Service Interaction: Patterns, Formalization, and Analysis

Wil M. P. van der Aalst[1], Arjan J. Mooij[1], Christian Stahl[1], and Karsten Wolf[2]

[1] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{W.M.P.v.d.Aalst, A.J.Mooij, C.Stahl}@tue.nl

[2] Universität Rostock, Institut für Informatik
18051 Rostock, Germany
Karsten.Wolf@uni-rostock.de

Abstract. As systems become more service oriented and processes increasingly cross organizational boundaries, interaction becomes more important. New technologies support the development of such systems. However, the paradigm shift towards service orientation, requires a fundamentally different way of looking at processes. This survey aims to provide some foundational notions related to service interaction. A set of service interaction patterns is given to illustrate the challenges in this domain. Moreover, key results are given for three of these challenges: (1) How to expose a service?, (2) How to replace and refine services?, and (3) How to generate service adapters? These challenges will be addressed in a Petri net setting. However, the results extend to other languages used in this domain.

Keywords: Service Orientation, Service Choreography, Open Nets, Verification, Service Interaction Patterns

1 Introduction

Information technology has changed business processes within and between enterprises. Traditionally, information technology was mainly used to support individual tasks ("type a letter") and to store information. However, today's business processes and their information systems are intertwined. Processes heavily depend on information systems and information systems are driven by the processes they support [1]. In the last decade, information systems have become *"process aware"*, i.e., processes are taken as the starting point [1].

At the same time, there is an increasing acceptance of *Service-Oriented Architectures* (SOA) as a paradigm for integrating software applications within and across organizational boundaries [2]. XML-based standards like SOAP and

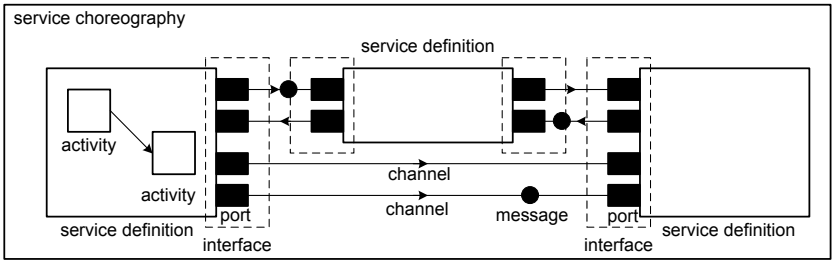


Fig. 1. An illustration showing the main terms used to describe services.

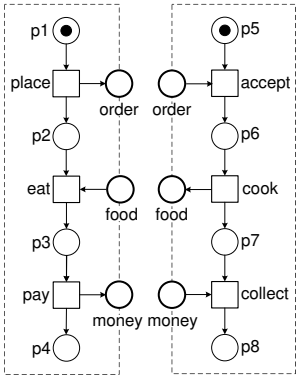
WSDL facilitate the realization of such loosely coupled architectures. Interestingly, SOA and associated technologies have blurred the classical distinction between intra-organizational processes and inter-organizational processes. Whether work is subcontracted to an internal service or an external service, is no longer relevant from a technological point of view.

However, the *importance of interaction is increasing as more and more monolithic systems are broken down into smaller services*. The importance of interaction has been stressed by many authors [3–8]. Moreover, interaction is also considered a first-class citizen in various industry standards. For example, the *Web Services Business Process Execution Language* (BPEL) has primitive activities such as *invoke* (invoking an operation on a web service), *receive* (waiting for a message from an external source), and *reply* (replying to an external source) [9]. Moreover, the *pick* construct that can be used for race conditions based on external triggers is clearly inspired by the needs of service interaction.

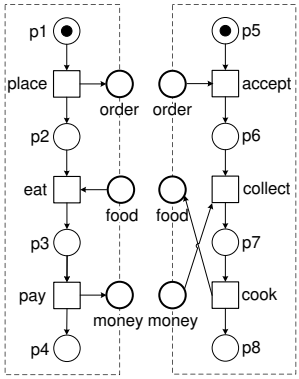
Foundations In this paper, we focus on the *foundations of service interaction*. We will not review the industry standards and associated tools. Instead we focus on fundamental concepts that are independent of a particular implementation language.

