The Leading Role of Software and Systems Architecture in the Age of Digitization

Manfred Broy

1 Introduction: Software Is Eating the World

The age of digitization can be characterized by the fact that digital technology is more and more closely integrated with business models. It provides key support of everyday life both in business and the private. In economy, more and more enterprises directly depend on their abilities to combine all the potentials of digital technology with their business models and their possibilities in the market generating customer experience and customer journeys. Here, it is absolutely clear that software is the most important part of digital technology since, in the end, it determines the functionality of digital systems and plays a decisive role (see [1]). For companies, it becomes a key capability to understand how software can be used to create innovative products, innovative processes, and instruments to reach the customer. As a consequence, more and more software platform companies are typical for software technology with platforms which can guarantee a short time to market, high quality, and low costs with respect to maintenance and evolution. For platform companies and generally for software-intensive systems, architectural issues become most decisive (see [2]).

Quite hard to understand, programming languages to a large extent are still in the infant state of computer hardware of the 1960s. Most programming languages including object-oriented languages, in principle, follow a conventional von Neumann-style programming paradigm where sequential execution without any considerations of real-time properties, concurrency, parallelism, and continuous interaction are the foundational concepts. This is surprising in a world which is full of network systems, software that is embedded in physical devices and interacts over

M. Broy (✉)

Institut für Informatik, Technische Universität München, München, Germany

e-mail: broy@in.tum.de

© The Author(s) 2018

V. Gruhn, R. Striemer (eds.), *The Essence of Software Engineering*,

https://doi.org/10.1007/978-3-319-73897-0_1

powerful communication networks. For such systems, paradigms are needed that provide support for distribution, interaction, parallelism, and real-time properties. However, we cannot wait until appropriate programming languages are available. Today, we have to build software systems using the programming technology which is around.

Nevertheless, there are options to deal with the important aspects at the level of architecture.

2 Structuring Architecture: Future Reference Architecture

Following the ideas of platform-dependent and platform-independent views onto software systems, a software architecture has to be structured basically into the following three views.

Functional Service Architecture Here we describe for software systems all the functionalities that are offered by the software systems to the outside world. Since today software systems like smartphones or cars offer a large number of services, it is important to structure those services into an architecture of services and to describe both their interface behavior which is the functionality provided to the outside their mutual relationships which define their functional dependencies, often called feature interactions. Therefore, we need description techniques capturing both interface behaviors, their structuring, and their dependencies (see [3]). In fact, specifications of interfaces might also rely on assumptions about their context.

Platform-Independent Component Architecture To realize software systems, we structure and decompose them into a set of components. From the viewpoint of architecture, these components have to be described by their roles, captured by the services they offer in terms of their interfaces including their interface behavior. The description of the components is independent of any execution platform, but, nevertheless, in the interfaces we have to be able to specify parallel behavior, real-time behavior, and also probabilistic behavior (see [4]).

Platform-Dependent Software and Hardware Architecture In the end, the abstract system described in terms of the functional service architecture and the platform-independent component architecture has to be implemented on a well-chosen hardware platform. This means that we have to define the deployment of the software components and their scheduling. Here, it is important that the components of the component-oriented architectures are independent deployable units that are only related to their environments only by their interfaces.

For all these approaches, we need an appropriate concept of interface, of interface behavior, and of system composition in terms of interfaces as well as techniques for their specification.

Finally, we have to be able to describe the context of software systems and assumptions about and its behavior. A well-understood way to describe the prop-

erties of the context that are relevant for the systems is assumptions in terms of interface assertions (see [5, 6]). Therefore, the core of the whole approach from its foundations is an approach to describe interfaces and the concept of composition including assumptions formulated by assertions (see [5]).

3 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 called component architecture. 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 input and output 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 (see [7]). However, not all interface assertions guarantee these properties.

3.1 About Architecture

Architecture of systems and also of software systems is about the structuring of systems. There are many different aspects of structuring systems and therefore of architecture. Examples are functional feature architectures which structure systems in terms of their offered services—also called functional features. We speak of a *functional architecture* or of a *service feature architecture* (see [3]). 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 *subsystem* or of a *component architecture* (see [5]).

