Composition and architecture of IT-systems

Johannes Reich[1 [0000−0002−0378−853X]]

SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germary
johannes.reich@sap.com

Abstract. In this article, I investigate the role of composition for the architecture of IT-systems as it results from their interactions. This gives rise to the notion of the interface of a system, as the set of all its composition-related information, as well as the notion of a component as a system constructed for a particular composition and therefore equipped with a well defined interface.
I distinguish between two main composition classes. On the one hand we have functional composition, exposing the system function as a whole and establishing a directed "is-part-of" relationship between a supersystem and its respective subsystems ("vertical relation"). On the other hand, the composition of system projections, named "roles", leads to protocols which are closely related to games in economics ("horizontal relation").
Several well-known component models are discussed in terms of their support for these composition classes, including distributed objects, SOA, REST and client-server. I also discuss some architecture reference models such as the OSI model, the LCIM and RAMI4.0. Finally, I sketch the idea of an interaction-oriented architecture as an answer to the tension between the need to describe interactive IT-systems with executable functions and the non-deterministic character of their horizontal interactions.

1 Introduction

The forword of [6] opens with the question, *"Why can we get the drawings for a house that's several decades old, but we are unable to see the architecture of software written last year?"* In my impression there is still some truth in this sentence and this article's contribution is to provide some understanding for this state of affairs and how to improve it.

It is consensus in computer science to understand the structure of a system in terms of the compositional relationships of its interacting components as the concept-defining aspect of its architecture [28,18,39,6]. In this sense, the structure of a system – or its architecture – is not left to the arbitrary consideration of some engineer or even some power point artist, but is inherent in the system as a whole. In my understanding structure is actually the concept with which we can give "wholeness" an intelligible meaning [29]. Thus, the concept of composition of IT systems not only determines a meaningful framework for the discourse of their interoperability [34], but, by definition, also determines the discourse of their architecture.

In this article, I work out this insight by first introducing concept of compositionality in section 2 to derive three quick benefits. In section 3, I apply the composition concept to discrete systems. I demonstrate that hierarchical system relations stem from homogenous composition where (sub)-systems compose to (super)-systems and non-hierarchical system relations stem from nonhomogeneous composition where system projections, or roles, compose to protocols. Additionally, in section 4 I propose an interaction-oriented architecture to render interactive IT-systems robust against changes in their interactions. In section 5 and 6 I discuss several component models and so called reference architectures. I close the article with a short summary and outlook.

2 Compositionality

Mathematically composition[1] means making one out of two or more mathematical objects with the help of a mathematical mapping. For example, we can take two functions $f, g : \mathbb{N} \to \mathbb{N}$, which map the natural numbers onto themselves, and, with the help of the concatenation operator $\circ$, we can define a composed function $h = f \circ g$ by $h(n) = f(g(n))$.

If we apply this notion to interacting systems, which we denote by $\mathcal{S}_1, \ldots, \mathcal{S}_n$, then, regardless of the concrete representation of these systems, we can define their composition into a supersystem by means of a corresponding composition operator $comp_{\mathcal{S}}$ as a partial function[2] for systems:

$$\mathcal{S}_{tot} = comp_{\mathcal{S}}(\mathcal{S}_1, \ldots, \mathcal{S}_n). \tag{1}$$

The first benefit we can derive from this definition is that we can now classify the properties of the supersystem into those that arise from the same properties of the subsystems and those that arise from other properties of the subsystems:

Definition 1. *A property $\alpha : S \to A$ of a system $\mathcal{S} \in S$ is a partial function which attributes values of some attribute set A to a system $\mathcal{S} \in S$. I call a property α of a composed system $\mathcal{S}_{tot}$ "(homogeneous) compositional" with respect to the composition $comp_{\mathcal{S}}$, if there exists an operator $comp_\alpha$ such that $\alpha(S_{tot})$ results as $comp_\alpha(\alpha(\mathcal{S}_1), \ldots, \alpha(\mathcal{S}_n))$, thus, it holds:*

$$\alpha(comp_{\mathcal{S}}(\mathcal{S}_1, \ldots, \mathcal{S}_n)) = comp_\alpha(\alpha(\mathcal{S}_1), \ldots, \alpha(\mathcal{S}_n)) \tag{2}$$

Otherwise I call this property "emergent".

[1] It was mainly Arend Rensink in his talk "Compositionality huh?" at the Dagstuhl Workshop "Divide and Conquer: the Quest for Compositional Design and Analysis" in December 2012, who inspired me to these thoughts and to distinguish between composition of systems and the property of being compositional for the properties of the systems.

[2] "Partial" means that this function is not defined for all possible systems, i.e. not every system is suitable for every composition

For mathematical structures α is a homomorphism. Emergent properties may result also from other properties α_i of the parts if $\alpha\left(comp_{\mathcal{S}}(\mathcal{S}_1,\ldots,\mathcal{S}_n)\right) = comp_{\alpha}\left(\alpha_1(\mathcal{S}_1),\ldots,\alpha_n(\mathcal{S}_n)\right)$.

A simple example of a homogeneous compositional property of a physical systems is their mass: The mass of a total system is equal to the sum of the masses of the individual systems. A simple example of an emergent property of a physical system is the resonance frequency of an oscillating circuit consisting of a coil and a capacitor. It results from their inductance and capacity.

