Acta Informatica (2019) 56:229–253
https://doi.org/10.1007/s00236-018-0328-7

ORIGINAL ARTICLE

Associative composition of components with double-sided interfaces

Wolfgang Reisig[1]

Received: 21 December 2017 / Accepted: 19 October 2018 / Published online: 29 October 2018

Abstract
Distributed systems are often organized in chains of components (e.g. business process chains), where each component naturally has a double-sided (left and right) interface. We suggest a corresponding, highly abstract and general framework (in mathematical terms: a *monoid*) of components and their composition, with minimal assumptions on the underlying global infrastructure (in fact, just a global set of symbols). As a fundamental property, decisive for the composition of more than two components, composition of such properties turns out to be *associative*. We discuss a number of instantiations of this framework (mainly classes of Petri nets), some of which preserve important properties (such as *soundness* of workflows) under composition. We glance at a number of generalizations and specializations.

1 Introduction and motivation

In the history of computing and programming, three particularly successful paradigms prevailed: conventional, imperative algorithms since the 1970ies, object-oriented computing since the 1980ies, and finally, distributed, component-, agent-, and service-oriented computing (SOC) since the 1990ies. The first two of these paradigms came with a solid formal, mathematical basis: computability theory, and general algebra together with first-order logic. In contrast, a dazzling array of models and formalisms have been published for the third paradigm. A lot of areas contributed such models and formalisms, including concurrency theory, multi-agent systems, coordination languages, component architectures, service-oriented architectures, and others. Those models and formalisms evolved without a perspective for unifying or unified fundamental principles and concepts. The-few-concepts common to all these models and formalisms include *components* in very broad terms, and the *composition* of components.

In this paper we suggest a general framework for modeling the composition of components. This framework is inspired by service-oriented computing and business processes. It is applicable in other areas too, e.g. in multi-agent systems.

✉ Wolfgang Reisig
reisig@informatik.hu-berlin.de

[1] Humboldt-Universität zu Berlin, 10099 Berlin, Germany

1.1 Components

As mentioned above, models for *components* are the central notion of the forthcoming framework. A central aspect of components is their *behavior*: A behavior may include some kind of *communication* with the component's *environment*. Typically, a component may send or receive messages to or from other components. Or a component may execute a step synchronously with a step of an other component.

1.2 Lightweight infrastructure

In the context of service-orientation and business processes, components are made to be *composed* with other components. Composition of components requires an *infrastructure* that organizes and realizes composition. We strive at an utmost general modeling framework for components and their composition. So, we keep the assumptions about the infrastructure at a minimum. In particular, we assume a uniform way of communication for all components. Heterogeneous forms of communication, e.g. handling the decision whether or not two components A and B communicate correctly or sensible according to a given protocol, should not be a matter of the communication infrastructure. This kind of questions may be treated by a third component, C, placed between A and B.

In technical terms, the requirement for a lightweight infrastructure for any sensible class $\mathbb{S}$ of models of components motivates the requirement that

$$\text{composition is } \textit{total} \text{ on } \mathbb{S}. \tag{1}$$

One may understand (1) as a manifestation of the popular axiom on interpersonal communication as stated by Paul Watzlawick (and others) that "one cannot not communicate" [31].

1.3 Associativity

Useful and manageable systems are frequently composed of *many* components. Typical examples include business process chains, consisting of e.g. an ore mine, a steel mill, a cutlery manufacturer, a wholesaler, a retailer and a consumer. In such a chain $A_1 \cdot ... \cdot A_n$ of component models, the order of indices i in which A_i and A_{i+1} (or A_i and A_{i-1}) are composed, must remain irrelevant. In technical terms this means for a class $\mathbb{S}$ of component models that composition is associative, i.e. for any three models A, B, C in $\mathbb{S}$,

$$(A \cdot B) \cdot C = A \cdot (B \cdot C). \tag{2}$$

1.4 The component monoid

As outlined above, composition should be a total and associative operation in any sensible set $\mathbb{S}$ of component models: then any model composed of smaller elements can easily and locally be extended, updated, revised, etc. It will turn out that there is always a *neutral* element N in $\mathbb{S}$, i.e. for all elements $A \in \mathbb{S}$ holds

$$N \cdot A = A \cdot N = A. \tag{3}$$

The properties (1), (2) and (3) imply that $\mathbb{S}$ exhibits the algebraic structure of a *monoid*. This in turn implies that we can consider component models in the same way as we consider words of formal languages.

1.5 Expressivity

Lightweight infrastructure and simplicity of composition, shaped $A_1 \cdot ... \cdot A_n$, nevertheless are maximally expressive: Any finite network with components A_i can be represented in the forthcoming framework.

2 The formal framework

2.1 Components with interfaces

A component with states and steps, as discussed in Sect. 1.1 already, can abstractly be conceived as a set of *elements* and a *relation* over the elements. Examples include transition systems and any kind of automata models, Petri nets, BPMN diagrams, statecharts, MSCs, etc. As an example, Fig. 1 shows three component models, shaped as Petri nets. Circles and squares (the Petri net's places and transitions) are the elements of the model. The arcs represent its relation (the Petri net's flow relation).

Adequacy and usefulness of an abstract formal framework for components heavily depend on the framework's way to *compose* components. Components are usually composed along their *interfaces*. A component's interface is usually a subset of its elements. As an example, the interface elements of Fig. 1 are colored blue and red. Interface elements are labeled. In

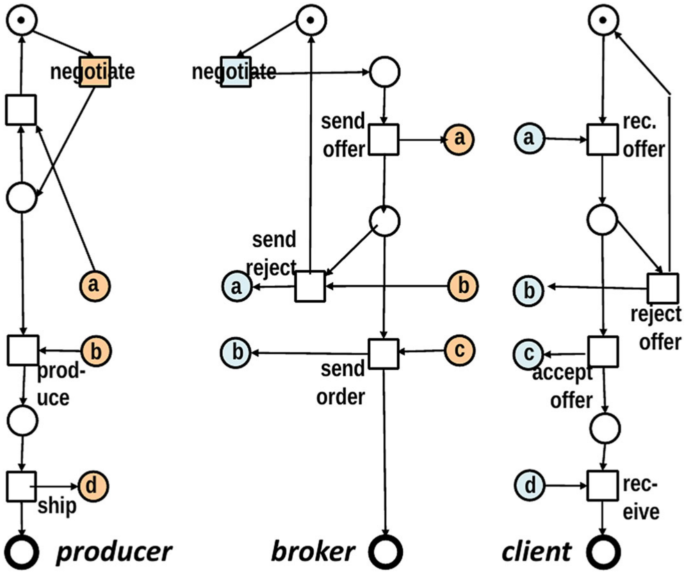


Fig. 1 Three service models (left and right interface elements in blue, and red, respectively (color figure online)

fact, a set Λ of labels, known to all components, is one of the few universal concepts of the underlying infrastructure. In Fig. 1, each interface element is inscribed by its label. So, for the rest of this paper, we assume

$$\text{a finite set } \Lambda \text{ of } \textit{labels}, \tag{4}$$

intended to label interface elements of components.

2.2 Double sided interfaces

The pivotal idea of the notion of components and their composition in this paper is based on the observation that the composition of *many* components A_0, ..., A_n is mostly organized in a row $A_0 \cdot ... \cdot A_n$. Every A_i $(i = 1, ..., n-1)$ has apparently A_{i-1} as its *left partner* and A_{i+1} as its *right partner*. A broker, as in Fig. 1, definitely makes sense only if it has two clear-cut partners. Mediators and adapters, frequently occurring in service-oriented computing, are further typical examples.

Consequently, a component A has a *double-sided* interface: a set $^*\!A$ of *left* interface elements, and a set A^* of *right* interface elements. The components of Fig. 1 in fact have this kind of interface: **producer** has no left interface elements (***producer** $= \emptyset$). Its right interface, **producer***, includes a transition, labeled **negotiate**, and three places, labeled **a**, **b** and **d**. Correspondingly, the labels of the broker's two interfaces, ***broker** and **broker*** are **negotiate**, **a**, **b** in the left, and **a**, **b**, **c**, in the right interface, respectively. The client's right interface is empty (**client**$^* = \emptyset$).

2.3 Composing double sided interfaces

The composition $A \cdot B$ of a component A with a component B is gained by "gluing" equally labeled elements (a *matching pair*) of A^* and $^*B$. This yields a new element, that inherits the arcs from the glued elements, and is turned into an inner element of $(A \cdot B)$. Each element in A^* without a matching partner in $^*B$ turns into an element of $(A \cdot B)^*$. Likewise each element of $^*B$ without a matching partner in A^* turns into element of $^*(A \cdot B)$. As an example, in Fig. 1, the two sets **producer*** and ***broker** have three matching pairs, labeled **negotiate**, **a** and **b**, respectively. They turn into inner elements in Fig. 2. The **d**-labeled place of **producer*** has no matching partner in ***broker**. Thus, this element goes to (**producer**$\cdot$**broker**)* in Fig. 2. Now (**producer** $\cdot$ **broker**)* and ***client** have four matching pairs. Thus, as Fig. 3 shows, all previous interface elements turn into inner elements in **producer** $\cdot$ **broker** $\cdot$ **client**, which remains with empty left and right interfaces.

Matters are not always as simple: In the technical example of Fig. 4, ***A** has two **a**-labeled elements. They are tagged by *yellow indices*, identifying them individually as the *first* and the *second* **a**-labeled element of ***A**, respectively. The two **c**-labeled elements of **A*** are likewise tagged. Generally, each interface element has an index. The tag for the index "1" is usually omitted in the—frequent—case where a left or a right interface has no two or more equally labeled elements. Figures 1 and 2 are examples for this case.