This shows that architecture is the structuring of a system into smaller elements, a description of 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.

3.2 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 consists of 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 subsystem 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 as described by the design of an architecture in terms of their interfaces that follow the rules and principles of the architectural design.

3.3 Logical Subsystem Architectures

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

- A set of elements called subsystems or components, each equipped with a set of interfaces
- An architectural 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.

4 Interfaces Everywhere

As can be seen from the description of architectures and their concepts, interfaces are a key issue. In our terminology, interface comprises both a syntactic interface and an interface behavior. Interfaces are used to describe the functions of sub-components. Interfaces are used to describe the functionality of systems and their structuring into sub-services that again can be described by interfaces. This shows that the notion of an interface is essential in architectural design. Thus, interfaces occur everywhere, in the functional service architecture, in the logical subsystems or components architecture, and in the technical architecture as well.

There are several ways to describe interfaces. Specifying interfaces can be done by assertions. Assertions can be formulated quite formally or rather informally. We refer to a fully formalized notion of interface and interface behavior that is powerful enough to support specification and that is modular for parallel composition. Although fully formalized, it can nevertheless be used in an informal or as well as in a semiformal way.

4.1 *Property-Oriented Specification of Interfaces of Systems*

Since the components have to run in parallel and systems, in general, have to run in parallel, since they are distributed and they have to communicate and to interact in order to be part of networks, the interface concept has to be chosen appropriately. We chose an interface concept which is able to describe services that are interactive, run in parallel in a property-oriented way.

We denote a syntactic interface by $(I \blacktriangleright O)$, where I denotes the set of input channels of the interface and O denotes the set of output channels. A fully formalized instance of a theory of interfaces is found in the appendix. This interface theory includes the specification of real-time properties and can be extended to a probabilistic view (see [4]). It supports the description of probabilistic interface properties.

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 consists 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 .

Given a syntactic interface $(I \blacktriangleright O)$, we write $F: (I \blacktriangleright O)$ for an interface behavior F for the syntactic interface $(I \blacktriangleright O)$. An interface assertion Q for the syntactic interface $(I \blacktriangleright O)$ is used to specify properties for interface behaviors $F: (I \blacktriangleright O)$. We write $F \in Q$ to express that behavior F fulfills the specification Q (note that Q specifies a set of interface behaviors).

Given syntactic interface $(I \blacktriangleright O)$ we call the syntactic interface $(O \blacktriangleright I)$ then *inverse* interface. It is denoted by $(I \blacktriangleright O)^{-1}$.

4.2 Composition

Given two systems S_1 and S_2 with syntactic interfaces $(I_n \blacktriangleright O_n)$ and interface specifications Q_1 and Q_2 , and corresponding sub-interfaces $(I'_n \blacktriangleright O'_n)$ such that $(I'_1 \blacktriangleright O'_1) = (I'_2 \blacktriangleright O'_2)^{-1}$ and where the $(I_n \blacktriangleright O_n) \setminus (I'_n \blacktriangleright O'_n)$ are disjoint for $n = 1, 2$, we may compose S_1 and S_2 over their corresponding sub-interfaces and get a system with syntactic interface

$$(I_1 \blacktriangleright O_1) \setminus (I'_1 \blacktriangleright O'_1) \cup (I_2 \blacktriangleright O_2) \setminus (I'_2 \blacktriangleright O'_2)$$

and interface assertion (here $H = I'_1 \cup O'_1 \cup I'_2 \cup O'_2$ denotes the internal channels that connect the two systems and $G = (I_1 \cup O_1 \cup I_2 \cup O_2) \setminus H$ denotes the external channels of the composite system):

$$\exists H : Q_1 \wedge Q_2$$

This shows a modular composition of systems over their corresponding sub-interfaces. We can also specify the property of the connector H of the two systems:

$$\exists G \setminus H : Q_1 \wedge Q_2$$

This property of the connector can be used to formulate a watchdog like an assert statement in object-oriented programming.

