On Architecture Specification

Manfred Broy(✉)

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

Abstract. The design, specification, and correct implementation of an architectural design are after the task of requirements specification the perhaps most important design decisions, when building large software or software based systems. Architectures are responsible for software quality, for a number of quality attributes such as maintainability, portability, changeability, reusability but also reliability, security, and safety. Therefore, the design of architectures is a key issue in system and software development. For highly distributed, networked systems and for cyber-physical systems we need a design concept which supports composition, parallelism, and concurrency and finally real time but keeps all of the general advantages of object-oriented programming. We describe an approach to specify and implement systems along the lines of some of the established concepts of object-orientation – such as inheritance and class instantiation. This leads to an approach that nevertheless provides an execution model which is parallel and concurrent in nature and supports real time and modular composition. This way, it lays the foundation of a software and systems engineering style where classical object-orientation can be extended to cyber-physical systems in straightforward way.

Keywords: Specification · Design · Contracts · Assumptions · Commitments
System specification · Interface · Architecture

1 Introduction

Object-oriented programming is currently the perhaps most widely used programming style in software development. It combines a number of useful concepts in programming in a way that, in particular, the development of large software systems is supported by it. Nevertheless, object-oriented programming shows a number of deficiencies when dealing with distributed cyber-physical systems. First of all, in classical object-oriented programming the execution model is inherently sequential. All attempts to extend or generalize it to parallel execution models without significant changes in the underlying execution model make the understanding and design of object-oriented programs utterly complicated. Secondly, the composition of object-oriented programs shows some weaknesses and open issues. This is related to the recognized lack of a clear notion of component, a lack of parallel composition, and the lack of a parallel execution model as needed usually for the development of

A M. Tjoa et al. (Eds.): SOFSEM 2018, LNCS 10706, pp. 19–39, 2018.
https://doi.org/10.1007/978-3-319-73117-9_2

cyber-physical systems as we see them nearly everywhere nowadays. A further issue is time and probability which are first class citizens in cyber-physical applications.

When looking at software families and product lines, architecture becomes even more significant, because it determines the possibilities and options of variability and reusability (see [8]). With this in mind, it is a key issue to have an appropriate methodology with a calculus for the design of architectures. This includes a number of ingredients.

- A key concept for *subsystems*, also called components, as building blocks of architectures: this means that we have to determine what the concept of a subsystem is and, in particular, what the concept of an *interface* and *interface behavior* is. Interfaces are the most significant concept for architectures. Subsystems are composed and connected via their interfaces.
- The second ingredient is *composition*. We have to be able to compose systems by composition via their interfaces. Composition has to reflect parallel execution.
- This requires that interfaces of subsystems can be structured into a *family* of sub-interfaces, which are then the basis for the composition of subsystems, more precisely the composition of sub-interfaces of subsystems with other sub-interfaces of subsystems. For this we need a syntactic notion and a notion of behavior interface.
- In addition, we are interested in options to specify properties of interface behaviors in detail.
- Moreover, we have to be able to deal with interface types and subsystem types. These concepts allow us to introduce a notion of subsystems and their types, called system classes as in object-oriented programs, and these can also be used to introduce types of interfaces, properties of assumptions of the interfaces of subsystems which we compose.
- As a result, we also talk about the concept of refinement of systems and their interfaces as a basis of inheritance.

A key is the ability to specify properties of subsystems in terms of their interfaces and to compose interface specifications in a modular way.

In the following, we introduce a logical calculus to deal with interfaces and show how we can use it to define subsystems via properties of their interface assumptions also be able to deal with architectural patterns such as layered architectures.

2 A Formal Model of Interfaces

The key to software and system design is interface specifications where we do not only describe syntactic interfaces but also specify interface behavior.

2.1 Data Models

Systems exchange messages. Messages are exchanged between systems and their operational context and also between subsystems. Systems have states. States are composed of attributes. In principle, we can therefore work out the data model for a

service-oriented architecture which consists, just as an object-orientation, of all the attributes which are part of the local states of the subsystems which consists of the description of the data which are communicated over the interfaces between the subsystems.

2.2 Syntactic Interfaces and Interface Behavior

We choose a very general notion of interface where the key is the concept of a channel. A channel is a directed typed communication line on which data of the specified type are transmitted. As part of an interface, a channel is a possibility to provide input or output to a system. Therefore, we speak about input channels and output channels.

Syntactic Interfaces
An interface defines the way a system interacts with its context. Syntactically an interface is specified by a set C of channels where each channel has a data type assigned that defines the set of messages, events, or signals that are transmitted over that channel.

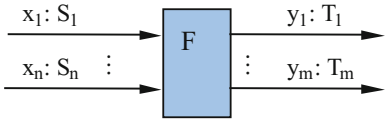

Fig. 1. Graphical representation of a system F as a data flow node with its syntactic interface consisting of the input channels x_1, …, x_n of types S_1, …, S_n and the output channels y_1, …, y_m of types T_1, …, T_m, resp.