One may be tempted to stick to injective labelings of interfaces. However, components A and B with injectively labeled interfaces may yield a composition $A \cdot B$, violating this property. As an example, replace one of the labels **a** in A^* of Fig. 4 by **b**. Formulated differently, composition is not total on the set of components with injectively labeled interfaces.

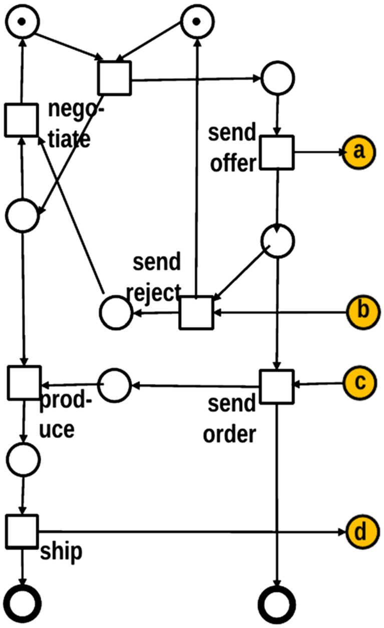


Fig. 2 The composed system **producer** · **broker**

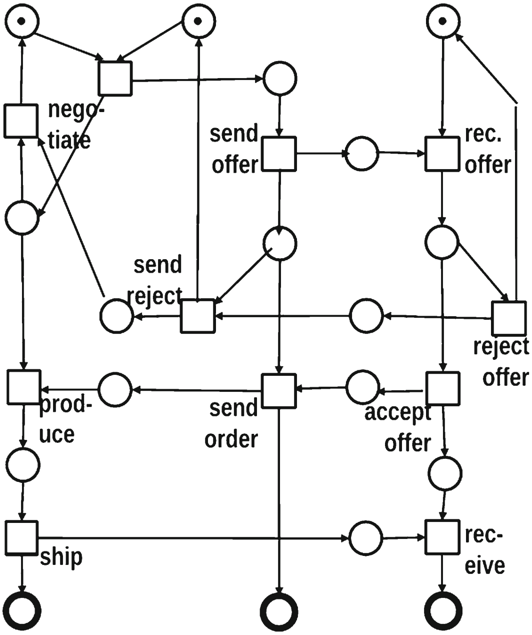
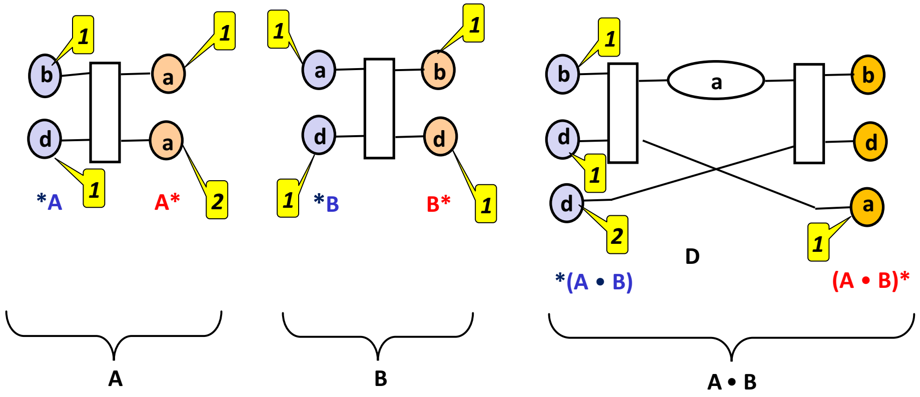


Fig. 3 The composed system **producer** · **broker** · **client**

In Fig. 4, both $\mathbf{A}^*$ and ${}^*\mathbf{B}$ contain a **a**-labeled, **1**-indexed element. As the labels and the indices of these elements coincide, they are said to *match*. This implies that in $\mathbf{A} \cdot \mathbf{B}$ they both turn into one “glued” inner element.



Left interfaces ***A** , ***B** , and ***(A • B)** : ***blue*** ; right interfaces **A*, B* and (A • B)*** : ***red*** .
Indices of equally labelled elements are flagged yellow (index „**1**“ is occasionally skipped). The inner nodes of **A** and **B** are sketched as boxes.

Fig. 4 Sketch of two component models **A** and **B**, and their composition **A** · **B** (color figure online)

The **a**-labeled, **2**-indexed element of $\mathbf{A}^*$ has no matching partner in $^*\mathbf{B}$. Hence, this element goes to $(\mathbf{A} \cdot \mathbf{B})^*$, retains its label **a**, and is indexed **1** in $(\mathbf{A} \cdot \mathbf{B})^*$, because $\mathbf{B}^*$ contains no **a**-labeled element. Likewise, the **d**-labeled **1**-indexed element of $^*\mathbf{B}$ has no matching partner in $\mathbf{A}^*$. So, this element goes to $^*(\mathbf{A} \cdot \mathbf{B})$ with index **2** in $^*(\mathbf{A} \cdot \mathbf{B})$, because $^*\mathbf{A}$ contains already a **d**-labeled element.

2.4 Index labeled sets

Labeled and indexed elements, as required for left and right interfaces of components, yield *index labeled sets*, defined as follows, (w.r.t. Λ as in (4)).

Definition 1 Let M be a set, and let $\lambda_M : M \to \Lambda$ be a mapping. For each $x \in \Lambda$, let $M_x =_{\text{def}} \{m \in M | \lambda_M(m) = x\}$. Let $\delta_M : M \to \mathbb{N}$ be a mapping such that for each $x \in \Lambda$ and each $i \in \{1, ..., |M_x|\}$ there exists exactly one $m \in M_x$ with $\delta_M(m) = i$.

Then (λ_M, δ_M) is an *index label* of M.

Hence, the x-labeled elements of M are indexed 1,2, …etc. In graphical representations, each element of an index labeled set is inscribed by its label, and tagged by its index. The tag for index "1" is frequently skipped. In Fig. 4, all the left and right interfaces $^*\mathbf{A}$, $\mathbf{A}^*$, $^*\mathbf{B}$, $\mathbf{B}^*$,$^*(\mathbf{A} \cdot \mathbf{B})$, $(\mathbf{A} \cdot \mathbf{B})^*$ are index labeled sets.

2.5 The notion of component models

We are now prepared to define abstract models of components: A component model consist of *elements* (nodes), a *relation* to represent the component's structure, and its two *interfaces*.

Definition 2 Let A be a finite set, let $R_A \subseteq A \times A$ and let $^*\!A, A^* \subseteq A$ be disjoint and index labeled. Then A together with R_A, $^*\!A$ and A^* is a *component model*. The set A is its *carrier*, R_A its relational *structure*, $^*\!A$ its *left interface*, and A^* its *right interface*. The set $\text{inner}(A) =_{\text{def}} A \backslash (^*\!A \cup A^*)$ contains the *inner* elements of A.

Figure 4 shows graphical representations of three components: **A**, **B** and $\mathbf{A} \cdot \mathbf{B}$. The inner elements of **A** and **B** are abstracted as boxes. Elements of left and of right interfaces are colored blue and red, respectively, and depicted left and right to the box. Likewise, the Petri nets of all above figures are components, all with **1**-indexed elements. Some of them have empty interfaces.

In the sequel, we denote by

$$\mathbb{S} \text{ the set of all components with interface labels in } \Lambda. \tag{5}$$

With the convention that indexing goes top-down, index tags can be skipped in drawings of components, thus simplifying the pictures and their reading.

2.6 Matching pairs

Two components A and B are composed by "gluing" *matching* elements of A^* and $^*B$. Elements of two index labeled sets *match* iff their index labels coincide:

Definition 3 Let M, N be index labeled sets. A tuple (m, n) is a *matching pair* of (M, N), iff $\lambda_M(m) = \lambda_N(n)$ and $\delta_M(m) = \delta_N(n)$.

As an example, in Fig. 4, $(\mathbf{A}^*, {}^*\mathbf{B})$ has *one* matching pair. Its label is **c**, and its index is **1**. Replacing in $^*\mathbf{B}$ the label **a** by **c** would yield a second matching pair with label **c** and index **2**. In Figs. 1 and 2, **producer*** and ***broker** have three matching pairs (all indexed **1**). (**producer** · **broker**)* and ***client** have four matching pairs.

2.7 Composing component models

As outlined in Sect. 2.3 and in Fig. 4, composition of two component models A and B turns each matching pair (a, b) of $(A^*, {}^*B)$ into an inner element of $A \cdot B$. Each arc connected to a or to b in A or B, is redirected to (a, b) in $A \cdot B$. Each remaining element of A^* turns into an element of $(A \cdot B)^*$, and likewise each remaining element of $^*B$ turns into an element of $^*(A \cdot B)$. Upon this procedure, each element retains its label. But its index is recalculated: To compute $\delta_{(A \cdot B)^*}(p)$ for an x-labeled element $p \in A^*$, we first calculate the index of p in $A^* \backslash \text{pr}_1(Q)$, where Q is the set of all matching pairs of $(A^*, {}^*B)$, and pr_i (for $i = 1, 2$) denotes the ith projection. This results in the number $\delta_{A^*}(p) - |\text{pr}_1(Q)_x|$. In $(A \cdot B)^*$, the element p goes "on top" of B^*. Hence, $\delta_{(A \cdot B)^*}(p) = |(B^*)_x| + \delta_{A^*}(p) - |\text{pr}_1(Q)_x|$. By entirely symmetrical arguments we obtain for each x-labeled element $p \in^* B$: $\delta_{*(A \,\cdot\, B)}(p) = |(^*\!A)_x| + \delta_{*B}(p) - |\text{pr}_2(Q)_x|$.

