(x,y))$. If this holds, we can replace the layer L by its refinement L'

$$L' \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : Q(a,b,x,y)$$

Actually, there are special forms of refinements of layers; if we have interface predicates R' and P' such that:

$$R(a,b) \Rightarrow R'(a,b)$$

and:

$$P'(x,y) \Rightarrow P(x,y),$$

Then layer L', which is specified by:

$$L' \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : R'(a,b) \Rightarrow P'(x,y)$$

is a refinement of L since $(R' \Rightarrow P') \Rightarrow (R \Rightarrow P)$. This is shown by a simple proof in propositional logic.

4.7. Behavior of Layers in Case of Partially Correct Required Services

In the following, we consider the case where the required service is delivered correctly at least to a certain time t. Then the provided service is delivered at least until time $t+1$. This is expressed as follows:

$$(x : X, b : B \blacktriangleright y : Y, a : A) : \forall\, t \in \mathbb{N} : (R \downarrow t)(a,b) \Rightarrow (P \downarrow t+1)(x,y)$$

where:

$$(R \downarrow t)(a,b) = \exists\, a' \in \vec{A}, b' \in \vec{B} : R((a \downarrow t)\hat{}a', (b \downarrow t+1)\hat{}b')$$
$$(P \downarrow t)(x,y) = \exists\, x' \in \vec{X}, y' \in \vec{Y} : P((x \downarrow t)\hat{}x', (y \downarrow t+1)\hat{}y')$$

Finally we get, as a consequence the layer specification:

$$(x : X, b : B \blacktriangleright y : Y, a : A) : (\forall\, t \in \mathbb{N} : (R \downarrow t)(a,b) \Rightarrow (P \downarrow t+1)(x,y)) \wedge (R(a,b) \Rightarrow P(x,y))$$

which is a stronger specification of a layer. This states that the provided service will be correct up to time t, provided the required service is correct up to time t. This property is a consequence of strong causality as explained in detail in (Broy 2023b). Note $\exists\, t \in \mathbb{N} : \neg(R \downarrow t)(a,b) \Rightarrow \neg R(a,b)$.

Now let us consider two layers L_1 and L_2 specified as in the preceding section. From $P_1 \Rightarrow R_2$ we can conclude that:

$$(P_1 \downarrow t)(x_1,y_1) \Rightarrow (R_2 \downarrow t)(x_1,y_1)$$

and thus, for a stacked composition of layers:

$$\begin{aligned} L = L_1 \otimes L_2 \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : \\ \exists\, x_1 \in \vec{X_1}, y_1 \in \vec{Y_1} : (R_1(a_1,b_1) \Rightarrow P_1(x_1,y_1)) \\ \wedge (R_2(x_1,y_1) \Rightarrow P_2(x_2,y_2)) \\ \wedge (\forall\, t \in \mathbb{N} : (R_1 \downarrow t)(a_1,b_1) \Rightarrow (P_1 \downarrow t+1)(x_1,y_1)) \\ \wedge (\forall\, t \in \mathbb{N} : (R_2 \downarrow t)(x_1,y_1) \Rightarrow (P_2 \downarrow t+1)(x_2,y_2)) \end{aligned}$$

We obtain, since $(P_1 \downarrow t)(x_1,y_1) \Rightarrow (R_2 \downarrow t)(x_1,y_1)$ follows from $P_1(x_1,y_1) \Rightarrow R_2(x_1,y_1)$:

$$\begin{aligned} L \vdash & (x : X, b : B \blacktriangleright y : Y, a : A) : (R_1(a_1,b_1) \Rightarrow P_2(x_2,y_2)) \\ & \wedge (\forall\, t \in \mathbb{N} : (R_1 \downarrow t)(a_1,b_1) \Rightarrow (P_2 \downarrow t+2)(x_2,y_2)) \end{aligned}$$

which implies:

$$\begin{aligned} L \vdash & (x : X, b : B \blacktriangleright y : Y, a : A) : (R_1(a_1,b_1) \Rightarrow P_2(x_2,y_2)) \\ & \wedge (\forall\, t \in \mathbb{N} : (R_1 \downarrow t)(a_1,b_1) \Rightarrow (P_2 \downarrow t+1)(x_2,y_2)) \end{aligned}$$

This formula expresses that *as long as the required services are guaranteed, the required behavior of the provided service will be delivered*. Moreover, it expresses that the provided service is correct up to time $t+1$ provided the required service is correct up to time t.

Two systems described in terms of assumed and provided services can be composed if the provided services are refinements of the assumed services. Note that service S1 is a refinement

of service S_2 if the interface assertion specifying S_1 implies the interface assertion specifying S_2.

(Note: The specification of layers working with partially correct required services is more complicated than the specification of the layers given in sect. 4.1. Fortunately, the specification of layers working for partially correct required services can be derived by the principle of strong causality as shown in (Broy 2023b) schematically (see the Appendix) from the specification of the layers given in sect.. 4.1.)

5. Layers with Specific Data Flow

So far we considered layers specified in terms of a provided and a required service where the provided service was guaranteed as long as the required service was available and there was no specified data flow between the required and the provided service. In the following, we deal with layers, where there is some additional specified data flow between the required and the provided services. Such layers are called *expanded* layers.

5.1. Data Flow Between the Required and the Provided Service

So far, we have not explicitly discussed any specific data flows between a required and a provided service. As noted, the specification of a layer expresses that the provided service is guaranteed if the required service is available. This, however, does not even specify whether the required service is used at all nor how it is used or in which way the input for the provided service influences the output to the required service or in which way the input coming from the required service influences the output to the provided service.

Consider the following specification of layer L:

$$L \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : R(a,b) \Rightarrow P(x,y)$$

Actually, for a number of applications, an explicit data flow between provided service and the input and the required service may be of interest or even mandatory. This data flow can be expressed by an assertion $Q(x,b,y,a)$. In that case, the specification of the layer LDF with specific data flow reads as follows:

$$LDF \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : (R(a,b) \Rightarrow Q(x,b,y,a))$$

where we assume:

$$Q(x,b,y,a) \Rightarrow P(x,y)$$

This case with explicit data flow constraints is obviously a refinement of the specification of layer L, which does not specify any data flow and where interface predicate Q is only guaranteed to hold if the required service is available. However, a layer LDF' may be specified as a refinement of layer L, in situations where the additional data flow holds independently of the availability of the correct required service:

$$LDF' \vdash (x : X, b : B \blacktriangleright y : Y, a : A) : \\ (R(a,b) \Rightarrow P(x,y)) \wedge Q'(x,b,y,a)$$

Note that the specification for LDF' is stronger that the specification for LDF, but both are refinements while both are refinements of the specification of layer L. Nevertheless, both LDF and LDF' are refinements of L.

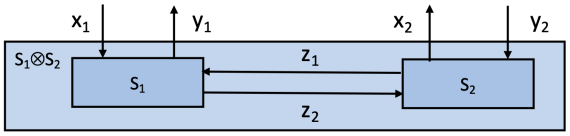

Figure 12 Composition of systems S_1 and S_2 (hiding feedback channels z_1 and z_2)

Examples of specifications for some data flow between the provided and the required service are given in sect. 5.3 and in the extended example in sect. 7.

5.2. The Layering Pattern versus General System Composition

Consider systems composed as shown in Fig. 12. Again, for simplicity of notation, we consider only channels that carry streams instead of signatures with histories carrying families of streams. Nevertheless, this discussion can be generalized to histories (families of streams). Consider systems S_k with given specifications

$$S_k \vdash (x_k, z_k \blacktriangleright y_k, z_{3-k}) : Q_k(x_k, z_k, y_k, z_{3-k})$$

for $k = 1, 2$, that are composable.

The channels z_1 and z_2 denote the feedback channels in the composition. We define $(S_1 \otimes S_2)$ hiding feedback channels as output by:

$$(S_1 \otimes S_2) \vdash (x_1, x_2 \blacktriangleright y1, y2) : \\ \exists\, z_1, z_2 : Q_1(x_1, z_1, y_1, z_2) \wedge Q_2(x_2, z_2, y_2, z_1)$$

We consider two instances of specifications of the systems S_1 and S_2 to show the difference between "vertical" composition (which means layering – see sect. 2) and "horizontal" composition (peer composition). For layer stacking, let us assume that we have two layers L_1 and L_2 for the systems S_1 and S_2:

$$L_1 \vdash (x_1, z_1 \blacktriangleright y_1, z_2) : R_1(x_1, y_1) \Rightarrow P_1(z_1, z_2)$$
$$L_2 \vdash (x_2, z_2 \blacktriangleright y_2, z_1) : R_2(z_1, z_2) \Rightarrow P_2(x_2, y_2)$$

In other words, we consider systems L_1 and L_2 that follow the layer specification pattern.

If $P_1 \Rightarrow R_2$ holds, we get:

$$(L_1 \otimes L_2) \vdash (x_1, x_2 \blacktriangleright y_1, y_2) : R_1(x_1, y_1) \Rightarrow P_2(x_2, y_2)$$

This shows the key properties of the layer pattern: restricted dependencies between input and output under the assumption of the presence of the required services.

In the general case, in system S_1 the streams y_1, z_2 depend on the streams x_1, z_1 and in system S_2 the streams y_2, z_1 depend on the streams x_2, z_2. Therefore, in general, the streams y_1, y_2 depend on the streams x_1, x_2 in $(S_1 \otimes S_2)$. This shows that the layering pattern is a special case of general ("horizontal") composition.

