Chapter 1
Behavioral Service Substitution

Christian Stahl and Wil M. P. van der Aalst

Abstract Service-oriented design supports system evolution and encourages reuse and modularization. A key ingredient of service orientation is the ability to *substitute* one service by another without reconfiguring the overall system. This chapter aims to give an overview of the state of the art and open challenges in the area of *service substitution*. Thereby, we restrict ourselves to changes of the *service behavior*. We present a formal model of service behavior, formalize service substitution, study algorithms to decide service substitution, and provide rules to construct services that are correct by design. Beside analysis at *design time*, we also investigate analysis at *runtime*, where we measure the deviation of a running service (or collection of services) from its specification based on recorded event data (e.g., message or transaction logs).

1.1 Introduction

Today's enterprises are challenged to continuously change their systems to address changes in their environment. On the one hand, systems are highly complex, run in heterogeneous environments, and are often distributed over several enterprises. On the other hand, because of the extensively growing acceptance of the Internet and Internet-related technologies, enterprises consider themselves to be exposed to intense competition and, therefore, have to act dynamically and to change and adapt their systems whenever necessary. For example, when some new functionality is added or some quality parameter of some functionality is improved, this causes a change in system development. Instead of designing a system from scratch, existing systems need to be redesigned and improved iteratively.

Christian Stahl and Wil M. P. van der Aalst
Dept. of Mathematics and Computer Science, Technische Universiteit Eindhoven,
PO Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: C.Stahl|W.M.P.v.d.Aalst@tue.nl

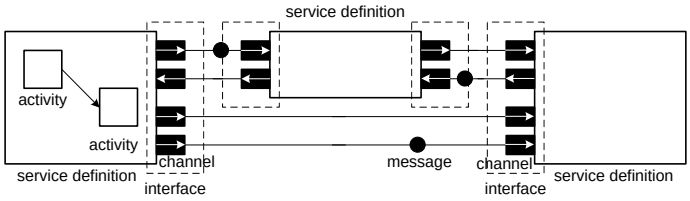


Fig. 1.1 An illustration showing the main terms used to describe services.

Service orientation [40] is a paradigm to design a complex distributed system by composing it from smaller building blocks called *services*. A service is an autonomous system that has an interface to interact with other services via message passing. As services are composed into more complex services, a service is usually *stateful*. The behavior of a service is described by a set of *activities*. An activity is the atomic unit of work in a service. The execution of an activity is either internal to the service or yields the sending or the receiving of a message. A service can be executed; that is, an *instance* of this service is created. An instance can execute activities. Figure 1.1 illustrates these terms.

An important property of a service composition is *compositionality*; that is, the composition is again a service. To achieve compositionality, a service composition must be *compatible*. The modular design of services enables enterprises to substitute one service by another one rather than changing the entire system. Substituting one service by another one should preserve compatibility of the overall system. Verification of compatibility is challenging, as one wants to derive correctness of the overall system from the correctness of the correctness of its services. *Service substitution*—that is, deciding whether a service can substitute another service—is considered to be one of the grand challenges [42].

In this chapter, we give an overview of the state of the art and open challenges in the area of service substitution. We thereby *restrict ourselves to changes of the service behavior*, which are also known as *business protocol changes* [41]. This restriction implies that we assume that nonfunctional and semantical properties are not violated when changing a service to another service; that is, we abstract from resources and consider only data and message types and not their content. Figure 1.2 illustrates how we approach this topic.

First, in Sect. 1.2, we formalize service behavior according to the illustration in Fig. 1.1. We introduce *open nets*, Petri nets extended with an interface, as a service model and formalize terms such as *compatibility*. Suitability of this model has been demonstrated by feature-complete open-net semantics for various languages such as BPMN and WS-BPEL [29].

In Sect. 1.3, we present two variants of service substitution and formalize them using a refinement relation between the specified (i.e., the old) service *Spec* and the implemented (i.e., the new) service *Impl*. A naive way to ensure correctness during service substitution is to compare the sets of all controllers (i.e., admissible contexts) of *Spec* and *Impl*. Only if all controllers of *Spec* are included in the set

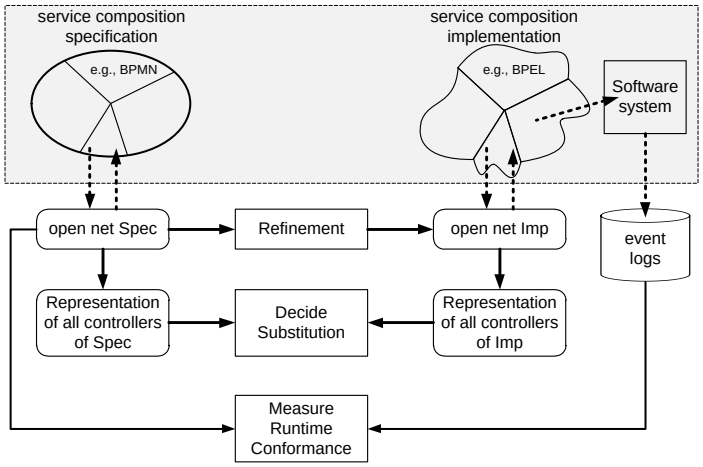


Fig. 1.2 Illustration of the proposed approach.

of controllers of *Impl*, compatibility is ensured. As these sets of controllers may be infinite, we introduce a *finite representation* of these sets to decide substitution. We also consider service substitution in a setting where a running instance of the old service has to be migrated to the new service.

Besides checking whether a service can actually be substituted by another one, one can also guide the construction of services that are *correct by design*. This approach can then be integrated in design tools and helps to speed up the design process. We investigate such techniques in Sect. 1.4.

Compatibility and substitutability can be studied at *design time* under the assumption that services behave as modeled. However, organizations may – deliberately or accidently – implement a different service or services may evolve over time. As a result, the real service behavior deviates from the modeled behavior. The availability of event logs of the actually implemented service *Impl* and the existence of the specified service *Spec* enables us to check *conformance at runtime*; that is, we investigate to what extent "the real *Impl*" deviates from *Spec*. In Sect. 1.5, we present techniques for *offline* conformance checking (diagnosis of deviations based on historic event data) and *online* conformance checking (generating alerts the moment a deviation occurs).

Section 1.6 concludes the chapter by summarizing our main findings and by discussing open research challenges.

1.2 Formalizing Service Behavior

Petri nets have proven to be successful for the modeling of business processes and workflows [8, 11]. In this section, we introduce our modeling formalism for ser-

vices, *open nets*. We focus on service behavior, and abstract from nonfunctional properties, semantical information, and data. As the formalism of open nets refines place/transition Petri nets, we first provide the basic definitions on Petri nets.

1.2.1 Basic Definition on Petri Nets

A Petri net [46] consists of two kinds of nodes, *places* and *transitions*, and a *flow relation* on nodes. Whilst transitions represent dynamic elements, for example an activity in a service, places represent static elements, such as causality between activities or an interface port. Graphically, a circle represents a place, a box represents a transition, and the directed arcs between places and transitions represent the flow relation. A *state* of the Petri net is represented by a marking. A marking is a distribution of tokens over the places. Graphically, a black dot represents a token.

Definition 1 (Net). A *net* $N = (P, T, F, m_N, \Omega)$ consists of

- a finite set P of *places*,
- a finite set T of *transitions* such that P and T are disjoint,
- a *flow relation* $F \subseteq (P \times T) \cup (T \times P)$,
- an *initial marking* m_N, where a marking $m \in \mathscr{B}(P)$ is a multiset over P, and
- a set Ω of final markings.

A *labeled net* is a net N together with an *alphabet* $\mathscr{A}$ of actions and a *labeling function* $l \in T \to \mathscr{A} \cup \{\tau\}$, where $\tau \notin \mathscr{A}$ represents an invisible, internal action.

Let $x \in P \cup T$ be a node of a net N. As usual, ${}^\bullet x = \{y \mid (y,x) \in F\}$ denotes the *preset* of x and $x^\bullet = \{y \mid (x,y) \in F\}$ the *postset* of x. We interpret presets and postsets as multisets when used in operations also involving multisets.

A marking $m \in \mathscr{B}(P)$ is a multiset over the set P of places; for example, $[p_1, 2p_2]$ denotes a marking m with $m(p_1) = 1$, $m(p_2) = 2$, and $m(p) = 0$ for $p \in P \setminus \{p_1, p_2\}$. We define $+$ and $-$ for the sum and the difference of two markings and $=, <, >, \leq, \geq$ for comparison of markings in the standard way. If $m_1 \in \mathscr{B}(P_1)$ and $m_2 \in \mathscr{B}(P_2)$, then $m_1 + m_2 \in \mathscr{B}(P_1 \cup P_2)$ (i.e., the underlying set of elements is adjusted when needed).

The *behavior* of a net N relies on changing the markings of N by firing transitions of N. A transition $t \in T$ is *enabled* at a marking m, denoted by $m \xrightarrow{t}$, if for all $p \in {}^\bullet t$, $m(p) > 0$. If t is enabled at m, it can *fire*, thereby changing the marking m to a marking $m' = m - {}^\bullet t + t^\bullet$. The firing of t is denoted by $m \xrightarrow{t} m'$; that is, t is enabled at m and firing it results in m'.

The behavior of N can be extended to sequences: $m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} m_k$ is a *run* of N if for all $0 < i < k$, $m_i \xrightarrow{t_i} m_{i+1}$. A marking m' is *reachable from* a marking m if there exists a (possibly empty) run $m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} m_k$ with $m = m_1$ and $m' = m_k$; for $w = \langle t_1 \ldots t_{k-1} \rangle$, we also write $m \xrightarrow{w} m'$. Marking m' is *reachable* if $m_N = m$. The set

M_N represents the set of all reachable markings of N, and the set of runs of N from the initial marking to a final marking is $Ru(N) = \{w \in T^* \mid \exists m_f \in \Omega : m_N \xrightarrow{w} m_f\}$.

In the case of labeled nets, we lift runs to traces: If $m \xrightarrow{w} m'$ and v is obtained from w by replacing each transition by its label and removing all τ-labels, we write $m \overset{v}{\Longrightarrow} m'$. For example, if $w = \langle t_1, t_1, t_2, t_1, t_2, t_3 \rangle$, $l(t_1) = a$, $l(t_2) = \tau$, and $l(t_3) = b$, and $m \xrightarrow{w} m'$, then $m \overset{v}{\Longrightarrow} m'$ with $v = \langle a, a, a, b \rangle$. We refer to v as a *trace* whenever $m_N \overset{v}{\Longrightarrow} m_f$ with $m_f \in \Omega$ and $Tr(N) = \{\sigma \in \mathscr{A}^* \mid \exists m_f \in \Omega : m_N \overset{\sigma}{\Longrightarrow} m_f\}$ is the set of all traces of N.