As an example, for the "**a**"-labeled, "2"-indexed element p of $\mathbf{A}^*$ in Fig. 4, we obtain $|(\mathbf{B}^*)_\mathbf{a}| = 0$, $\delta_{A^*}(p) = 2$ and $|\text{pr}_1(Q)_\mathbf{a}| = 1$. Hence, $\delta_{(\mathbf{A} \cdot \mathbf{B})^*}(p) = 0 + 2 - 1 = 1$. Likewise, for the **d**-labeled element q of $^*\mathbf{B}$ we obtain $|(^*\mathbf{A})_\mathbf{d}| = 1$, $\delta_{*\mathbf{B}}(q) = 1$ and $|\text{pr}_2(Q)_\mathbf{a}| = 0$. Hence, $\delta_{*(\mathbf{A} \,\cdot\, \mathbf{B})}(q) = 1 + 1 - 0 = 2$. The formal definition of composing two component models reads as follows (with R_A etc. as defined in Definition 2):

Definition 4 Let A, B be component models with disjoint carriers. Let Q be the set of matching pairs of $(A^*, {}^*B)$, and let $K =_{\text{def}} (A \cup B) \backslash (\text{pr}_1(Q) \cup \text{pr}_2(Q))$. Then the *composition* $A \cdot B$ is the component model defined as follows:

(i) The carrier of $A \cdot B$ is $K \cup Q$.

(ii) For the relation $R_{A \cdot B}$ of $A \cdot B$ holds:
$(p, q) \in R_{A \cdot B}$ iff one of the following three conditions holds:

- $(p, q) \in R_A \cup R_B$, and $p, q \in K$,
- $p = (p_1, p_2) \in Q$ and $(p_1, q) \in R_A$ or $(p_2, q) \in R_B$,
- $q = (q_1, q_2) \in Q$ and $(p, q_1) \in R_A$ or $(p, q_2) \in R_B$.

(iii) $p \in {}^*(A \cdot B)$ iff
either $p \in {}^*A$, with $\lambda_{*(A \cdot B)}(p) = \lambda_{*A}(p)$, and $\delta_{*(A \cdot B)}(p) = \delta_{*A}(p)$
or $p \in {}^*B$ and with $x =_{\text{def}} \lambda_{*B}(p)$ holds:
$\lambda_{*(A \cdot B)}(p) = x$ and $\delta_{*(A \cdot B)}(p) = |({}^*A)_x| + \delta_{*B}(p) - |pr_2(Q)_x|$.

(iv) $p \in (A \cdot B)^*$ iff
either $p \in B^*$, with $\lambda_{(A \cdot B)^*}(p) = \lambda_{B^*}(p)$ and $\delta_{(A \cdot B)^*}(p) = \delta_{B^*}(p)$,
or $p \in A^*$ and with $x =_{\text{def}} \lambda_{A^*}(p)$ holds:
$\lambda_{(A \cdot B)^*}(p) = x$ and $\delta_{(A \cdot B)^*}(p) = |(B^*)_x| + \delta_{A^*}(p) - |pr_1(Q)_x|$.

All above figures show examples of components and their composition. Figure 5a shows a component model, in fact a Petri net N that models a workflow on a very abstract level. Figure 5b, c show the composition of two and three instances of N. Notations such as "$(N \cdot N)$" imply the implicit assumption that *instances* of a model are composed. This is a usual assumption for graph based models. In particular, any two component models to be composed are assumed to be disjoint in the sequel.

This completes our formal framework for components and their composition. Literature suggests a large variety of formal techniques to model components and their composition. Frequently, a component model comes with an interface, consisting of labeled elements. Component models are then composed by "gluing" equally labeled interface elements. Typical examples include Petri net models of workflows and services [28,32,33], but also models such as interface automata [7], and stream based calculi [5], some implemented languages employ interfaces with elements glued upon composition, including the coordination language REO [1] or many software architecture definition languages (ADls) [15]. Our formal framework differs fundamentally from all so far available formal techniques to model components and their composition, as will be outlined in Chapter 3.

3 The component monoid

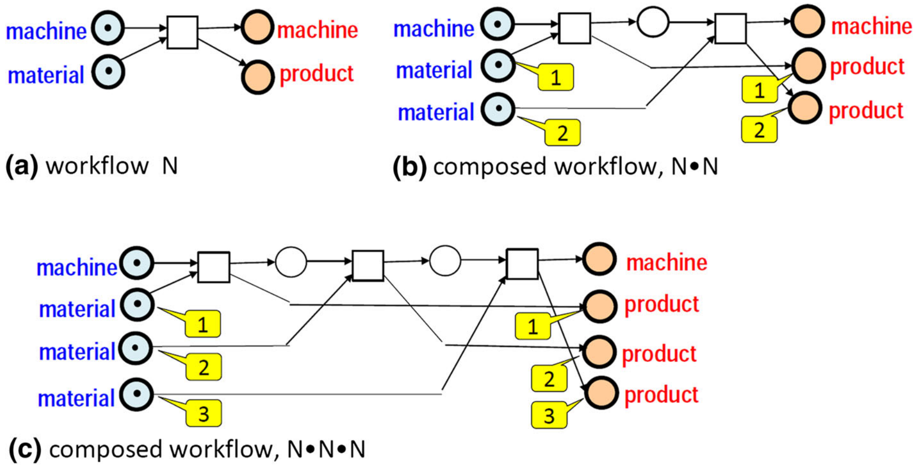
Composition of components, as defined above, exhibits a number of useful properties, that together add up to a *monoid*. Classes of instantiations of components then yield submonoids.

3.1 The monoid S

As mentioned in (5) in Sect. 2.5 already, a fixed set Λ of interface labels characterizes a fixed set $\mathbb{S}$ of component models. This set exhibits very useful properties. Firstly, composition is *total* on $\mathbb{S}$, i.e.

$$\text{for any } A, B \in \mathbb{S} \text{ holds: } A \cdot B \in \mathbb{S}. \tag{6}$$

This is obviously true according to Definition 4.



(a) workflow N

(b) composed workflow, N•N

(c) composed workflow, N•N•N

Fig. 5 Component model N of a simple workflow, modeled as a Petri net, transforming material into products by help of a machine, together with $N \cdot N$ and $N \cdot N \cdot N$

Composition is not total in many formal component frameworks. Frequently, a *composability* predicate is assumed, defining composition only for composable components. This applies for instance to automata based models such as I/O-automata [14] and interface automata [7]: components are composable only if their inner actions, as well as their output actions, are disjoint. Composability of two components A and B according to [3] requires that an interface element shared by A and B is an output element of A and an input element of B, thus resembling our matching pairs of A^* and $^*B$. Similar to our composition operator, non-shared output elements of A and input elements of B turn into output elements of $A \cdot B$ and input elements of $A \cdot B$, respectively.

Wolf [32] partitions a component's interface into sets of ports, and requires shared elements of two composable components A and B to belong to one port of A and one port of B. Coordination languages as well as software architecture languages allow for complex composability requirements, oriented at the intended semantics of components and their composition.

In contrast to those approaches, we separate the technical concern of *composing* two components A and B, from the semantical concern whether or not the composed component $A \cdot B$ is meaningful. Composition itself should be technically most simple and general, because the underlying infrastructure that would *organize* composition, should only be burdened with a minimum of requirements. Semantical concerns, e.g. the distinction between "pessimistic" and "optimistic" assumptions about the environment, as discussed in [7], can then be treated independently of the composition operator, either inside the components, or by means of an adapter (c.f. forthcoming Sect. 5.2).

We adopt an algebraic stance of components and composition, as e.g. also taken in process algebras: two process algebraic terms t and u may both contain a term s as a subterm. The two occurrences of s are entirely detached. This allows to write terms such as $s|s$, without any semantical problem. Accordingly, we allow composing several instances of a component, as in the example of Fig. 5.

The second useful property of the composition operator is its *associativity*, as addressed in Sect. 1.3 already. Formulated as a theorem:

Theorem 1 *Let A, B, C be component models. Then*

$$A \cdot (B \cdot C) = (A \cdot B) \cdot C. \tag{7}$$

The proof of this Theorem is intricate. It requires the distinction of 7 cases in a tree structure. For example, an element $p \in {}^*C$ may move to ${}^*(B \cdot C)$ with its index updated, and then to ${}^*(A \cdot (B \cdot C))$ with its index updated again. This must be mimicked by a single index update of p in ${}^*((A \cdot B) \cdot C)$. The detailed proof is given in the "Appendix".

Thirdly, there exists a *neutral element*, E, i.e. for any $A \in \mathbb{S}$ holds:

$$E \cdot A = A \cdot E = A. \tag{8}$$

E is the extreme case of the component model without any element. Consequently, the relation of E is the empty relation, and ${}^*E = E^* = \emptyset$. Definition 4 implies (8). In mathematical terms, the three properties (6), (7) and (8) characterize a *monoid*. So we come up with the fundamental

Observation 1 $\mathbf{S} =_{\text{def}} (\mathbb{S}; \cdot, E)$ is a monoid.

Thus we can consider the set $\mathbb{S}$ of component models over the label set Λ [c.f. (4)] in the same way as we consider the set Σ^* of words over a finite alphabet Σ.

For applications and implementations this means that *any* two components may be composed, without the fear of inconsistencies, illegal states, etc. Furthermore, composing components $A_1, \ldots, A_n$ can be written

$$A_1 \cdot \ldots \cdot A_n, \tag{9}$$

without brackets.

3.2 Interface equivalence

Occasionally we are not interested in the details of the inner structure of a component, but only in its interface. Then two component models with isomorphic interfaces are *interface equivalent*.

Definition 5 Let M and N be two index labeled sets. M and N are *isomorphic* iff for some bijection $f : M \to N$ and all $m \in M$ holds: $\lambda_M(m) = \lambda_N(f(m))$ and $\delta_M(m) = \delta_N(f(m))$.