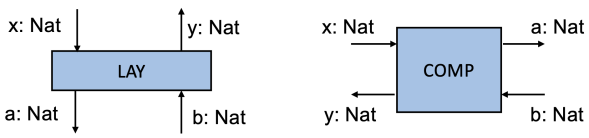

Figure 13 A layer *LAY* and a general component *COMP*

The interesting question here is, whether there is a difference between a component as used for composition in the horizontal way and a layer in a layered architecture. We illustrate this difference by two rather simple example components depicted in Fig. 13.

We specify layer *LAY* and a component *COMP* which have the same syntactic interface, specified as follows (let Nat be the type of natural numbers). First, we specify the layer element:

LAY
in x, b: TSTR Nat
out y, a: TSTR Nat

$R(a,b) \Rightarrow P(x,y)$

We specify its provided service as:

$$P(x,y) = (\forall\, d \in Nat : (d\#x > 0 \Rightarrow d\#y = \infty) \\ \wedge\, (d\#x = 0 \Rightarrow d\#y = 0))$$

and its required service as:

$$R(a,b) = \forall\, d \in Nat : d\#b = d\#a$$

These specifications of the provided service and the required service leads to the interface assertion:

$$(\forall\, d \in Nat : d\#b = d\#a) \Rightarrow \\ (\forall\, d \in Nat : (d\#x > 0 \Rightarrow d\#y = \infty) \\ \wedge\, (d\#x = 0 \Rightarrow d\#y = 0))$$

which specifies layer *LAY*. Note that imposing causality for the layer leads to additional assertions about the timing.

Next, we consider the system *COMP* with an identical syntactic interface as LAY, but with different interface assertions.

COMP
in x, b: TSTR Nat
out y, a: TSTR Nat

$\forall\ d \in Nat :$
$\quad d\#a = min(d\#x, d\#b)$
$\wedge\ d\#y = d\#x + d\#b$

Obviously, *COMP LAY* is not realizable. Consider

$$\forall\, d \in Nat : d\#a = min(d\#x, d\#b) \wedge d\#y = d\#x + d\#b \\ \wedge (\forall\, d \in Nat : d\#b = d\#a) \Rightarrow \\ (\forall\, d \in Nat : (d\#x > 0 \Rightarrow d\#y = \infty) \\ \wedge\, (d\#x = 0 \Rightarrow d\#y = 0))$$

This is shown by a simple example. Choose $d\#x = 1$ and $\forall\, d \in \mathbb{N} : d\#b = 0$; then *COMP* implies $d\#a = 0$ and $d\#y = 1$ while LAY $d\#a = 0$ implies $d\#y = \infty$. *COMP LAY* is not realizable.

For the specification of component *LAY*, we may specify, in addition, some data flow for the layer LAY between input x and b and output a:

$$\forall\, d \in Nat : d\#a = d\#x + d\#b \wedge (d\#x = 0 \Rightarrow d\#a = 0)$$

as well as between input b and output y:

$$\forall\, d \in Nat : d\#y = d\#b$$

This specifies how *LAYI* makes use of the required service to produce the provided service. We get the refined specification:

LAYI
in x, b: TSTR Nat
out y, a: TSTR Nat

$(\forall\, d \in Nat : d\#b = d\#a) \Rightarrow$
$\quad (\forall\, d \in Nat : d\#y = d\#b \wedge d\#a =$
$\quad d\#x + d\#b \wedge (d\#x = 0 \Rightarrow d\#a = 0))$

Note that the specifying assertion of *LAYI* implies the specifying assertion of *LAY* since:

$$(\forall\, d \in Nat : d\#y = d\#b \\ \wedge\, \#a = d\#x + d\#b \wedge (d\#x = 0 \Rightarrow d\#a = 0)) \Rightarrow \\ \forall\, d \in Nat : d\#a = d\#x + d\#b \wedge (d\#x = 0 \Rightarrow d\#a = 0)$$

This shows that the specification *LAYI* is a refinement of *LAY*. Specification *LAYI* describes again a layer between the provided service and the required service with a particular way (based on the data flow specification) to make use of the required service. This demonstrates how we may refine a layer specification by adding a specification of the data flow between the provided and the required service.

A specification of a system which fulfills the layer specification is exactly an example of a specification that, in addition to the layer specification, may include further properties about the data flow between the provided and the required service.

Formulating this more generally, there are many specifications which are refinements of a layer specification L. They are created by specifying additional properties, in particular, of the data flow between the provided and the required service. However, there are other specifications for the same syntactic interface that are not refinements of the layer specification. If they are logically weaker than the layer specification, then the layer specifications is a refinement of such weaker specifications. In that case we can, of course, refine those specifications into a layer specification.

In the end, we distinguish the following three situations for the relation between a specification for a layer L and a specification of system G for the same syntactic interface. We specify a system by the weakest specification which is a refinement both of the specification of layer L and the specification of system G. If Q is the specifying predicate of layer L and E is the

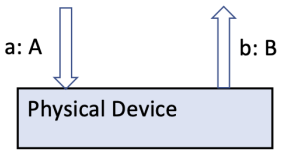

Figure 14 A compound physical device with its provided service

specifying predicate of system G, then the conjunction $Q \wedge E$ is the specifying predicate of the weakest specification which is a refinement both of the layer L and the system G:

- First, if E is logically weaker than G, i.e. if $G \Rightarrow E$, then $Q \wedge E = Q$. Layer L is a refinement of G.
- Second, if the system specified by $Q \wedge E$ is realizable, then there exists an implementation which is a refinement both of L and G (see sect. 5).
- Third, if the system specified by $Q \wedge E$ is not realizable, then there does not exist an implementation which is a refinement both of L and G (see the example $L = LAY$ and $G = COMP$ above).

Given systems with specifications that (depending on how they are specified) fulfill a layer specification (are refinements of a layer specification), they can also be used as layers but also in a different way.

For a layer specification L, of course, it is not unreasonable to use L in a horizontal manner. However, if we combine L with another system J, which does not deliver the required service, then we do not know anything according to the layer specification about the behavior of the result with respect to the provided service. Nevertheless, in cases where the system J fulfills the specification of the required service, then according to the specification of L we get the provided service.

5.3. Cyber-Physical Systems: Controlling Physical Devices

In a cyber-physical system, a physical device (PD) is typically controlled by a service. The physical device may have sensors that provide information about the state of the PD and it may have actuators that offer the possibility to control the device. Both sensors and actuators can be modeled by services.

We may choose to combine all the sensors and actuator services into one compound service. Then a PD can be graphically illustrated as shown in Fig. 14.

A *control layer* is often placed on top of a physical device and offering a provided service for controlling the device that is more structured, more abstract, and simpler way to use the device rather than manipulating the device itself. This control layer uses the provided service of the physical device for its required services (see Fig. 15). In this case, the control layer specification should include an explicit data flow both from the provided to the required service and back from the required to the provided service.

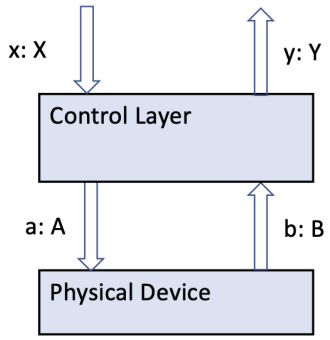

Figure 15 Control layer combined with a compound physical device

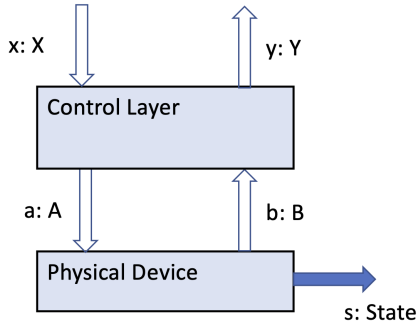

Figure 16 Control layer combined with a compound physical device

Consider, for example, the case of a brake control system, where the Control Layer is responsible for realizing an anti-lock braking function. This function is activated by signals received on input x, which are translated by the Control Layer into appropriate "operate-brake" output signals to the actual physical brake, which then achieve the corresponding antilock braking mode.

For more complex devices it may be necessary to work with a more explicit model. In that case, the physical device can be modelled by a state machine with inputs and outputs, where the states represent the state of the physical device. The input is given to the actuators of the physical device as part of a cyber-physical system, while the output on the channels in B models the original sensor outputs (see Fig. 16). Here the stream of states generated by the state machine this way provides observations about the behavior of the state machine beyond what can be observed through the sensors that generate the history b.

The streams s represents a stream of states which reflects the reaction of the physical device to the input by the stream a to the actuators of the physical device. Of course, layers that control physical devices like the Control Layer in Fig. 16 are classic expanded layers since typically, there is a data flow between its provided and its required service.

6. Structuring Services into Subservices and the Feature Interactions Problem

In the preceding text, we have given a syntactic requirement for layers and a semantic one which is based on the following idea: Each layer provides a set of services, which can be invoked by applications in its immediate upper layers based on their specific needs. In general, a supporting layer makes no assumptions or restrictions on when and to what purpose these applications use its services. That is the responsibility of its applications.

Note, however, that this does not preclude the possibility of specific protocols that may constrain the order and timing in which individual services are invoked (see summary). More technically, the specification of the required services may include some assumptions that have to be maintained to get valid access to a required layer service. This can be expressed in the interface assertion for a service and such protocols have to be observed by an application using the service. There might be rather complex and complicated procedures to get access to a service, but – as shown in (Broy 2018) – such special access protocols can easily be part of a service specification.

Although the interface specification of a supporting layer includes a specification of its own required services (and, possibly, the even data flows between provided and required services), this does not violate information hiding, because applications using the provided services of a layer can only access those.