2.1 Computable functionality

One of the most important properties of IT-systems is the computability of their system function. Based on considerations of Kurt Gödel, Stephen Kleene was able to show [20] that this property is indeed compositional by construction. Starting from given elementary operations (successor, constant and identity), all further computable operations on natural numbers can be constructed by the following 3 rules (let F_n be the set of all functions on the natural numbers with arity n):

1. *Comp:* Be $g_1,\ldots,g_n \in F_m$ computable and $h \in F_n$ computable, then $f = h(g_1,\ldots,g_n)$ is computable.
2. *PrimRec:* Are $g \in F_n$ and $h \in F_{n+2}$ both computable and $a \in \mathbb{N}^n$, $b \in \mathbb{N}$ then also the function $f \in F_{n+1}$ given by $f(a,0) = g(a)$ and $f(a,b+1) = h(a,b,f(a,b))$ is computable .
3. *μ-Rec:* Be $g \in F_{n+1}$ computable and $\forall a \exists b$ such that $g(a,b) = 0$ and the μ-Operation $\mu_b[g(a,b) = 0]$ is defined as the smallest b with $g(a,b) = 0$. Then $f(a) = \mu_b[g(a,b) = 0]$ is computable.

Comp states that given computable functions, their successive as well as their parallel application is in turn a computable function. *PrimRec* defines simple recursion, where the function value is computed by applying a given computable function successively a predefined number of times. In imperative programming languages it can be found as FOR loop construct as well as operation construct for one step recursion. The third rule *μ-Rec* states that a recursive calculation can also exist as an iterative solution of a computable problem, in the case of the natural numbers as a determination of roots. In imperative programming languages, this corresponds to the WHILE loop construct.

2.2 The notion of interface and component

The second benefit we can derive from our definition of composition is a clear definition of the notion of interface and component. We are now in the need for a term that comprises everything a composition operator needs to know about a system. [42,9] use the term *interface* for this, a suggestion I am happy to endorse. This makes the question of what an interface actually is decidable. The recipe is that the claim for a mathematical object to be an interface must be

substantiated first by providing a system model, and secondly, by presenting a composition operator so that thirdly, it becomes comprehensible which parts of a system model belong to the system's interface.

A component becomes a system that is intended for a particular composition and therefore has a correspondingly well-defined interface that, by definition, express the intended composition. If we additionally require the composition to be somehow "simple", then a component model also introduce a complexity boundary. For example, usually engineers try to avoid mutually recursive operation calls between components and thereby, in fact, avoid general recursive composition – which is much more complex then parallel or sequential composition or the one-step recursion of a single operation (see below).

2.3 Substitutability and compatibility

The third benefit of our composition concept comes from the possibility to define substitutability and to distinguish downward from upward compatibility.

For this purpose we first consider two systems $\mathcal{A}$ and $\mathcal{B}$ which can be composed by a composition operator C into the supersystem $\mathcal{S} = C(\mathcal{A}, \mathcal{B})$. System $\mathcal{A}$ can certainly be substituted by another system $\mathcal{A}'$ in this composition if $\mathcal{S} = C(\mathcal{A}', \mathcal{B})$ also holds. Please note that substitutability may relate seemingly unrelated things, such as raspberry juice and cod liver oil, both of which are suitable fuels for a Fliwatüt [21].

Now, let us assume that $\mathcal{A}'$ additionally extends $\mathcal{A}$ in a sense to be concretised, notated as $\mathcal{A} \sqsubseteq \mathcal{A}'$, and another operator C' composes $\mathcal{A}'$ and again $\mathcal{B}$ into the supersystem $\mathcal{S}' = C'(\mathcal{A}', \mathcal{B})$, so that the property of extension is preserved, thus also $\mathcal{S} \sqsubseteq \mathcal{S}'$ holds. Then we speak of *"compatibility"* under the following conditions: if C' applied to the old system $\mathcal{A}$ still produces the old system $\mathcal{S} = C'(\mathcal{A}, \mathcal{B})$, then $\mathcal{A}$ behaves *"upward compatible"* in the context of the composition C'. And if $\mathcal{A}'$ can replace $\mathcal{A}$ in the old context C in the sense that $\mathcal{S} = C(\mathcal{A}', \mathcal{B})$, we say that $\mathcal{A}'$ behaves *"downward compatible"* in context C.

A simple example is a red light-emitting diode (LED) $\mathcal{A}$ and a socket $\mathcal{B}$ composing to a red lamp $\mathcal{S} = C(\mathcal{A}, \mathcal{B})$. A new LED A' that adds to the old one the ability to glow green when the current is reversed is called downward compatible with $\mathcal{A}$ if it fits into the same socket and makes the lamp behave as with the original LED.

Actually, of course, the bi-colour LED is intended for a new lamp circuit that supports this bi-colour. Since the socket remained unchanged, inserting the old LED into the new lamp results in the old, single-colour red lamp despite the new circuit. The old LED behaves upwards compatible in the new context.

3 The system model and different composition classes

I focus on discrete systems which map their input $in(t)$ and internal state $q(t)$ at time t in one time step $t \to t'$ onto their output $out(t')$ and (new) internal state $q(t')$ in one step: $(q(t'), out(t')) = f(q(t), in(t))$.