Definition 6 Two component models A and B are *interface equivalent* iff ${}^*A$ and ${}^*B$, as well as A^* and B^*, are isomorphic.

It is easy to see that this equivalence is in fact an equivalence relation.

Notations As usual, we write for $A, B \in \mathbb{S}$:

- $A \sim B$ if A and B are interface equivalent,
- $[A]$ for the equivalence class of A, i.e. the set of all component models B with $A \sim B$.
- $\mathbf{S}_\sim$ denotes the set of all equivalence classes of component models.

Interface equivalence is in fact a congruence w.r.t. composition:

Observation 2 Let A, B, C be components, $A \sim B$. Then $A \cdot C \sim B \cdot C$ and $C \cdot A \sim C \cdot B$.

Composition of components expands canonically to equivalence classes:

Definition 7 For two equivalence classes $[A]$, $[B]$ of components, let $[A] \cdot [B] =_{\text{def}} [A \cdot B]$.

This definition is sensible according to Observation 2. Furthermore, the equivalence classes again yield a monoid:

Observation 3 $\mathbf{S}_{\sim} =_{\text{def}} (\mathbb{S}_{\sim}; \cdot, [E])$ is a monoid.

Proof Associativity: $([A] \cdot [B]) \cdot [C] = [A \cdot B] \cdot [C] = [(A \cdot B) \cdot C] = [A \cdot (B \cdot C)] = [A] \cdot [B \cdot C] = [A] \cdot ([B] \cdot [C])$.
Neutral object: $[A] \cdot [E] = [A \cdot E] = [A]$ and $[E] \cdot [A] = [E \cdot A] = [A]$ □

The neutral element $[E]$ of $\mathbf{S}_{\sim}$ contains all component models X with ${}^*X = X^* = \emptyset$.

We may ask for an *inverse* A^{-1} of a component model A such that $[A] \cdot [A^{-1}] = [A^{-1}] \cdot [A] = [E]$. Such an inverse does in general not exist. However, to each A there exist two component models L and R, such that $[L \cdot A \cdot R] = [E]$.

3.3 Commutative composition

Composition is in general not commutative, i.e. $A \cdot B$ and $B \cdot A$ are usually different. Nevertheless, in some cases we would expect commutativity. For example, a business process A may trigger a process B to book a flight, as well as a process C to book a hotel. We expect B and C may be triggered in any order, i.e. $A \cdot B \cdot C = A \cdot C \cdot B$. Intuitively, this assumption is justified, because the interface labels of B and C would be entirely disjoint. This, in fact, holds in general, according to the following definition and observation:

Definition 8 For an index labelled set M, let Λ_M be the set of all labels that occur in M.

Observation 4 Let A and B be components. Then $A \cdot B = B \cdot A$ iff $(\Lambda_{{}^*A} \cup \Lambda_{A^*}) \cap (\Lambda_{{}^*B} \cup \Lambda_{B^*}) = \emptyset$.

4 The Petri monoid and its submonoids

The above formalism provides a framework for components and their composition. Now we discuss concrete component models that naturally fit into this framework.

4.1 The Petri monoid

We assume the reader's familiarity with the basics of Petri nets. For details we refer to [21]. In this section, only the static structure of Petri nets is considered.

Definition 9 Let P and T be two finite, disjoint sets ("places" and "transitions"), and let $F \subseteq (T \times P) \cup (P \times T)$ ("flow relation"). Then

(i) $K = (P, T, F)$ is a *net structure*.
(ii) With $A =_{\text{def}} P \cup T$ and freely chosen disjoint, index labeled sets ${}^*A$, $A^* \subseteq A$, the set A together with F, ${}^*A$ and A^* is a *Petri component of* K.

With the usual graphical conventions of Petri nets (circles, boxes and arrows represent places, transitions, and the flow relation), all above figures show Petri components, with left and right interface elements colored blue and red, and drawn at the figures' left and right edge, respectively.

For Petri components we assume the set Λ of labels to be partitioned into two disjoint subsets Λ_P and Λ_T, i.e. with (4),

$$\Lambda = \Lambda_P \mathbin{\dot\cup} \Lambda_T, \tag{10}$$

to label places and transitions, respectively.

The interface of a Petri component may contain places as well as transitions, as for example **producer*** and ***broker** in Fig. 1. Requirement (10) guarantees that never a place and a transition are glued upon composition. Thus, composition of two Petri components yields again a Petri component. The empty component E is obviously a Petri component, too. So, with $\mathbb{P}$ denoting the set of all Petri components, we obtain

Observation 5 $\mathbf{P} =_{\text{def}} (\mathbb{P}, \cdot, E)$ is a monoid

P is the *Petri monoid*, in fact a submonoid of **S**.

4.2 The asynchronous and the synchronous Petri submonoids

In the interface of Petri components, places realize asynchronous communication, and transitions realize synchronous communication. Though both forms of communication may be mixed in one interface (as in Fig. 1), many applications engage just *one* of the two forms of communication. This fits with the observation that each of the two communication forms generates its own submonoid.

Definition 10 Let A be a Petri component.

(i) A is *asynchronous* iff $^*A$ and A^* contain only places.
(ii) A is *synchronous* iff $^*A$ and A^* contain only transitions.

Notations By $\mathbb{Q}$ and $\mathbb{T}$ we denote the set of asynchronous and synchronous Petri components, respectively.

The **client** in Fig. 1 and the components in Figs. 2, 3, 4 and 5 are all asynchronous. **producer** and **broker** of Fig. 1 are neither synchronous nor asynchronous.

Observation 6 $\mathbf{Q} =_{\text{def}} (\mathbb{Q}, \cdot, E)$ and $\mathbf{T} =_{\text{def}} (\mathbb{T}, \cdot, E)$ are both monoids, called the *asynchronous* and the *synchronous* Petri submonoid.

The asynchronous monoid is particularly interesting: Services are asynchronously composed. Pautasso and Wilde [19] motivates and justifies asynchronous composition in detail.

4.3 The soundness preserving monoid

An important class of components are *business processes*. Figure 6 shows a component model, in fact a special kind of Petri net, called *workflow net*, that describes an internet based purchase (The model itself is composed of a buyer- and a seller model. This aspect is however not relevant here). Workflow nets are based on net structures, as in Definition 1, and the basic notions of markings (c.f. [21]):

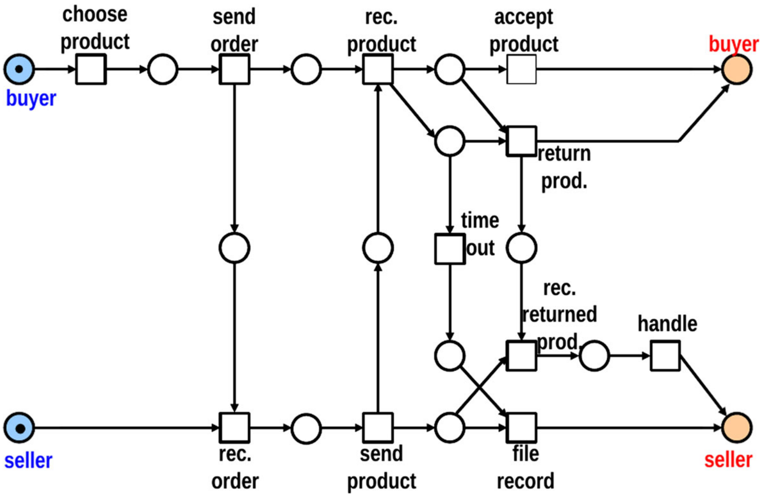


Fig. 6 Sound model of internet purchase

Definition 11 Let $N = (P, T, F)$ be a net structure. Each $Q \subseteq P$ defines the marking $\widehat{Q} : P \rightarrow \{0, 1\}$ with $\widehat{Q}(p) = 1$ iff $p \in Q$.

Definition 12 Let N be an asynchronous Petri component such that no arc starts at any place of N^*. Then N together with the initial marking $\text{init}_N =_{\text{def}} \widehat{{}^*N}$ and the final marking $\text{fin}_N =_{\text{def}} \widehat{N^*}$ is the *workflow component of* N.

By Observation 6, the asynchronous Petri components constitute a monoid, and so do the workflow components, by the above definition. We are interested in component models that are based on *sound* workflow components. This notion is based on the *reachability* of markings (c.f. [21]).

Definition 13 Let N be a workflow component. N is sound iff

– to each transition t there exists a marking reachable from init_N that enables t;
– from each marking that is reachable from init_N, the marking fin_N is reachable
– no marking $M \neq \text{fin}_N$ with $M(p) \geq \text{fin}_N(p)$ for each place p is reachable in N.

As an example, Fig. 6 shows a workflow component. It is easy to see that it is sound.

Soundness is a fundamental property: Essentially each reasonable business process is sound. *Worklfow nets* capture soundness. They have extensively been studied in [28]. A data sensitive version presents [9]. So it is very useful that composition of sound workflow components is sound. As an example, Fig. 7 sketches the composition of two instances of Fig. 6. Notice that the buyer may have started his second purchase, while the seller is still busy with the first purchase. We will show elsewhere that composition of sound workflow nets retains soundness.