In this section, we briefly introduce syntactic and semantic notions of discrete models of *systems* and their *interfaces*. This theoretical framework is in line with [1] called the FOCUS approach. Systems own input and output channels over which streams of messages are exchanged. In the following we denote the universe of all messages by IM.

Let I be a syntactic interface of typed input channels and O be a syntactic interface of typed output channels that characterize the syntactic interface of a system. (I▶O) denotes this *syntactic interface*. Figure 1 shows system F with its syntactic interface in a graphical representation as a data flow node.

System Interaction: Timed Data Streams
Let $\mathbb{N}$ denote the natural numbers (including 0) and $\mathbb{N}^+$ denote the strictly positive natural numbers.

The system model is based on the concept of a global clock. The system model can be described as time synchronous and message asynchronous. In the following, we work with streams that include discrete timing information. Such streams represent histories of communications of data messages transmitted within a time frame. By this model of discrete time, time is structured into an infinite sequence of finite time intervals of equal length. We use the natural numbers $\mathbb{N}^+$ to number the time intervals.

Definition. Timed Streams

Given a message set $M \subseteq IM$ of data elements of type T we represent a *timed stream* s of type T by a function

$$s : \mathbb{IN}^+ \to M^*$$

In a timed stream s a sequence of messages s(t) is given for each time interval $t \in \mathbb{IN}^+$; $s(t) = \varepsilon$ indicates that in time interval t no message is communicated. By $(M^*)^\infty$ we denote the set of timed streams. ❑

Throughout this paper, we work with a couple of basic operators and notations for streams over the message set that are shortly summarized as follows:

$\langle\ \rangle$	empty sequence or empty finite stream,
$\langle m \rangle$	one-element sequence containing m as its only element,
a^s	concatenation of the finite sequence with the finite or infinite sequence s,
s(t)	element in the t-th time interval of the stream s,
$s\downarrow t$	prefix of length $t \in \mathbb{IN}$ of the stream s (which corresponds to a sequence of message in t time intervals),
$s\uparrow t$	the stream s without its first t time intervals,
#s	number of messages in stream s,
M#s	number of copies of messages of stream s that are in a given set $M \subseteq IM$ (for {m}#x we also write m#x),
$\bar{x}$	denotes the result x(1)^x(2)^ … of concatenating the sequences x(1), x(2), x(3), … resulting in a finite stream in M* or an infinite stream in $(\mathbb{IN}^+ \to M)$.

A channel history for a set C of typed channels (which is a set of typed identifiers) assigns to each channel $c \in C$ a timed stream of messages communicated over that channel.

Let C be a set of typed channels; a (total) *channel history* x is a mapping

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

such that x(c) is a timed stream of type Type(c) for each channel $c \in C$. We denote the set of all channel histories for the channel set C by $\overrightarrow{C}$. A finite (partial) channel history is a mapping

$$x : C \to (\{1, \ldots, t\} \to M^*)$$

with some number $t \in \mathbb{IN}$ such that x(c) respects the channel type of c. ❑

As for streams, for every history $z \in \overrightarrow{C}$ and every time $t \in \mathbb{IN}$ the expression $z\downarrow t$ denotes the partial history (the communication on the channels in the first t time intervals) of z until time t. $z\downarrow t$ yields a finite history for each of the channels in C represented by a mapping of the type $C \to (\{1, \ldots, t\} \to IM^*)$. $z\downarrow 0$ denotes the history with the empty sequence associated with all its channels.

Interface Behavior

For a given syntactic interface (I▶O) a relation that relates the input histories in $\overrightarrow{I}$ with output histories in $\overrightarrow{O}$ defines its behavior. It is called *system interface behavior* (see [10]). We represent the relation by a set-valued function. In the following we write $\wp$ (M) for the power set over M.

Definition. Interface Behavior and Causal Interface Behavior

A function

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

is called an I/O-behavior; F is called *causal in input x* if (for all times $t \in \mathbb{N}$ and input histories $x, z \in \overrightarrow{I}$):

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

F is called *strongly causal* if (for all times $t \in \mathbb{N}$ and input histories $x, z \in \overrightarrow{I}$):

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

Causality indicates consistent time flow between input and output histories (for an extended discussion of causality see [1]).

Notation: Extension of predicates on infinite histories to finite ones. Throughout the paper, we use the following notation: Given a predicate

$$p: \overrightarrow{C} \to \mathbb{B}$$

on infinite histories, we extend it also to finite histories x of length t by the definition:

$$p(x) \equiv \exists\, x' \in \overrightarrow{C}, t \in \mathbb{N} : x = x'{\downarrow}t \wedge p(x')$$ ❑

In other words, assertion p(x) holds for a finite history x if there exists some infinite history x' for which predicate p holds and which is identical to x till time t. This notation is easily extended to n-ary predicates on histories.

Interface Assertions

The interface behavior of systems can be specified in a descriptive logical style using interface assertions.