In fact, there are many formalism to describe discrete systems and their behaviour [15,3,32]. I use I/O-transition systems whose transitions map a start state p and an input character i to a target state q and an output character o. In my opinion, they are particularly suitable because of their possible coupling by some "exchanged" character, representing information transport while their transitions represent information processing[3]. They can also be used to show the relationship of protocols to games [30,32] and to deduce an interesting model for the "meaning" of the exchanged characters [33].

So, with the convention that ϵ is the empty character and for any alphabet A, $A^\epsilon = A \cup \{\epsilon\}$, we describe the behaviour of such a discrete system with an I/O-transition system (I/O-TS):

Definition 2. *An I/O-transition system $\mathcal{A}$ is defined as $\mathcal{A} = (Q, I, O, \Delta)_\mathcal{A}$. $Q_\mathcal{A}$, $I_\mathcal{A}$ and $O_\mathcal{A}$ are alphabets, whereas only Q has to be non-empty. $q_0 \in Q$ is the initial value and $\Delta_\mathcal{A} \subseteq I^\epsilon \times O^\epsilon \times Q \times Q$ is the transition relation.*

To fully represent a system function, the I/O-TS must be deterministic and I must not be empty.

3.1 Homogenous or functional composition

Composing systems homogeneously means to compose (sub-) systems to (super-) systems or the corresponding deterministic sub-I/O-TSs to a deterministic super-I/O-TS. According to our schema to compose computable functionality, we can distinguish between simple composition, comprising serial and parallel composition, primary recursion and μ-recursion.

In parallel composition two systems work in parallel on a single input. We therefore combine two deterministic I/O-TSs $\mathcal{A}_{tot} = \mathcal{A}_1 | \mathcal{A}_2$ working on the same input alphabet $I_{tot} = I_1 = I_2$ and we get: $Q_{tot} = Q_1 \times Q_2$, $O_{tot} = O_1 \times O_2$ and $(i, (o_1, o_2), (p_1, p_2), (q_1, q_2)) \in \Delta_{tot}$ iff $(i, o_1, p_1, q_1) \in \Delta_1$ and $(i, o_2, p_2, q_2) \in \Delta_2$.

In sequential composition of two systems, the second of the corresponding deterministic I/O-TSs $\mathcal{A}_2$ process the output of the first I/O-TS $\mathcal{A}_1$ and it must hold $O_1 \subseteq I_2$. I then define $\mathcal{A}_{tot} = \mathcal{A}_2 \circ \mathcal{A}_1$ such that $I_{tot} = I_1$, $O_{tot} = O_2$, $Q_{tot} = Q_1 \times Q_2$ and $(i, o, (p_1, p_2), (q_1, q_2)) \in \Delta_{tot}$ iff there exists a character $c \in O_1$ such that $(i, c, p_1, q_1) \in \Delta_1$ and $(c, o, p_2, q_2) \in \Delta_2$.

For primary recursive composition of a system $\mathcal{A}$ to a recursive system $\mathcal{A}_{tot}$, the input alphabet $I_\mathcal{A}$ of the corresponding I/O-TS must consist of three components $I_\mathcal{A} = I_1 \times I_2 \times I_3$ with $I_2 = \mathbb{N}$ and $O_\mathcal{A} \subseteq I_3$. The recursion operator creates the supersystem with $I_\mathcal{S} = I_1 \times I_2$, $O_\mathcal{S} = O_\mathcal{A}$, $Q_\mathcal{S} = Q_\mathcal{A}$ and the transi-

[3] There are many interaction models based on transition systems with named transitions, where the coupling of different systems is achieved by identically (or complementarily) named transitions, e.g. [14,24].

tion relation $\Delta_{\mathcal{S}}$ with

$$\begin{aligned}((a,n),y_n,p,q)\in\Delta_{\mathcal{S}} \;\Leftrightarrow\; &\text{there exist characters } y_{n-1},\ldots,y_0\in O_{\mathcal{A}} \text{ such that}\\ &((a,n-1,y_{n-1}),y_n,p,q)\in\Delta_{\mathcal{A}}\\ &\text{and } ((a,n-2,y_{n-2}),y_{n-1},p,q)\in\Delta_{\mathcal{A}}\\ &\vdots \qquad\qquad \vdots\\ &\text{and } ((a,1,y_0),y_1,p,q)\in\Delta_{\mathcal{A}}\\ &\text{and } ((a,0,*),y_0,p,q)\in\Delta_{\mathcal{A}}\end{aligned}$$

μ-recursion works similar.

As every system comes with its own time scale, the time scale of the composed system potentially differs from the time scale of its subsystems. Related to that, the I/O-states that created the connections for sequential composition, and the internal counting states, necessary for recursive composition, vanish in the composed supersystem – in line with the notion that information is only transported between systems and is processed within systems. For our corresponding I/O-TS this means that the composed I/O-TS does not contain the respective alphabet components any longer. Thus, by homogenous composition, we "hide" state which is relevant only for the composition itself.

To illustrate functional composition, Fig 1 shows a simple system composition where three systems $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ compose to a supersystem $\mathcal{S}$ with system function $f_{\mathcal{S}}(x) = 2x+5$. System $\mathcal{S}_2$ contributes its system function $f_{\mathcal{S}_2}(x) = 2x$, $\mathcal{S}_3$ contributes $f_{\mathcal{S}_3}(x) = x+5$, and $\mathcal{S}_1$ coordinates system $\mathcal{S}_2$ and $\mathcal{S}_3$ in three steps in a non-trivial recursive manner. As we can easily see, there is no interaction between the subsystems and their supersystem, but instead, the supersystem is created by the deterministic interactions of the subsystems, and its interface represents the function $f_{\mathcal{S}}(x) = 2x + 5$.