5 Outlook and conclusion

In this chapter we glance at the expressive power as well as the limitations of components with double-sided interfaces, together with their composition. This includes modeling of

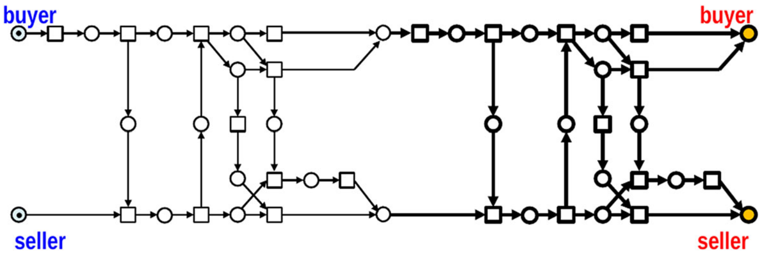
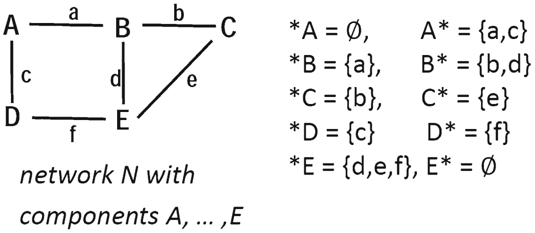


Fig. 7 Two purchases in a row (second one: bold faced)

Fig. 8 $N = A \cdot B \cdot C \cdot D \cdot E$



cyclic structures, variants of the composition operator, overlapping interfaces, characterizing sets of composed components, and suggesting a further operator on components. Further arguments on conceptual and applied aspects can be found in [22].

5.1 Representing components networks

The concept of representing a composed system as a sequence $A_1 \dots A_n$ of components A_i does not imply that interface elements of A_i would match only with interface elements of A_{i-1} and A_{i+1}, as Fig. 8 exemplifies. In fact, **any** finite network of components can be represented, with components composed this way: It is just a matter of placing an interface element of a component X into the left or the right interface $^*X$ or X^*, or re-labeling two component's interface elements.

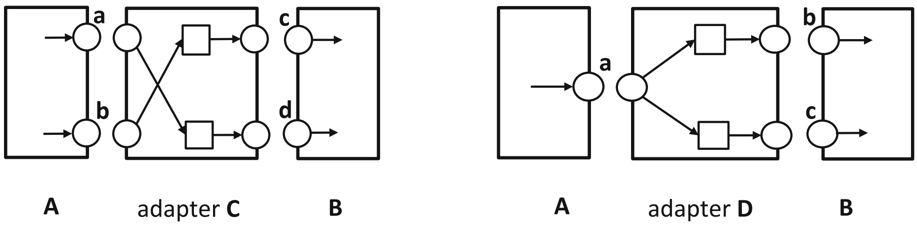
5.2 Modeling variants of composition by means of adapters

One may invent variants of composition that don't immediately fit to our framework. To cope with this issue, instead of extending our framework by further operators or "generalizing" it in whatever way, we suggest to formulate such variants for a composition $\mathbf{A} \cdot \mathbf{B}$ in terms of an *adapter* **C**, such that $\mathbf{A} \cdot \mathbf{C} \cdot \mathbf{B}$ exhibits the intended behavior. We present typical examples:

Our composition operator determines the matching of equally labelled elements of two interfaces: Equally indexed elements match. For example, with the top-down convention of indexing (as suggested in Sect. 2.5), in Fig. 9a, the matching pairs of $\mathbf{A}^*$ and $^*\mathbf{B}$ are **(a,c)** and **(b,d)**. The adapter **C** emulates the matching pairs **(a,d)** and **(b,d)**.

Our composition operator is *deterministic*: With places $\mathbf{a} \in \mathbf{A}^*$ and $\mathbf{b}, \mathbf{c} \in {}^*\mathbf{B}$ as in Fig. 9b we can't immediately express to nondeterministically choose either **(a,b)** or **(a,c)** as a matching pair for $\mathbf{A} \cdot \mathbf{B}$. This choice, however, is emulated by the adapter **D**.

On the abstract, semantics-free level of the graph monoid S as in Sect. 3.1, variants of composition may be defined by help of Sobociński's linking diagrams [25].



(a) *Adapting swap of matching interface elements:* **a** *is matched with* **d** *and* **b** *is matched with* **d**.

(b) *Adapting nondeterministic choice: matching either* (**a,b**) *or* (**a,c**).

Fig. 9 Adaption of swap of interface elements and of nondeterministic choice

5.3 Overlapping left and right interfaces

So far we assumed (as in Definition 3) that the left and the right interfaces of a component A are disjoint ($^*\!A \cap A^* = \emptyset$). However, skipping this requirement, the notion of composition as in Definition 4 can slightly be generalized such that the monoid property is retained. Depending on the structure of B, an element of $^*\!A \cap A^*$ moves to $^*(A \cdot B)$ or to $(A \cdot B)^*$ or to both theses sets. The intuitive idea of a left and a right interface no longer applies. Nevertheless, this punchy concept yields a lot of interesting applications (not detailed in this contribution).

The extreme case are components A with $^*\!A = A^*$. This kind of components again forms a monoid, which is even commutative. However, an interface element never turns into an inner element upon composition in this case.

5.4 Generating components from components

In analogy to the Σ^*-monoid of words generated from a finite set Σ of symbols, each finite set $\mathbb{C}$ of component models generates an infinite set of composed component models. Business administration typically starts out with a finite set of business processes, that in a company can be composed in many ways. Formal language theory may help to characterize sets ("languages") of "wanted" processes of companies. It goes without saying that one can easily remain in one of the submonoids, e.g. the synchronous or asynchronous Petri components, or the sound workflow nets.

5.5 The closure operator

In Sect. 5.1 we discussed how to construct cyclic structures. For example, one may compose the well-known five philosophers system as $P_1 \cdot \ldots \cdot P_5$, where with forks f_i, $^*\!P_1 = \emptyset$, $P_1{}^* = \{f_1, f_5\}$, $^*\!P_i = \{f_{i-1}\}$, $P_i{}^* = \{f_i\}$ for $i = 2, 3, 4$, $^*\!P_5 = \{f_4, f_5\}$ and $P_5{}^* = \emptyset$. However one would prefer to compose the system from five instances of a generic philosopher model P with one-elementary interfaces $^*\!P$ and P^* (both elements equally labeled). With $P \cdot P \cdot P \cdot P \cdot P$ we obtain five philosophers in a row. To close the cycle of five philosophers, however, one has to glue elements of $(P \cdot P \cdot P \cdot P \cdot P)^*$ with the matching elements of $^*(P \cdot P \cdot P \cdot P \cdot P)$. So, for a component A, the closure A^c is defined by composing A^* with $^*\!A$ in exactly the same way as A^* is composed with $^*\!B$, for some component B. For a philosopher model P, five philosophers in a cycle are gained as $(P \cdot P \cdot P \cdot P \cdot P)^c$. This yields a rich algebraic structure $(\mathbb{S}, \cdot, ^c, E)$ that deserves careful examination.

5.6 Other modeling techniques with double-sided interfaces

The Petri monoid and its submonoids, some of which we discussed in Sect. 4, are certainly prominent instantiations of the general framework. Nevertheless, the notion of a component, as well as the composition operator (Definitions 3, 4) require no specific aspects of Petri nets, but just a set of nodes and a relation over these nodes, viz., a graph. Many system models exhibit the structure of graphs. However, many of them assume *one* global thread of control. Upon composing two components, our framework naturally would yield more than one such thread. This is why our composition of classical automata and of transition systems does not return an automaton or a transition system. Our framework however comprises modeling techniques such as BPMN diagrams, statecharts and Message-Sequence-Diagrams. Variants of those techniques would be interesting, that follow the general principles of double-sided composition.

Related work

As outlined already several times, formal techniques to model components and their composition frequently extend or specialize known models. For example, some models are based on automata [7,14]. It is quite intuitive to define the semantics of a component (or service) as a relation over (infinite) streams of symbols, with a double-sided interface for input and output, as suggested in [3–5]. The REO framework [1] suggests to compose components by means of specialized channels and fitting connectors. We would suggest to formulate the intended effect of such channels and connectors as adapters. This combines high expressive power with a simple composition mechanism. [6] generalizes this approach to component networks. Composition of components is a central concern of software architecture languages (c.f. [12,15] for detailed surveys). In fact, many software architecture languages can be embedded into our framework. This helps to define their semantics and to guarantee associativity (among many other properties). In particular, microservice architectures fit in our framework, by composing many (small) components [17].

In the world of process algebras, the *chaining* operator “>>” (c.f. [23]) assumes processes P_i with in- and out-channels, where the out-channel of P_i is “glued” with the in-channel of P_{i+1}. This resembles Broy’s architecture [3–5], and can be conceived as a special case of our composition operator.

A number of contributions construct variants of the π-calculus: Vieira et al. [30] defines a π-calculus like formalism to represent service-based systems, including “context” as an infrastructure for communication among components. Nierstrasz and Achermann [18] considers questions similar to ours, and develops a π-calculus based formalism. Gößler et al. [10] employs “port sets” and a set of connectors that “glue” sets of port elements.

A logic based approach is given in [13].

A couple of Petri net based formalisms define building blocks and several composition operators (in sequence, alternatively, cyclic, parallel, etc.) in an algebraic style, such as the box calculus [2], or [11]. van Glabbeek et al. [29] identifies a class of “distributable” Petri nets, and shows that any distributable net may be implemented on a network of asynchronously communicating components. A number of contributions extend the fundamental notion of workflow nets [28] by data [9], [20].

A number of papers contribute non-trivial analysis techniques for workflow- and service models, that exploit the particular structure of Petri Nets. This includes [8,9,11,20,26–28,32, 34]. The idea of composing Petri Nets in an algebraic framework is not new. Fundamental algebraic and categorial constructs of Petri net composition offer [16,24].

We are strict on the structure of interfaces, but liberal on the inner structure of components: the interface of a component consists of two index labeled sets. The inner structure is any set together with a relation between inner and interface elements. No other modeling technique suggests a composition operator similar to ours. Totality of composition on a given network of components is rare. Associativity is frequently assumed, more or less implicitly, or is violated.