6.1. Decomposition of Services into Subservices: Feature Interactions

In this section, we show how to decompose services of a system, such as the provided service or the required service of a layer, into a set of subservices. Such subservices of a system may be called *functional features*. To keep the presentation simple, we treat only the decomposition of a service into two subservices in detail. This is easily generalized to larger sets of subservices.

Given a service $S = (X \blacktriangleright Y) : Q$ where $X \cap Y = \emptyset$ specified by the interface assertion Q, we may decompose it into two (or more) distinct subservices $S_i = (X_i \blacktriangleright Y_i) : Q_i$ for $i = 1, 2$ provided that:

$$X = X_1 \cup X_2 \qquad \textit{where } X_1 \cap X_2 = \emptyset$$
$$Y = Y_1 \cup Y_2 \qquad \textit{where } Y_1 \cap Y_2 = \emptyset$$

such that for all:

$$Q(x, y) = Q_1(x \mid X_1, y \mid Y_1) \wedge Q_2(x \mid X_2, y \mid Y_2)$$

In that case:

$$S = S_1 \otimes S_2$$

This indicates that the input on X_1 determines the output on Y_1 which is not dependent on the input on X_2 and vice versa the input on X_2 determines the output on Y_2 which is not dependent on the input on X_1 (see Fig. 17). This means that, in this case, there are no interactions between subservices S_1 and S_2. Service S can be decomposed into independent subservices S_1 and S_2 as well as composed from subservices $S1$ and S_2 (see Fig. 18).

Figure 17 Decomposition of service S into independent subservices S_1 and S_2

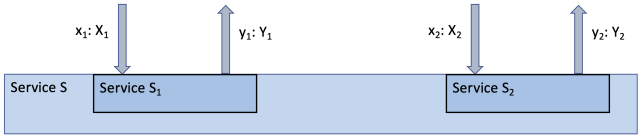

Figure 18 Composition of service S by combining subservices S_1 and S_2

However, if such a clean separation into two independent subservices is not possible, this implies that there exist *feature interactions* between subservices S_1 and S_2 (see (Broy 2010)). That is, there are dependencies between the inputs or outputs of subservice S_1 and the inputs or outputs of subservice S_2. Typically, this is due to some form of sharing by the (hidden) implementations of the subservices involved, such as sharing of resources or inputs and outputs. The effect of such interactions can be highly problematic for applications that use both subservices because, although this internal coupling is visible at the application level, it may not be noticed and cause faults.

Fortunately, it is possible to identify if such feature interactions between subservices exist without having to examine the internal implementation details of service S. Simply by examining the interface behavior of service S we can determine whether S can be decomposes into distinct subservices S_1 and S_2 in the way described above.

Dependencies according to feature interactions between services are modelled by capturing possible interactions between two distinct services, S_1' and S_2' (see Fig. 19), each with a set of

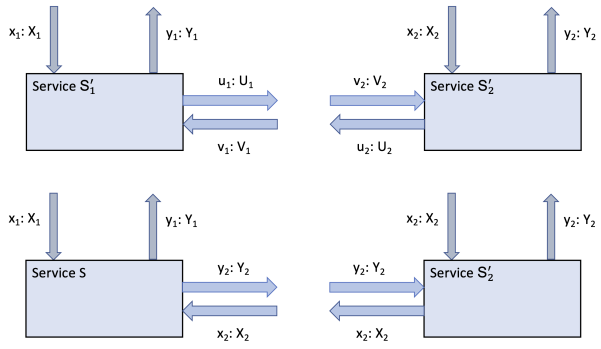

Figure 19 Two services, S_1' and S_2' and their brute force realization where service S_1' is replaced by service S and S_s' is a dummy that forwards its input x_2 to S and receives its output y_2 from S

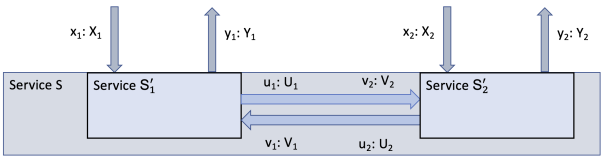

Figure 20 Composition of services S_1' and S_2'

"interaction" channels, u_i and v_i, for $i = 1, 2$, where:

$$S_i' \vdash (x_i : X, v_i : V_i \blacktriangleright y_i : Y_i, u_i : U_i) : Q_i'(x_i, v_i, y_i, u_i)$$

We then compose S_1' and S_2' into a joint combined service S such that $u_1 = v_2$ and $u_2 = v_1$ for $i = 1, 2$ (Fig. 20).

Then, if there do not exist mutually independent subservices S_1 and S_2 such that $S = S_1 \otimes S_2$, it implies that there are, in fact, feature interactions between both S_1 and S_2. That is, there are services S_1' and S_2' such that (here we write $S[v/u]$ for a specification S to rename the channels in u to the channels in v):

$$S = S_1'[u_2/v_1] \otimes S_2'[u_1/v_2]$$

Channel sets V_1 and V_2 model the feature interactions between the subservices of S with their distinct syntactic sub-interfaces $(X_1 \blacktriangleright Y_1)$ and $(X_2 \blacktriangleright Y_2)$ of the overall syntactic interface of S, $(X \blacktriangleright Y)$.

It is easy to prove that such channel sets U_i and V_i and interface predicates Q_i' can always be constructed. This is shown by a brute force argument in Fig. 19. This proves that a decomposition of the pattern shown in Fig. 20 is always possible. Of course, in practice, it is usually better to select more case-appropriate abstractions for the histories u_i and v_i.

A typical example of feature interactions between the subservices of provided services are communication platforms.

Figure 21 Composition of the peers with the message passing platform

Example: Message Passing Platform

We specify the provided service of the message passing platform (see Fig. 21). Its syntactic interface and its interface behavior is given by its provided service:

$$MPP \vdash (u : U \blacktriangleright v : V) : MPPS(u, v)$$

where:

$$U = \{u_1 : \text{TSTR}\, M, \ldots, u_n : \text{TSTR}\, M\}$$
$$V = \{v_1 : \text{TSTR}\, M, \ldots, v_m : \text{TSTR}\, M\}$$

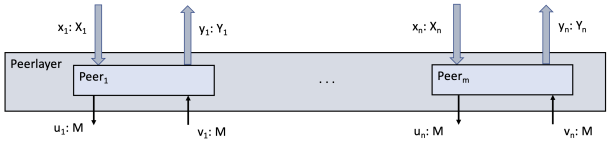

Figure 22 Peer layer

and M is the type of messages, where each message has a sender and a receiver and a content:

$$Sender : M \to \{1, \ldots, n\}$$
$$Receiver : M \to \{1, \ldots n\}$$
$$Content : M \to Data$$

We define interface assertion $MPPS(u, v)$ as follows:

$$\begin{aligned} MPPS(u, v) = \exists z \in \text{TSTR}\, M : \forall k, 1 \le k \le n : \\ \{m \in M : Sender(m) = k\} \copyright \bar{z} = \bar{u}_k \\ \wedge \quad \{m \in M : Receiver(m) = k\} \copyright \bar{z} = \bar{v}_k \end{aligned}$$

Here $\bar{z}$ denotes, for a timed stream z, the untimed stream $\bar{z}$ where all sequences of messages in z are concatenated into an untimed stream. Also, for a set $N \subseteq M$ and for a stream z, we denote by $N \copyright z$ the stream deduced from z, where all messages that are not in N are deleted.

We next specify a provided service for the Peerlayer (see Fig.22).

Given two sets of typed channel names:

$$X = x_1 : \text{TSTR}\, T_1, \ldots, x_n : \text{TSTR}\, T_n$$
$$Y = y_1 : \text{TSTR}\, S_1, \ldots, y_m : \text{TSTR}\, S_m$$

let the interface behavior:

$$PPS = (x : X \blacktriangleright y : Y) : PP(x, y)$$

denote the *provided service* of the peer layer PeerPS.

We then get the following specification of the peer layer (see Fig.22):

$$PL = (x : X, v : V \blacktriangleright y : Y, u : U) : MPPS(u, v) \Rightarrow PP(x, y)$$

It specifies that, if the required service *MPPS* is provided via the channels in u and v, then the provided service *PP* is offered via the channels x and y by the peer layer. In this description of a layer, we do not specify a data flow between the provided and the required service.

It is possible to specify, in addition, the provided service of the peer layer explicitly, which, in general, would rely on the required service as shown in Fig.23.

The provided service of the peer layer PeerPS may be decomposed in the provided services of the subservices $Peer_k$ with the syntactic interface $(x_k : \text{TSTR}\, T_k \blacktriangleright y_k : \text{TSTR}\, S_k)$ where there is a massive highly intended feature interaction between them. We may specify the behavior of $Peer_k$ as follows introducing

Figure 23 Provided service *PPS* of the peer layer PeerPS

interaction channels v_k and u_k as explained above to deal with the feature interactions as shown above above:

$$Peer_k \vdash (x_k : \text{TSTR}\ T_k, v_k : V_k \blacktriangleright y_k : \text{TSTR}\ S_k, u_k : U_k) : \\ PP_k(x_k, v_k, y_k, u_k)$$

Here PP_k denotes the interface predicate for the system $Peer_k$. Then we can construct the specification provided service as follows:

$$PPS = (x : X \blacktriangleright y : Y) : \exists z \in \text{TSTR}\ M : \\ \forall k, 1 \leq k \leq n : PP_k(x_k, v_k, y_k, u_k) \\ \wedge \{m \in M : Sender(m) = k\}\copyright \bar{z} = \bar{u}_k \\ \wedge \{m \in M : Receiver(m) = k\}\copyright \bar{z} = \bar{v}_k$$

It is clear that only with the availability of the required service of the peer layer PeerPS, which is the service *MPP* provided by the of the message passing platform, can the peer layer offer its provided service *PPS*. This is a typical example of an expanded layer.