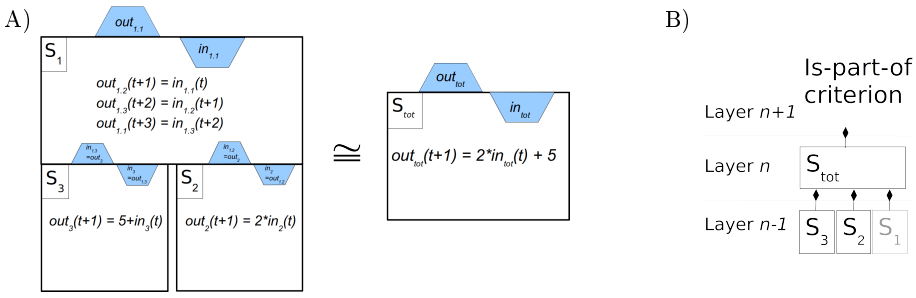

Fig. 1. Three systems $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ compose to a supersystem $\mathcal{S}$ with the overall (super)system function $f_{\mathcal{S}}(x) = 2x + 5$. The right part shows the ordering according to their "is-part-of"-relation, represented by a filled diamond and a solid line. There is no information flow between the layers.

This latter hierarchy is used in imperative programs and the object oriented world with their method-construct. A method represents a function which — if not elementary — depends on other methods. Thus, with methods we do not follow an interaction-oriented interface concept, but a structur-oriented interface concept, supporting the "is-part-of" relation. I propose to speak of a *"unilateral"* interface, as it relates only to its own system and therefore is composable with arbitrary other such interfaces within an appropriate hierarchical composition context.

3.2 Inhomogeneous or protocol composition

But, systems can also be composed inhomogenously by the same mechanism of "character exchange". But unlike section 3.1, I now use the I/O-TS to represent only parts of the systems in the sense of a projection. Thus, these I/O-TSs will generally be non-deterministic. I call these system parts *"roles"*.

Due to space constraints, I can only sketch the proceeding here (for details see [32]). Now the goal of the composition changes. It is not intended to create a new I/O-TS with some external in- and output, but to compose all involved roles $\mathcal{R}_i$, each from a different system, to create a closed transition system with no external in- and output any longer, namely a *"protocol"* $\mathcal{P}$ – which obviously does not represent a system in the original sense. For deterministic transition systems no input would mean no possible transitions. Thus, the nondeterminism of the roles is essential for this composition to make sense.

$$\mathcal{P} = C_{\mathcal{R}}^{Prot}(\mathcal{R}_1, \ldots, \mathcal{R}_n) \tag{3}$$

Thereby the focus of the composition changes from the mere existence of a (deterministic) transition relation towards a couple of properties, summarizable as "consistency", which directly relate to the execution of the protocol. Consistency comprises to be well formed in the sense that all sent characters can also be received. It also comprises to be interruptible which means that no infinite interaction chains exist. And it comprise to "work as intended". Aiming at consistency, we therefore have to extend our I/O-TSs to I/O-automata with an additional initial state q_0 and an acceptance component F that represents an additional local criterion for "works as intended". Essentially, coupling the roles to a protocol contrains the transition relation of the product transition system.

Leveraging the nondeterminism of the protocol roles, we can go a step further and introduce "decisions" as an additional internal input alphabet which determines the transitions of the otherwise nondeterministic roles. Thus, we can say that successfully taking part in a protocol implies knowing its rules and being able to take (for example by calculation) the necessary decisions.

Due to the nondeterministic character of all protocol roles, both, the internal state as well as the coupling I/O-states are inherent building blocks of the resulting protocol and cannot be hidden by composition. What protocols do hide is the calculation of the decisions. I propose to speak of a role as a *"multilateral"* interface, as it has to be combined with its complementary other interfaces of its protocol.

4 System architecture

4.1 Semantic layers

With the functional and the protocol composition, two different compositions exist, which we can express two-dimensionally in a graphical system model, as Fig. 2 shows. The layers in the vertical direction represent the order by the "is-part-of"-relation, created by functional composition. The horizontal direction relates systems of the same layer through protocols. Importantly, in this representation, information is exchanged only between systems, that is in the horizontal direction but *not* within systems, that is in the vertical direction. The system boundaries are determined by the interactions and *not* by the suggestive power of the ordering of some boxes.