Definition. Interface Assertion

Given a syntactic interface (I▶O) with a set I of typed input channels and a set O of typed output channels, an *interface assertion* is a formula in predicate logic with channel identifiers from I and O as free logical variables which denote streams of the respective types. ❑

We specify the behavior F_S for a system with name S with syntactic interface (I▶O) and an *interface assertion* Q by a scheme:

spec S
in I **out** O
Q

Q is an assertion containing the input and the output channels as free variables for channels. We also write q(x, y) with $x \in \overrightarrow{I}$ and $y \in \overrightarrow{O}$ for interface assertions. This is only another way to represent interface assertions which is equivalent to the formula $Q[x(x_1)/x_1, \ldots x(x_n)/x_n), y(y_1)/y_1, \ldots y(y_m)/y_m]$.

Definition. Meaning of Specifications and Interface Assertions

An interface behavior F fulfills the specification S with interface assertion q(x, y) if

$$\forall x \in \overrightarrow{I}, y \in \overrightarrow{O} : y \in F(x) \Rightarrow q(x, y)$$

S and q(x, y) are called *(strongly) realizable* if there exists a "realization" which is a strongly causal function f: $\overrightarrow{I} \to \overrightarrow{O}$ that fulfills S. ❑

The purpose of a specification and an interface assertion is to specify systems.

Composing Interfaces

Finally, we describe how to compose systems from subsystems described by their interface behavior. Syntactic interfaces (I_k▶O_k) with k = 1, 2 are called *composable*, if their channel types are consistent and $O_1 \cap O_2 = \varnothing, I_1 \cap O_1 = \varnothing, I_2 \cap O_2 = \varnothing$.

Definition. Composition of Systems – Glass Box View

Given for k = 1, 2 composable interface behaviors $F_k : (I_k \blacktriangleright O_k)$ with composable syntactic interfaces; let $I = I_1 \backslash O_2 \cup I_2 \backslash O_1$, $O = O_1 \cup O_2$ and $C = I_1 \cup I_2 \cup O_1 \cup O_2$; we define the composition $(F_1 \times F_2) : (I \blacktriangleright O)$ by

$$(F_1 \times F_2)(x) = \{y \in \overrightarrow{O} : \exists z \in \overrightarrow{C} : x = z|I \wedge y = z|O \wedge z|O_1 \in F_1(z|I_1) \wedge z|O_2 \in F_2(z|I_2)\}$$

where | denotes the usual restriction operator for mappings. ✠

In the glass box view the internal channels and their valuations are visible. In the black box view the internal channels are hidden. From the glass box view we can derive the black box view of composition.

Definition. Composition of Systems – Black Box View – Hiding internal channels

Given two composable interface behaviors $F_k : (I_k \blacktriangleright O_k)$ with k = 1, 2; let $I = I_1 \backslash O_2 \cup I_2 \backslash O_1$ and $O = O_1 \backslash I_2 \cup O_2 \backslash I_1$ and $C = I_1 \cup I_2 \cup O_1 \cup O_2$

$$(F_1 \otimes F_2)(x) = \{y \in \overrightarrow{O} : \exists z \in \overrightarrow{C} : y = z|O \wedge z \in (F_1 \times F_2)(x)\}$$

Shared channels in $(I_1 \cap O_2) \cup (I_2 \cap O_1)$ are hidden by this composition. ✠

Black box composition is commutative and associative as long as we compose only systems with disjoint sets of input channels.

A specification approach is called *modular* if specifications of composed systems can be constructed from the specification of their components. The property of modularity of composition of two causal interface specifications F_k, k = 1, 2, where at least one is strongly causal is as follows. Given system specifications by specifying assertions P_k:

spec F_1
in I_1 **out** O_1
P_1

spec F_2
in I_2 **out** O_2
P_2

We obtain the specification of the composed system $F_1 \otimes F_2$ as a result of the composition of the interface specification F_1 and F_2 as illustrated in Fig. 3: $L_1 \cup L_2$ denotes the set of shared channels.

spec $F_1 \otimes F_2$
in $I_1 \backslash L_2 \cup I_2 \backslash L_1$ **out** $O_1 \backslash L_1 \cup O_2 \backslash L_2$
$\exists L_1, L_2$: $P_1 \wedge P_2$

The specifying assertion of $F_1 \otimes F_2$ is composed in a modular way from the specifying assertions of its components by logical conjunction and existential quantification over streams denoting internal channels (Fig. 2).

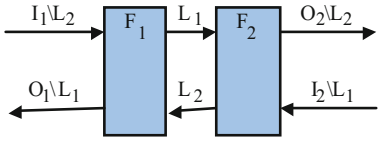

Fig. 2. Composition $F_1 \otimes F_2$

In a composed system, the internal channels are used for internal communication.

The composition of strongly causal behaviors yields strongly causal behaviors. The set of systems together with the introduced composition operators form an algebra. For properties of the resulting algebra, we refer to [1, 4]. Since the black box view hides internal communication over shared channels, the black box view provides an abstraction of the glass box composition.

Note that this form of composition works also for instances. Then, however, often it is helpful to use not channels identified by instance identifiers but to connect the channels of classes and to use the instance identifiers to address instances.