We first present some of the terms we will be using. A service has a *definition*. This definition describes the behavior and the interface of the service. A service can be *instantiated*. An instance corresponds to an execution of a service, and hence it can execute *activities*, and receive and send *messages*. Activities are the atomic units of work in a service and are specified in the service definition. The interface of a service consists of a set of *ports*. A pair of ports can be connected using a *channel*, thus enabling the exchange of *messages*. Services can be *composed* by connecting the interfaces. We use the term *service choreography* to refer to a set of fully-connected service definitions. Figure 1 illustrates these terms.

In Sect. 2 we categorize some recurring service interactions in terms of a set of *service interaction patterns*. Inspired by the Workflow Patterns Initiative (cf. `www.workflowpatterns.com`), in particular the control-flow patterns [10] and the set of interaction patterns [3] presented by Barros et al., we use a patterns-



(a) Two services: a guest service GS_1 and a friendly restaurant service RS_1



(b) Two services: a guest service GS_1 and an unfriendly restaurant service RS_2

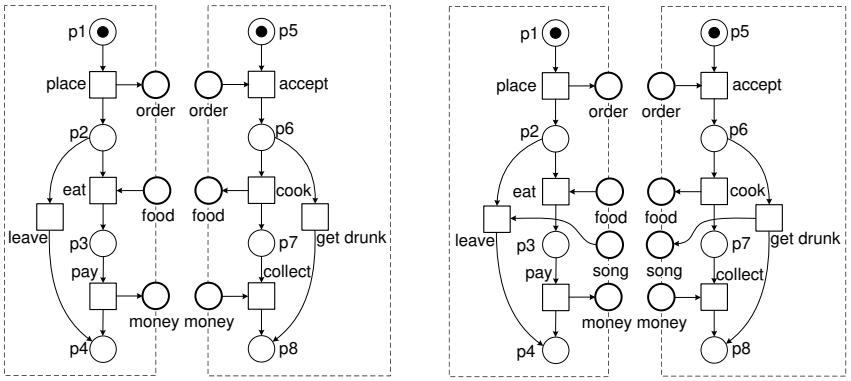
Fig. 2. Two pairs of services: $GS_1 \oplus RS_1$ and $GS_1 \oplus RS_2$.

based approach [10, 3] to introduce the foundational concepts and challenges of service interaction in a language and tool independent manner.

In Sect. 3 we introduce *open nets* [11] as a basic tool to explain and formalize services. Open nets are a refinement of Petri nets [12–14] with interface places for communication and designated initial and final markings. Open nets can be seen as a generalization of workflow nets [15], extended with communication and a more relaxed net structure. A service definition is modeled as an open net. Ports are modeled as interface places. Service definitions can be composed by merging interface places into channel places. These channel places are internal to the composed service. Messages correspond to tokens passed from one service to another via interface places. A service choreography is the result of composing a set of open nets such that all interface places become internal.

We define service composition in terms of open nets. The composition of services may lead to all kinds of (behavioral) problems, e.g., deadlocks, livelocks, inability to terminate, etc. Two or more services are *compatible* if their composition "behaves well", but there exist several notions of compatibility in the literature. Based on a particular compatibility notion, one can define *controllability*, i.e., "Does a service have a compatible service?".

Examples Let us look at some examples of open nets representing very simple "toy services". Figure 2(a) shows two service definitions. The composition of these two service definitions can be achieved by merging equally labeled interface places (depicted on the frame). Each of the service definitions corresponds to an open net and in the composition the interface places are fused. The open net on the left (GS_1) represents a guest that first places an order, eats the food, and finally pays. The three activities of this service, i.e., place, eat, and pay, are modeled in terms of transitions. These activities are connected to the other service via ports and channels. In our open-net representation these have been



(a) Two incompatible services GS_2 and RS_3

(b) Two compatible services GS_3 and RS_4

Fig. 3. Another two pairs of services: $GS_2 \oplus RS_3$ and $GS_3 \oplus RS_4$.

mapped onto the places order, food, and money. In this toy example, real objects are passed via the connecting places (e.g., food). In the context of web services, of course only messages are exchanged via such places. The open net on the right-hand side of Fig. 2(a) (RS_1) models the other service that consist of activities accept, cook, and collect. This is the "friendly restaurant service" RS_1, which can be seen as the obvious counterpart of GS_1. The composition of GS_1 and RS_1, denoted by $GS_1 \oplus RS_1$, has only one possible execution trace: place, accept, cook, eat, pay, and collect. It seems obvious that the two services are compatible, even without having a precise definition of compatibility in mind.