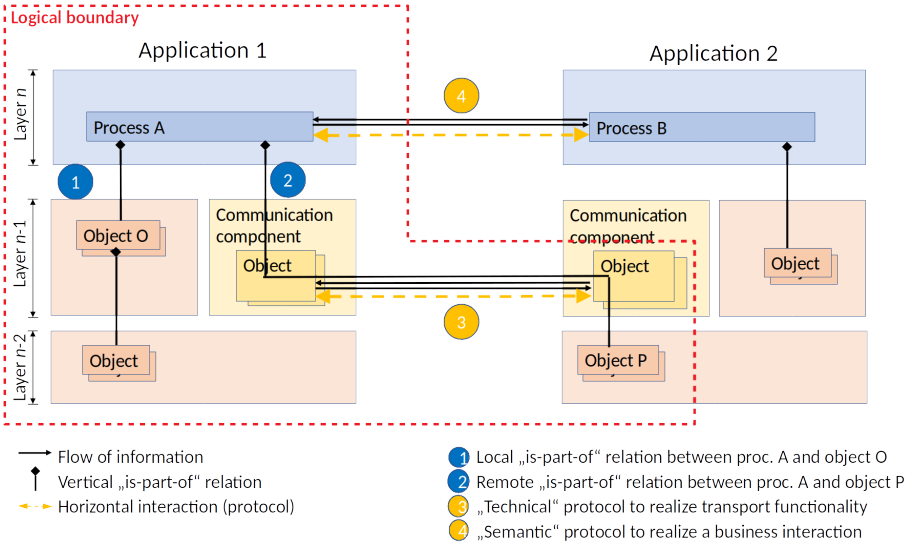

Fig. 2. A layered IT system architecture. Remote object P logically belongs to process A.

Why is it justified to talk about "semantic" layers? This attribute results from the model of information. Information is what is being transported between IT-systems and its semantics is attributed by its processing [33]. Thus any hierarchy of information processing implies a corresponding semantic hierachy.

It is clear that in a finite-size IT system with layered structure there must be a top layer. In order not to be part of a corresponding supersystem, the system it contains must be *"interactive"* in the sense that it is involved in multiple nondeterministic, stateful, asynchronous interactions – an activity I call *"coordination"* (see below). Other terms for such systems are "reactive systems" [12] or "processes". These top layer processes carry the "business semantics".

The top layer processes can delegate all their reusable functionality to dependent objects. This applies in particular to general recursive functions. Consequentially, each layer is the place where the software engineer has to accumulate the non-reusable aspects of the more concrete system logic. Conversely, an unclear layer structure can significantly impair reuse, since then, software engineers can only poorly separate the reusable from the non-reusable.

From our considerations, it immediately follows that protocols on higher semantic layers are an essential tool for "programming in the large" [38], since their roles demarcate complex, stateful IT- systems from one another.

4.2 Interaction oriented architecture

Perhaps the most interesting finding of our investigation is, that building interactive IT systems, we are exposed to a tension between describing systems via functions and being able to integrate these systems into the desired interaction networks only via nondeterministic, stateful, and asynchronous interactions. If we ignore this tension and describe interactive systems with an explicitly formulated system function, then we unfortunately lose any guarantee that small changes in individual interactions will result in correspondingly small changes in the structure of the application.

This leads to the obvious requirement to find a balance by an *"interaction-oriented IT system architecture"*. A hint in the right direction gives us the inhomogeneity of the composition of roles to protocols of Eq. (3). Hence, for the composition of (simple) protocols to (more complex) protocols, we have to look for another composition rule, which again has to refer to roles. If we consider the protocol composition as an *"outer"* composition, then the composition rule we are looking for to link the roles within a system would be an *"inner"* composition in the sense of a role coordination [32]. Building a system while preserving the roles would have the consequence that the correctness of the implementation of a protocol role is not affected by changes in other roles.

From a constructive point of view, the internal coordination of roles constrains the transition of their product automaton in the sense that each role provides all the information at the right time such that the desired coordination with all the other roles of the system can take place and the remaining nondeterminism is filled by decisions. It's the nondeterminism of each role which plays a major role for enabling a flexible inner coordination.

We thereby reach a reference architecture of interactive systems where a system $\mathcal{S}$ is composed from all its roles $\mathcal{R}_i$ by a composition operator $C_{\mathcal{R}}^{System}$ based on coordination rules and decisions.

$$\mathcal{S} = C_{\mathcal{R}}^{System}(\mathcal{R}_1, \ldots, \mathcal{R}_n) \tag{4}$$

5 Component models in the literature

According to Gerard J. Holzmann [16], the term software component was coined at the 1968 NATO Conference on Software Engineering in Garmisch by Doug

McIlroy [23]. The tight relation between the component and the interface concept makes this relation well suited for an evaluation of component models.

Hierarchically composing component models are meanwhile ubiquitous [7]. But what about support for horizontal interactions? In their overview of component models, Ivica Crnkovic et al. [8] distinguish between "operation-based" and "port-based" interface support, showing that many currently important component models do not in fact support role declarations for protocol-based interactions, like "implements role X of protocol y" at all.

5.1 Components as distributed objects

Distributed object models were developed under the idea that the encapsulation of the internal state by a beforehand defined set of operations in the sense of an abstract data type "hides" this state against the external world, providing some "autonomy" and thereby achieving a "loose" coupling between IT systems [5]. Examples are the Common Object Request Broker Architecture (CORBA), the Distributed Component Object Model (DCOM) or also the Open Platform Communications Unified Architecture (OPC-UA).

However, according to our definition, only the compositional structure can be hidden behind an object-oriented interface, but not the respective mapping. Thus, from a logical perspective, remote objects become just a part of the "one IT-system" as local objects do and there is no question of "loose coupling". We simply cannot reach out of a system by the call of an operation. And we cannot – by definition – express horizontal relations with the interfaces of operations.