A net N is *bounded* if there exists a bound $b \in \mathbb{N}$ such that for all reachable markings $m \in M_N$ and $p \in P$, $m(p) \leq b$. A reachable marking $m \notin \Omega$ of N is a *deadlock* if no transition $t \in T$ of N is enabled at m. Net N is deadlock free if at least one transition of N is enabled at every reachable non-final marking.

1.2.2 Open Nets

A service consists of a control structure describing its behavior and an interface to communicate asynchronously with other services. Thereby an interface consists of a set of input and output *ports*. In order that two services can interact with each other, an input port of the one service has to be connected with an output port of the other service. These connected ports then form a *channel*.

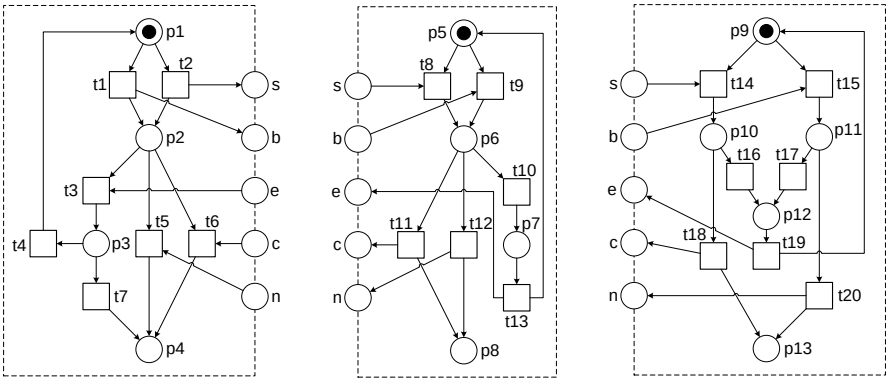
The control structure of a service can be adequately modeled as a net. We use the set of final markings of a net to model the states, in which a service may successfully terminate. In addition, it is necessary to model ports. To this end, we add an *interface* to our model. The service interface is reflected by two disjoint sets of *input and output places*. Thereby, each interface place corresponds to a port. An input place has an empty preset and is used for receiving messages from a distinguished channel, whereas an output place has an empty postset and is used for sending messages via a distinguished channel. In the model, we abstract from data and identify each message by the label of its message channel. The resulting service models are *open nets* [28, 53].

Definition 2 (Open net). An *open net* N is a tuple $(P, T, F, m_N, I, O, \Omega)$ with

- $(P \cup I \cup O, T, F, m_N, \Omega)$ is a net such that P, I, O are pairwise disjoint;
- for all $p \in I \cup O$, $m_N(p) = 0$, and for all $m \in \Omega$ and $p \in I \cup O$, $m(p) = 0$;
- the set I of *input places* satisfies for all $p \in I$, ${}^\bullet p = \emptyset$; and
- the set O of *output places* satisfies for all $p \in O$, $p^\bullet = \emptyset$.

If $I = O = \emptyset$, then N is a *closed net*. The net $inner(N)$ results from removing the interface places and their adjacent arcs from N. Two open nets are *interface equivalent* if they have the same sets of input and output places.

A closed net can be used to model a service choreography, whereas the inner of an open net reflects the interior of a service. Graphically, we represent an open net



(a) Customer N_C (b) Translator N_T (c) Translator N_B

Fig. 1.3 Open nets modeling a customer ($\Omega_C = \{[p_1], [p_4]\}$) and two translation services ($\Omega_T = \{[p_5], [p_8]\}$ and $\Omega_B = \{[p_9], [p_{13}]\}$).

like a net with a dashed frame around it. The interface places are depicted on the frame.

Figure 1.3 depicts our running example. It is a simplified (behavioral) model of a translation service. The example is inspired by the translation APIs offered by Bing and Google. The service in Fig. 1.3(b) receives a text file that must be translated. Our example allows customers to send a small or a large file, modeled by messages s and b, respectively. Depending on the pricing model used, the service asks for a cheap or a normal price (messages c and n). Sometimes the service may not work properly, for example, if too many requests are sent. In this case, the service sends an error message e. After having successfully translated a text file, the service enters a final state. In addition, also the initial state is a final state to allow customers to stop at any time if the translation service does not work properly. Figure 1.3(a) depicts the open net of a customer who may send small and large file and, in the case of an error, may send the file again or terminate (final marking $[p_1]$).

Communication between two services takes place by connecting pairs of ports using a channel and exchanging messages via these channels. We model this by composing the respective open nets, thereby merging shared interface places and turn these places into internal places. Such a merged interface place models a channel and a token on such a place corresponds to a pending message in the respective channel.

For the composition of open nets, we assume that the sets of transitions are pairwise disjoint and that no internal place of an open net is a place of any other open net. In contrast, the interfaces intentionally overlap. We require that all communication is *bilateral* and *directed*; that is, every shared place p has only one open net that sends into p and one open net that receives from p. We refer to open nets that fulfill these properties as *composable*.

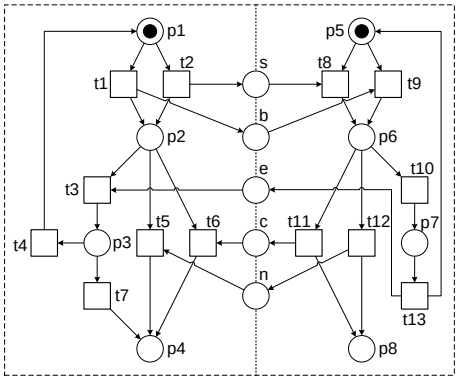


Fig. 1.4 Open net modeling a contract.

Definition 3 (Open net composition). Open nets N_1 and N_2 are *composable* if $(P_1 \cup T_1 \cup I_1 \cup O_1) \cap (P_2 \cup T_2 \cup I_2 \cup O_2) = (I_1 \cap O_2) \cup (I_2 \cap O_1)$. The *composition* of two composable open nets N_1 and N_2 is the open net $N_1 \oplus N_2 = (P, T, F, m_N, I, O, \Omega)$ where

- $P = P_1 \cup P_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1)$;
- $T = T_1 \cup T_2$;
- $F = F_1 \cup F_2$;
- $m_N = m_{N_1} + m_{N_2}$;
- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$;
- $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$; and
- $\Omega = \{m_1 + m_2 \mid m_1 \in \Omega_1 \wedge m_2 \in \Omega_2\}$.

Ignoring the dotted line along the former interface places, Fig. 1.4 shows the composition of the two open nets N_C and N_T of Fig. 1.3.

Open net composition models asynchronous message passing. Asynchronous message passing means that communication is nonblocking; that is, after a service has sent a message it can continue its execution and does not have to wait until this message is received. Furthermore, messages can 'overtake' each other; that is, the order in which the messages are sent is not necessarily the order in which they are received.

We want the composition of a set of services to be *compatible*. There exist a multitude of compatibility notions in the literature, making it is impossible to list them all. However, almost all of these notions can be classified into three dimensions:

- the composition terminates (e.g., deadlock freedom or the possibility to always reach a final marking, i.e., weak termination);
- the composition fulfills a scenario (e.g., a client must always pay by credit card); and
- the communication schema used by the composition (beside asynchronous communication following the idea of SOA, in practice also synchronous, queued, or mixed communications are implemented).

Throughout this chapter, compatibility refers deadlock freedom or weak termination. It will always be clear from the context, which of the two notions we consider. Compatibility is only of interest for a service choreography, which is modeled by a closed net. A user that communicates with a service, such that the composition is compatible, can also be seen as a *controller* of the service. In addition, we restrict ourselves to finite state services—more precisely, a composition with a controller must be bounded. The motivation for this restriction is that we model the control flow of service compositions and assume finitely many control states and a (finite) capacity of the message channels. For a reasonable concept of a service, we assume the inner of an open net, modeling the service interior, to be finite state. To ensure boundedness in the composition, controllers must not send a message if there the bound in the respective message channel has been reached already. Technically, this enforces that the composition has a finite number of reachable states. Pragmatically, it could either represent a reasonable buffer size in the middleware—for example, the result of a static analysis of the communication behavior of a service—or be chosen sufficiently large.

Definition 4 (Controllability). An open net C is a *controller* of an open net N if the composition $N \oplus C$ is closed, compatible (i.e., deadlock free or weakly terminating), and bounded. If such a C exist, then N is *controllable*.

If N is not controllable, then N is obviously ill-designed because it cannot properly interact with any other open net.

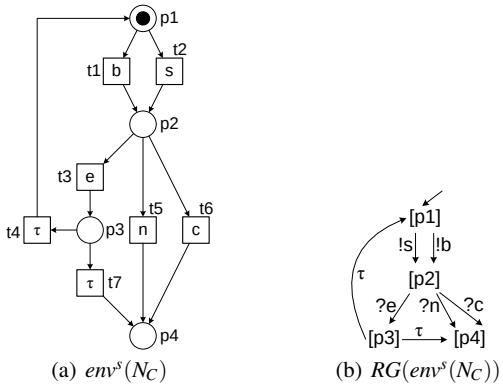
The contract in Fig. 1.4 is closed, 1-bounded, and weakly terminating (i.e., it is always possible to reach a final marking). As a consequence, open net N_C is a controller of open net N_T, and vice versa.

Later in this chapter, we want to analyze the behavior of a service modeled as an open net. The behavior of an open net is basically the reachability graph of its inner. To highlight which transitions are sending and receiving actions and which are only internal actions, we label each transition of an open net. The idea is to add to each transition adjacent to an interface place the respective place label and to all other transitions the label τ (denoting the internal action). To simplify the labeling, we restrict ourselves to open nets where each transition is connected to at most one interface place. We refer to those open nets as *elementary communicating*. That way, a transition is labeled by a single label rather than a set. This restriction is not significant as every open net can be transformed into an equivalent elementary communicating open net [28]. The respective inner is a labeled net to which we refer as *the synchronous environment*.

The behavior of an open net N can now be defined by the reachability graph $RG(env^s(N))$ of its synchronous environment. This graph has reachable markings $M_{env^s(N)}$ as its nodes and a $l(t)$-labeled edge from m to m' whenever $m \xrightarrow{t} m'$.

Figure 1.5 depicts the synchronous environment $env^s(N_C)$ of the customer service and its behavior described by the reachability graph of $env^s(N_C)$.