Acknowledgements I am very grateful to the anonymous reviewers for their very detailed and constructive review. They pointed at formal inconsistencies, debatable intuitive statements, and necessary clarifications of a previous version of this paper. Heinz Schmidt from RMIT University (Melbourne) showed me the relevance of our framework for the definition of architecture definition languages to model component based software architectures.

Appendix: Proof of the theorem

For component models X,Y, in the sequel we write

$$XY \text{ for } Y \cdot Y \tag{11}$$

Let G, H and K be component models. To prove the Theorem, with Definition 2 we have to show:

$$\text{The nodes of } (GH)K \text{ and of } G(HK) \text{ coincide.} \tag{12}$$

$$\text{The arcs of } (GH)K \text{ and of } G(HK) \text{ coincide,} \tag{13}$$

$$^*((GH)K) = {}^*(G(HK)) \tag{14}$$

$$((GH)K)^* = (G(HK))^* \tag{15}$$

$$\lambda_{^*((GH)K)} = \lambda_{^*(G(HK))} \tag{16}$$

$$\lambda_{((GH)K)^*} = \lambda_{(G(HK))^*} \tag{17}$$

$$\delta_{^*((GH)K)} = \delta_{^*(G(HK))} \tag{18}$$

$$\delta_{((GH)K)^*} = \delta_{(G(HK))^*} \tag{19}$$

Proof of (12) Definitions 2 and 4 immediately imply for component models X, Y and $f \in \text{inner}(X)$:

$$f \in \text{inner}(XY) \text{ and } f \in \text{inner}(YX).$$

This in turn implies for all $f \in \text{inner}(G) \cup \text{inner}(H) \cup \text{inner}(K)$: $f \in \text{inner}((GH)K)$ and $f \in \text{inner}(G(HK))$. The proposition then follows with forthcoming properties (23) and (24), and Definition 2. □

Proof of (13) Property (12) implies that the matching pairs of $(GH)K$ and of $G(HK)$ coincide. Hence, the replacements of arcs in the arc sets of $(GH)K$ and of $G(HK)$, as in Definition 4, coincide. □

Proof of (16) For component models X and Y, the λ-label of an element e in $^*X$ or $^*Y$ remains when e moves to $^*(XY)$, according to Definition 4. Likewise, the λ-label of an element e in X^* or Y^* remains when e moves to $(XY)^*$. The proposition the follows with (14) and (15). □

Proof of (17) Analog to Proof of (16). □

(14) and (18) are proven in the sequel. (15) and (19) follow analogously to (14) and (18) and are left to the reader.

Proof of (14) and (18) In the sequel, a *port* P is the left or right interface $^*X$ or X^* of one of the component models

$$X = G, H, K, GH, HK, (GH)K \text{ or } G(HK)\ . \tag{20}$$

With respect to a port P and $f \in P$ we write

$$\max(P) \text{ for } |\{p \in P \mid \lambda_P(p) = \lambda_P(f)\}|, \quad \text{and} \tag{21}$$

$$P(f) \text{ for } \delta_P(f)\ . \tag{22}$$

In (21), $\max(P)$ refers to an element f of P that is clear in each context, and is not noted for the sake of readability. Intuitively formulated, $\max(P)$ is the number of nodes in P with labels coinciding with the label of f. These nodes are numbered by δ_P, and $P(f)$ is the corresponding number of f. Obviously holds □

Lemma 1 *Let P be a port, let $f \in P$. Then $P(f) \leq \max(P)$.*

In order to show (14) and (18) we observe (by Definition 4) that, $^*((GH)K)$ and $^*(G(HK))$ contain only nodes of $^*G$, $^*H$ and $^*K$. Therefore it suffices to show for each $f \in {}^*G \cup {}^*H \cup {}^*K$:

$$f \in {}^*((GH)K) \text{ and } f \in {}^*(G(HK)),\ \text{ with } {}^*((GH)K)(f) = {}^*(G(HK))(f) \tag{23}$$

or for some node g,

$$(g, f) \in \text{inner}((GH)K) \text{ and } (g, f) \in \text{inner}(G(HK)). \tag{24}$$

We start with some obvious properties that are later on used to justify deduction steps:

Lemma 2 *Let X, Y be component models.*

(i) *Let $x \in {}^*X$. Then $x \in {}^*(XY)$ and $^*(XY)(x) = {}^*X(x)$.*
(ii) *Let $y \in Y^*$. Then $y \in (XY)^*$ and $(XY)^*(y) = Y^*(y)$.*
(iii) *Let $y \in {}^*Y$ and $\max(X^*) \geq {}^*Y(y)$. Then there exists a unique $x \in X^*$ with $\lambda_{^*Y}(y) = \lambda_{X^*}(x)$, $X^*(x) = {}^*Y(y)$ and $(x, y) \in \text{inner}(XY)$.*
(iv) *Let $y \in {}^*Y$ and let $\max(X^*) < {}^*Y(y)$. Then $y \in {}^*(XY)$ and $^*(XY)(y) = \max({}^*X) + {}^*Y(y) - \max(X^*)$*
(v) *Let $x \in X^*$ and $X^*(x) > \max(Y^*)$. Then $x \in (XY)^*$ and $(XY)^*(x) = X^*(x) + \max(Y^*) - \max({}^*X)$.*
(vi) *Let $\max(X^*) > \max({}^*Y)$. Then $\max({}^*(XY)) = \max({}^*X)$.*
(vii) *Let $\max(X^*) \geq \max({}^*Y)$. Then $\max((XY)^*) = \max(Y^*) + \max(X^*) - \max({}^*X)$.*
(viii) *Let $\max(X^*) < \max({}^*Y)$. Then $\max({}^*(XY)) = \max({}^*X) + \max({}^*Y) - \max(X^*)$.*
(ix) *Let $\max(X^*) < \max({}^*Y)$. Then $\max((XY)^*) = \max(Y^*)$*

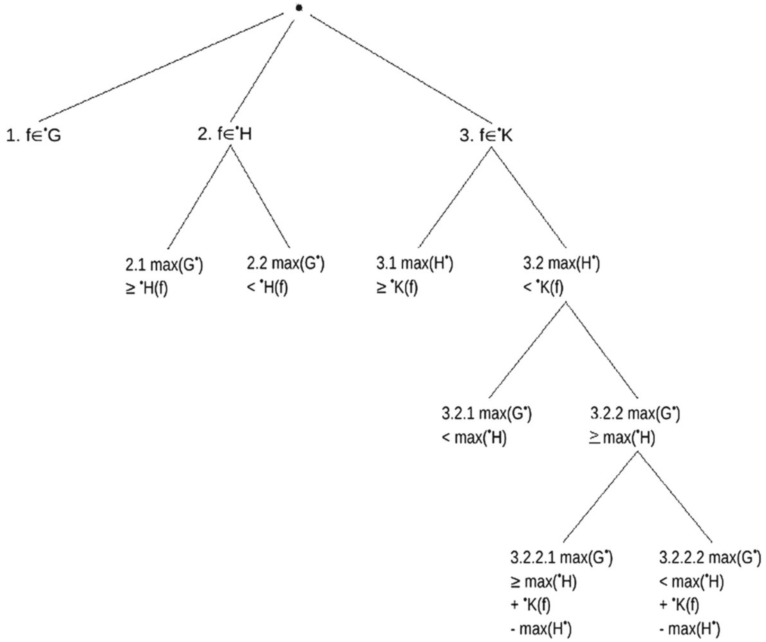


Fig. 10 Structure of the proof of (14)

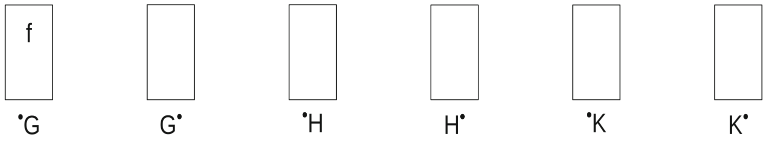


Fig. 11 Case 1

(x) $\max({}^*X) \le \max({}^*(XY))$.
(xi) *Let* $f \in X^*$. *Then* $\max(X^*) \ge X^*(f)$.

For the proof of (14) and (18) we distinguish seven cases, structured as shown in Fig. 10. Property (24) applies to cases 2.1, 3.1 and 3.2.2.1. For all other cases we show property (23). Figures outline the (increasingly more involved) proof steps of the seven cases.

Case 1 $f \in {}^*G$ (Fig. 11). We prove (23):

a) $f \in {}^*(G(HK))$ and ${}^*(G(HK))(f) = {}^*G(f)$, by Case 1 and Lemma 2(ii).
b) $f \in {}^*(GH)$ and ${}^*(GH)(f) = {}^*G(f)$, by Case 1 and Lemma 2(ii). Then $f \in {}^*((GH)K)$ and ${}^*((GH)K)(f) = {}^*G(f)$, by Lemma 2(i).

Case 2 $f \in {}^*H$.

Case 2.1 $\max(G^*) \ge {}^*H(f)$ (Fig.12). We prove (24).
There exists a unique $g \in G^*$ with $\lambda_{G^*}(g) = \lambda_{{}^*H}(f)$ and $G^*(g) = {}^*H(f)$, by Case 2.1 and Lemma 2(iii).
a) $f \in {}^*(HK)$ and ${}^*(HK)(f) = {}^*H(f)$, by Case 2 and Lemma 2(i).
Then $\max(G^*) \ge {}^*(HK)(f)$, by Case 2.1. Then $(g, f) \in \mathrm{inner}(G(HK))$, by Case 2 and Lemma 2(iii).