3 Specifying Contracts

Contracts are used in architectures (see [7, 9, 11–16]). In the following we show how to specify contracts.

3.1 Interface Assertions for Assumption/Commitment Contracts

Specifications in terms of assumptions and commitments for a system S with syntactic interface (I▶O) and with input histories $x \in \overrightarrow{I}$ and output histories $y \in \overrightarrow{O}$ are syntactically expressed by interface assertions asu(x, y) and cmt(x, y). We write A/C-contracts by the following specification pattern:

assume : $\quad \mathrm{asu}(x, y)$
commit : $\quad \mathrm{cmt}(x, y)$

with interface assertions asu(x, y) and cmt(x, y). In the following section we explain why, in general, in the assumption not only the input history occurs but also the output history y. We interpret this specification pattern as follows:

- Contracts as Context Constraints: the assumption asu(x, y) is a specifying assertion for the context with syntactic interface (I▶O)

Understanding the A/C-contract pattern as context constraints leads to the following meaning: if the input x to the system generated by the context on its input y, which is the system output, fulfills the interface assertion given by the assumption asu(x, y) then the system fulfills the promised assertion cmt(x, y). This leads to the specification:

$$\mathrm{asu}(x, y) \Rightarrow \mathrm{cmt}(x, y)$$

Assertion asu(x, y) is a specification indicating which inputs x are permitted to be generated by context E fulfilling the assumption given the output history y.

3.2 Contracts in Architectures

In this section, we discuss methodological applications of the A/C-pattern in system development with emphasis on system architecture design. We study contracts for

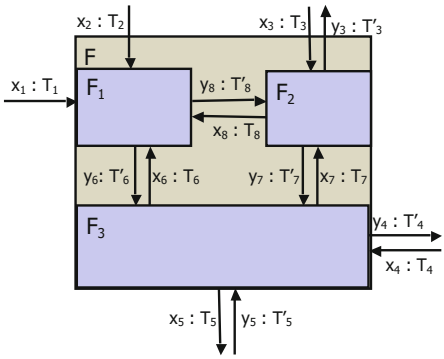

Fig. 3. Architecture of a system with interface behavior $F = F_1 \otimes F_2 \otimes F_3$

subsystems and their role in designing and reasoning about architectures and their relationship to the A/C-contract of the composite system. This provides a basis for a method for supporting steps in architecture design.

Architectures are blue prints to build and structure systems. Architectures contain descriptions of subsystems and specify how to compose the subsystems. In other words, architectures are described by the sets of subsystems where the subsystems are described by their syntactic interfaces and their interface behavior. Shared channels describe internal communication between the subsystems.

In the following we assume that each system used in an architecture as a component has a unique identifier k.

4 On Systems, Their Interfaces and Properties

In the following, we use the term system in a specific way. We address discrete systems, more precisely discrete real-time system models with input and output. For us, a system is an entity that shows some specific behavior by interacting with its operational context. A system has a boundary, which determines what is inside and what is outside the system. Inside the system there is an encapsulated internal structure. The set of actions and events that may occur in the interaction of the system with its operational context at its border determines the syntactic ("static") interface of the system. At its interface, a system shows some *interface behavior*.

From the behavioral point of view, we distinguish between

- the *syntactic interface* of a system that describes which actions may be executed at the interface and which kind of information is exchanged by these actions across the system border,
- the *semantic interface* (also called *interface behavior*) which describes the behavior evolving over the system border in terms of the specific information exchanged in the process of interaction by actions according to the syntactic interface.

For specifying predicates there are further properties that we expect. We require that system behaviors fulfill properties such as causality and realizability. However, not all interface assertions guarantee these properties (see [3, 5]).

4.1 About Architecture

Architecture of systems and also of software systems is about the structuring of systems. There are many different aspects of structuring of systems and therefore of architecture. Examples are functional architectures which structure systems in terms of their offered services (see [2]) – also called functional features. We speak of a *functional architecture* or of a *service feature architecture* (see [6]). Another very basic concept of architecture is the decomposition of a larger system into a number of subsystems that are composed and provide this way the behavior of the overall system. We speak of a *sub-system architecture*.

This shows that architecture is the structuring of a system into smaller elements, a description how these elements are connected and behave in relationship to each other. A key concept of architecture is the notion of element and interface. An interface shows at the border of a system how the system interacts with its operational context.

4.2 On the Essence of Architecture: Architecture Design is Architecture Specification

Architecture is not what is represented and finally implemented in code but a description of architectural structures and rules which are required by the design for implementations leading to code that is correct w.r.t. the specified architecture. The rules, structure, and therefore the principles of architecture usually cannot be reengineered from the code but provide an additional design frame that is documented in the architecture specification. An architecture design is the specification of the system's structures, rules, and principles.

Implemented systems realize architectures, more precisely architecture designs described by specifications. Architectures define the overall structure of systems. Consequently, architectures have to be specified. Designs of sub-system architectures are specifications of the sets of subsystems, relevant properties of their interfaces including their interface behavior, and the way the interfaces are connected. This defines the way the subsystems are composed following the design of an architecture in terms of their interfaces that follow the rules and principles of the architectural design.