5.2 Service oriented architecture (SOA)

The idea of a SOA goes back to R.W. Schulte and Y. V. Natis of the Gartner Group. [37]. OASIS defines a SOA quite unspecifically as a *"paradigm for organising and utilising distributed capabilities that may be under the control of different ownership domains."* [25]. A *"service"* is defined as *"The performance of work (a function) by one for another." and as a "mechanism by which needs and capabilities are brought together"*. A SOA is currently being propagated, for example, for Industry 4.0 [10] or in NATO [1] (Vol. 2).

In fact, none of the service definitions address the transformation behaviour of a "service", such as whether it represents a function or not. The WSDL 1.1 specification [43] defined four *"transmission primitives"* called *"operations"*. WSDL 2.0 [44] in section 2.2.1 speaks of an *"interface component"* as a *"sequences of messages that a service sends and/or receives"* and of an *"operation"* as an *"interaction with the service consisting of a set of (ordinary and fault) messages exchanged between the service and the other parties involved in the interaction"*. Hence, according to our composition concept, SOA interfaces are not well-defined (see also [31]) .

Applying these SOA-"concepts" to involve humans into processes leads to functionalisation: people become task completers. This is nicely illustrated in the WS-Task [26] specification where a "task", represented by a WSDL-operation, is

the basis *"to the integration of human beings in service-oriented applications"*. The term "decision" does not appear a single time.

The naming as "service" makes mutual understanding with economists quite difficult because in economics a "service" does not represent itself as a simple function, as the SOA with its WSDL interface syntax suggests. For example, to get a wall painted in a newly built house, one has to solicit bids, accept a bid, arrange and, if necessary, rearrange appointments, check the result, if accepted, pay the bill, and finally even keep the documents for tax declaration – a relationship with the craftsmen at eye level, full of state, asynchrony, and nondeterminism. It is precisely for this reason that economics describes these interactions as games, which are – as we have seen – closely related to protocols.

5.3 Representational State Transfer (REST)

REST [11] can be seen as an attempt to apply the principles of stateless communication along with semantic agnosticism – both principles of the highly successful Hypertext Transfer Protocol (HTTP) – to network system interactions. Currently, it is often positioned as a simpler variant of SOA.

A REST call is said to satisfy the principles of addressability, that each resource must have a unique URI, and statelessness, that each REST message should contain all the information necessary for the processing it initiates. Sometimes idempotence (e.g., [27]) is also mentioned, that a REST call should always have the same effect regardless of its timing.

In some way, I think, REST rests on a misunderstanding of the role of state in network system interactions. Computing functionality, state occurs only transiently, but for coordination it is essential. For example, if I seriously apply the idea to interact statelessly to my bank transfer, then the bank would not be aware of my account balance. An interesting thought.

The actual transformational behaviour is explicitly not part of the semantics of a REST interface, nor is any relationship between different REST interfaces. Accordingly, REST "interfaces" do not represent interfaces in the sense of this article, but represent only a transport semantic.

5.4 Client server model

The client server model is usually not understood as a component model. However, it is an interaction model and as such client and server act as two components. Thus the question arises, considering its practical relevance, what their relationship constitutes in the sense of a component model.

At the beginning of the 1990s the client server model was understood as a request/reply scheme [2,40]. With the emergence of SOA, the understanding shifted to a division of the system function into "services" that are provided by different servers and could be used by a client [39].

The client-server model is also relevant in the context of database-based applications. There, engineers often talk of 2-tier, 3-tier, or multi-tier architecture

in terms of layering, where the database forms the bottom tier 1 and user interaction forms the top tier.

In terms of our model of systems and their interactions, client and server first of all represent systems whose interaction let them fall into any of the different composition classes. The only significant difference between client and server is that between a caller and the called party – which is exactly what you find looking at the specification, for example, of the TCP/IP protocol (RFC 793, 7323), where the server waits at a TCP/IP port for calls from a client.

This criterion can indeed be used to declare an order. The only question is, how meaningful it is. True, the division into caller vs. called does fit the semantic direction of a remote operation call. But it would be a complete misunderstanding, in my view, if we attribute the success and prevalence of the client-server model to this fit.

Let's take a look at the so-called 3-tier architecture of modern enterprise applications, consisting of database, application server and user interface (UI) component. Its scalability has been one of the technological reasons for SAP's business success: after the introduction of the R/3 system, which featured such an architecture, SAP's per capita revenue rose from about 150,000€ in 1992 to 250,000€ in 2000, while the number of employees increased from about 3,000 to 20,000 in the same period. This development can only be understood, in my view, by recognising that with a 3-tier architecture, there is – at least from an interaction perspective — no layering, but 3 comparatively independent, stateful applications, each representing its domain of expertise, coordinating multiple non-deterministic, horizontal interactions.

6 Reference architectures in the literature

According to [22], a reference model is *"an abstract framework for understanding significant relationships among the entities of some environment. It enables the development of specific reference or concrete architectures using consistent standards or specifications supporting that environment."*.

There exist many so called "reference architectures", most of them provide some model of layering. It is quite astonishing that, at least based on the ideas presented in this work, many of them actually lack a consistent criterion for their claimed layers.

6.1 The Open Systems Interconnection (OSI) model

One of the most influential reference models is certainly the Open Systems Interconnection (OSI) model [19] which every informatics student learns in her first semester. It established the idea of a multi-layer software architecture. However, the assumption *"OSI is concerned with the exchange of information between open systems (and not the internal functioning of each individual real open system)."* is inconsistent. One cannot make assertions about an information processing

system's internal structure, such as layering, while refraining from saying anything about the structure of its information processing. Also, the OSI model was not precise enough about the nature of the hierarchy. For example, the OSI assumption that information processing between components by means of protocols always takes place in the same layer proved to be false in the case of remote function calls.