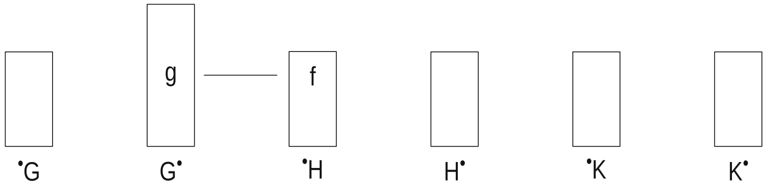


Fig. 12 Case 2.1

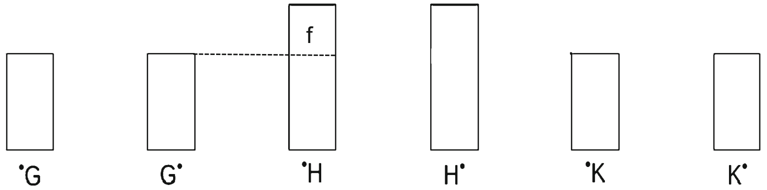


Fig. 13 Case 2.2

b) $(g, f) \in \text{inner}(GH)$, by Case 2, Case 2.1 and Lemma 2(iii). Then $(g, f) \in \text{inner}((GH)K)$, by Definition 4.

Case 2.2 $\max(G^*) < {}^*H(f)$ (Fig. 13). We prove (23).
As a shorthand, let

$$n =_{def} \max({}^*G) + {}^*H(f) - \max(G^*). \tag{25}$$

a)

$$f \in {}^*(HK) \text{ and } {}^*(HK)(f) = {}^*H(f) \tag{26}$$

by Case 2 and Lemma 2(i).
Then $\max(G^*) < {}^*(HK)(f)$, by Case 2.2. Then by Lemma 2(iv) $f \in {}^*(G(HK))$ and

$$\begin{aligned}&{}^*(G(HK))(f)\\ &= \max({}^*G) + {}^*(HK)(f) - \max(G^*), \text{ by Lemma 2}(iv)\\ &= \max({}^*G) + {}^*H(f) - \max(G^*), \text{ by (26)}\\ &= n, \text{ by (25)}\end{aligned}$$

b) $f \in {}^*(GH)$ and

$$^*(GH)(f) = \max({}^*G) + {}^*H(f) - \max(G^*) \tag{27}$$

by Case 2 and Lemma 2*(iv)*.
Then $f \in {}^*((GH)K)$ and

$$\begin{aligned}&{}^*((GH)K)(f)\\ &= {}^*(GH)(f), \text{ by Lemma 2}(i)\\ &= \max({}^*G) + {}^*H(f) - \max(G^*), \text{ by (27)}\\ &= n, \text{ by (25).}\end{aligned}$$

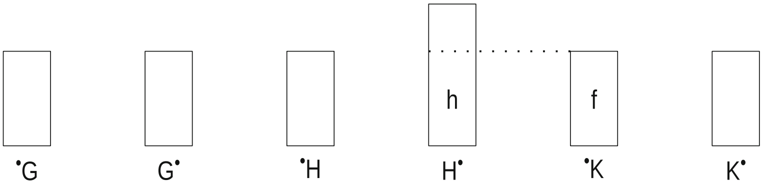


Fig. 14 Case 3.1

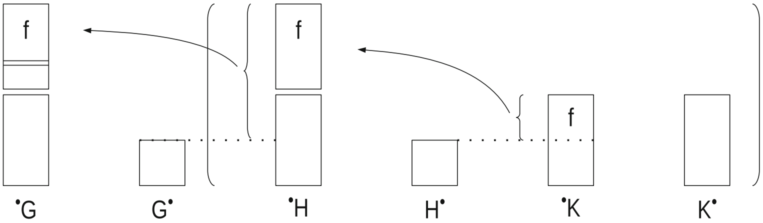


Fig. 15 Case 3.2.1 (a)

Case 3 $f \in {}^*K$.

Case 3.1 $\max(H^*) \geq {}^*K(f)$ (Fig. 14). We prove (24).

There exists a unique $h \in H^*$ with

$$H^*(h) = {}^*K(f) \tag{28}$$

and $(h, f) \in \text{inner}(HK)$, by case 3, case 3.1 and Lemma 2(iii).

a) Then $(h, f) \in \text{inner}(G(HK))$ by Definition 4.

b) $f \in {}^*K$ and $\max(H^*) \geq {}^*K(f)$, by case 3 and case 3.1.
Then $f \in {}^*K$ and $\max((GH)^*) \geq {}^*K(f)$, by (21), and, obviously, $H^* \subseteq (GH)^*$.
Then there exists a unique $g \in (GH)^*$ with $\lambda_{{}^*K}(f) = \lambda_{(HK)^*}(g)$ and $(GH)^*(g) = {}^*K(f)$ and $(g, f) \in \text{inner}((GH)K)$, by Lemma 2(iii).
Furthermore, $g = h$, because $\delta_{(GH)^*}$ is injective.
Then $(h, f) \in \text{inner}((GH)K)$.

Case 3.2 $\max(H^*) < {}^*K(f)$.

In the rest of this proof, let

$$n =_{\text{def}} \max({}^*G) + \max({}^*H) + {}^*K(f) - \max(G^*) - \max(H^*) \tag{29}$$

Furthermore it holds:

$$f \in {}^*(HK)\text{and}\,{}^*(HK)(f) = \max({}^*H) + {}^*K(f) - \max(H^*) \tag{30}$$

by 3.2 and Lemma 2(iv).

Case 3.2.1 $\max(G^*) < \max({}^*H)$ (Figs. 15, 16). We prove (23).

a) $f \in {}^*(HK)$ and $\max(G^*) < {}^*(HK)(f)$, by (29) and (30).
Then $f \in {}^*(G(HK))$ and

$$\begin{aligned}
&{}^*(G(HK))(f)\\
&\quad = \max({}^*G) + {}^*(HK)(f) - \max(G^*)\ \text{, by (30) and Lemma 2(iv)}\\
&\quad = \max({}^*G) + \max({}^*H) + {}^*K(f) - \max(H^*) - \max(G^*)\ \text{, by (30)}\\
&\quad = n\ \text{, by (29)}
\end{aligned}$$

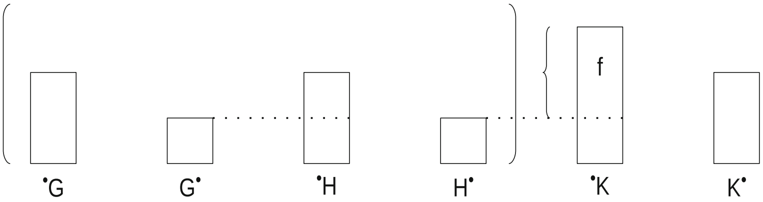


Fig. 16 Case 3.2.1 (b)

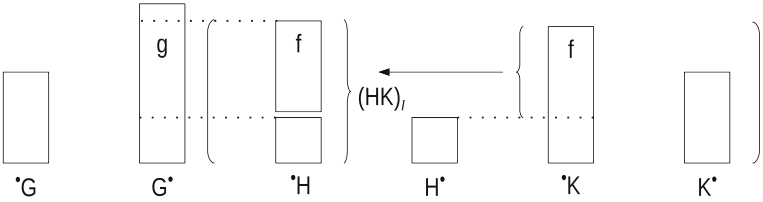


Fig. 17 Case 3.2.2.1 (a)

b) $\max((GH)^*) = \max(H^*)$, by 3.2.1 and Lemma 2(ix). Then $\max((GH)^*) < {}^*K(f)$ by 3.2. Then $f \in {}^*((GH)K)$ and

$$\begin{aligned}
&{}^*((GH)K)(f)\\
&= \max({}^*(GH)) + {}^*K(f) - \max((GH)^*), \text{ by case 3 and Lemma 2(iv)}\\
&= \max({}^*G) + \max({}^*H) - \max(G^*) + {}^*K(f) - \max((GH)^*),\\
&\quad \text{by Lemma 2(viii)}\\
&= \max({}^*G)+\max({}^*H) - \max(G^*)+{}^*K(f)-\max(H^*), \text{ by Lemma 2(ix)}\\
&= n, \text{ by (29).}
\end{aligned}$$

Case 3.2.2 $\max G^* \geq \max({}^*H)$.
Case 3.2.2.1 $\max(G^*) \geq \max({}^*H) + {}^*K(f) - \max(H^*)$ (Figs. 17, 18). We prove (24).

There exists a unique $g \in G^*$ with

$$G^*(g) = \max({}^*H) + {}^*K(f) - \max(H^*), \tag{31}$$

by 3.2.2 and Lemma 2(v).

a) $f \in {}^*K$ and $\max(H^*) \geq {}^*K(f)$, by case 3 and case 3.2.
Then $f \in {}^*(HK)$ and ${}^*(HK)(f) = \max({}^*H) + {}^*K(f) - \max(H^*)$, by Lemma 2(iv) and (31).
Then $(g, f) \in \text{inner}(G(HK))$, by Lemma 2(iii).

b) $G^*(g) > \max({}^*H)$ by (31) and case 3.2.
Then $g \in (GH)^*$, and $(GH)^*(g) = \max(H^*) + G^*(g) - \max({}^*H)$ by (31) and Lemma 2(v).
Then $g \in (GH)^*$ and $(GH)^*(g) = {}^*K(f)$, by (31).
Then $\max((GH)^*) \geq {}^*K(f)$ and $(GH)^*(g) = {}^*K(f)$ by Lemma 1.
Then $(g, f) \in \text{inner}((GH)K)$ by Lemma 2(iii).

Case 3.2.2.2 $\max(G^*) < \max({}^*H) + {}^*K(f) - \max(H^*)$ (Figs. 19, 20). We prove (23).