4.3 Logical Sub-system Architectures

Logical sub-system architectures including service-oriented architectures are execution platform independent. They consist of the following ingredients

- A set of elements called sub-systems, each equipped with a set of interfaces
- A structure connecting these interfaces

This shows that a key issue in architectural design is the specification of interfaces including their interface behavior and the description of the architectural structure.

5 Interfaces and Their Composition

An interface is structured into a syntactic part, which describes the set of available activities on the system border, separated in activities of the context (which are input to the system) and activities of the system (which are output of the system). The syntactic part is the basis for the behavior part which describes the logic of behavior.

A system has a syntactic interface and an interface behavior. The interface can be formalized as we shown by sets of input and output channels and the relationship of their valuations. The interface of a system can be structured into a set of sub-interfaces that may serve as connectors to other sub-systems.

Two interfaces that fit syntactically together can be connected by a composition of the two systems over their interfaces, if their syntactic interfaces fit together (formally, what is an input channel of one of the interfaces is an output channel of the other system and vice versa).

A system has an interface the behavior of which is specified by an interface assertion L. If we want to use the system in context with a number of other systems we partition the syntactic interface into a number of sub-interfaces. Each sub-interface can be specified by an assertion L' for which we require

$$L \Rightarrow L'$$

In the following, we show how to deal with export, import and assumption/commitment interfaces.

5.1 Export Interfaces

We consider the following example illustrated by Fig. 4. We specify subsystem K1 as follows: y1, z2: {req}, x1, z1: D

$$L1 \equiv [\#z1 = \min(\#x1, \#z2) \wedge \forall d \in D : d\,\#z1 \leq d\,\#x1 \wedge y1 = z2]$$

We specify K2 in analogy as follows: z2, x2: {req}, z1, y2 : D

$$L2 \equiv [\#y2 = \min(\#z1, \#x2) \wedge \forall d \in D : d\,\#y2 \leq d\#z1 \wedge z2 = x2]$$

Composing the two components results into the following interface assertions

$$\begin{gathered}\#z1 = \min(\#x1, \#z2) \wedge \forall d \in D : d\#z1 \leq d\#x2 \\ \wedge\, \#y2 = \min(\#z1, \#x2) \wedge \forall d \in D : d\#y2 \leq d\#z1 \wedge y1 = z2 \wedge z2 = x2\end{gathered}$$

Hiding z1 and z2 by existential quantification we get

$$\#y2 = \min(\#x1, \#x2) \wedge \forall d : d\#y2 \leq d\#x2 \wedge y1 = x2$$

In this special case, the composed system fulfills the same assertion as the two sub-systems.

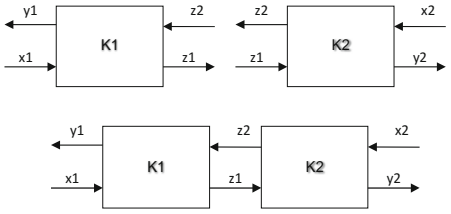

Fig. 4. Example of two sub-systems and their composition

For the hidden channels, we get the assertions

$$z2 = y1 \wedge z2 = x2$$

and

$$\#y2 = \min(\#z1, \#x2) \wedge \forall\, d : d\#y2 \leq d\#x1 \leq d\#x1 \wedge \#z1 = \min(\#x1, \#z2)$$

This assertion characterizes the properties of the internal channels of this little architecture.

However, we may also specify an assertion for the internal channels:

$$\exists\, x1, y1, x2, y2 : \#z1 = \min(\#x1, \#z2) \wedge \forall\, d \in D : d\#z1 \leq d\#x2$$
$$\wedge\, \#y2 = \min(\#z1, \#x2) \wedge \forall\, d \in D : d\#y2 \leq d\#z1 \wedge y1 = z2 \wedge z2 = x2$$

which can be simplified to

$$\exists\, x1, y2 : \#z1 = \min(\#x1, \#z2) \wedge \forall\, d \in D : d\#z1 \leq d\#z2$$
$$\wedge\, \#y2 = \min(\#z1, \#z2) \wedge \forall\, d \in D : d\#y2 \leq d\#z1$$

This condition is an assertion for the internal channels z1 and z2.

We call interfaces that describe the service offered of a system *export* interfaces. They describe the export a service without any assumptions about properties of their context.