This form of composition is beautiful in the sense that it reflects two major concepts of architecture, namely, parallel composition and channel hiding, by two key concepts of logic, namely, logical “and” reflecting parallel composition and existential quantification reflecting hiding. In fact, this relationship between architectural concepts and logic is not only of interest from an aesthetic point of view. It is also very helpful that we can map architectural concepts one-to-one onto such basic logical concepts.

4.3 Structuring Interfaces

Another important point is that we have to be able to structure interfaces. A system, in particular a large system, has a very large interface with a huge interface behavior, impossible to describe in a monolithic form. Therefore, it is important to introduce a concept of interface *decomposition*, which allows us to decompose and structure an interface into a number of services. Since these services may be not independent, we also have to be able to model the dependency between these services.

A syntactic interface can be structured into a set of sub-interfaces. Given a syntactic interface $(I \blacktriangleright O)$, we write $(I' \blacktriangleright O') \subseteq (I \blacktriangleright O)$ to express that $(I' \blacktriangleright O')$ is a sub-interface of $(I \blacktriangleright O)$. If there is a set of sub-interfaces $S_1, \dots, S_k$ that are disjoint with union $(I \blacktriangleright O)$, we speak of a decomposition of $(I \blacktriangleright O)$.

Given interface assertions $Q_1, \dots, Q_k$ for interfaces $S_1, \dots, S_k$, we speak of a *faithful* decomposition for interface $(I \blacktriangleright O)$ with Q if (see [3])

$$Q \equiv Q_1 \wedge \dots \wedge Q_k$$

Often such a faithful decomposition does not exist. Then we need an auxiliary syntactic interface H for the exchange of mode messages between the sub-interfaces and we have to include the rule of feature interactions in the assertions Q_i such that

$$Q \equiv \exists H : Q_1 \wedge \dots \wedge Q_k$$

Such a decomposition is both useful for the functional service architecture and, in particular, leads to functional service architecture but it is also useful for using systems as components in platform-independent component architecture where a component offers a set of sub-interfaces which are used to compose these components with others.

5 Composition: Interfaces in Architectures

Given specifications of IF1 and IF2 by interface assertions P_1 and P_2 , we define the interaction assertion

$$P_1 \wedge P_2$$

which specifies the interaction between the subsystems that are connected via their interfaces. Note that here we did not add the concept of channel hiding.

Fig. 1 Connecting subsystem S_1 with subsystem S_2 via their matching interfaces

5.1 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 (see [8]).

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 (see the internal channels in H of the composition shown in Fig. 1).

5.2 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 of composing systems via their interfaces:

- Connecting export and import interfaces: Given an export interface of one system described by interface assertion P and an import interface of another system 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 P and Q that fit together syntactically, we speak of a sound connection annotated by

$$P \wedge Q$$

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

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

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

we speak of a sound connection; the connection is annotated by the assertion

$$P_1 \wedge P_2$$

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.

5.3 Layered Architectures

A layer in a layered architecture (see [9]) the interface looks as shown in Fig. 2. Its interface specification reads as follows:

$$A \Rightarrow P$$

Layered architectures have many advantages. In many applications, therefore layered architectures are used.

Fig. 2 Interface of a layer

Fig. 3 Composition of two layers

In a layered architecture as shown in Fig. 3, 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 3 shows the composition of layer $S2$ providing service A_1 with system $S1$ requiring this service. We get

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

which hiding interface A_1 results in the interface assertion

$$A_2 \Rightarrow P$$

If we replace the component $S2$ with the interface assertion A_2 by the component $S2'$ 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 subsystems of a layered architecture are partitioned in layers. The set of layers is in a linear order and subsystems 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.

6 On the Asset of Foundations

A powerful approach to architectural design needs a formal foundation provided by a model for the types of interfaces that are used, which has all the properties that are needed for designing systems. These are in particular abstraction, modularity of composition, and property-oriented specification. This way, we are able to understand all of the principal properties of the approach.

It is important, however, that we are not forced always to describe architectures and their elements such as functional services and components in full formality. It is important that we offer an approach which gives the freedom to the developer to be as formal as needed.