Consider Fig. 2(b) modeling two similar services consisting of the same activities. However, the restaurant service now is less friendly and requires the guest to pay before preparing the food (i.e., collect should occur before cook). This "unfriendly restaurant service" is named RS_2. The composition $GS_1 \oplus RS_2$ always runs into a deadlock, i.e., after executing place and accept both services are blocked waiting for one another. Clearly these two services are not compatible.

To illustrate the complexity and intricate subtleties of service interaction, consider Fig. 3(a). This time the guest may leave without eating and paying for the food. Moreover, the cook is an alcoholic and may get drunk instead of cooking the food. The resulting service composition $GS_2 \oplus RS_3$ clearly has problems. It may be the case that the customer leaves while the food has been or will be delivered. One may wonder which service would be compatible with RS_3, as it is unclear for the outside whether the food will be delivered or not. We will come back to this question in Sect. 3.

Figure 3(b) shows improved versions of the "potentially leaving guest service" and the "drunk cook restaurant service": GS_3 and RS_4. In the new service choreography the cook starts singing Irish folk songs when he gets drunk. As a result, the customer knows when to leave. The composition $GS_3 \oplus RS_4$ has only two possible execution traces: (1) the original scenario: place, accept, cook, eat,

pay, and collect, and (2) an added scenario: place, accept, get drunk, and leave. Hence these two services are compatible.

Challenges In the second part of the paper, we discuss three main challenges of service interaction in terms of open nets.

Exposing services (Sect. 4). In order to find compatible pairs of services, services need to know each other (to some degree). Hence services need to be "exposed" to cooperate in a meaningful way. For example, the guest should know that he should leave when the cooks starts singing Irish folk songs. One common approach is where a service shows its own specification or implementation. The drawback of this approach is that the environment starts using short-lived particularities, or that sensitive information is shown without reason. Another, less common, approach is to describe the class of compatible services. The challenge is to characterize a possibly infinite set of services in a compact manner. Operating guidelines are a way of specifying the class of services a service can work with, without exposing irrelevant or sensitive information.

Replacing and refining services (Sect. 5). One of the advantages of using an SOA is that things can be changed more easily, i.e., one service may be replaced by another service, or an unavailable service is replaced by several simpler services. However, all of these changes may cause various errors that break the service choreography. The challenge is to provide rules for replacing and refining services while guaranteeing forward/backward compatibility. Note for example that RS_3 in Fig. 3(a) can be replaced by RS_1 in Fig. 2(a), but not the other way around. How to capture this in a generic rule?

Integrating services using adapters (Sect. 6). In reality, existing services need to be composed to achieve a specific goal. However, services are often not compatible. This triggers many questions, e.g., how to repair a service, how to diagnose problems, etc. In Sect. 6 we focus on the challenge of adapter synthesis, i.e., the (semi-)automatic generation of "glue logic" that makes incompatible services compatible while achieving a given goal. For example, suppose that GS_4 is an anorexic guest that just wants to order and pay without actually eating. It is easy to make an adapter service AS that throws away the food such that the service choreography $GS_4 \oplus AS \oplus RS_1$ functions without any problems.

After discussing these challenges and providing an overview of the known results for these problems, we present tool support for these methods in Sect. 7. Finally, Sect. 8 concludes the paper.

2 Service Interaction Patterns

Before formalizing service definitions and addressing the various challenges in this domain, we provide some examples of service interaction patterns. The goal is not to summarize the existing patterns or to present new ones. Instead various service interaction patterns are presented informally using a notation close to open nets. Since we do not aim to describe the patterns in any detail, we do not use the typical patterns format describing various aspects of a pattern

(e.g., description, examples, forces, motivation, overview, context, implementation, issues, and solutions) like in [10, 3, 16–19]. Instead, we just show a figure and provide a brief description for each pattern.