5.2 Import Interfaces

If a component requires a certain interface to be able to fulfill its task this is expressed by an import interface. An import interface is a specification of a requested interface. Given a system with interface assertion G and the input assumption A we compose it with a system with interface B if the composition condition holds

$$B \Rightarrow A$$

A is called *assumption*. We get the interface specification of the composed component

$$(A \Rightarrow G) \wedge B$$

which is equivalent to

$$G \wedge B$$

Example: Consider the component K1 in Fig. 4 with the specification as before and the additional assumption

$$Asu_1 \equiv \forall t : \#z2{\downarrow}t \leq (\#z1{\downarrow}t) + 1$$

For the second component, we add the assertion Asu_1 to the specifying assertion of sub-system K2 leading to interface assertion

$$L2' = (L2 \wedge Asu_1)$$

We get obviously

$$L2' \Rightarrow Asu_1$$

Only if component K2 with specification $L2'$ fulfills the assumption Asu_1 we may compose the components and get the assertion

$$(L1 \wedge L2') \equiv (L1 \wedge L2 \wedge Asu_1)$$

We get assumptions as additional condition for the internal channels. To fulfill these assumptions, we have to add the following assumption to K2 – note z2 = y1 $\wedge$ z2 = x2

$$\forall t : \#x2{\downarrow}t \leq (\#z1{\downarrow}t) + 1$$

This demonstrates how assumptions are part of specifications and how they have to be distributed.

5.3 Assumption/Commitment Specifications

We may also work with interfaces that provide assumptions and commitments at the same time. Consider interface specifications with interface assertions L_1 and L_2 where

$$L_1 \Rightarrow (A_1 \Rightarrow C_1) \qquad L_2 \Rightarrow (A_2 \Rightarrow C_2)$$

and interface specifications with an assumption A_1 and a commitment C_1 and with assumption A_2 and commitment C_2 such that

$$(A_1 \Rightarrow C_1) \Rightarrow A_2$$
$$(A_2 \Rightarrow C_2) \Rightarrow A_1$$

In other words, the specifications fulfil mutually the resp. assumptions. Then the interface assertion

$$L_1 \wedge L_2$$

implies assumption A_1 as well as assumption A_2.

Example: We introduce an additional assumption Asu_2 and commitment Com_2 for the second component

$$Asu_2 \equiv \forall\, t : \#(z_1{\downarrow}t) \leq \#(z_2{\downarrow}t)$$
$$Com_2 \equiv \forall\, t : \#(z_1{\downarrow}t) \leq \#(z_2{\downarrow}t) + 1$$

and a commitment for the first component

$$Com_1 \equiv \forall\, t : \#z_1{\downarrow}t \leq \#z_2{\downarrow}t$$

and add Asu_2 to L1 and Asu_1 to L2:

$$L1' \equiv Asu_2 \wedge L1 \qquad\qquad L2' \equiv Asu_1 \wedge L2$$

We get obviously

$$L1' \Rightarrow Asu_2 \qquad\qquad L2' \Rightarrow Asu_1$$

and by composition a component that fulfills the specification

$$L1' \wedge L2'$$

This demonstrates how we compose systems specified by assumptions and commitments. ❑

We get a logical calculus of interface assertions for the composition of systems.

5.4 Using Different Types of Interfaces Side by Side

We distinguish the following three types of interfaces:

- *export interfaces*: they describe services offered by the system to its outside world
- *import interfaces*: they describe services required by the system from its outside world
- *assumption/commitment interfaces*: they describe assumptions about the behavior of the outside world and the commitment of the system under the condition that the assumption holds.

We consider the following cases:

- Connecting export and import interfaces: Given an export interface described by interface assertion P and an import interface described by interface assertion Q which fit together syntactically we speak of a sound connection if

$$P \Rightarrow Q$$

- Connecting two export interfaces: Given two export interfaces with interface assertions A_1 and A_2 that fit together syntactically then we speak of a sound connection annotated by (see Fig. 6)

$$A_1 \wedge A_2$$

- Connecting two assumption/commitment interfaces: Given two assumption/commitment interfaces with assumptions A_1 and A_2 and commitment P_1 and P_2 that fit together syntactically and where if

$$(A_2 \Rightarrow P_2) \Rightarrow A_1$$
$$(A_1 \Rightarrow P_1) \Rightarrow A_2$$

We speak of a sound connection; the connection is annotated by $P_1 \wedge P_2$.

The case of connecting an export interface with an assumption/commitment interface is considered as a special case of connecting two assumption/commitment interfaces where one assumption is true.

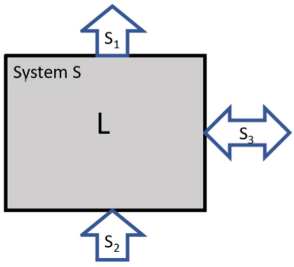

Fig. 5. System S with 3 sub-interfaces of different types

Similarly, the composition of an export interface with an input interface can be understood as a special case where one assumption is true (for the export interface) and one commitment is true (for import interface). This shows that the general case is the assumption of two assumption/commitment interfaces that cover all other cases as special cases.

A system has an interface that can be structured into a family of sub-interfaces each of which is determined by an interface specification. Now we show how these sub-interfaces can be combined into a comprehensive interface. Let us consider a simple example of a system with three sub-interfaces S_1, S_2, and S_3 as described in Fig. 5.