6.2. Hiding Feature Interactions by Means of Underspecification

In certain cases, a simple ("brute force" without abstraction) method of dealing with feature interactions is to hide their effects by replacing the precise description of the impact on the outputs v_i so that the result is an underspecified nondeterministic behavior specification.

Consider subservices S_i' with feature interactions occurring through channels U_i and V_i, as illustrated in Fig. 20 for $i = 1, 2$:

$$S_i' \vdash (x_i : X, v_i : V_i \blacktriangleright y_i : Y_i, u_i : U : i) : Q_i'(x_i, v_i, y_i, u_i)$$

If we replace the input channels in U_i with corresponding underspecifications by abstracting out the influence of the V_i channels, we get:

$$S_i'' \vdash (x_i : X \blacktriangleright y_i : Y_i, u_i : U_i) : \exists\, v_i \in \vec{V_i} : Q_i(x_i, v_i, y_i, u_i)$$

Note that subservice S_i' is a refinement of subservice S_i''. Going even further, we could even completely ignore the interactions occurring via channels U_i and V_i, such that:

$$S_i^- \vdash (x_i : X \blacktriangleright y_i : Y_i) : \exists\, u_i \in \vec{U_i}, v_i \in \vec{V_i} : Q_i(x_i, v_i, y_i, u_i)$$

In this case, subservice S_i' is a refinement of subservice S_i^- (allowing increasing the syntactic interface).

This method may be appropriate in cases of unintended or inconsequential feature interactions that result only in "weak" differences which are semantically immaterial between behaviors of S_i' and S_i^-. Of course, the practicality of such an approach depends on the magnitude of the impact that the hidden feature interactions have on the service. Consider, for example, the situation where two subservices share a common resource, such as a CPU. The overall effect of the contention for the CPU might be only small delays in responses, which may not have any material impact on the desired behavior.

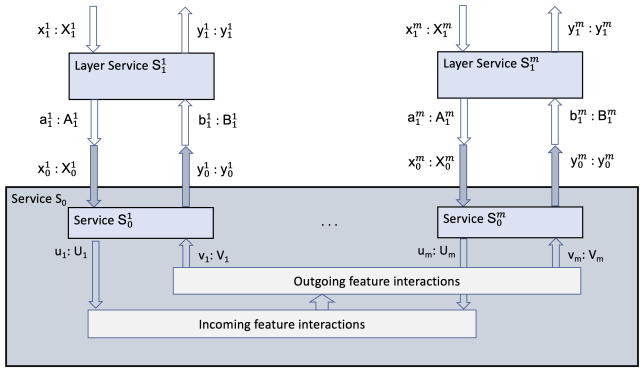

Figure 24 Layer with sublayer and subservices connected with subservices of the layer above

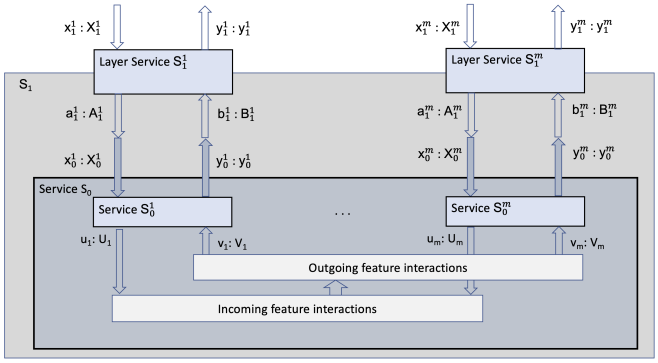

Figure 25 Provided service S_1

6.3. Propagation of Feature Interactions through Multiple Layers

When decomposing a large given service:

$$(X \blacktriangleright Y) : Q$$

into a set $(X_j \blacktriangleright Y_j)_{j \in J}$ of subservices S_j where $\bigcup_{j \in J} X_j = X$ and $\bigcup_{j \in J} Y_j = Y$, the cause of feature interactions may be some kind of sharing of resources and exchanging messages between the subservices inside the layer.

However, the feature interactions may occur in *layers* that are further down the stack, as illustrated in Fig.24. In this case, there are feature interactions between provided sub-services $S_0^1, \ldots, S_0^m$ in lower levels of the stack hierarchy. However, these services are connected as required services for higher layers $S_1^1, \ldots, S_1^m$. As a result, the provided services of the sub-layers $S_1^1, \ldots, S_1^m$ may "inherit" the feature interactions of

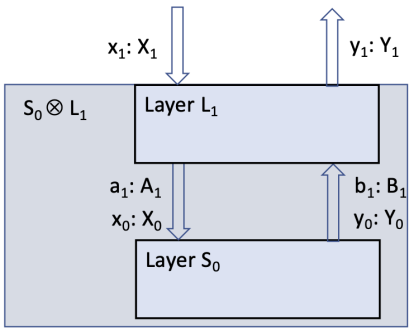

Figure 26 Composition of layer L_1 with bottom layer L_0

services $S_0^1, \ldots, S_0^m$. This explains how feature interactions may propagate through sub-layers of a layered architecture.

If we compose layer S_0^j with layer S_1^j in the upper layer, we get:

$$S_0^j \otimes S_1^j = (x_1^j : X_1^j \blacktriangleright y_1^j : Y_1^j) : \\ \exists\, a_0^j \in \vec{A_0^j}, b_0^j \in \vec{B_0^j}, u_j \in \vec{U_j}, v_j \in \vec{V_j} : \\ Q_j(a_0^j, v_j, b_0^j, u_j) \wedge P_j(x_1^j, b_0^j, y_1^j, a_0^j)$$

where $Q_j(a_0^j, v_j, b_0^j, u_j)$ denotes the interface assertion that specifies the provided service of sub-layer S_0^j and $P_j(x_1^j, b_0^j, y_1^j, a_0^j)$ denotes the interface assertion that specifies the provided service of sub-layer S_1^j. Service $S_1 = S_0 \otimes L_1$ is a refinement of service $\otimes_{1 \le j \le m}(S_0^j \otimes S_1^j)$ (see Fig. 24, Fig. 25, and Fig. 26).

There are cases where feature interactions occur in a lower layer that are tolerable since they do not critically affect the behavior of a subservice. As long as we are not interested in the individual behavior of the feature interactions and may accept them as underspecified in the way described in sect. 6.2 we may accept using $S_0^j \otimes S_1^j$ instead of $S_j \otimes R_j$.

The propagation of feature interactions for a DAG-like layer architecture can be modelled the same way. Note that in the decomposition of a large service into subservices we can always identify feature interactions between the subservices just looking at the service interface specifications according to sect. 6.1 – without inspection of possible sources of feature interactions down the stack of layers.

7. Case Study: A Layered Architecture with Feature Interactions at Deeper Layers

The example in Fig. 27 27 is rather simple and may appear somewhat artificially contrived. However, it was chosen to illustrate the feature interaction issue in a way that is both easily understood and yet sufficiently close to a practical system to be credible. In particular, it is intended to explain the effect of feature interactions deep down in layers and how that effects the provided services at higher layers.

7.1. The Architecture

The architecture of this layered system is inspired by one of the original layered architecture designs: the T.H.E operating system described by E.W. Dijkstra (see (Dijkstra 1983)).

The architecture is in the form of a hierarchy of software layers designed to incrementally transform the underlying computing hardware into a virtual machine/platform that targets a specific category of concurrent applications:

- **Layer L1**: includes a set of standard OS capabilities and services, but, for our purpose, we are only concerned with the File System service.
- **Layer L2**: Transaction Service layer, which adds a transaction-based interface to the underlying File System service. It is realized in the form of a *single* concurrent process, which supports a standard transaction type capability to its clients.
- **Layer L3**: A "programming framework" layer that provides a set of services customized for the domain of the applications in layer L4. This includes:
  - *Device Abstraction Services*, which provide a convenient abstract view of individual input and output devices (like the controller approach described in sect. 5.3). These take the form of concurrent "device handler" processes, which interact with their respective application clients through distinct concurrent message queues/buffers. Separate queues are used for receiving and sending messages. (These are suitable for modeling as channels for carrying message streams of the appropriate type.)
  - A *Logging Services* for keeping a record of communications between applications and their corresponding input and output devices. To support clear separation of communications associated with each application, there is a dedicated Log Service Agent associated with each application client. However, to allow crisp separation of records maintained for the individual application clients, the Transaction Service handles only one transaction at a time. (This means that large records may first need to be broken up into smaller chunks so that the corresponding transactions don't take up excessive time.)[4]
- **Layer L4**: the "applications" layer, where multiple *independently* designed application programs can execute concurrently.

In practical terms, all service calls between layers in this system are executed by means of *synchronous procedure calls*.

7.2. The Applications

Two independently conceived and realized applications executing concurrently, each running on its own process/thread:

- **AppA**: This simple application accepts *sporadic* inputs from the corresponding device handler (via the corresponding message queue processes). When it receives a sufficient

[4] Admittedly, this is a suboptimal design, but it is realistic enough to serve our purpose.