We introduced open nets as a formal model for service behavior. Open nets can be used to model service compositions, asynchronous communication, and proper termination. Open nets abstract from data and identify each message by the label of its



(a) $env^s(N_C)$ (b) $RG(env^s(N_C))$

Fig. 1.5 Synchronous environment of the customer and its reachability graph. Symbol "?" denotes a receiving and "!" a sending event of N_C.

message channel. This abstraction is necessary, because the analysis techniques that shall be introduced in the forthcoming sections are only applicable for finite state models. However, our approach allows to deal with finite data domains as those domains can be unfolded—for example, if a message returns a Boolean, then we could unfold this domain yielding two channels, one for exchanging value true and the other for value false. There also exist techniques to derive a finite abstraction from an infinite data domain, but they are out of scope for this chapter. To simplify our analysis techniques, open nets also abstract from time information. Moreover, open nets are a well-suited model for service behavior due to their link to workflows [8]. Nevertheless, also other modeling techniques have been successfully applied, for example, transition systems, finite automata, process algebra, and session types.

1.3 Service Substitution

In this section, we formalize multiparty contracts, introduce two substitutability notions and algorithms to decide these notions, and present variants of service substitution.

1.3.1 Multiparty Contracts and Accordance of Services

The service-oriented paradigm enables enterprises to publish their services via the Internet. These services can then be automatically found and used by other enterprises. However, this approach has not become accepted in practice, mainly because enterprises usually cooperate only with enterprises they already know. Therefore, in practice, a more pragmatic approach is used instead. The parties that will participate in an interorganizational cooperation specify together an abstract description of the overall service. This description is a choreography. The choreography consists

of a set of activities. Each activity is assigned to one party. A connection between two activities is either internal—that is, both activities belong to the same party—or external—that is, both activities belong to different parties. A party's share of the choreography (i.e., its *public view*) is then the projection of the choreography to the party's activities. The choreography serves as a common *contract* among the parties involved in the cooperation.

The challenge of the contract approach is to balance the following two conflicting requirements: On the one hand, there is a strong need for *coordination* to optimize the flow of work in and among the different parties. On the other hand, the parties involved in the cooperation are essentially *autonomous* and have the freedom to create or modify their services at any point in time. Furthermore, the parties do not want to reveal their trade secrets. Therefore, it has been proposed to use a contract that defines "rules of engagement" without describing the internal services executed within each party [2, 13].

After the parties have specified the contract, each party will implement its public view on its own. The implementation, the *private view*, will usually deviate significantly from its public view. Obviously, these local modifications have to *conform* to the agreed contract. This is, in fact, a nontrivial task, because it may cause global errors, such as deadlocks. As all parties are autonomous, none of them owns the overall service (i.e., the implemented contract). Therefore, none of the parties can verify the overall service. As a result, an approach is needed such that each party can check locally whether its private view guarantees global correctness of the overall service.

The basic idea of the contract approach can be seen in Fig. 1.2. The starting point is the specification on the top left which serves as a contract. It is partitioned into four parties, each illustrated as a fragment of the specification. By substituting each fragment by its implementation, we obtain the implementation on the top right.

Basically, we see a contract as a closed net N, where every transition is assigned to one of the involved parties $X_1, \ldots, X_k$. We impose only one restriction: If a place is accessed by more than one party, it should act as a directed bilateral communication place. This restriction reflects the fact that a party's public view of the contract is a service again. A contract N can be cut into parts $N_1, \ldots, N_k$, each representing the agreed public view of a single party X_i ($1 \leq i \leq k$). Hence, we define a contract as the composition of the open nets $N_1, \ldots, N_k$.

Definition 5 (Multiparty contract). Let $\mathscr{X} = \{X_1, \ldots, X_k\}$ be a set of parties. Let $\{N_1, \ldots, N_k\}$ be a set of pairwise interface compatible open nets such that $N = N_1 \oplus \cdots \oplus N_k$ is a closed net. Then, N is a *contract for* $\mathscr{X}$. For $i = 1, \ldots, k$, open net N_i is the *public view of* X_i *in* N.

Figure 1.4 shows a multiparty contract involving only two parties: a customer and a translation service. The dotted line is used to divide the composition into the two shares, open nets N_C and N_T.

As previously mentioned, we want that every party involved in the contract can independently substitute its public view N_i with a private view N_i'. Clearly, this substitution should not violate compatibility of the contact. Informally spoken, all the

other services forming the environment of N_i must not distinguish between N_i and N_i'. This can be achieved if every controller of N_i is also a controller of N_i'. This relation between the two open nets forms a refinement relation to which we refer as *accordance*.

Definition 6 (Accordance). Let *Spec* and *Impl* be interface equivalent open nets. Open net *Impl accords with* open net *Spec*, denoted by $Impl \sqsubseteq_{acc} Spec$, if every controller C of *Spec* is also a controller of *Impl*.

The main result for multiparty contracts is that each party can substitute its public view by a private view independently. If each of the private views accords with the corresponding public view, then compatibility (here deadlock freedom) of the implemented contract is guaranteed.

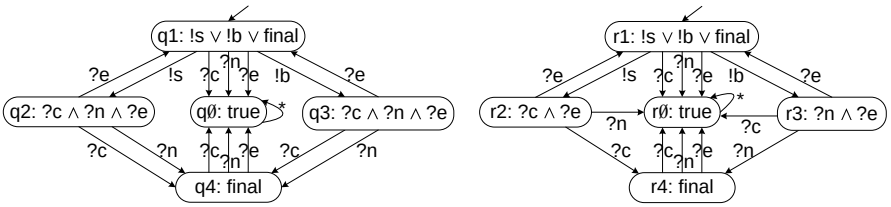
Theorem 7 (Implementation of a contract). *Let N be a contract between parties $\{X_1,\ldots,X_k\}$ where N is compatible. If, for all $i \in \{1,\ldots,k\}$, N_i' accords with N_i, then $N' = N_1' \oplus \cdots \oplus N_k'$ is compatible.*

Figure 1.3(c) depicts a private view N_B of the translation service N_T. This service implements a more concrete pricing model compared to N_T: Translating a small file is cheap whereas for large files a normal price is asked for. If we consider deadlock freedom as a compatibility notion, then N_B accords with N_T. With Theorem 7, we conclude that substituting N_T by N_B preserves deadlock freedom. Next we show, how we can decide accordance for two open nets.

1.3.2 Deciding Accordance Using Operating Guidelines

An algorithm to decide accordance for two open nets *Spec* and *Impl* must decide whether every controller of *Spec* is also a controller of *Impl*. As an open net has potentially infinitely controllers, we must check inclusion of two infinite sets. For *deadlock freedom*, we can overcome this problem, because the set of all controllers of open nets can be represented in a finite manner using a data structure called *operating guideline* [28].

An operating guideline $OG(N)$ of a service N describes how a user should successfully communicate with N; technically, it characterizes the behavior of the possibly infinite set of controllers of N in a finite manner. It is based on the observation that there exists a behavior that has the least restrictions, the *most permissive behavior*. The most permissive behavior is a deterministic transition system TS^* and serves as the first ingredient for our operating guideline. Every behavior of any controller is then just a restriction of TS^*. We can specify those restrictions by annotating every state of TS^* with a Boolean formula, specifying which of the outgoing transitions must be present. Thus, a literal of such a Boolean formula is a transition label of N or the literal *final*, specifying that N is in a final state. The Boolean formula are the second ingredient of an operating guideline. Technically, an operating guideline is an annotated automaton.



(a) $OG(N_T)$ (b) $OG(N_B)$

Fig. 1.6 Operating guidelines for the two translator services. Symbol $*$ is a short hand for every element of the alphabet.

Figure 1.6(a) shows the operating guideline of open net N_T. It is a finite automaton with five annotated states and can be read as follows. Initially, a controller of N_T either sends a (small or large) file or is in a final state. After sending for example, a small file (state q_2), the controller must be able to receive messages f,c, and n. The conjunction thereby emphasizes that any of these three messages can be sent and hence has to be received. Receiving f yields the initial state; receiving any of the other messages yields state q_4, where the controller must terminate. State $q_\emptyset$ denotes a nonreachable state: It does not harm if a controller can receive more messages than the service will send. For example, initially N_T will not send the translated file but it is not wrong if a controller can receive such a message.

To determine whether an open net C is a controller of an open net N, we check whether the behavior $TS_C = RG(env^s(C))$ of C matches with the operating guideline of N. Matching consists of two steps. First, we check whether TS_C is a potential restriction of TS^* by applying a procedure similar to a weak simulation relation—the difference is, whenever a τ-labeled transition can be performed in TS_C, then TS^* remains in the same state. Second, for each pair (q_C, q^*) of states in the relation, we verify whether the outgoing transitions of q_C and the information whether q_C is a final state or not evaluate the Boolean formula ϕ assigned to q^* to true. That way, we check whether TS_C is a valid restriction of TS^*.

Matching the behavior of the customer N_C with the operating guideline of N_T yields relation $\{([p_1],q_1),([p_2],q_2),([p_2],q_3),([p_3],q_1),([p_4],q_1),([p_4],q_4)\}$. For example, in $([p_1],q_1)$, $[p_1]$ assigns true to all three literals of $\phi(q_1)$, thereby evaluating this formula to true. Similar, in $([p_2],q_2)$, $[p_2]$ assigns true to all three literals of $\phi(q_2)$, thereby evaluating this formula to true. The same holds for all other elements of this relation, and therefore we conclude that N_C matches with $OG(N_T)$—which must be the case because N_C is a controller of N_T.

An *operating guideline $OG(N)$ of an open net N is the annotated automaton that represents all controllers of N* [28]. With the operating guidelines $OG(Spec)$ and $OG(Impl)$ of two open nets *Spec* and *Impl*, we can decide whether *Impl* accords with *Spec*.

Theorem 8 (Deciding accordance [50]). *For open nets Spec and Impl, with operating guidelines $OG(Spec)$ and $OG(Impl)$, we have that Impl $\sqsubseteq_{acc}$ Spec iff there*

exists a minimal simulation ρ *of* $OG(Spec)$ *by* $OG(Impl)$ *and, for each pair of nodes* $(q_{Spec}, q_{Impl}) \in \rho$, $\phi(q_{Spec})$ *implies* $\phi(q_{Impl})$ *is a tautology.*

Intuitively, the existence of the minimal simulation relation [33] guarantees that $OG(Impl)$ simulates the behavior of every controller of *Spec*, and the implication of the formulae ensures that whenever a service deadlocks with *Impl* it does so with *Spec*. The operating guideline algorithm has been implemented in the tool Wendy [30] and the accordance check in the tool Cosme [31].

Consider the operating guidelines $OG(N_T)$ and $OG(N_C)$. The minimal simulation relation is $\rho = \{(q_1, r_1), (q_2, r_2), (q_3, r_3), (q_4, r_4), (q_\emptyset, r_\emptyset)\}$. For each element of ρ, the tautology holds; for example, for (q_2, r_2) we have $?c \wedge ?n \wedge ?f$ implies $?c \wedge ?f$ is a tautology. Thus, N_B accords with N_T.