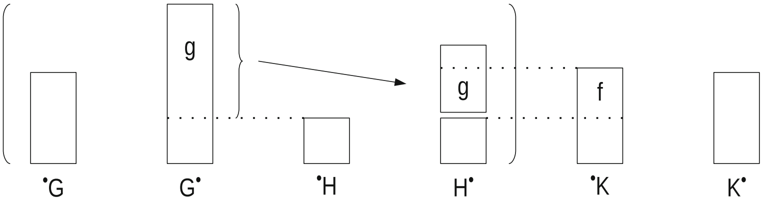


Fig. 18 Case 3.2.2.1 (b)

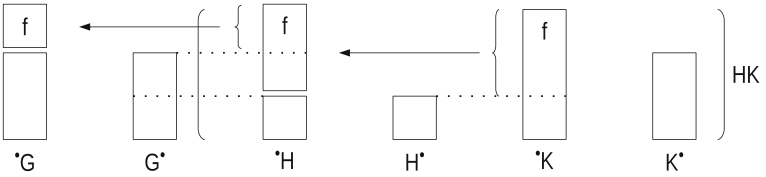


Fig. 19 Case 3.2.2.2 (a)

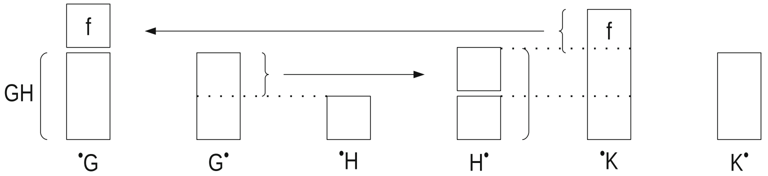


Fig. 20 Case 3.2.2.2 (b)

a) $f \in {}^*(HK)$ and ${}^*(HK)(f)$

$$= \max({}^*H) + {}^*K(f) - \max(H^*) \tag{32}$$

by case 3.2 and Lemma 2(iv).
Then $f \in {}^*(HK)$ and $\max(G^*) < {}^*(HK)(f)$, by case 3.2.2 and case 3.2.2.2.
Then $f \in {}^*(G(HK))$ and

$$\begin{aligned}
&{}^*((GH)K)(f)\\
&\quad= \max({}^*G) + {}^*(HK)(f) - \max(G^*) \text{ , by Lemma 2(iv)}\\
&\quad= \max({}^*G) + \max({}^*H) + {}^*K(f) - \max(H^*) - \max(G^*) \text{ , by (32)}\\
&\quad= n \text{ , by (29).}
\end{aligned}$$

b) $\max(G^*) + \max(H^*) - \max({}^*H) < {}^*K(f)$, by 3.2.2.2.
Then $\max((GH)^*) < {}^*K(f)$ by Lemma 2(vii). Then $f \in {}^*((GH)K)$ and

$$\begin{aligned}
&{}^*((GH)K)(f)\\
&\quad= \max({}^*(GH)) + {}^*K(f) - \max((GH)^*) \text{ , by case 3. and Lemma 2(iv)}\\
&\quad= \max({}^*G) + {}^*K(f) - \max((GH)^*) \text{ , by case 3.2.2 and Lemma 2(vi)}\\
&\quad= \max({}^*G) + {}^*K(f) - (\max(G^*) + \max(H^*) - \max({}^*H)) \text{ , by case}\\
&\qquad \text{3.2.2 and Lemma 2(vii)}\\
&\quad= n \text{ , by (29).}
\end{aligned}$$

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
2. Best, E., Devillers, R.R., Koutny, M.: Petri net algebra. In: Monographs in Theoretical Computer Science. An EATCS Series. Springer (2001)
3. Broy, M.: On architecture specification. In: Tjoa, A.M., Bellatreche, L., Biffl, S., van Leeuwen, J., Wiedermann, J. (eds.) SOFSEM 2018: Theory and Practice of Computer Science—44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29–February 2, 2018, Proceedings, vol. 10706 of Lecture Notes in Computer Science. Springer, pp. 19–39 (2018)
4. Broy, M., Krüger, I.H., Meisinger, M.: A formal model of services. ACM Trans. Softw. Eng. Methodol. **16**(1), 5 (2007)
5. Broy, M., Stølen, K.: Specification and Development of Interactive Systems-Focus on Streams, Interfaces, and Refinement. Monographs in Computer Science. Springer, Berlin (2001)
6. Dastani, M., Arbab, F., de Boer, F.S.: Coordination and composition in multi-agent systems. In: Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M.P., Wooldridge, M. (eds.) 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), July 25–29, 2005, Utrecht, The Netherlands. ACM, pp. 439–446 (2005)
7. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Tjoa, A.M., Gruhn, V. (eds.) Proceedings of the 8th European Software Engineering Conference Held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001., ACM, pp. 109–120 (2001)
8. Dong, W., Yu, H., Zhang, Y.: Testing bpel-based web service composition using high-level petri nets. In: Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16–20 October 2006, Hong Kong, China. IEEE Computer Society, pp. 441–444 (2006)
9. Esparza, J., Hoffmann, P.: Reduction rules for colored workflow nets. In: Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (2016), Stevens, P., Wasowski, A. Eds., vol. 9633 of Lecture Notes in Computer Science, Springer, pp. 342–358
10. Gößler, G., Graf, S., Majster-Cederbaum, M.E., Martens, M., Sifakis, J.: An approach to modelling and verification of component based systems. 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, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20–26, 2007, Proceedings, vol. 4362 of Lecture Notes in Computer Science. Springer, pp. 295–308 (2007)
11. Hamadi, R., Benatallah, B.: A petri net-based model for web service composition. In: Schewe, K., Zhou, X. (eds.) Database Technologies 2003, Proceedings of the 14th Australasian Database Conference, ADC 2003, Adelaide, South Australia, February 2003, vol. 17 of CRPIT. Australian Computer Society, pp. 191–200 (2003)
12. Hesami Rostami, N., Kheirkhah, E., Jalali, M.: Web services composition methods and techniques: a review. Int J Comp Sci Eng Inf Technol. **3** (2013)
13. Knapp, A., Marczynski, G., Wirsing, M., Zawlocki, A.: A heterogeneous approach to service-oriented systems specification. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C. (eds.) Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22–26, 2010. ACM, pp. 2477–2484 (2010)
14. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Schneider, F.B. (ed.) Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10–12, 1987. ACM, pp. 137–151 (1987)
15. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**(1), 70–93 (2000)
16. Meseguer, J., Montanari, U.: Petri nets are monoids. Inf. Comput. **88**(2), 105–155 (1990)
17. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture, 1st edn. O'Reilly Media, Inc., Newton (2016)
18. Nierstrasz, O., Achermann, F.: A calculus for modeling software components. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5–8, 2002, Revised Lectures, vol. 2852 of Lecture Notes in Computer Science. Springer, pp. 339–360 (2002)
19. Pautasso, C., Wilde, E.: Why is the web loosely coupled?: a multi-faceted metric for service design. In: Quemada, J., León, G., Maarek, Y.S., Nejdl, W. (eds.) Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20–24, 2009. ACM, pp. 911–920 (2009)

20. Polyvyanyy, A., Weidlich, M., Weske, M.: Connectivity of workflow nets: the foundations of stepwise verification. Acta Inf. **48**(4), 213–242 (2011)
21. Reisig, W.: Understanding Petri Nets—Modeling Techniques, Analysis Methods, Case Studies. Springer, Berlin (2013)
22. Reisig, W.: Towards a conceptual foundation of service composition. Comput. Sci. R&D **33**(3–4), 281–289 (2018)
23. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science. Springer, Berlin (2010)
24. Sassone, V.: On the category of petri net computations. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22–26, 1995, Proceedings, vol. 915 of Lecture Notes in Computer Science. Springer, pp. 334–348 (1995)
25. Sobocinski, P.: Nets, relations and linking diagrams. In: Heckel, R., Milius, S. (eds.) Algebra and Coalgebra in Computer Science—5th International Conference, CALCO 2013, Warsaw, Poland, September 3–6, 2013. Proceedings, vol. 8089 of Lecture Notes in Computer Science. Springer, pp. 282–298 (2013)
26. Stahl, C., Wolf, K.: Deciding service composition and substitutability using extended operating guidelines. Data Knowl. Eng. **68**(9), 819–833 (2009)
27. Tan, W., Fan, Y., Zhou, M.: A petri net-based method for compatibility analysis and composition of web services in business process execution language. IEEE Trans. Autom. Sci. Eng. **6**(1), 94–106 (2009)
28. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. Form. Asp. Comput. **23**(3), 333–363 (2011)
29. van Glabbeek, R.J., Goltz, U., Schicke-Uffmann, J.: On distributability of petri nets—(extended abstract). In: Birkedal, L. (ed.) Foundations of Software Science and Computational Structures—15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24–April 1, 2012. Proceedings, vol. 7213 of Lecture Notes in Computer Science. Springer, pp. 331–345 (2012)
30. Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: a model of service-oriented computation. In: Drossopoulou, S. (ed.) Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings, vol. 4960 of Lecture Notes in Computer Science. Springer, pp. 269–283 (2008)
31. Watzlawick, P., Bavelas, J., Jackson, D., O'Hanlon, B.: Pragmatics of Human Communication: A Study of Interactional Patterns, Pathologies and Paradoxes. W. W. Norton, New York (2011)
32. Wolf, K.: Does my service have partners? Trans. Petri Nets Other Models Concurr. **2**, 152–171 (2009)
33. Xiong, P., Fan, Y., Zhou, M.: A petri net approach to analysis and composition of web services. IEEE Trans. Syst. Man Cybern. Part A **40**(2), 376–387 (2010)
34. Yang, Y., Tan, Q., Xiao, Y.: Verifying web services composition based on hierarchical colored petri nets. In: Hahn, A., Abels, S., Haak, L. (eds.) Proceedings of the First International ACM Workshop on Interoperability of Heterogeneous Information Systems (IHIS'05), CIKM Conference, Bremen, Germany, November 4, 2005. ACM, pp. 47–54 (2005)