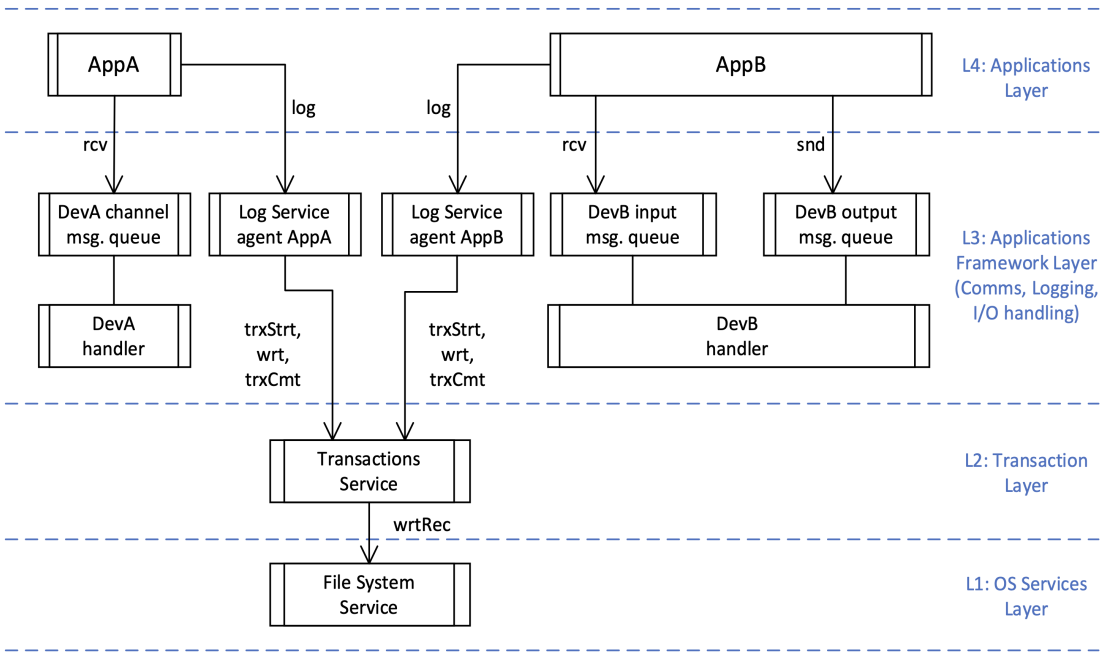

Figure 27 Layered architecture of the case study

number of inputs, it bundles them (in order of arrival) into a single message, and sends that to its corresponding Log Service Agent for transmission to the File System via the Transaction Service. Note that, because the inputs from the device are sporadic, the time interval between successive inputs is non-deterministic.

– **AppB**: This is a time sensitive application, which takes *periodic* inputs from its corresponding device handler (via its input message queue process) and reacts to them by sending an appropriate control command to the device (via its output message queue process). This application is time sensitive, and the output must be returned within a fixed time interval T, or an error situation will occur.

7.3. The Feature Interaction Scenario

The interleaving between transactions driven by AppA and those driven by AppB is non-deterministic, since the inputs sent to AppA from its device are sporadic rather than periodic. Thus, it may happen that an excessively large logging transaction initiated by AppA takes place *just before* a logging transaction from AppB is supposed to commence. Because all inter-layer service calls are realized by synchronous procedure calls, this will block the AppB process from executing its response to the most recent input. If the timing of the near-simultaneous logging requests is just right, the result may be a *missed deadline* and an error state for AppB.

The result is a classic example of unintended feature interaction, whereby two independently designed applications interfere with each other because they share a common resource (the Transaction Service process in L2). Note that, formally, this error occurs at the application layer (L4), although it is caused by a shared resource conflict that occurs at two layers down (L2).

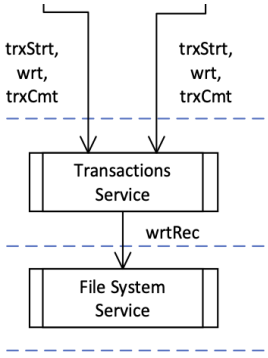

Figure 28 Fragment from the layered architecture of the case study in Fig. 27

7.4. Modelling the Effects of the Feature Interaction

We consider the fragment shown by Fig. 28 of the architecture shown in Fig. 27.

For a timed stream s we use the following notation where $n \in \mathbb{N}$ (also see the Appendix):

$$\begin{array}{rl} \#x & \textit{number of messages in stream } x \\ x(n) & \textit{the nth message in } x\ (1 < n \leq \#x) \\ n@x & \textit{time of the nth message in } x\ (1 < n \leq \#x) \end{array}$$

The example introduced above works via procedure calls. We model a procedure call in the frameworks of streams by two messages: the call as a message issued by the caller over a call channel cc from the caller to the callee, and the reply by a message sent in response by the callee over a return channel cr from the callee to the caller. Because message processing takes time, for each call (calls are numbered, and we assume at

least n calls), we get for the timing of the call number n and the corresponding return message

$$n@cc < n@cr$$

It specifies that a return message for a call is issued after the call has been received. Next, we specify the call protocol that contains a specification *Cspec* for the caller and a specification *Rspec* for the callee:

$$\begin{aligned} &Cspec(cc,cr) = (\#cc \leq 1 + \#cr \wedge \\ &\quad \forall n \in \mathbb{N} : 1 < n \leq \#cc \Rightarrow n@cc > n-1@cr) \\ &Rspec(cc,cr) = (\#rc \leq \#cc \wedge \\ &\quad \forall n \in \mathbb{N} : 1 \leq n \leq \#cr \Rightarrow n@cr > n@cc) \end{aligned}$$

Note that in the procedure call interaction between a caller C and a callee B over call channel cc and return channel cr, the callee B assumes $Cspec(cc,cr)$ and guarantees $Rspec(cc,cr)$, while the caller C assumes $Cspec(cc,cr)$ and guarantees $Rspec(cc,cr)$. We then obtain the following two parts of their specification:

$$\begin{aligned} Rspec(cc,cr) &\Rightarrow Cspec(cc,cr) \quad Caller \\ Cspec(cc,cr) &\Rightarrow Rspec(cc,cr) \quad Callee \end{aligned}$$

This looks strange at a first sight, since their composition (a conjunction), yields:

$$\begin{aligned} &(Rspec(cc,cr) \Rightarrow Cspec(cc,cr)) \wedge \\ &(Cspec(cc,cr) \Rightarrow Rspec(cc,cr)) \end{aligned}$$

which does not appear very helpful to conclude anything. However, when we apply induction over a series of calls, and take into account the specification of the timing, and the rule of strong causality, we can show that this leads to:

$$Cspec(cc,cr) \wedge Rspec(cc,cr)$$

The above can be simplified in the case where calls are always returned. Hence, $\#cr = \#cc$ so that the call protocol can be specified as follows:

$$\begin{aligned} &Cprot(cc,cr) = (\#cr = \#cc \wedge \\ &\quad \forall n \in \mathbb{N} : 1 \leq n \leq \#cc \Rightarrow \\ &\quad ((1 < n \Rightarrow n@cc > n-1@cr) \wedge n@cr > n@cc)) \end{aligned}$$

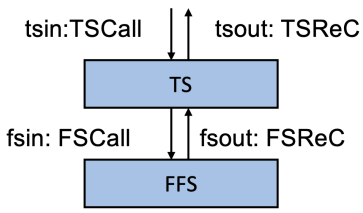

Figure 29 Fragment of the composition of transaction Service *TS* with the file system service *FSS*

We model the layer containing the Transaction Service *TS* as follows (note that we only consider the time flow but not the data flow):

$$\begin{aligned} TS \vdash &(tsin : TSCall, fsout : FSReC \blacktriangleright \\ &\quad tsout : TSReC, fsin : FSCall) : \\ &Cspec(tsin,tsout) \wedge Rspec(fsin,fsout) \Rightarrow \\ &\quad Rspec(tsin,tsout) \\ &\wedge Cspec(fsin,fsout) \\ &\wedge \#tsin = \#fsin \\ &\wedge \forall n \in \mathbb{N} : 0 < n \leq \#fsout \Rightarrow \\ &\quad (\exists e \in \mathbb{N} : e \in exectimets(tsin(n)) \wedge \\ &\quad n@tsout = (n@fsout) + e) \end{aligned}$$

where:

- *TSCall* denotes the type of calls to the Transaction Service *TS*,
- *TSReC* denotes the type of returns from the Transaction Service *TS*,
- *FSCall* denotes the type of calls to the File System Service *FFS*,
- *FSReC* denotes the type of returns from the File System Service *FFS*
- *tsin* denotes the call channel for calls to the Transaction Service *TS*,
- *tsout* denotes the return channel for return messages of the Transaction Service *TS*,
- *fsin* denotes the call channel for calls to the File System Service *FFS*,
- *fsout* denotes the return channel for return messages of the File System Service *FFS*
- *exectimefs(c)* denotes the set of execution times for the File System Service *FFS TS* for input c
- *exectimets* denotes the set of execution times for the Transaction Service *TS*
- *exectimeabts* denotes the set of execution times for the transaction service *ABTS*

For the File System Service *FSS*, given the set of possible execution times *exectimefs(c)* for each call $c \in FSCall$, we get:

$$\begin{aligned} FFS \vdash &(fsin : FSCall \blacktriangleright fsout : FSReC) : \\ &Cspec(fsin,fsout) \Rightarrow (Rspec(fsin,fsout) \\ &\wedge \forall n \in \mathbb{N} : 0 < n \leq \#fsin \Rightarrow \\ &\quad (\exists e \in \mathbb{N} : e \in exectimefs(fsin(n)) \wedge \\ &\quad n@fsout = (n@fsin) + e)) \end{aligned}$$

This specification asserts that each call arriving over *tsin* is forwarded at some later time to the File System Service *FSS* and, as soon as the answer is received, it is sent back.