1.3.3 Substitution in a Less Restrictive Setting

One of the main drivers for service evolution is that organizations have to increase their profit and therefore continuously improve their services. Service improvement includes figuring out bottlenecks and unprofitable lines of business. On the level of the service behavior, service improvement leads to restructuring the process. Restructuring may also result in excluding some business functionality. For example, the translation service may stop offering the translation of large files if this is not profitable. As a consequence, the improved service may have fewer controllers than the current service. Thus, another refinement notion than accordance is necessary to cope with this scenario. To this end, we introduce *preservation*, a refinement relation in which the implementation preserves only a subset of the controllers of the specification.

Definition 9 (Preservation). Let *Spec* and *Impl* be interface equivalent open nets and $\mathscr{C}$ be a set of controllers of *Spec*. Open net *Impl accords with open net Spec under preservation of* $\mathscr{C}$, denoted by $Impl \sqsubseteq_{acc,\mathscr{C}} Spec$, if every controller $C \in \mathscr{C}$ is also a controller of *Impl*.

If $Impl \sqsubseteq_{acc,\mathscr{C}} Spec$ and thus preserves $\mathscr{C} \subseteq Con(Spec)$, then *Impl* controls every $C \in \mathscr{C}$. This, however, is equivalent to *Impl* matches with the operating guideline for any C. For a finite set $\mathscr{C}$, we can represent the intersection of the individual sets of services represented by these operating guidelines as one operating guideline. Technically, this operating guideline has the synchronous product of the underlying transition systems as its structure and to every synchronized state we assign the conjunction of the respective Boolean formulae. The following theorem formalizes the informally sketched decision procedure.

Theorem 10 (Deciding preservation [50]). *Let Spec and Impl be interface equivalent open nets and* $\mathscr{C} = \{C_1, \ldots, C_k\}$ *be a set of controllers of Spec. Let* $OG(C_i)$, $1 \leq i \leq k$, *be the operating guideline of* C_i, *and let* $OG_\otimes$ *denote the product of all* $OG(C_i)$. *Then,* $Impl \sqsubseteq_{acc,\mathscr{C}} Spec$ *iff Impl matches with* $OG_\otimes$.

The limitation of this result is that set $\mathscr{C}$ must be finite. However, it is also possible to constrain the set of controllers of a service by excluding or enforcing certain scenarios. The idea is similar: A constraint can be modeled as an (annotated) automaton. By constructing the product of this automaton and the operating guideline of *Spec*, we can constrain the controllers of *Spec*; see [27, 50]. In addition, [52] shows how certain activities of a service (i.e., transitions in the respective open net) can be covered.

1.3.4 Accordance in a Purely Service-Oriented Setting

In this section, we investigate service substitution in a purely service-oriented setting where services are composed from other published services. We want to decide when a service *Spec* published by some party can be substituted by a modified version *Impl*. Clearly, service *Impl* must accord with *Spec*. The actual challenge is that the party (i.e., a service provider) does not even know in which environment its service is executed. To illustrate that this fact matters, consider a closed net $N \oplus N_1 \oplus Spec$ with N shares interface places with N_1 and *Spec* but N_1 and *Spec* do not share interface places. In the contract setting, the provider of *Spec* knows the public views of N_1 and N and hence can substitute *Spec* by an accordant *Impl*. In the current setting, the provider of *Spec* does not know N nor N_1, but the refinement of *Spec* must be compositional in the sense that an accordant *Impl* must guarantee that also the composition $Impl \oplus N$ accords with $Spec \oplus N$. This argumentation is more difficult, because the two latter open nets are not closed. A refinement relation, such as accordance, that satisfies this property is a *precongruence*. A precongruence is a preorder such that if two open nets *Spec* and *Impl* are related by the precongruence so are $Impl \oplus N$ and $Spec \oplus N$ for any composable open net N. In contrast, the refinement relation necessary for ensuring Theorem 7, does not need to be a precongruence but only a preorder. For more details on this setting and a precongruence result for accordance, we refer to [51].

1.3.5 Service Instance Migration

So far, we have considered service substitution on the level of the service definition. However, running services may have long running instances. An example is the service of a life insurance company. A new legal regulation may cause a service to change, while instances of this service have been running for decades. In this case, each running instance of the old service has to be migrated to an instance of the new service. This problem is known as *service instance migration*.

Given a running instance in a state q_{Spec} of *Spec*, instance migration is the task of finding some state q_{Impl} of *Impl* such that resuming the execution in state q_{Impl} does not affect any controller of *Spec*. We call the transition from q_{Spec} to q_{Impl} a *jumper*

transition. Clearly, there may be states q_{Spec} for which there does not exist a jumper transition to a state q_{Impl}. Sometimes it might be necessary to continue the instance on *Spec* until a state is reached, where a migration is then possible.

In [26], an algorithm based on operating guidelines has been proposed to calculate jumper transitions for accordance in the case of deadlock freedom. The algorithm has been implemented in the tool Mia. For the translation services N_T and N_B, the following pairs (m_T, m_B) would be calculated: $([p_5],[p_9])$, $([p_6],[p_{10}])$, $([p_6],[p_{11}])$, $([p_7],[p_{12}])$, and $([p_8],[p_{13}])$. For example, if the old instance is in marking $[p_7]$, then it can be migrated to marking $[p_{12}]$, thereby ensuring that no controller of N_T is affected by this migration.

1.3.6 Discussion and Related Work

We introduced two refinement relations, accordance and preservation, and a data structure to decide these relations. The algorithmic solutions are tailored to deadlock freedom, but in recent (yet unpublished) work the procedure has been lifted to weak termination. Case studies in [26,30] show that our approach is applicable for service models of industrial size. To improve the efficiency of the accordance check, a more compact operating guideline representation has been proposed in [31]. An operating guideline can also be encoded as an automaton without annotations, called the maximal controller in [34]. This allows for another decision procedure for accordance: the composition of the maximal controller of service *Spec* and service *Impl* must be deadlock free. Kaschner and Wolf [23] showed that all noncontrollers of a service can be represented in a finite manner. This result (i.e., controller negation) together with the product of operating guidelines (i.e, intersection of sets of controllers) and an emptiness check yields an algebra on sets of controllers that generalizes the techniques for deciding preservation [23]. An approach to tackle preservation for more general properties has been proposed in [39,43].

Closest to our work is the work of Vogler [53] who considers a more general notion of composition. Different refinement relations in a process-algebraic setting have been investigated, for example, in [17,20,25]. The termination criterion is usually stronger than deadlock freedom but the communication schema is mostly synchronous. Bravetti and Zavattaro [18] consider different communications schemes for weak termination. Benatallah et al. [16] investigate accordance and preservation in a synchronous setting and in [44], results for a timed model are presented. Dumas et al. [19] investigate accordance and preservation using the more expressive π-calculus.

Instance migration has been studied by many researchers, in particular, in the field of workflows; see [5,45,47] for an overview. For services, several variants of this problem have been investigated in [49].

1.4 Constructing Substitutable Services

In the previous section, we presented an algorithm to decide for two given open nets *Spec* and *Impl* whether *Impl* accords with *Spec* and thus can substitute *Spec* without violating any controller of *Spec*. However, designing *Impl* is a nontrivial and error-prone task even for experienced service designers. In order to support service designers, we introduce an approach to construct open nets that are *correct by design*.

1.4.1 Approach

Given an open net *Spec*, we want to incrementally transform *Spec* to an open net *Impl* such that every transformation step preserves accordance by construction. To this end, fragments of *Spec* are incrementally substituted by other fragments. In this approach, a fragment Z of *Spec* is substituted by another fragment Z' yielding the open net *Impl*. We prove that if Z' accords with Z, then *Impl* accords with *Spec*.

An open net Z is a *fragment* of an open net N if there is an open net N_{rest} and the composition of Z and N_{rest} is the open net N. The set of interface places of Z is divided into two sets: some interface places of N and some internal places $R \cup S$ of N. We use R to denote these input places and S to denote these output places. For technical reasons, we require that the initial marking of Z is the empty marking and the set of final markings is the singleton set with the empty marking.

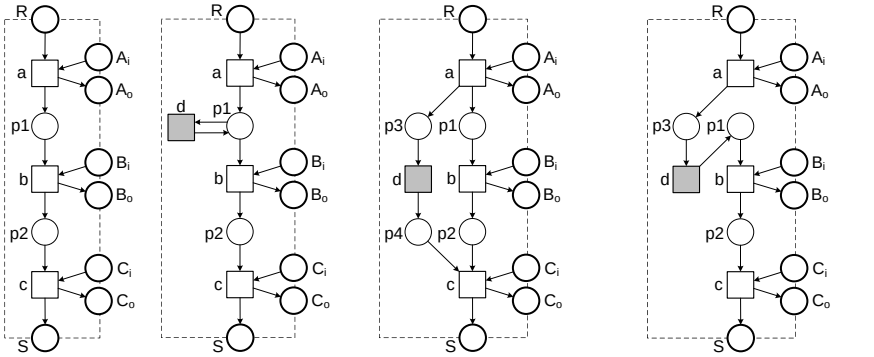
Definition 11 (Fragment). Let Z be an open net with $m_Z = 0$ and $\Omega = \{[]\}$. Open net Z is a *fragment* of an open net N if there exists an open net N_{rest} such that $N = Z \oplus N_{rest}$.

If an open net N has a fragment Z and there is another fragment Z' that accords with Z, then we can substitute Z by Z' without affecting any controller of N. Such transformations can be applied incrementally and thus refine a service specification to an implementation by applying transformation steps. The resulting implementation is correct by construction; that is, it preserves all controllers of the specification.

Theorem 12 (Justification [9]). *Let $N_1 \oplus N_2$ be a weakly terminating open net composition. Let Z be a fragment of N_1, and let N_{rest} be an open net such that $N_1 = Z \oplus N_{rest}$. For any open net Z' that accords with Z, the composition $(Z' \oplus N_{rest}) \oplus N_2$ is weakly terminating.*

1.4.2 Transformation Rules

The first three rules correspond to design patterns for extending a service to incorporate new behavior: (1) adding an internal loop, (2) putting a new internal transition in



(a) Z_0. (b) Z_1: Adding a loop to Z_0. (c) Z_2: Putting transition d in parallel to b. (d) Z_3: Inserting transition d in-between a and b.

Fig. 1.7 Transformation rules to change internal transitions: transition d is added (when applied left to right) or removed (when applied right to left).