Thus, the system and interaction model of this article provides a formal justification for the intuition of the OSI model to consider software applications as layered. However, it also explains at the same time why the OSI model has found its way into reality only up to its 4th layer. The management of a "session" state cannot be assigned to a dedicated layer in the general case. Only in the case of vertical relation can the interaction-related state be hidden in a state of an intermediate "session" layer. In the case of horizontal interaction, the interaction-related state actually belongs to the components of the same semantic layer that interact with each other.

6.2 The Level of Conceptual Interoperability Model (LCIM)

Another example of a more frequently referenced reference model (e.g. lately in the IIC Connectivity Framework [17]) is the *"Level of Conceptual Interoperability Model (LCIM)"* [41], which consists of the 7 alleged layers: no [interoperability], technical, syntactic, semantic, pragmatic, dynamic, and conceptual interoperability.

Obviously, it is not interaction that constitutes this hierarchy, but what else? Even for the technical transport of information, e.g. by the Internet protocol, a certain structure (=syntax) of the transported information is necessary. It is unclear how to separate semantic from pragmatic aspects. For example, how can the meaning of a bank transfer be described without referring to an action that a bank should perform? In my view, "syntactic interoperability" is most likely to mean mutual understanding at the level of data types, that is, that an incoming document can be mapped to an internally used typed data structure. In my understanding, this is already "semantics" and comes into play in every protocol-based, horizontal interaction and is accordingly not limited to one level of interaction.

6.3 The Reference Architecture Model Industry 4.0 (RAMI4.0)

A third example is the Reference Architecture Model Industry 4.0 (RAMI4.0) [10]. At the centre of this model is the "asset" as an entity that is used in the context of industrial production and has some value to an organisation. RAMI4.0 aims to structure the standardisation efforts of the Plattform Industrie 4.0, a network led by the German government to further develop and implement its high-tech strategy in the field of industrial manufacturing, along the three axes of: (1) architecture, (2) life cycle, and (3) hierarchy.

As noted in [36], on the one hand, no clear ordering criterion can be identified for the architecture axis and, on the other hand, the hierarchy axis is based on

an "is-part-of" relationship that is not conceptually independent of the architecture axis. Also, the categories of the hierarchy axis are not mutually exclusive — a company could also be a product, for example. Finally, the different compositional mechanisms for interacting systems are not considered: A technical system that is part of a company is so in a different way than, say, a taillight is part of a car.

7 Summary and outlook

With this article, I have tried to show that one important source of the current vagueness of the concept of IT-system architecture is the seemingly unnoticed vagueness of basic concepts like interface and component. In this sense it's a nice example of the effect of an unknown unknown. Even though reference architectures make heavily use of the concepts of horizontal vs. vertical system relations, they nevertheless lack a clear enough understanding of them. And introducing additional concepts like "service" or "architectural style" doesn't help very much in this respect.

My proposal with this article is to use the composition notion and apply it to the information processing of systems to sort things out. It is interesting to see that we reach at virtually the same conclusions using a simple classification of interactions [35]. Composing systems requires us to refer to their mapping relations they represent. If we use them completely we get functional or supersystem composition, and if we use them partially we get protocols. We get on very slippery terrain it we refrain from using them at all. It was actually the different compositional behavior which motivated David Harel and Amir Pnueli to distinguish between *"transformational"* and *"reactive"* systems [12].

We thereby reach at a decidable interface and component concept and actually at a very dynamic system concept. Systems are being created or destroyed depending on the creation or destruction of a system function. The systems' structure is nothing we determine by nice box drawings but it is the structure of their system function, that is how it is composed out of its parts. And to compose functions from parts is essentially what informatics is all about.

It's the composition concept that allows us to understand first, that discrete systems interacting on an equal semantic level generally have to be interactive; secondly, that creating interactive IT-systems poses a dilematic problem to the engineer, which, thirdly, might be solved by an interaction-oriented architecture.

Why are clear interfaces are important? Because they demarque system boundaries. Unclear system boundaries entail security risks. We can make a virtue out of necessity by using security controls like signature and encryption to enforce the demarcation of system boundaries. Thereby proper security becomes a testing probe for a good system architecture [4].

Beyond that, well-defined boundaries of systems being constructed are of direct relevance for organising the constructing organisations. The presented model suggests that the interaction structures of the constructed IT-systems should shape the interaction structures of the organisations constructing them

[13] – the inverse of "Conway's Law". The definition of vertical interfaces should be a means to entrust departments with differently abstract topics, and the definition of protocol-based interfaces should be a good instrument for structuring larger software-developing organisations in departments with separate domains of expertise.