The interface specification of a sub-system defines the contract for the subsystem between its implementer and the architect that uses the subsystem. Each implemented subsystem may fulfill many contracts. The sub-interfaces shown in Fig. 4 describe

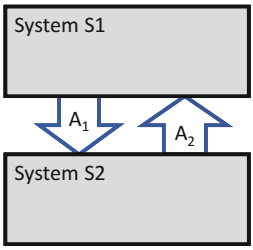

Fig. 6. Connecting subsystem S1 with subsystem S2 via their interfaces

three different types of interfaces. S_1 is the assertion specifying a service, offered by the system (called provided service). S_3 is a service that is structured into an assumption A_3 and a commitment C_3. S_2 is an interface assertion that specifies a service which is assumed to be provided called required service.

The three sub-services are put together into the over-service specified by the following interface specification in terms of interface assertions. This finally leads to a complete overall interface specification for the system S.

$$(S_2 \wedge A_3) \Rightarrow (C_3 \wedge S_1)$$

Here the assertion $S_2 \wedge A_3$ defines an assumption while the assertion defines a commitment $C_3 \wedge S_1$.

Channels allow us, in addition, the structuring of interfaces. Interfaces consist of channels where each channel has a data type indicating which data are communicated.

An important aspect in structuring interfaces is the separation of the set of channels of the interface into input and output channels. This has semantic consequences. We require causality which is a notion similar to monotonicity in a domain theoretic approach. Causality for an interface consisting of a set of input channels and output channels where the input and output are timed streams indicating the asymmetry between input and output. Causality basically says that the output produced till time t does only depend on input received before time t. The reverse does not hold. Input generated at time t can be arbitrary and does not have to depend on the output produced till time t.

6 Composition: Interfaces in Architectures

Given specifications of S1 and S2 by interface assertions A_1 and A_2 we define the interaction assertion

$$A_1 \wedge A_2$$

which specifies the interaction between the subsystems that are connected via their interfaces.

Another specification may give only the interaction assertion Q which describes the result $A_1 \wedge A_2$.

We may introduce a layering between subsystems, if we specify only one interface, by assertion P and do only specify the behavior of the other one by assertion A. For instance, for a layer in a layered architecture the interface looks as shown in Fig. 7.

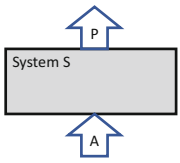

Fig. 7. Interface between two layers

6.1 Interaction Assertions

Given a set of systems with interface assertions we may compose them into an architecture, provided the semantic interfaces fit together. We call the architecture *well-formed*, if all assumptions are implied by the interface assertions the interfaces they are composed with.

For each pair of connected interfaces, we speak of a *connector*, we derive an *interaction assertion* which describes the properties of the data streams that are communicated over this connector. An example of an interaction assertion is given at the end of Sect. 5.1 specifying the properties of the internal channels z1 and z2 of the composition shown in Fig. 4.

6.2 Layered Architectures

Layered architectures have many advantages. In many applications, therefore layered architectures are applied. In a layered architecture as shown in Fig. 8 the key idea is that system S2 offers some service that does not include any assumptions about the way it is used. Therefore, we describe the service by some interface assertion A_2. The interface P of system S1 can be arbitrary. However, the specification of the interface Q of S1 reads as follows

$$Q = [A_1 \Rightarrow P]$$

and P is an interface specification for the reverse interface, then the interface can only be used in a meaningful way if the assumption is fulfilled by system S1. Note that S2 does not rely in any way on the behavior of S1 – it is supposed only to offer export interface A.

Figure 8 shows the composition of layer S2 providing service A_1 with system S1 requiring this service. We get

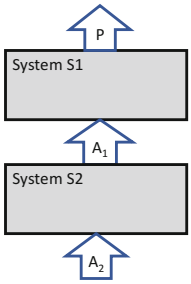

Fig. 8. Composition of two layers

$$(A_1 \Rightarrow P) \wedge (A_2 \Rightarrow A_1)$$

which hiding interface A_1 results in

$$A_2 \Rightarrow P$$

If we replace the component S2 with the interface assertion A_2 by the component S′ with interface assertion $A_2 \Rightarrow B$ where

$$B \Rightarrow A_1$$

then the arguments work as well. S2′ is a refinement of S2 and we get for the composition

$$(A_2 \Rightarrow B) \wedge (B \Rightarrow A_1)$$

which results the hiding interface B again into

$$A_2 \Rightarrow P$$

The sub-systems of a layered architecture are partitioned in layers. The set of layers is in a linear order and sub-systems of layer k are only connected to layer $k - 1$ or $k + 1$.

However, this definition is not sufficient. The key idea of a layered architecture is that layer k offers services to layer $k + 1$ but does not assume anything about layer $k + 1$. Layer k may use services offered by layer $k - 1$ but has to know nothing more about layer $k - 1$. In other terms, a layer imports a number of services (from layer $k - 1$) and exports a number of services (for layer $k + 1$). The only relationship between the layers is by the services that are exported to the next layer.

The idea of layered architecture thus is therefore not captured by data flow (by the idea that data may only flow from lower to higher layers or vice versa) nor by control flow (by the idea that calls may only be issued by higher to lower layers) but by the "design flow". Lower layers can be designed without any knowledge of the higher layers – only knowing the services that are requested at the higher layer.