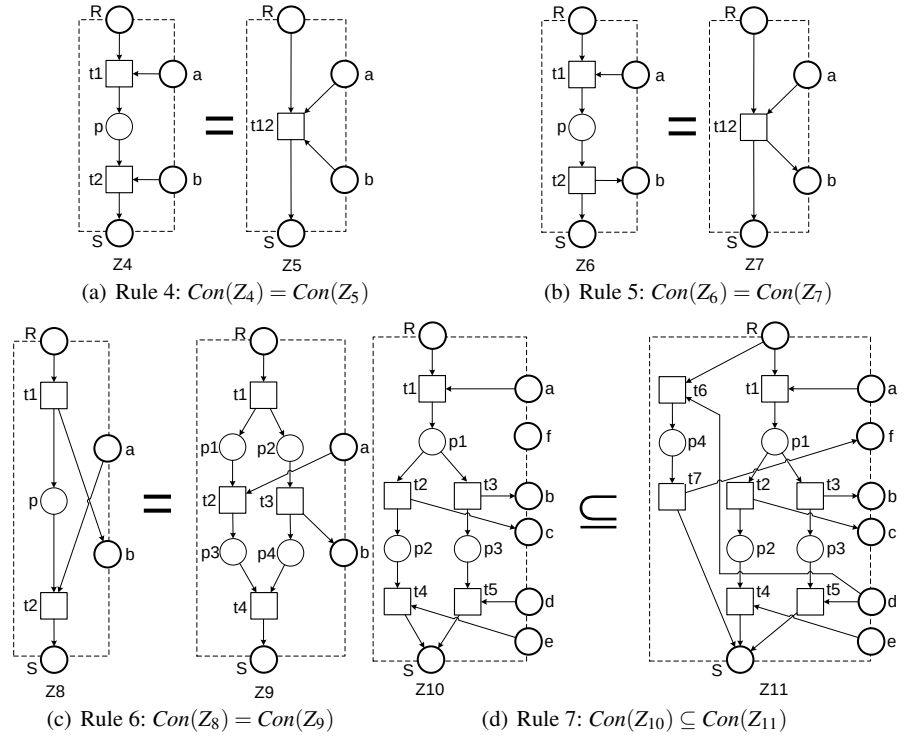
parallel with existing transitions, and (3) inserting an internal transition in-between existing transitions. These rules have been introduced in [5].

We exemplify these rules in Fig. 1.7. Figure 1.7(a) represents a fragment Z_0 of an open net N. Z_0 contains transitions a, b and c. By Definition 11, there are no other connections of a, b, c, $p1$ and $p2$ than those shown in Fig. 1.7(a). Each transition is connected to an input and an output place. However, as indicated by the capital letters, each interface place may correspond to a set of places. Further, A_i, A_o, B_i, B_o, C_i, C_o do not need to be disjoint. Places R and S denote the input and output places to N. Again, R and S may be sets of places. Similar remarks hold for the other three fragments Z_1, Z_2, and Z_3. For example, Z_1 is obtained by adding transition d to Z_0.

The three transformation rules only add (or remove if applied in reverse direction) internal transitions of an open net. However, there are also transformation rules that directly impact the interface behavior. We present four example transformation rules that affect transitions that are adjacent to an interface place.

Rule 4 is depicted in Fig. 1.8(a) and specifies that a sequence of receiving transitions can be merged, and the messages can be sent simultaneously. It is also possible to reorder a sequence of receiving transitions or to execute them concurrently (not shown). The same rule holds for a sequence of sending transitions. Rule 5 in Fig. 1.8(b) combines sending and receiving transitions. A receiving transition followed by a sending transition can be executed simultaneously. Due to Rule 4, Rule 5 can be generalized to a sequence of receiving transitions followed by a sequence of sending transitions. Rule 6, depicted in Fig. 1.8(c), specifies that first sending and then receiving a message can also be executed concurrently, and vice versa. Rules 4–6 preserve accordance in both directions.

This is in contrast to Rule 7 which specifies a way to add an alternative branch to a fragment Z_{10} depicted on the left hand side of Fig. 1.8(d). The fragment Z_{10} first receives a and then enters either the left or the right branch. In the left (right) branch,



(a) Rule 4: $Con(Z_4) = Con(Z_5)$ (b) Rule 5: $Con(Z_6) = Con(Z_7)$

(c) Rule 6: $Con(Z_8) = Con(Z_9)$ (d) Rule 7: $Con(Z_{10}) \subseteq Con(Z_{11})$

Fig. 1.8 Transformation rules to change interface transitions.

message b (c) is sent, and then message d (e) is received. The fragment M_{10} can be transformed into M_{11} by adding an alternative branch. In this branch, d is received, and then a message f is sent. Rule 7 preserves accordance in one direction only. The intuition behind this rule is that a controller of Z_{10} has to wait for the decision of Z_{10} which branch it will enter. Otherwise, it could happen that an environment sends d, but Z_{10} enters the left branch and waits for message e.

For an overview of all these rules and additional antipatterns, we refer to [9, 10].

1.4.3 Discussion and Related Work

We presented seven accordance-preserving transformation rules. Six of these rules preserve accordance in both directions and one rule preserves accordance only in one direction. Although these transformation rules are sound (i.e., correctness preserving), they are not complete, meaning they do not cover all possible service implementations. This is actually the weak point when dealing with transformations.

Refinement of Petri nets has been addressed by many researchers. However, most of the results require restricted Petri net classes or Petri nets without interfaces. The Murata rules [38] also maintain accordance, if we consider every input place as a place with some additional incoming arcs, and every output place as a place with some additional outgoing arcs. Refinement of places and transitions in Petri nets that preserves compatibility of the whole net is studied in [53]. These results could be applied in our setting. Soundness preserving transformation rules have been proposed in [1, 5, 54]. The rules proposed by Van Hee et al. [22] refine sets of places in service compositions, but they require additional reachability checks. In [24], the authors show how the presented rules can be translated into BPEL. That way, BPEL processes can directly be refined without transforming them into a formal service model.

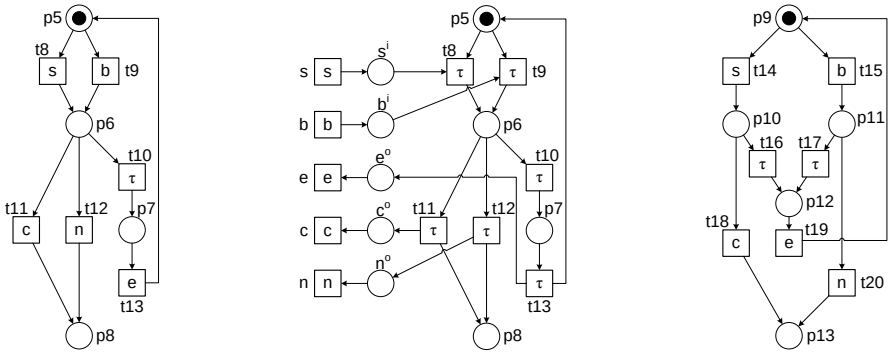
1.5 Conformance Checking of Services Based on Observed Behavior

Thus far, we only considered *modeled* service behavior. For example, we described requirements linking the public view of one party in the multiparty contract to the corresponding private view (implementation view). The analysis techniques did not consider actually observed behavior. However, in the context of services it is often not realistic to assume that all parties will indeed execute their processes as agreed upon at design time. Services may have been implemented incorrectly or change over time. Therefore, we now focus on *conformance checking based on events logs* (i.e., recorded behavior).

Process mining is a relatively young research discipline that sits between computational intelligence and data mining on the one hand, and process modeling and analysis on the other hand. Process mining research resulted in mature conformance checking techniques and tools that are able to align observed and modeled behavior [3, 4]. As a result it is possible to detect and quantify deviations.

In the remainder, we first introduce some basic process mining terminology. Then, we show how event log and model can be *aligned*. Based on this, we show how conformance checking techniques can be used to compare observed behavior (i.e., event data) with the public view of one or more parties in the multiparty contract. We also define conformance checking problems in a less restrictive setting where, from a behavioral point of view, parties may deviate from the contract as long as it does not harm the overall choreography.

Initially, we focus on conformance checking based on historic data (“offline” conformance checking). However, all techniques can be applied on-the-fly (“online” conformance checking); that is, streaming event data can be monitored at runtime and deviations can be detected immediately.



(a) $env^s(N_T)$ (b) $env^a(N_T)$ (c) $env^s(N_B)$

Fig. 1.9 Synchronous and asynchronous environment for open net N_T in Fig. 1.3(b) ($\Omega = \{[p_5],[p_8]\}$) and synchronous environment for open net N_B in Fig. 1.3(c) ($\Omega = \{[p_9],[p_{13}]\}$).

1.5.1 Process Mining

Process mining aims to *discover, monitor and improve real processes by extracting knowledge from event logs* readily available in today's information systems [3]. Starting point for process mining is an *event log*. Each event in such a log refers to an *activity* (i.e., a well-defined step in some process) and is related to a particular *case* (i.e., a *process instance*). The events belonging to a case are *ordered* and can be seen as one "run" of the process. An event log contains only *example* behavior; that is, we cannot assume that all possible runs have been observed. In fact, an event log often contains only a fraction of the possible behavior.

In this chapter, we define an event log as a multiset of *traces*. Each trace describes the life-cycle of a particular case in terms of the activities executed.

Definition 13 (Event, Trace, Event log). Let $\mathscr{A}$ be a set of activities. $\sigma \in \mathscr{A}^*$ is a *trace*, i.e., a sequence of events. $L \in \mathscr{B}(\mathscr{A}^*)$ is an *event log*, i.e., a multiset of traces.

In this simple definition of an event log, an event refers to just an *activity*. Often event logs may store additional information about events. For example, many process mining techniques use extra information, such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event (e.g., the size of an order). In this paper, we abstract from such information. An example log is $L = [\langle b,c\rangle^{10}, \langle s,e,s,c\rangle^5, \langle s,e,b,n\rangle^5]$. L contains information about 20 cases, e.g., 10 cases followed trace $\langle b,c\rangle$. There are $10\times 2+5\times 4+5\times 4=60$ events in total.

To relate event logs to process models, we use labeled nets. The behavior of such a model is described by the set $Tr(N)$ of traces which is computed from the set $Ru(N)$ of runs leading from the initial marking to a final marking; see Sect. 1.2.1 for a definition.

Figure 1.9(c) shows the labeled Petri net $env^s(N_B)$. The set of runs of $env^s(N_B)$ is the set $Ru(env^s(N_B)) = \{\langle t_{14}, t_{18}\rangle, \langle t_{15}, t_{20}\rangle, \langle t_{14}, t_{16}, t_{19}, t_{15}, t_{20}\rangle, \langle t_{15}, t_{17}, t_{19}, t_{14}, t_{18}\rangle, \ldots\}$. Every of these runs starts in $[p_9]$ and ends in $[p_{13}]$. The labeled net in Fig. 1.9(c) is weakly terminating, because all partial runs can be extended into a run in $Ru(env^s(N_B))$. If the labeled net has deadlocks or livelocks, then the problematic traces are simply discarded by $Ru(env^s(N_B))$.