6.1 *Not Formal Methods but Formal Foundation*

Our introduced concept to deal with architectures provides a purely functional and logical approach to architectures. We considered the value of this approach not so much in the fact that we can use it in the sense of formal methods by giving formal specifications of architectures and computing the results of compositions and even doing computer-based formal verification. This is certainly one way to make use of it. But what we consider much more important is to provide a scientific foundation for forming architectures. What we get is a complete understanding of what the properties of architectures are that are relevant and of the logical properties we can

express by that. So, we get a scientific underpinning of the notion of an architecture which can be used as a guideline in many more practical approaches, for instance, when trying to understand what approaches like UML and SysML are trying to do. This architectural logical framework gives a comprehensive and complete scientific foundation which allows us to justify all kinds of practical issues like diagrams to describe architecture's rules, to do verification and analysis with architectures as well as the verification of design patterns or the description of architectural styles like layered architectures. What we get is a foundation of architectures.

Now this foundation follows the principle of Occam's razor. It keeps everything as simple as possible. As can be seen from the appendix, the whole theory to describe architectures can be written down with only a few pages. It does not use very difficult mathematics. We have taught this approach of architectures to engineers in many different companies, and they never faced difficulties in understanding the whole approach and understanding what it gives to them.

6.2 Flexibility and Universality of the Presented Approach

The approach is powerful enough to describe not only the classical components of digital systems and their interfaces. It can also describe all kinds of particular additional elements such as protocols, sensors, and actuators, man-machine interaction, and many more aspects. It finally leads into a universal modeling instrument. When being interested in adapting the whole approach to the physical, it is possible to generalize it to physical systems. Then, in addition to the discrete interface concepts, we have to introduce continuous interface concepts where we replace discrete streams by continuous ones generalizing the whole approach in the direction of control theory.

6.3 System Components as Schedulable and Deployable Units

The notion of component is described in detail in Szyperski's work (see [10]). What he considered important is a component that is independently deployable and independently schedulable. This should also apply to our notion of components.

In principle, we could also see physical units, in particular cyber-physical systems as components, but when reduced in the view to study only software components, we stick to the concept of Szyperski that a component should be independently deployable and independently schedulable. In particular, components should be designed in the way that they run in parallel and can be connected also by real-time properties over their interfaces.

6.4 Modularity

A key idea in the approach of components is the idea that components can be described to the outside world only by their interfaces. Interfaces are as described highly parallel, allowing the description of the real-time properties, and supporting the formulation of components contracts.

It is decisive that the specification of components can be exclusively done by describing their interface properties following the idea of Jeff Bezos of a truly service-oriented architecture. This also supports the classical idea of modularity. In this approach, a component concept is modular if we can derive all the relevant interface properties of a composed component by the specifications of their components given the architecture of the components.

6.5 Strict Property Orientation: Architecture Designs by Specifications

In a strict property orientation, we describe interfaces by property-oriented specifications. When we compose interfaces, we get communication links for those interfaces for which we can describe the histories of the data streams communication over those communication links again in terms of property-oriented specifications. As a result, we get a perfect property-oriented view onto architectures (see [6]).

An additional issue is the fact that often in systems and their interfaces we need assumptions about their context that are required with respect to be able to guarantee certain commitments. Other assumptions can be seen purely as properties and so finally we get also a property-oriented view onto assumptions.

When designing architecture following our lines of thought, we describe the components of systems, their interfaces, and the interface behavior in terms of their properties by interface assertions. These properties are specifications. Implementation of architecture may introduce additional properties beyond those required in the architectural design.

6.6 Real Time and Probability: Functional Quality Properties

There is a long debate in the literature about functional and nonfunctional properties. However, there is not a clear definition of what functional properties are and what nonfunctional properties are. Following our concept, we call all properties functional that can be expressed in terms of interface properties.

If we introduce the notion of interfaces with talks about real time, then real-time properties are functional properties and this is, in contrast to IEEE standards, the

more adequate view. If we are interested in using functions where real time is an issue, it is essential that real-time properties are part of our functional requirements.