These conditions are combined by the composition of *TS* with

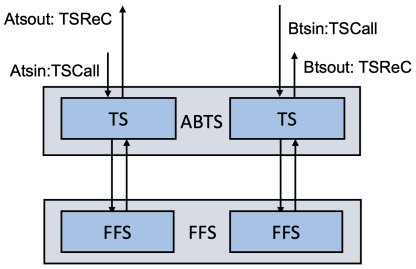

Figure 30 Fragment from the layered architecture with two separated copies of FFS without feature interaction

FFS, as explained in sect. 4.2:

$$
\begin{aligned}
&TS \otimes FFS \vdash (tsin : TSCall \blacktriangleright tsout : TSReC) : \\
&\quad \exists\, fsout \in FSReC, fsin \in FSCall : \\
&\qquad Cspec(tsin, tsout) \wedge Rspec(fsin, fsout) \Rightarrow \\
&\qquad\quad Rspec(tsin, tsout) \\
&\qquad \wedge\, Cspec(tsin, tsout) \\
&\qquad \wedge\, \#tsin = \#fsin \\
&\qquad \wedge\, (\forall\, n \in \mathbb{N} : 0 < n \leq \#fsout \Rightarrow \\
&\qquad\quad \exists\, e \in \mathbb{N} : e \in exectimets \wedge \\
&\qquad\qquad n@tsout = (n@fsout) + e) \\
&\qquad \wedge\, Cspec(fsin, fsout) \Rightarrow (Rspec(fsin, fsout) \\
&\qquad\quad \wedge\, \forall\, n \in \mathbb{N} : 0 < n \leq \#fsin \Rightarrow \\
&\qquad\qquad \exists\, e \in \mathbb{M} : e \in exectimefs(fsin(n)) \\
&\qquad\qquad\quad \wedge\, n@fsout = (n@fsin) + e)
\end{aligned}
$$

which implies (by induction proof):

$$
\begin{aligned}
&\exists fsout \in FSReC, fsin \in FSCall : \\
&\quad Cspec(tsin, tsout) \Rightarrow \\
&\qquad Cprot(tsin, tsout) \\
&\quad \wedge\, \#tsin = \#fsin \\
&\quad \wedge\, (\forall\, n \in \mathbb{N} : 0 < n \leq \#fsout \Rightarrow \\
&\qquad \exists\, e \in \mathbb{N} : e \in exectimets \wedge n@tsout = (n@fsout) + e) \\
&\quad \wedge\, Cprot(fsin, fsout) \\
&\quad \wedge\, \forall\, n \in \mathbb{N} : 0 < n \leq \#fsin \Rightarrow \\
&\qquad (\exists\, e \in \mathbb{N} : e \in exectimefs(fsin(n)) \\
&\qquad\quad \wedge\, n@fsout = (n@fsin) + e)
\end{aligned}
$$

It is straightforward to define a similar architectural fragment with the two services dedicated to *AppA* and *AppB* respectively (see Fig. 30).

This gives us information about the specified timing *for the case without feature interaction* (see Fig. 30).

To model the feature interaction, we give a specification of

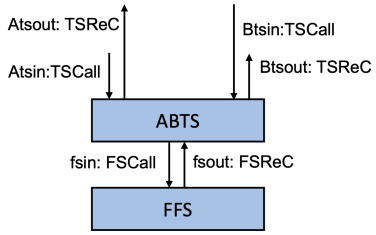

Figure 31 Fragment from the layered architecture with sharing of FFS with feature interaction

the transaction service *ABTS* (see Fig. 31):

$$
\begin{aligned}
&ABTS \vdash (Atsin, Btsin : TSCall, fsout : FSReC \blacktriangleright \\
&\qquad Atsout, BTsout : TSReC, fsin : FSCall) : \\
&\quad Cspec(Atsin, Atsout) \wedge Cspec(Btsin, Btsout) \\
&\qquad \wedge\, Rspec(fsin, fsout) \Rightarrow \\
&\quad Rspec(Atsin, Atsout) \\
&\quad \wedge\, Rspec(Btsin, Btsout) \\
&\quad \wedge\, Cspec(fsin, fsout) \\
&\quad \wedge\, Scheduling(Atsin, Btsin, fsout, Atsout, Btsout, fsin)
\end{aligned}
$$

where *Scheduling(Atsin, Btsin, fsout, Atsout, Btsout, fsin)* represents the scheduling function, which is defined as follows.

Scheduling(Atsin, Btsin, fsout, Atsout, Btsout, fsin) is specified as a first come first served strategy as follows:
Here we use two functions *ha* and *hb* to indicate in which order the incoming messages on the streams *Atsin* and *Btsin* are scheduled into the stream *fsin*. Furthermore, *an* and *bn* represent natural numbers that correspond to time instants of the corresponding streams. The notation $[1 : n]$ represents an interval of naturals from 1 to $n \in \mathbb{N}$. There exist functions:

$$ha : [1 : an] \rightarrow [1 : (an + bn)]$$
$$hb : [1 : bn] \rightarrow [1 : (an + bn)]$$

that fulfill the following equations (*ha* and *hb* are strictly increasing and their images are distinct):

$$\forall\, i, j \in [1 : an] : i < j \Rightarrow ha(i) < ha(j)$$
$$\forall\, i, j \in [1 : bn] : i < j \Rightarrow hb(i) < hb(j)$$
$$\forall\, i \in [1 : an], j \in [1 : bn] : ha(i) \neq hb(j)$$

This ensures, that *ha* and *hb* define a schedule. It has to preserve the timing constraints:

$$
\begin{aligned}
&\forall\, i \in [1 : an],\, j \in [1 : bn] : \\
&\quad (i@Atsin < j@Btsin \Rightarrow ha(i) < hb(j)) \\
&\quad \wedge\, (j@Btsin < i@Atsinimplieshb(j) < ha(i))
\end{aligned}
$$

The calls arising on *Atsin* and *Btsin* are scheduled as soon as possible, which means as soon as they arrive and the last call has returned (in case this is not the first call to be scheduled):

$$
\begin{aligned}
&\forall\, i \in [1 : an], k \in [1 : (an + bn)] : \\
&\quad ha(i) = k \Rightarrow \exists\, e \in \mathbb{N} : e \in exectimeabts \\
&\qquad (k = 1 \wedge k@fsin = (i@Atsin) + e) \\
&\qquad \vee\, (k \neq 1 \wedge k@fsin = max(k - 1@fsout, i@Atsin) + e) \\
&\forall\, i \in [1 : bn], k \in [1 : (an + bn)] : \\
&\quad hb(i) = k \Rightarrow \exists\, e \in \mathbb{N} : e \in exectimeabts \\
&\qquad (k = 1 \wedge k@fsin = (i@Btsin) + e) \\
&\qquad \vee\, (k \neq 1 \wedge k@fsin = max(k - 1@fsout, i@Btsin) + e)
\end{aligned}
$$

For of streams *Atsout* and *Btsout* that are the output of *ABTS* the timing is determined by the output on *fsout*:

$$\forall i \in [1:an] : \exists e \in \mathbb{N} :$$
$$e \in exectimeabts \wedge (i@Afsout) = (ha(i)@fsout) + e$$
$$\forall j \in [1:bn] : \exists e \in \mathbb{N} :$$
$$e \in exectimeabts \wedge (j@Bfsout) = (hb(j)@fsout) + e$$

The scheduling is done in a first come first served order.

If we compare the two specifications for systems shown in Fig. 30 and Fig. 31, they deliver different provided services *ATS* and *BTS*. For the latter case, the feature interaction is visible in the interface behavior of the subservice provided by $(Btsin, Btsout)$. This is easy to seen by a simple example: assume that we have

$$(1@Atsin) + 1 = 1@Btsin$$

(i.e., *Btsin* occurs after *Atsin*). Then $ha(1) < hb(1)$ and, consequently

$$hb(1)@tsout > (ha(1)@fsin) + e + e'$$

for some $e, e' \in exectimeabts$ and thus

$$1@Btsout > (1@fsin) + e + e'$$

while in the case of the feature free architecture we get

$$1@Btsout = (1@fsin) + e'$$

for some $e' \in exectimeabts$. The higher delay of the service *ATS* as offered by *ABTS* due to the sharing of service *FFS* is, of course, visible at the interface of the Transaction Service, and consequently also at the interface of the Log Service. If $1@Atsin > 1@Btsin$ then there is no delay for $Btsout(1)$. Conversely, too long a delay may result in a missed deadline and an error state in *AppB*. This demonstrates that we can detect a feature interaction simply by examining the interface specification of a system, *without having to delve into its internal implementation*.

Note that here we mapped remote procedure calls onto data flow and stream processing. In addition, there are examples beyond remote procedure calls that can be handled by this kind of model. As noted in sect. 5.1 in this example we have specified an explicit time flow between the provided and the required service.

8. Concluding Remarks

The approach that we have presented provides an abstract, idealized view of layered architectures. It does not cover a number of more specific technical issues that might arise if layered architectures are not completely described in a modular style as discussed in sect. 4 (although, even such effects can be described directly in our approach). However, what we feel is most important is that we have provided a precise syntactic and semantic description of layered architectures.

8.1. Related Work

Plenty of work on system architecture deals with layered systems, tiered architectures, client/server architectures, hierarchical and vertical composition – all terms and concepts related somewhat to layering. However, the terminology used in the literature is often vague and confusing.