Each trace in $Tr(env^s(N_B))$ corresponds to one or more runs in $Ru(env^s(N_B))$. A transition t is removed from the sequence if $l(t) = \tau$, otherwise it is replaced by $l(t)$. Therefore, $Tr(env^s(N_B)) = \{\langle s, c\rangle, \langle b, n\rangle, \langle s, e, b, n\rangle, \langle b, e, s, c\rangle, \ldots\}$ for the labeled net in Fig. 1.9(c). In $Tr(env^s(N_B))$, transitions are mapped onto their corresponding labels and τ transitions are not recorded; that is, t_{16} and t_{17} do not leave a trail in $Tr(env^s(N_B))$.

Event logs can be used to discover, monitor and improve services based on observations rather than hand-made models. There are three main types of process mining:

- *Discovery*: Take an event log and produce a model without using any other a-priori information. There are dozens of techniques to extract a process model from raw event data. For example, the classical α algorithm can discover a labeled net by identifying basic process patterns in an event log [12]. This algorithm takes an event log $L \in \mathscr{B}(\mathscr{A}^*)$ and produces a labeled net N. For many organizations it is surprising to see that existing techniques are indeed able to discover real processes based on merely example executions recorded in event logs. Process discovery is often used as a starting point for other types of analysis.
- *Conformance*: An existing process model is compared with an event log of the same process. For example, an event log $L \in \mathscr{B}(\mathscr{A}^*)$ is compared with the traces of some labeled net N. Ideally, any trace in L also appears in $Tr(N)$. Conformance checking reveals where the real process deviates from the modeled process. Moreover, it is possible to quantify the level of conformance and differences can be diagnosed. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model, and vice versa.
- *Enhancement*: Take an event log and process model and extend or improve the model using the observed events. Whereas conformance checking measures the alignment between model and reality, this third type of process mining aims at changing or extending the a-priori model. For instance, by using timestamps in the event log one can extend the model to show bottlenecks, service levels, throughput times, and frequencies [3].

In the remainder, we focus on the second type of process mining: conformance checking.

1.5.2 Conformance Checking Approaches

Conformance checking techniques investigate how well an event log $L \in \mathscr{B}(A^*)$ and a model—in our case a labeled net—fit together. Conformance checking can be done for various reasons; for example, it may be used to audit processes to see whether reality conforms to some normative or descriptive model [4,6,7,48]. Deviations may point to fraud, inefficiencies, and poorly designed or outdated procedures. *In the services setting, the different parties should operate in accordance with their public views.* Therefore, we elaborate on conformance checking techniques and show how they can be used to check the conformance of *running services*.

There are four quality dimensions for comparing model and log: (1) *fitness*, (2) *simplicity*, (3) *precision*, and (4) *generalization* [3, 4]. A model with good *fitness* allows for most of the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end. The *simplest* model that can explain the behavior seen in the log is the best model. This principle is known as Occam's Razor. Fitness and simplicity alone are not sufficient to judge the quality of a discovered process model. For example, it is very easy to construct an extremely simple labeled net ("flower model") that can replay all traces in an event log (but also any other event log referring to the same set of activities). Similarly, it is undesirable to have a model that only allows for the exact behavior seen in the event log. Remember that the log contains only example behavior and that many traces that are possible may not have been seen yet. A model is *precise* if it does not allow for "too much" behavior. Clearly, the "flower model" lacks precision. A model that is not precise is "underfitting". Underfitting is the problem that the model over-generalizes the example behavior in the log (i.e., the model allows for behaviors very different from what was seen in the log). At the same time, the model should generalize and not restrict behavior to just the examples seen in the log. A model that does not *generalize* is "overfitting". Overfitting is the problem that a very specific model is generated whereas it is obvious that the log only holds example behavior (i.e., the model explains the particular sample log, but there is a high probability that the model is unable to explain the next batch of cases).

In the remainder, we will focus on fitness. Ideally, all traces in the log correspond to a possible run of the model.

Definition 14 (Perfectly fitting log). Let $L \in \mathscr{B}(A^*)$ be an event log and let N be a labeled net. L is *perfectly fitting* N if $\{\sigma \in L\} \subseteq Tr(N)$.

Consider the event log $L = [\langle b,c\rangle^{20}, \langle s,e,s,c\rangle^{10}, \langle s,e,b,n\rangle^{10}]$. Clearly, L is perfectly fitting the labeled net $env^s(N_T)$ in Fig. 1.9(a) but it is not perfectly fitting the labeled net $env^s(N_B)$ in Fig. 1.9(c).

There are various ways to quantify fitness [3, 4, 15, 21, 32, 36, 37, 48], typically on a scale from 0 to 1, where 1 means perfect fitness. A naive approach would be to simply count the *fraction of fitting traces*. However, such an approach is too simplistic for two reasons:

- Whether traces in the log "almost" fit the model or not is irrelevant for such a metric. Traces $\sigma_1 = \langle e,b,c \rangle$ and $\sigma_2 = \langle e,e,e,b,c \rangle$ both do not fit the model in Fig. 1.9(c). However, it is obvious that σ_1 fits "better" than σ_2.
- It is important to also map the non-fitting behavior onto the model in order to do further analysis (performance analysis, predictions, etc.).

To address these issues, we need to *align* traces in the event log to traces of the process model. Some example alignments for L and the labeled net $env^s(N_B)$ are:

$$\gamma_1 = \begin{array}{|c|c|c|}\hline b & c & \gg \\ \hline b & \gg & n \\ t_{15} & & t_{20} \\ \hline \end{array} \quad \gamma_2 = \begin{array}{|c|c|c|c|c|}\hline b & \gg & \gg & \gg & c \\ \hline b & \tau & e & s & c \\ t_{15} & t_{17} & t_{19} & t_{14} & t_{18} \\ \hline \end{array} \quad \gamma_3 = \begin{array}{|c|c|c|c|c|}\hline s & \gg & e & s & c \\ \hline s & \tau & e & s & c \\ t_{14} & t_{16} & t_{19} & t_{14} & t_{18} \\ \hline \end{array} \quad \gamma_4 = \begin{array}{|c|c|c|c|c|}\hline s & \gg & e & b & n \\ \hline s & \tau & e & b & n \\ t_{14} & t_{16} & t_{19} & t_{15} & t_{20} \\ \hline \end{array}$$

The top row of each alignment corresponds to "moves in the log" and the bottom two rows correspond to "moves in the model". There are two bottom rows because there may be multiple transitions having the same label. If a move in the log cannot be mimicked by a move in the model, then a "$\gg$" ("no move") appears in the bottom row. For example, in γ_1 the model cannot do the c move. If a move in the model cannot be mimicked by a move in the log, then a "$\gg$" ("no move") appears in the top row. For example, all "silent steps" in the model (occurrences of τ transitions) cannot be mimicked by the event log. Moreover, in γ_2 the log did not do an "e move" whereas the model has to make this move to reach the end. Given a trace in the event log there may be many possible alignments. The goal is to find an alignment with the least number of $\gg$ elements, e.g., γ_1 is better than γ_2.

To establish an alignment between process model and event log, we need to relate "moves" in the log to "moves" in the model. However, as shown, there may be some moves in the log that cannot be mimicked by the model, and vice versa. For convenience, we introduce the set $A_L = \mathscr{A} \cup \{\gg\}$ where $x \in A_L \setminus \{\gg\}$ refers to "move x in log" and $\gg \in A_L$ refers to "no move in log". Similarly, we introduce the set $A_M = \{(a,t) \in \mathscr{A} \times T \mid l(t) = a\} \cup \{\gg\}$ where $(a,t) \in A_M$ refers to "move a in model" and $\gg \in A_M$ refers to "no move in model".

One step in an alignment is represented by a pair $(x,y) \in A_L \times A_M$ such that

- (x,y) is a *move in log* if $x \in \mathscr{A}$ and $y = \gg$,
- (x,y) is a *move in model* if $x = \gg$ and $y \in A_M \setminus \{\gg\}$,
- (x,y) is a *move in both* if $x \in \mathscr{A}$ and $y \in A_M \setminus \{\gg\}$, and
- (x,y) is an *illegal move* $x = \gg$ and $y = \gg$.

$A_{LM} = \{(x,y) \in A_L \times A_M \mid x \neq \gg \ \vee \ y \neq \gg\}$ is the set of all *legal moves*.

Let $\sigma_L \in L$ be a trace in the event log and let $\sigma_M \in Ru(N)$ be a run from the initial to a final marking of labeled net N. An *alignment* of σ_L and σ_M is a sequence $\gamma \in {A_{LM}}^*$ such that the projection on the first element (ignoring $\gg$) yields σ_L and the projection on the second element (again ignoring $\gg$ and only considering transitions and not the corresponding labels) yields σ_M. Consider again the four example alignments based on the labeled net $env^s(N_B)$. We represent the moves vertically, e.g., the first move of γ_1 is $(b,(b,t_{15}))$ indicating that both the log and the model

make a b move. γ_1 is an alignment of $\sigma_L = \langle b,c \rangle$ and $\sigma_M = \langle t_{15},t_{20} \rangle$. γ_2 is an alignment of $\sigma_L = \langle b,c \rangle$ and $\sigma_M = \langle t_{15},t_{17},t_{19},t_{14},t_{18} \rangle$.

To qualify the quality of an alignment, one can define a *distance function* on legal moves: $\delta \in A_{LM} \rightarrow \mathbb{N}$. The distance function associates costs to moves in an alignment:

- $\delta(a,\gg)$ is the cost of "move a in log" (with $a \in \mathscr{A}$),
- $\delta(\gg,(b,t))$ is the cost of "move b in model" (with $t \in T$ and $l(t) = b$), and
- $\delta(a,(b,t))$ is the cost of "move a in log and move b in model" (with $a \in \mathscr{A}$, $t \in T$ and $l(t) = b$).