However, we could go even one step further. Based on the concept of interfaces as we have introduced it, it is, at least from a conceptual point of view, an easy step to generalize interfaces to probabilistic properties. Formally, our interfaces are described in terms of functions that map real-time input data streams onto a set of allowed real-time output data streams. It is a straightforward idea to introduce a probability distribution over the set of real-time data output streams (see [4]). This way, we can assign a probability to sets of behaviors. Having done so, we are even able to specify concepts like reliability, availability, safety, and many more (see [11]). A large amount of the classical quality properties that are considered in the old style of looking at systems as nonfunctional properties are then functional properties, and this is correct if we are interested in the reliability of a system. Then, it is absolutely clear that we understand reliability as a required functional property.

7 Concluding Remarks

Following the idea as introduced and explained, we get an extended view onto architecture of systems. Looking at the architecture of systems, we see all the required things that are necessary. We see the functional service architecture with the set of services and functions offered to the outside; we see the relationship in terms of feature interactions between those functions which give a comprehensive structure description of the functionality of a system including functional properties such as real time and also probabilistic properties such as reliability or safety if needed.

It uses the same techniques to describe the decomposition of a system into a set of components that are connected over their interfaces and provide the required functionality.

It is a straightforward step to introduce also concepts of classes and instantiations into the approach being able to define and describe dynamic architectures, and another extension would be to go into continuous data streams and also to introduce control theory. Finally, we get along that lines a comprehensive foundational model of architecture and system behavior in terms of interaction which also to do a comprehensive documentation of state-of-the-art software systems and also of cyber-physical systems.

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 [12]). 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 subsys-

tem 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.

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.

Appendix: A Formal Model of Interfaces

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

Data Models

System exchange messages. Messages are exchange 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.

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.

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 [13] 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. $(I \blacktriangleright O)$ denotes this *syntactic interface*. Figure 4 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. We work with streams that include discrete timing information. Such streams represent histories of

Fig. 4 Graphical representation of a system F as a data flow node with its syntactic interface consisting of the input channels $x_1, \dots, x_n$ of types $S_1, \dots, S_n$ and the output channels $y_1, \dots, y_m$ of types $T_1, \dots, T_m$, resp.

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 IN^+ to number the time intervals.

Definition Timed Streams

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

$$s : \text{IN}^+ \rightarrow M^*$$

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

A *channel history* (also called *channel valuation*) 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

$$x : C \rightarrow (\text{IN}^+ \rightarrow M^*)$$

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

$$x : C \rightarrow (\{1, \dots, t\} \rightarrow M^*)$$

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

As for streams, for every history $z \in \vec{C}$ and every time $t \in \text{IN}$, 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 $C \rightarrow (\{1, \dots, t\} \rightarrow \text{IM}^*)$. $z \downarrow 0$ denotes the history with the empty sequence associated with all its channels.

Interface Behavior

For a given syntactic interface $(I \blacktriangleright O)$, a relation that relates the input histories in $\vec{I}$ with output histories in $\vec{O}$ defines its behavior. It is called *system interface behavior* (see [2]). 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

$$F : \vec{I} \rightarrow \wp(\vec{O})$$

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

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

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

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

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

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 $(I \blacktriangleright O)$ 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. $\square$

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

$$\frac{\text{spec } S}{\frac{\text{in } I}{\frac{\text{out } O}{Q}}}$$

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

Definition Meaning of Specifications and Interface Assertions

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

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

S and $q(x, y)$ are called (*strongly*) *realizable* if there exists a “realization” which is a strongly causal function $f : \vec{T} \rightarrow \vec{O}$ that fulfills S . $\square$

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 $(I_k \blacktriangleright O_k)$ with $k = 1, 2$ are called *composable*, if their channel types are consistent and $O_1 \cap O_2 = \emptyset$, $I_1 \cap O_1 = \emptyset$, $I_2 \cap O_2 = \emptyset$.