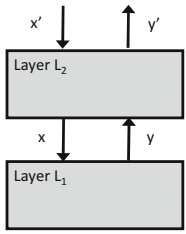

Fig. 9. Composition of the two layers L_1 and L_2

Example: Layered Architecture of a Question Answering System

We describe a simple layered architecture of two layers L_1 and L_2 as shown in Fig. 9. We start by defining two types of messages

Qst	the set of questions
Asw	the set of answers

Let the predicate

$$\text{asw} : \text{Qst} \times \text{Asw} \rightarrow \mathbb{B}$$

specify by asw(q, a) that a is an answer for question q. We define for $x \in (\text{Qst}^*)^\infty$ and $y \in (\text{Asw}^*)^\infty$ the two assertions

$$P = \forall k \in \mathbb{N} : k \leq \#\bar{x} \Rightarrow \text{asw}(\bar{x}(k), \bar{y}(k))$$
$$A = \forall t \in \mathbb{N} : \#x{\downarrow}t \leq 1 + \#y{\downarrow}t$$

P expresses that all questions are answered and A expresses that no further question is asked before all previous questions are answered. We specify the layer L_1 with input channel x and output channel y by

$$A \Rightarrow P$$

We can add a layer L_2 with input channel x′ and y and output channel x and y′ which controls x and satisfies this way the assumption. Let x′ be an infinite sequence of questions. A solution for the layer is given by the specification p(x, y, x′, y′) which holds if

$$y' = y$$

and (for all t)

$$x(t) = \begin{cases} \varepsilon & \text{otherwise} \\ x'(k) & \textit{if } \#(y{\downarrow}(t-1)) = k \wedge \#(x'{\downarrow}(t-1)) = k \end{cases}$$

The layer makes sure that the system gets only one question at a time. ❑

The example shows a classical assumption/commitment specification.

7 Concluding Remarks and Future Work

The purpose of this paper is to show that architecture can be specified by assertions similar to assertion logic in programs. This includes also on assertion calculus for architecture. The key here is a denotation for interaction in our case in terms of timed streams.

An interesting question is the logical flow of the assertions through an architecture. An example are assumptions and how they propagate through the architecture.

Acknowledgement. It is a pleasure to thank my colleagues for stimulating discussions.

References

1. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement. Monographs in Computer Science. Springer, New York (2001). https://doi.org/10.1007/978-1-4613-0091-5
2. Broy, M., Krüger, I., Meisinger, M.: A formal model of services. TOSEM - ACM Trans. Softw. Eng. Methodol. **16**, 5 (2007)
3. Broy, M.: Interaction and realizability. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 29–50. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69507-3_3
4. Broy, M.: A logical basis for component-oriented software and systems engineering. Comput. J. **53**(10), 1758–1782 (2010)
5. Broy, M.: Computability and realizability for interactive computations. Inf. Comput. **241**, 277–301 (2015)
6. Broy, M.: Multifunctional software systems: structured modeling and specification of functional requirements. Sci. Comput. Program. **75**, 1193–1214 (2010)
7. Broy, M.: Theory and Methodology of assumption/commitment based system interface specification and architectural contracts, to appear
8. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley, Boston (2010)
9. Derler, P., Lee, E.A., Tripakis, S., Törngren, M.: Cyber-physical system design contracts. In: Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems (ICCPS 2013), pp. 109–118. ACM, New York, (2013)
10. Henzinger, Th.A., Qadeer, S., Rajamani, S.K.: Decomposing refinement proofs using assume-guarantee reasoning. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD), pp. 245–252. IEEE Computer Society Press (2000)
11. Meyer, B.: Applying "Design by Contract". Computer **25**(10), 40–51 (1992). IEEE
12. Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming Dr. Frankenstein contract-based design for cyber-physical systems. Europ. J. Control **18**(3), 217–238 (2012)
13. Soderberg, A., Vedder, B.: Composable safety-critical systems based on pre-certified software components. In: 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 343–348, November 2012

14. Toerngren, M., Tripakis, S., Derler, P., Lee, E.A.: Design contracts for cyber-physical systems: making timing assumptions explicit. Technical report UCB/EECS-2012–191, EECS Department. University of California, Berkeley, August 2012
15. Tripakis, S., Lickly, B., Henzinger, Th.A., Lee, E.A.: A theory of synchronous relational interfaces. ACM Trans. Program. Lang. Syst. **33**(4), 14:1–14:41 (2011)
16. Westmann, J.: Specifying safety-critical heterogeneous systems using contracts theory. KTH, Industrial Engineering and Management. Doctoral thesis Stockholm, Sweden (2016)

http://www.springer.com/978-3-319-73116-2

SOFSEM 2018: Theory and Practice of Computer Science
44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29 - February 2, 2018, Proceedings
Tjoa, A.M.; Bellatreche, L.; Biffl, S.; van Leeuwen, J.; Wiedermann, J. (Eds.)
2018, XV, 698 p. 141 illus., Softcover
ISBN: 978-3-319-73116-2