Following (Schmidt et al. 1996), the term "layer" designates a logical structuring of groups of components in a software system. In contrast, "client/server" refers to a generic invocation relationship between services offered within a system and their clients, *independent of their respective placements within the structure of the system or their deployment*. That is, it simply characterizes a generic invocation relationship, whereby a client requests another server to perform a service task for its benefit. Therefore, the two are distinct concepts, despite the fact that both deal with relationships. For example, in a layered architecture, when a service in one layer invokes a service of a layer directly below, these two services are in a "vertical" client/server relationship (see sect. 1 and also sect. 5.2). However, the same client/server pattern also applies in the "horizontal" case when a service in a layer calls another service within the same layer.

Layered architectures are often confused with *tiered* architectures, although the latter are defined in terms of their physical deployment structure. In this case, a tier stands for a (cluster of) computing nodes, with each tier providing a set of specialized services. A client residing in one tier invokes the services of a remote server in a different tier, in what constitutes a "horizontal" client/server relationship. For example, for their Java Enterprise technology, Sun Microsystems has developed a 4-tier architecture comprising a Client Tier, Presentation Tier, Business Services Tier, and a Data Tier, each residing on a separate computing node[5].

Our work is a continuation of (Herzberg & Broy 2005). Of course, we can think even further here and move on from tier architecture to mesh architecture. There is clearly potential for significant future research in this domain, in terms of suitable architectural patterns, reference architectures, and much more.

8.2. On Modeling Layers

In the example in sect. 7 we have demonstrated that classical remote procedure call mechanism can be modeled using FOCUS streams; i.e., by representing them as streams of invocations and matching streams of responses. This is a typical example of an expanded layer. Of course, procedure calls are just one way of how the interaction between layers of a layered architecture could be realized. Due to the high expressive power of this specification technique, we can represent different kinds of protocols by the stream processing data flows.

We distinguish basic and expanded layers. Basic layers follow the platform idea, where layers are seen simply as platforms that support implementations of their upper-layer clients. They follow the idea of information hiding, where the specifics of how required services are used by the upper layer makes no observable difference to its own provided services. However, in contrast, expanded layers include explicit specifications of data

[5] https://docs.oracle.com/cd/E19636-01/819-2326/aavdm/index.html

flow between the provided and required service interfaces of a layer.

8.3. On the Formal Model

The logical view we have introduced and used allows us to describe and deal with a large number of concepts, patterns, and issues connected to layered architectures at a relatively high level of abstraction. We do not consider more implementation-oriented technical issues of communication but rather used the more abstract concepts of channels and streams. We also considered effects of feature interactions with systems and their services residing on the same platform, leading to corruption of required services (see the example in sect. 7). However, we did not delve into *quality of service* aspects, such as response times and the like, which might be pertinent in time-sensitive applications, such as in the case study in sect. 7.

The key intent of our approach is to give a formal definition of a layered architecture that goes beyond conventional implicit or ambiguous definitions and explanations, such as those based on graphical placement, which are open to misinterpretation. They are simply insufficient to fully define what makes layering unique.

8.4. Methodology

In our model, one can see that, in a layered architecture, the higher layers are only guaranteed to produce their provided services if the required services are delivered correctly by its supporting layers. This specification method can be used in a number of ways: supporting modular design, doing proofs about layered architectures that involve specific data flows, feature interactions, and as a basis for defining tests.

Using the proposed method for a formal specification of layers, a specification of a concrete layer can be given, which consists of a specification of the provided and the required service. When constructing the implementation of the layer, the provided service can be implemented assuming that the required service is given. Then, the layer can be verified, for example, by defining test cases based on the specification of the provided and the requested services. Thus, the implementation of layer can be realized and verified in modular way, independent of the implementations of the layers above and below.

Another interesting aspect is that, in this model, it is clear and formally provable that we can replace layers in a layered architecture by different equivalent layers. That is, it is guaranteed that replacing layers in a layered architecture will not affect the overall correctness of the layered architecture as long as its original pattern of provided and required services, as specified, is still guaranteed. After all, layering introduces a discipline in system design (see (Sarkar et al. 2009)).

8.5. Future Work

The key idea in formalizing the layering pattern is the introduction of required services. This idea is not just restricted to the pattern of layering. We may, in general, look at specifications of systems which provide certain services and require certain services: The composition of those systems is not necessarily restricted to layering and could be done in a number of different ways. It seems sufficient to guarantee that, when composing systems, the services that are required by one of the systems are provided by the other systems. Therefore, it would be interesting to study more general patterns of system composition based on relationships between provided and required services.

The design pattern of a layer is an instance of an *assumption/commitment* specification (see (Broy 2018)). The idea of required services therefore can be applied in a more general way to architectures, not only for the case of layered systems, called layers, but also in more general architectural structures such as, for instance, mesh architectures.

Appendix: Streams, Time, and Behaviors

Throughout this paper, we use the terms system and service in a specific way. We address discrete systems, more precisely, models of discrete concurrent time-critical systems with input and output. In this appendix, we introduce a corresponding model of concurrent, distributed interactive systems according to FOCUS (see (Broy & Stølen 2012)). In this model, the interface behavior of *systems* is described by relations between input and output streams. This way, for us, a system is an entity that shows some specific interface behavior by interacting over its interface with its surrounding *operational context*. This behavior can be structured into *services* also called *functional features*. A system has a *boundary*, which defines what is inside and what is outside the system. Inside the system there is an encapsulated internal structure, often consisting of states or state attributes and an architecture composed of subsystems. This structure is hidden following the principle of *information hiding*. The set of actions and events that may occur in the interaction of the system with its operational context determines the *syntactic ("static") interface* of the system or the service. At its interface, a system shows a specific *interface behavior*. We specify interface behavior by *interface assertions*.

There are general logical properties that we assume for interface assertions. We require that system behaviors fulfill properties such as *strong causality* and *realizability* which are mandatory for a specification to be implementable. Causality requires that an output is only generated after a corresponding input has been received. Realizability requires the existence of an interaction strategy within the system such that, for an arbitrary input from the environment, the system will produce the correct output according to its specification (see (Broy 2023b)). However, not all interface assertions guarantee these two properties explicitly. Nevertheless, in such cases, causality or realizability can be added by a rule resulting in system specifications with these additional properties.

In FOCUS, being distributed, components of systems run in parallel and systems, in general, operate in parallel, evolving in a common physical time frame. We choose an interface concept to describe services in a property-oriented way, services that are interactive, time-critical, and run in parallel. In the following, we briefly introduce the concepts, models, and notations of FOCUS.

By M^* we denote the set of finite sequences of elements from set M which formally can be represented by functions ($\mathbb{N}$

denotes the natural numbers including 0, $\mathbb{N}_+ = \mathbb{N} - \{0\}$; $\mathbb{N}_+$ denotes the set of natural numbers without 0; $[1:n] \subseteq \mathbb{N}+$ denotes finite intervals of natural numbers for $n \in \mathbb{N}$):

$$M^* = \bigcup_{n \in \mathbb{N}} ([1:n] \to M)$$

By M^ω we denote the set of infinite sequences over M:

$$M^\omega = (\mathbb{N}_+ \to M)$$

The empty sequence as well as the empty stream is denoted by $\langle\rangle$.

By $(M^*)^\omega$ of infinite timed streams over a message set M we denote the set:

$$(M^*)^\omega = (\mathbb{N}_+ \to M^*)$$

In both cases, we use functions on the set of natural numbers or intervals thereof to define timed streams as well as time free streams. A stream $x \in (M^*)^\omega$ is called a *timed* stream, since we understand the set $\mathbb{N}_+$ to represent time in terms of an infinite sequence of time intervals. Each time interval is numbered by a number $t \in \mathbb{N}_+$. Then $x(t)$ with $t \in \mathbb{N}_+$ denotes the sequence of messages communicated by the timed stream x in time interval t. For $m \in M$, we denote the one element sequence and the one element time free stream by:

$$\langle m \rangle$$

Note that the role of natural numbers in $\mathbb{N}_+$ in the two expressions $(\mathbb{N}_+ \to M)$ and $(\mathbb{N}_+ \to M^*)$ is quite different. In case of timed streams x, the number $n \in \mathbb{N}_+$ denotes the sequence $x(n)$ transmitted in the nth time interval, whereas in time free streams $n \in \mathbb{N}_+$ simply denotes the ordinal position of a message $z(n)$ in a stream and does not represent time.

By $\#x \in \mathbb{N} \cup \{\infty\}$ we denote the number of elements from M occurring in a timed stream $x \in (M^*)^\omega$ or in a time free stream $x \in M^{*|\omega}$. For a timed stream $x \in (M^*)^\omega$ or a time free stream $x \in M^{*|\omega}$ over set M and a subset $D \subseteq M$ we denote by:

$$D\#x$$

the number of copies of elements from D in x; we also write $a\#x$ for $\{a\}\#x$ and $\#x$ for $M\#x$.

Given a sequence s of elements from M (or in the case of timed streams of elements, from M^*) and a stream x over M (finite or infinite, in the case of time streams of elements over M^*) we denote by:

$$s\hat{}x$$

the concatenation of sequence s with stream x.

Given $x \in (M^*)^\omega$ and $t \in \mathbb{N}$ we denote a *time cut* of length n as follows:

$$x \downarrow t \in ([1:n] \to M^*)$$

where:

$$(x \downarrow t)(n) = x(n) \Leftarrow 1 \leq n \leq t$$

$x \downarrow t$ denotes the first t sequences in the timed stream x, representing the messages transmitted in the first t time intervals.