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 $I = I_1 \setminus O_2 \cup I_2 \setminus O_1$, $O = O_1 \cup O_2$, 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

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

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 $I = I_1 \setminus O_2 \cup I_2 \setminus O_1$ and $O = O_1 \setminus I_2 \cup O_2 \setminus I_1$ and $C = I_1 \cup I_2 \cup O_1 \cup O_2$

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

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 :

$$\begin{array}{c} \text{spec } F_1 \\ \hline \text{in } I_1 \\ \text{out } O_1 \\ \hline P_1 \end{array}$$

Fig. 5 Composition $F_1 \otimes F_2$

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. 5; $L_1 \cup L_2$ denotes the set of shared channels:

spec $F_1 \otimes F_2$

in $I_1 \setminus L_2 \cup I_2 \setminus L_1$

out $O_1 \setminus L_1 \cup O_2 \setminus L_2$

$\exists L_1, L_2: P_1 \wedge P_2$

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.

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, 5]. 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.

Specifying Contracts

Contracts are used in architectures (see [4, 6, 9, 10, 14–17]). In the following, we show how to specify contracts.

Interface Assertions for Assumption/Commitment Contracts

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

assume : $\text{asu}(x, y)$

commit : $\text{cmt}(x, y)$

with interface assertions $\text{asu}(x, y)$ and $\text{cmt}(x, y)$. 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 $\text{asu}(x, y)$ is a specifying assertion for the context with syntactic interface $(I \blacktriangleright O)$.

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 $\text{asu}(x, y)$, then the system fulfills the promised assertion $\text{cmt}(x, y)$. This leads to the specification

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

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

Contracts in Architectures

Architectures are blue prints to build and structure systems (a simple example is shown in Fig. 6). 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.

We assume that each system used in an architecture as a component has a unique identifier k .

We get a logical calculus of interface assertions for the composition of systems. Each system is specified by a contract describing its interface behavior in terms of an interface assertion, possibly structured into sub-interfaces and into assumptions and commitments.

Fig. 6 Architecture of a system with interface behavior $F = F_1 \otimes F_2 \otimes F_3$

References

1. Andreessen, M.: Why Software Is Eating The World. <http://www.wsj.com/articles/SB10001424053111903480904576512250915629460>
2. Geisberger, E., Broy, M.: agendaCPS – Integrierte Forschungsagenda Cyber-Physical Systems (acatech STUDIE). Springer, Heidelberg (2012)
3. Broy, M.: Multifunctional software systems: structured modeling and specification of functional requirements. *Sci. Comput. Program.* **75**, 1193–1214 (2010)
4. Neubeck, P.: A probabilistic theory of interactive systems. Dissertation, Technische Universität München (2012)
5. Broy, M.: A logical basis for component-oriented software and systems engineering. *Comput. J.* **53**(10), 1758–1782 (2010)
6. Broy, M.: A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views. *Softw. Syst. Model.* (2017)
7. Broy, M.: Interaction and realizability, In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plasil, F. (eds.) *SOFSEM 2007: Theory and Practice of Computer Science*, Lecture Notes in Computer Science 4362, pp. 29–50. Springer (2007)
8. Broy, M.: Theory and methodology of assumption/commitment based system interface specification and architectural contracts. *Formal Methods Syst. Design* **52**(1), 33–87 (2018)
9. Herzberg, D., Broy, M.: Modeling layered distributed communication systems. *Form. Asp. Comput.* **17**(1), 1–18 (2005)
10. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*, 2nd edn. Addison-Wesley Professional (2002)
11. Broy, M.: System behaviour models with discrete and dense time. In: Chakraborty, S., Eberspächer, J. (eds.) *Advances in Real-Time Systems*, pp. 3–25. Springer, Berlin (2012)
12. 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)
13. Broy, M., Stølen, K.: *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer (2001)
14. 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 '13)*, pp. 109–118. ACM, New York, NY (2013)

15. 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 (2012)
16. Tripakis, S., Lickly, B., Henzinger, T.A., Lee, E.A.: A theory of synchronous relational interfaces. *ACM Trans. Program. Lang. Syst.* **33**(4), 14:1–14:41 (2011)
17. Westmann, J.: Specifying safety-critical heterogeneous systems using contracts theory. KTH, Industrial Engineering and Management. Doctoral Thesis Stockholm, Sweden 2016

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.