Distance function δ can be generalized to alignments by taking the sum of the costs of all individual moves: $\delta(\gamma) = \sum_{(x,y)\in\gamma} \delta(x,y)$.[1]

We define a *standard distance function* δ_S. For $a \in \mathscr{A}$, $t \in T$, and $b = l(t)$: $\delta_S(a,\gg) = 1$, $\delta_S(\gg,(b,t)) = 1$ if $b \neq \tau$, $\delta_S(\gg,(b,t)) = 0$ if $b = \tau$, $\delta_S(a,(b,t)) = 0$ if $a = b$, and $\delta_S(a,(b,t)) = \infty$ if $a \neq b$. Only moves where log and model agree on the activity or internal τ moves of the model have no associated costs. Moves in just the log or model have cost 1. δ_S associates high costs to moves where both log and model make a move but disagree on the activity. In "$\delta_S(a,(b,t)) = \infty$", ∞ should be read as a number large enough to discard the alignment (see below). Using the standard distance function δ_S: $\delta_S(\gamma_1) = 2$, $\delta_S(\gamma_2) = 2$ (note that the move in model involves a τ transition), $\delta_S(\gamma_3) = 0$, and $\delta_S(\gamma_4) = 0$. So the sum of the costs is $\delta_S(\gamma) = 4$ for $env^s(N_B)$ and $\delta_S(\gamma) = 0$ for $env^s(N_T)$ (because L is a perfectly fitting log for $env^s(N_T)$). Note that δ_S is just an example; various cost functions can be defined.

Thus far we considered a *specific* run (from the initial to a final marking) in the model. However, our goal is to relate traces in the model to the *best matching* run in the model. Therefore, we define the notion of an *optimal alignment*. Let $\sigma_L \in L$ be a trace in event log L and let N be a labeled net. $\Gamma_{\sigma_L,N} = \{\gamma \in {A_{LM}}^* \mid \exists_{\sigma_M \in Ru(N)}\ \gamma \textit{ is an aligment of } \sigma_L \textit{ and } \sigma_M\}$. An alignment $\gamma \in \Gamma_{\sigma_L,N}$ is *optimal* for log trace $\sigma_L \in L$ and model N if for any $\gamma' \in \Gamma_{\sigma_L,N}$: $\delta(\gamma') \geq \delta(\gamma)$.

If $Ru(N)$ is not empty, there is at least one (optimal) alignment for any given log trace σ_L. However, there may be multiple optimal alignments for σ_L. Since our goal is to align traces in the event log to traces of the model, we deterministically select an arbitrary optimal alignment. Therefore, we can construct a function λ_M that provides an "oracle": Given a log trace σ_L, λ_M produces *one* best matching run from the initial to a final marking and hence to a best matching trace $\lambda_M(\sigma_L) \in Tr(N)$. In [14, 15], various approaches are given to create an optimal alignment with respect to some predefined distance function. These approaches are based on the A^* algorithm; that is, an algorithm originally invented to find the shortest path between two nodes in a directed graph. The A^* algorithm can be adapted to find an optimal alignment between model and log. The process mining framework *ProM* supports various techniques to create such an alignment and use this for conformance checking and other types of log-based analysis [4].

[1] Summation is generalized to sequences; that is, if the same step occurs k times in γ its costs are counted k times.

The alignments produced by the "oracle" λ_M can be used to quantify fitness (typically a number between 0 and 1). If a trace appears multiple times in the event log, the associated costs are also counted multiple times. Moreover, once an optimal alignment has been established for every trace in the event log, these alignments can be used as a basis to quantify precision and generalization [4, 37]. Such alignments are also a prerequisite for other types analysis (e.g., performance analysis) [3].

In the remainder, we assume a function *conf* that computes the fitness of an event log and a model based on an optimal alignment; that is, $conf(L,N)$ yields a number between 0 (poor fitness) and 1 (perfect fitness).

1.5.3 Conformance Checking of the Public View

Earlier we defined a *multiparty contract* as a set of pairwise interface compatible open nets $\{N_1,\ldots,N_k\}$ such that $N = N_1 \oplus \cdots \oplus N_k$ is a closed net. Each of the open nets N_i represents the *public view* of one of the parties. Party i can substitute its public view N_i by a private view N_i'. N_i' may refine N_i but may also change the ordering of some of the activities (see Fig. 1.8). However, ideally, the environment of N_i must not distinguish between N_i and N_i'.

For conformance checking, we need to compare observed behavior (i.e., recorded events in some log L) with modeled behavior (N_i or N_i'). In order to align observed behavior and modeled behavior, we need as input an event log $L \in \mathscr{B}(\mathscr{A}^*)$ and a labeled net $N = (P,T,F,m_N,\Omega,l)$. We cannot simply take some *public view* N_i as input for conformance checking. The public view is an open net with input places I and output places O. Transitions consuming from I are dead when checking the private view N_i in isolation. Tokens produced on places in O cannot be removed by N_i. Hence, $Tr(N_i)$ cannot contain sequences involving interface transitions. Moreover, events need to be related to transitions rather than places.

This triggers the question "What kinds of events can be observed?". Obviously, relevant events are related to the interface places $I \cup O$. However, given the asynchronous nature of open nets, we can take two viewpoints depending on what/when events are actually recorded.

If events are recorded when party i consumes a message from I or produces a message for O, then we can use the *synchronous environment* $env^s(N_i)$ of N_i. As before, we assume (without loss of generality) that a transition is connected to at most one interface place.

However, the environment of party i may be unable to see when a message is consumed from I or produced for O. For example, the environment can put a token in input place $p \in I$, but this does not imply that the token is immediately consumed by party i. Hence, we can only record the event of producing a token for input place p, but not the actual consumption. To this end, we we construct the asynchronous environment $env^a(N)$ of an open net N by adding to each interface place p of N a p-labeled transition in $env^a(N)$ and renaming the place p to p^I (p^O) if p is an input (output) place in N. Figure 1.9(b) illustrates this construction for open net N_T.

To illustrate the difference between both types of environments, consider Fig. 1.9 showing the synchronous environment $env^s(N_T)$ and the asynchronous environment $env^a(N_T)$ for public view N_T in Fig. 1.3(b). Which of the two labeled nets is most suitable, depends on the events that are recorded. Consider for example a message passed via input place s. If the event of consuming a message from interface place s is recorded, then $env^s(N_T)$ is more suitable. If the event of producing a message for interface place s is recorded, then $env^a(N_T)$ is more suitable.

The choice of environment matters. Consider for example trace $\langle s,b,e,c\rangle$ which is impossible according to $env^s(N_T)$ but allowed by $env^a(N_T)$. For any labeled net N: $Tr(env^s(N)) \subseteq Tr(env^a(N))$; that is, the asynchronous environment allows for more behavior and will be more "forgiving" under conformance checking. However, the proper choice of environment depends on what is actually logged. In the remainder, we will often abstract from these subtle differences and simply write $env(N)$.

Definition 15 (Public view conformance). Let $N = N_1 \oplus \cdots \oplus N_k$ be a multiparty contract and $i \in \{1,\ldots,k\}$ is one of the parties with public view N_i and event log L_i. $conf(L_i, env(N_i))$ is the *public view conformance* for party i.

This discussion thus far assumed that the environment of party i wants to check whether i conforms to its public view N_i. However, it is also possible to reverse roles in the multiparty contract $N = N_1 \oplus \cdots \oplus N_k$ and check whether the partners of i conform to $N_i^{-1} = \bigoplus_{j \neq i} N_j$. Depending what is actually logged on the interface between i and the other parties, one can use synchronous environment $env^s(N_i^{-1})$ or asynchronous environment $env^a(N_i^{-1})$.

Once an event log L and suitably labeled net N (e.g., $env^s(N_i)$, $env^a(N_i)$, $env^s(N_i^{-1})$, or $env^a(N_i^{-1})$) have been determined, we can align each trace in the log with the best fitting execution path of the service(s) under investigation. As discussed earlier, such an alignment can be used to compute a conformance value $conf(L,N)$.

1.5.4 Conformance Checking of the Private View

In the previous section, we showed that it is possible to check whether the observed behavior of party i (or its collaborators) is consistent with the behavior specified in the multiparty contract. However, such a check may be too strict in a services setting.

Theorem 7 shows that each party can substitute its public view N_i by a private view N_i' as long as N_i' accords to N_i. Rules 4, 5 and 6 in Fig. 1.8 illustrate that the notion of accordance is different from classical equivalence notions (e.g., trace equivalence). Parties may reorder activities without necessarily jeopardizing accordance. Therefore, it may be inappropriate to directly compare the observed behavior with the contract composed of public views. One party may have changed its behavior without jeopardizing compatibility. Therefore, we can also try to check conformance using some private view N_i' rather than the public view N_i.

Definition 16 (Private view conformance). Let $N = N_1 \oplus \cdots \oplus N_k$ be a multiparty contract and $i \in \{1,\ldots,k\}$ is one of the parties with public view N_i and event log L_i.

- $\mathscr{P}(N_i) = \{N \mid N \sqsubseteq_{acc} N_i\}$ is the set of all private views that accord with N_i,
- $N \in \mathscr{P}(N_i)$ is a *best matching private view* for N_i and L_i if for any $N' \in \mathscr{P}(N_i)$: $conf(L_i, env(N)) \geq conf(L_i, env(N'))$,
- $conf(L_i, env(N))$ is the *private view conformance* for party i where $N \in \mathscr{P}(N_i)$ is a best matching private view for N_i and L_i.

Definition 16 cannot easily be transformed into an algorithm. The process mining tool ProM provides excellent support for computing optimal alignments between log and model while allowing a variety of distance functions [4,15]. However, there may be many (if not infinitely many) private views that accord with N_i. Definition 16 provides a well-defined conformance notation that can be parameterized with different compatibility notions (e.g., deadlock freedom versus weak termination) and different environments (e.g., $env^s(N)$ or $env^a(N)$). First results to select a best matching private view have been presented in [35].

1.5.5 Beyond Conformance Checking

Conformance can computed by establishing an optimal alignment between an event log and a service model (public view or best matching private view). Moreover, such an alignment can also be used for various other purposes. If conformance is good, alignments will have a high proportion of "move in both" steps. This means that attributes of events can be mapped onto model elements. For example, in most event logs each event has a timestamp. These timestamps can be mapped onto transitions in the corresponding Petri net and can be used to compute how much time tokens spend in places. Since log and model are aligned, waiting times, response times, and service times can be measured easily. This may be used to discover bottlenecks, analyze service-level-agreements, etc. Some logs also contain information about costs, resource usage, errors, etc. Attributes at the level of individual events—just like timestamps—can be associated to model elements using the "move in both" steps.

In this section, we focused on offline conformance checking. For example, we assume a model N_i and an event log L_i containing historic data. However, alignments can be computed on-the-fly; that is, even partially executed traces can be aligned with a model (partial alignments do not need to end in a final marking, but a final marking should remain reachable). This enables *online conformance checking*; that is, streaming event data can be monitored at runtime and deviations can be detected immediately. Similarly, partially aligned traces can be used for *predictions* and *recommendations* at runtime. For example, for a partially handled case we can predict the remaining flow time, predict the probability of a deviation, or recommend a next activity that minimizes costs [3].