Streams are send over channels that connect systems. Given a set X of typed channel names c_j with types T_j:

$$X = \{c_1 : T_1, \ldots, c_m : T_m\}$$

X is called a signature. By $\vec{X}$ we denote channel histories given by families of timed streams, one timed stream for each channel:

$$\vec{X} = (X \to (M^*)^\omega)$$

where we assume that, for $x \in \vec{X}$, the timed stream $x(c_k)$, $1 \leq k \leq m$, for the channel $c_k \in X$ carries messages of type S_k. All introduced notations, such as time abstraction $\bar{x}$, concatenation $s\hat{}x$, time slice $x \downarrow t$, continuation $x \uparrow t$, and prefix order carry over to channel histories $x \in \vec{X}$, as well.

Given signatures X and Y, a function on histories (the definition can be easily translated to functions on streams):

$$f : \vec{X} \to \vec{Y}$$

is called *strongly causal*, if:

$$\begin{aligned} \forall t \in \mathbb{N}, x, x' \in \vec{X} : x \downarrow t = x' \downarrow t \\ \Rightarrow f(x) \downarrow t+1 = f(x') \downarrow t+1 \end{aligned}$$

Then we write $SC[f]$.

An interface predicate Q (see sect. 4) for the syntactic interface $(X \blacktriangleright Y)$ is called *strongly causal* if

$$\begin{aligned} \forall t \in \mathbb{N}, x, x' \in \vec{X}, y \in \vec{Y} : \\ Q(x,y) \wedge x \downarrow t = x' \downarrow t \Rightarrow \exists y' \in \vec{Y} : \\ Q(x',y') \wedge y \downarrow t+1 = y' \downarrow t+1) \end{aligned}$$

Then we write $SC[Q]$.

An interface predicate Q for the syntactic interface $(X \blacktriangleright Y)$ is called *realizable*, if the there exists a strongly causal function f such that:

$$\forall x \in \vec{X} : Q(x, f(x))$$

Then the function f is called *realization* of Q. A system specification $(X \blacktriangleright Y) : Q$ is called *fully realizable* if for all $x \in \vec{X}$ and $y \in \vec{Y}$ for which $Q(x,y)$ holds, there exists a realization f for $(X \blacktriangleright Y) : Q$ such that $y = f(x)$. Every realization:

$$f : \vec{X} \to \vec{X}$$

has exactly one fixpoint which can be proved by Banach's fixpoint theorem since f is a contractive function (see (Banach 1922)).

An interface predicate Q for the syntactic interface $(X \blacktriangleright Y)$ is called *fully realizable* if:

$$Q(x,y) = (\exists f : \vec{X} \to \vec{Y} : y = f(x) \wedge Q[f])$$

where $Q[f]$ holds if f is a realization for Q:

$$Q[f] = (SC[f] \wedge \forall x \in \vec{X} : Q(x, f(x)))$$

We assumed that systems are implemented (realized) by generalized Moore machines (see (Moore et al. 1956)) which serve as the operational model. Since Moore machines compute interface behaviors that are fully realizable, every fully realizable interface predicate Q corresponds to a Moore machine that computes for input history x the output history y. If $Q(x,y)$ is a fully realizable specification, it is a correct and complete logical description of the behavior of the Moore machine that implements Q.

If an interface predicate Q for the syntactic interface $(X \blacktriangleright Y)$ is not fully realizable, we define the weakest fully realizable interface predicate $Q^{®}$ that is a refinement of interface predicate Q as follows (see (Broy 2023b)):

$$Q^{®}(x,y) = \exists f : \vec{X} \to \vec{Y} : y = f(x) \wedge Q[f]$$

The predicate $Q^{®}$ always exists – but may be equal to the specification false in case of a unrealizable specification Q. A fully realizable interface predicate is always strongly causal.

We assume that an interface specification:

$$S \vdash (x : X \blacktriangleright y : Y) : Q(x,y)$$

specifies a system S where S is realized by a Moore machine (see (Broy 2023b)). Since the behavior of every Moore machine is fully realizable, system S is fully realizable, so we can conclude that:

$$S \vdash (x : X \blacktriangleright y : Y) : \exists f : \vec{X} \to \vec{Y} : y = f(x) \wedge Q[f]$$

Since a fully realizable specification is strongly causal, this implies:

$$\begin{aligned} &S \vdash (x : X \blacktriangleright y : Y) : \\ &\quad (Q(x,y) \\ &\quad \wedge \forall t \in \mathbb{N}, x' \in \vec{X} : \\ &\qquad \exists y' \in \vec{Y} : R((x \downarrow t)^\frown x', (y \downarrow (t+1))^\frown y')) \end{aligned}$$

since the proposition:

$$\begin{aligned} &(Q(x,y) \wedge \forall t \in \mathbb{N}, x' \in \vec{X} : \exists y' \in \vec{Y} : \\ &\quad R((x \downarrow t)^\frown x', (y \downarrow (t+1))^\frown y')) \end{aligned}$$

is the weakest refinement of $Q(x,y)$ that is strongly causal. In most practical cases, a strongly causal specification is fully realizable. Therefore, in such cases the weakest strongly causal refinement is already sufficient to prove all properties that hold for all Moore machines that realize the specification.

Note that for a system S specified by an interface predicate Q:

$$S \vdash (x : X \blacktriangleright y : Y) : Q(x,y)$$

we can deduce a strongly causal specification:

$$\begin{aligned} &S \vdash (x : X \blacktriangleright y : Y) : \\ &\quad (Q(x,y) \wedge \forall t \in \mathbb{M}, x' \in \vec{X} : \exists y' \in \vec{Y} : \\ &\qquad R((x \downarrow t)^\frown x', (y \downarrow (t+1))^\frown y')) \end{aligned}$$

and even to a fully realizable specification:

$$S \vdash (x : X \blacktriangleright y : Y) : \exists f : \vec{X} \to \vec{Y} : y = f(x) \wedge Q[f]$$

A fully realizable interface predicate then permits derivation of all interface properties of implementations. Therefore, the calculus that comprises the rule to refine interface predicates into fully realizable ones is sound and relative complete.

Acknowledgments

The authors express their heartfelt gratitude to the "superhero" LaTex expert who helped convert the complex and intricate text of this document from its original MS Word format into the LaTex format required by the Journal. In addition, we are also extremely grateful to those who did such a professional and thorough job of completing what was undoubtedly a most daunting task.

References

Banach, S. (1922). Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta mathematicae*, *3*(1), 133–181.

Broy, M. (2010). Multifunctional software systems: Structured modeling and specification of functional requirements. *Science of Computer Programming*, *75*(12), 1193–1214.

Broy, M. (2018). Theory and methodology of assumption/-commitment based system interface specification and architectural contracts. *Formal Methods in System Design*, *52*, 33–87.

Broy, M. (2023a). A calculus for the specification and verification of distributed concurrent systems.
(To appear)

Broy, M. (2023b). Specification and verification of concurrent systems by causality and realizability.
(Submitted for publication)

Broy, M., Krüger, I. H., & Meisinger, M. (2007). A formal model of services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *16*(1), 5–es.

Broy, M., & Stølen, K. (2012). *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media.

Clements, P., Garlan, D., Little, R., Nord, R., & Stafford, J. (2003). Documenting software architectures: views and beyond. In *25th international conference on software engineering, 2003. proceedings.* (pp. 740–741).

Dijkstra, E. W. (1983). The structure of "THE"-multiprogramming system. *Communications of the ACM*, *26*(1), 49–52.

Herzberg, D., & Broy, M. (2005). Modeling layered distributed communication systems. *Formal Aspects of Computing*, *17*, 1–18.

Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied software architecture*. Addison-Wesley Professional.
Mary, S., & David, G. (1996). Software architecture: Perspectives on an emerging discipline. *Prentice-Hall*.
Mietzner, R., Fehling, C., Karastoyanova, D., & Leymann, F. (2010). Combining horizontal and vertical composition of services. In *2010 ieee international conference on service-oriented computing and applications (soca)* (pp. 1–8).
Moore, E. F., et al. (1956). Gedanken-experiments on sequential machines. *Automata studies*, *34*, 129–153.
Sarkar, S., Maskeri, G., & Ramachandran, S. (2009). Discovery of architectural layers and measurement of layering violations in source code. *Journal of Systems and Software*, *82*(11), 1891–1905.
Schmidt, D., Stal, M., Rohnert, H., & Buschmann, F. (1996). *Pattern-oriented software architecture, volume 1: a system of patterns.* John Wiley & Sons Chichester, UK.
Selic, B. (2020). The forgotten interfaces: A critique of component-based models of computing. *J. Object Technol.*, *19*(3), 3–1.
Selić, B. (2021). On the precise semantics of the software layering design pattern. *Journal of Object Technology*, *20*(2), 2:1-13.
Zdun, U., & Avgeriou, P. (2005). Modeling architectural patterns using architectural primitives. *ACM SIGPLAN Notices*, *40*(10), 133–146.

About the authors

Manfred Broy is professor emeritus for Software and Systems Engineering at the Faculty of Informatics at the Technische Universität München. He is doing research at the foundation and practice of software intensive cyper-physical systems. You can contact the author at broy@in.tum.de or visit https://wwwbroy.in.tum.de/~broy/.

Bran Selić is President of Malina Software Corp. and Adjunct Professor of Software Engineering at Monash University. He has over 50 years of industrial and research experience, focusing primarily on cyber-physical systems and model-based engineering technologies. In 2022, he was awarded an honorary doctorate from Mälardalen University in Sweden. You can contact the author at selic@acm.org.