References

1. NATO Standard Allied Data Publication ADatP-34, Edition M, Version 1 NATO Interoperability Standards and Profiles (2020)
2. Andrews, G.R.: Paradigms for process interaction in distributed programs. ACM Computing Surveys (CSUR) **23**(1), 49–90 (1991)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
4. Bitkom: Vorschlag zur systematischen Klassifikation von Interaktionen in Industrie 4.0 Systemen – Hinführung zu einem Referenzmodell für semantische Interoperabilität. White paper (2020)
5. Chin, R.S., Chanson, S.T.: Distributed, object-based programming systems. ACM Computing Surveys (CSUR) **23**(1), 91–124 (1991)
6. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures, Views and Beyond. Addison-Wesley, 2 edn. (2011)
7. Cox, R.: Surviving software dependencies. Communications of the ACM **62**(9), 36–43 (2019)
8. Crnkovic, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V.: A classification framework for software component models. Software Engineering, IEEE Transactions on **37**(5), 593–615 (2011)
9. De Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: International Workshop on Embedded Software. pp. 148–165. Springer (2001)
10. DIN: SPEC 91345:2016-04 Referenzarchitekturmodell Industrie 4.0 (RAMI4.0) (2016)
11. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
12. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems, pp. 477–498. Springer-Verlag, New York (1985)
13. Herbsleb, J.D., Grinter, R.E.: Splitting the organization and integrating the code: Conway's law revisited. In: Proceedings of the 21st international conference on Software engineering (ICSE). pp. 85–95 (1999). https://doi.org/10.1145/302405.302455
14. Hoare, C.: Communicating Sequential Processes. Prentice Hall (1985/2004)
15. Holzmann, G.J.: Design and validation of computer protocols. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1991)
16. Holzmann, G.J.: Software components. IEEE Softw. **35**(3), 80–82 (2018)
17. IIC: The Industrial Internet of Things, Volume G5: Connectivity Framework (2018)
18. Systems and software engineering - Architecture description (2011)
19. ITU-T: X.200 Information Technology - Open Systems Interconnection – Basic Reference Model (1994)
20. Kleene, S.: General recursive functions of natural numbers. Mathematische Annalen **112**(5), 727–742 (1936)

21. Lornsen, B.: Robbi, Tobbi und das Fliwatüt. Thienemann-Esslinger Verlag (1967)
22. MacKenzie, C.M., Laskey, K., McCabe, F., Brown, P.F., Metz, R. (eds.): Reference Model for Service Oriented Architecture 1.0. OASIS (2006)
23. McIlroy, M.D.: Mass-produced software components. In: Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany. pp. 88–98 (1968)
24. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes (parts I and II). Information and Computation **100**(1), 1–77 (1992)
25. OASIS: Reference model for service oriented architecture 1.0. `http://docs.oasis-open.org/soa-rm/v1.0/` (2006), aufgerufen am 2015-01-17
26. OASIS: Web Services – Human Task (WS-HumanTask) Specification Version 1.1 (2010), http://docs.oasis-open.org/bpel4people/ws-humantask-1.1.pdf called 2021-03-21
27. Pautasso, C.: RESTful web services: principles, patterns, emerging technologies. In: Web Services Foundations, pp. 31–51. Springer (2014)
28. Rechtin, E., Maier, M.W.: The art of systems architecting. Boca Raton, FL: CRC Press, 2009, 3 edn. (2009), 1st edition in 1997
29. Reich, J.: Über Struktur oder das Verhältnis der Teile zum Ganzen. Philosophia Naturalis **38**(1), 37–69 (2001)
30. Reich, J.: The relation between protocols and games. In: Fischer, S., Maehle, E., Reischuk, R. (eds.) 39. Jahrestagung der GI, Lübeck. LNI, vol. 154, pp. 3453–3464. GI (2009)
31. Reich, J.: Eine semantische Klassifikation von Systeminteraktionen. In: Cunningham, D., Hofstedt, P., Meer, K., Schmitt, I. (eds.) INFORMATIK 2015. pp. 1545–1559. Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn (2015)
32. Reich, J.: Composition, cooperation, and coordination of computational systems. preprint arXiv:1602.07065 (2016/2020)
33. Reich, J.: A theory of interaction semantics. preprint arXiv:2007.06258 (2020)
34. Reich, J.: Komposition und Interoperabilität von IT-Systeme. To appear in Spektrum Informatik, Springer (2021)
35. Reich, J., Schröder, T.: A simple classification of discrete system interactions and some consequences for the solution of the interoperability puzzle. ifac 2020 (2020)
36. Reich, J., Zentarra, L., Langer, J.: Industrie 4.0 und das Konzept der Verwaltungsschale – eine kritische Auseinandersetzung. HMD Praxis der Wirtschaftsinformatik pp. 1–15 (2020)
37. Schulte, R.W., Natis, Y.V.: "Service-Oriented" Architectures, Part 1 and 2. SSA Research Notes SPA-401-068, -069, Gartner Group (1996)
38. Singh, M.P., Chopra, A.K., Desai, N., Mallya, A.U.: Protocols for processes: programming in the large for open systems. ACM Sigplan Notices **39**(12), 73–83 (2004)
39. Sommerville, I.: Software engineering. Prentice Hall, 9 edn. (2009)
40. Tanenbaum, A.S.: Modern Operating Systems. Prentice Hall International, Inc (1992)
41. Tolk, A., Turnitsa, C.D., Diallo, S.Y., Winters, L.S.: Composable M&S web services for net-centric applications. The Journal of Defense Modeling and Simulation **3**(1), 27–44 (2006)
42. Tripakis, S.: Compositionality in the Science of System Design. Proceeding of the IEEE **104**, 960 – 972 (2016)
43. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl` (2001), called 2015-01-17
44. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. `http://www.w3.org/TR/wsdl20/` (2007), aufgerufen am 2015-01-17