1.6 Conclusion

The shift toward service-oriented systems enables enterprises to decompose their systems into several smaller services. That way, service orientation enables for faster changes, because an individual service can be substituted by another service rather than changing the overall system. Service substitution, however, also imposes new challenges as it should not effect compatibility of the overall system. As systems may be distributed over several enterprises, system correctness has to be derived from the correctness of its parts, which is nontrivial.

In this chapter, we have surveyed service substitution at design time and at runtime, thereby restricting ourselves to the service behavior. We have investigated this problem on the level of service models. For design-time support, we introduced several variants of service substitution and illustrated that the problem is parameterized w.r.t. the compatibility notion used. To decide that a service *Impl* can substitute a service *Spec*, we must compare the infinitely many admissible contexts of *Spec* and *Impl*. We proposed decision algorithms based on a finite representation of these sets. In addition, we proposed rules to construct substitutable services that are correct by design. Research challenges are to generalize these techniques to other compatibility notions and, in addition, to incorporate data, time, and resources. Other directions are diagnosing why a service cannot serve as a substitute and ideally propose how it can be repaired.

In this chapter, we did not limit ourselves to comparing *models* of services but also considered the actual behavior recorded in message and transaction logs. The actual service implementation may deviate from its model or the behavior of a service may change over time. We showed that *conformance checking* techniques can be used to detect and diagnose deviations between *observed* service behavior (i.e., event logs) and *modeled* service behavior. As shown, we can define conformance at the level of the public view (contractual level) and at the level of the private view (implementation level). For the public view, we can apply existing conformance checking techniques. However, checking conformance with respect to some unknown implementation is more challenging and requires further research. Moreover, the lion's share of attention has gone to fitness analysis whereas the analysis of "underfitting" and "overfitting" of services is equally important.

References

1. Aalst, W.M.P.v.d.: Workflow Verification: Finding Control-Flow Errors using Petri-net-based Techniques. In: Business Process Management: Models, Techniques, and Empirical Studies, *LNCS*, vol. 1806, pp. 161–183. Springer (2000)
2. Aalst, W.M.P.v.d.: Inheritance of interorganizational workflows: How to agree to disagree without loosing control? Information Technology and Management Journal **4**(4), 345–389 (2003)
3. Aalst, W.M.P.v.d.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)

4. Aalst, W.M.P.v.d., Adriansyah, A., Dongen, B.v.: Replaying History on Process Models for Conformance Checking and Performance Analysis. WIREs Data Mining and Knowledge Discovery **2**(2), 182–192 (2012)
5. Aalst, W.M.P.v.d., Basten, T.: Inheritance of Workflows: An Approach to Tackling Problems Related to Change. Theoretical Computer Science **270**(1-2), 125–203 (2002)
6. Aalst, W.M.P.v.d., Dumas, M., Ouyang, C., Rozinat, A., Verbeek, H.: Conformance Checking of Service Behavior. ACM Transactions on Internet Technology **8**(3), 29–59 (2008)
7. Aalst, W.M.P.v.d., Hee, K., Werf, J.v.d., Verdonk, M.: Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor. IEEE Computer **43**(3), 90–93 (2010)
8. Aalst, W.M.P.v.d., Hee, K.M.v.: Workflow Management: Models, Methods, and Systems. The MIT Press, Cambridge, MA (2004)
9. Aalst, W.M.P.v.d., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: From public views to private views - correctness-by-design for services. In: WS-FM 2007, *LNCS*, vol. 4937, pp. 139–153. Springer (2008)
10. Aalst, W.M.P.v.d., Mooij, A.J., Stahl, C., Wolf, K.: Service interaction: Patterns, formalization, and analysis. In: SFM 2009, *LNCS*, vol. 5569, pp. 42–88. Springer (2009)
11. Aalst, W.M.P.v.d., Stahl, C.: Modeling Business Processes – A Petri Net-Oriented Approach. The MIT Press, Cambridge, MA (2011)
12. Aalst, W.M.P.v.d., Weijters, A., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering **16**(9), 1128–1142 (2004)
13. Aalst, W.M.P.v.d., Weske, M.: The P2P approach to interorganizational workflows. In: CAiSE 2001, *LNCS*, vol. 2068, pp. 140–156. Springer (2001)
14. Adriansyah, A., Dongen, B.F.v., Aalst, W.M.P.v.d.: Towards Robust Conformance Checking. In: BPI 2010, *LNBIP*, vol. 66, pp. 122–133. Springer (2011)
15. Adriansyah, A., Dongen, B.v., Aalst, W.M.P.v.d.: Conformance Checking using Cost-Based Fitness Analysis. In: EDOC 2011, pp. 55–64. IEEE Computer Society (2011)
16. Benatallah, B., Casati, F., Toumani, F.: Representing, Analysing and Managing Web Service Protocols. Data Knowl. Eng. **58**(3), 327–357 (2006)
17. Bravetti, M., Zavattaro, G.: Contract Based Multi-party Service Composition. In: FSEN 2007, *LNCS*, vol. 4767, pp. 207–222. Springer (2007)
18. Bravetti, M., Zavattaro, G.: Contract-based discovery and composition of web services. In: SFM 2009, *LNCS*, vol. 5569, pp. 261–295. Springer (2009)
19. Dumas, M., Yang, Y., Zhang, L.: Towards a formalization of contracts for service substitution. In: SERVICES 2010, pp. 423–430. IEEE Computer Society (2010)
20. Fournet, C., Hoare, C.A.R., Rajamani, S.K., Rehof, J.: Stuck-Free Conformance. In: CAV 2004, *LNCS*, vol. 3114, pp. 242–254. Springer (2004)
21. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust Process Discovery with Artificial Negative Events. Journal of Machine Learning Research **10**, 1305–1340 (2009)
22. Hee, K.M.v., Sidorova, N., Werf, J.M.E.M.v.d.: Refinement of synchronizable places with multi-workflow nets - weak termination preserved! In: PETRI NETS 2011, *LNCS*, vol. 6709, pp. 149–168. Springer (2011)
23. Kaschner, K., Wolf, K.: Set algebra for service behavior: Applications and constructions. In: BPM 2009, *LNCS*, vol. 5701, pp. 193–210. Springer (2009)
24. König, D., Lohmann, N., Moser, S., Stahl, C., Wolf, K.: Extending the compatibility notion for abstract ws-bpel processes. In: WWW 2008, pp. 785–794. ACM (2008)
25. Laneve, C., Padovani, L.: The must preorder revisited. In: CONCUR 2007, *LNCS*, vol. 4703, pp. 212–225. Springer (2007)
26. Liske, N., Lohmann, N., Stahl, C., Wolf, K.: Another approach to service instance migration. In: ICSOC 2009, *LNCS*, vol. 5900, pp. 607–621. Springer (2009)
27. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: BPM 2007, *LNCS*, vol. 4714, pp. 271–287. Springer (2007)
28. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007, *LNCS*, vol. 4546, pp. 321–341. Springer (2007)

29. Lohmann, N., Verbeek, H.M.W., Dijkman, R.: Petri net transformations for business processes – a survey. In: ToPNoC II, LNCS 5460, pp. 46–63. Springer (2009)
30. Lohmann, N., Weinberg, D.: Wendy: A tool to synthesize partners for services. Fundam. Inform. **113**, 1–17 (2011)
31. Lohmann, N., Wolf, K.: Compact representations and efficient algorithms for operating guidelines. Fundam. Inform. **108**(1-2), 43–62 (2011)
32. Medeiros, A.K.A.d., Weijters, A., Aalst, W.M.P.v.d.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery **14**(2), 245–304 (2007)
33. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc. (1989)
34. Mooij, A.J., Parnjai, J., Stahl, C., Voorhoeve, M.: Constructing replaceable services using operating guidelines and maximal controllers. In: WS-FM 2010, *LNCS*, vol. 6551, pp. 116–130. Springer (2011)
35. Müller, R., Aalst, W.M.P.v.d., Stahl, C.: Conformance checking of services using the best matching private view. In: WSFM 2012, LNCS. Springer (2012). Accepted for publication
36. Munoz-Gama, J., Carmona, J.: A Fresh Look at Precision in Process Conformance. In: BPM 2010, *LNCS*, vol. 6336, pp. 211–226. Springer (2010)
37. Munoz-Gama, J., Carmona, J.: Enhancing Precision in Process Conformance: Stability, Confidence and Severity. In: CIDM 2011. IEEE (2011)
38. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
39. Oster, Z.J., Basu, S.: Extending substitutability in composite services by allowing asynchronous communication with message buffers. In: ICTAI 2009, pp. 572–575. IEEE Computer Society (2009)
40. Papazoglou, M.P.: Web Services: Principles and Technology. Pearson - Prentice Hall, Essex (2007)
41. Papazoglou, M.P.: The challenges of service evolution. In: CAiSE 2008, *LNCS*, vol. 5074, pp. 1–15. Springer (2008)
42. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. Int. J. Cooperative Inf. Syst. **17**(2), 223–255 (2008)
43. Pathak, J., Basu, S., Honavar, V.: On Context-Specific Substitutability of Web Services. In: ICWS 2007, pp. 192–199. IEEE Computer Society (2007)
44. Ponge, J., Benatallah, B., Casati, F., Toumani, F.: Analysis and applications of timed service protocols. ACM Trans. Softw. Eng. Methodol. **19**(4) (2010)
45. Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. LNCS ToPNoC **5460**(II), 115–135 (2009)
46. Reisig, W., Rozenberg, G. (eds.): Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, *LNCS*, vol. 1491. Springer (1998)
47. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems - a survey. Data Knowl. Eng. **50**(1), 9–34 (2004)
48. Rozinat, A., Aalst, W.M.P.v.d.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems **33**(1), 64–95 (2008)
49. Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., Saint-Paul, R.: Supporting the dynamic evolution of web service protocols in service-oriented architectures. TWEB **2**(2) (2008)
50. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding substitutability of services with operating guidelines. In: ToPNoC II, LNCS 5460, pp. 172–191. Springer (2009)
51. Stahl, C., Vogler, W.: A trace-based service semantics guaranteeing deadlock freedom. Acta Inf. **49**(2), 69–103 (2012)
52. Stahl, C., Wolf, K.: Deciding service composition and substitutability using extended operating guidelines. Data Knowl. Eng. **68**(9), 819–833 (2009)
53. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets, *LNCS*, vol. 625. Springer (1992)
54. Wynn, M.T., Verbeek, H.M.W., Aalst, W.M.P.v.d., Hofstede, A.H.M.t., Edmond, D.: Soundness-preserving Reduction Rules for Reset Workflow Nets. Information Sciences **179**(6), 769–790 (2009)