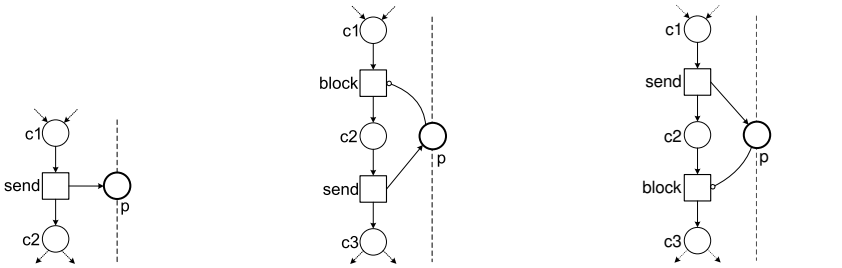
2.1 Workflow Patterns Initiative

The use of patterns is very appealing for identifying functionality in a system/language independent manner. The most well-known patterns collection in the IT domain is the set of design patterns documented by Gamma, Helm, Johnson, and Vlissides [16]. This collection describes a set of problems and solutions frequently encountered in object-oriented software design. This triggered many patterns initiatives in the IT field, including the Workflow Patterns Initiative. However, the idea to use a patterns-based approach originates from the work of the architect Christopher Alexander. In [20], he provides rules and diagrams describing methods for constructing buildings. The goal of the patterns documented by Alexander was to provide generic solutions for recurrent problems in architectural design.

The work described in this section is part of the *Workflow Patterns Initiative* (cf. `www.workflowpatterns.com`). This initiative is a joint effort of Eindhoven University of Technology and Queensland University of Technology which started in the late nineties. The aim of this initiative is to provide a conceptual basis for process technology. In particular, the research provides a thorough examination of the various perspectives (control flow, data, resource, and exception handling) that need to be supported by a workflow language or a business process modeling language. The results can be used for examining the suitability of a particular process language or process-aware information system, assessing relative strengths and weaknesses of various approaches to process specification, implementing certain business requirements in a particular system, and as a basis for language and tool development.

Originally the workflow patterns focussed exclusively on the control-flow perspective [10]. The initial set of 20 patterns was later extended to a set of more than 40 control-flow patterns [21]. In parallel, patterns were identified for the resource perspective [18], for the data perspective [19], and for exception handling [22]. Especially the control-flow patterns have had a huge impact on the selection of systems in practice and the definition of new standards. For example standardization efforts related to BPEL, XPDL, BPMN, etc. have been influenced by these patterns.

In the context of the Workflow Patterns Initiative, several collections for service interaction patterns have been collected. In [3], Barros, Dumas, and Ter Hofstede document such patterns and divide them into several groups: single-transmission bilateral interaction patterns (elementary interactions where a party sends/receives a message, and as a result expects/sends a reply), single-transmission non-routed patterns (also dealing with multi-lateral interactions), multi-transmission interaction patterns (a party sends/receives more than one message to/from the same logical party), and routed interaction patterns (involving complex routing of messages through the network). These patterns were



(a) SIP-1 Send pattern (b) SIP-2 Pre-Blocking Send pattern (c) SIP-3 Post-Blocking Send pattern

Fig. 4. Patterns related to sending a message.

described in an informal manner. In [4] some of these patterns are formalized using both π-calculus and Petri nets. A more systematic approach for the identification of service interaction patterns was conducted in the PhD thesis of Mulyar [17]. She identified five pattern families: multi-party multi-message request-reply conversation [6], renewable subscription, message correlation, message mediation, and bipartite conversation correlation. Using a generative approach, more than 1,500 service interaction patterns are documented in [17]. While the above service interaction patterns have been developed in the context of the Workflow Patterns Initiative, other relevant patterns have been identified in related domains. A notable example is the collection of enterprise integration patterns by Hophe and Woolf [5].

2.2 Basic Service Interaction Patterns

First, we describe the basic service interaction patterns. These patterns abstract from correlation, i.e., at this stage we do not worry about routing a message to a specific service or service instance.

The first pattern is the SIP-1 *Send* pattern. The basic idea is shown in Fig. 4(a). Note that in this open net fragment, transition send represents an activity with precondition c1 and postcondition c2, both modeled by a place. Place p represents an output port. The dashed line separates the interface from the rest of the service definition. In Fig. 2(a) and the other examples in the introduction, this pattern was used multiple times, e.g., to place an order or to pay money. Pattern SIP-2 *Pre-Blocking Send* shown in Fig. 4(b) is a variant of the same idea. However, now the sender blocks if the previously sent message was not yet consumed. This is modeled by a so-called inhibitor arc between block and p, i.e., block can only be executed if p is empty. Pattern SIP-3 *Post-Blocking Send* is another variant. Now the thread in the sender service blocks until the message sent is consumed, cf. Fig. 4(c). After sending the message, transition block waits until the message is removed from output port p. Note that SIP-2 and SIP-3 can be combined, i.e., the sender blocks before and after sending (if necessary).

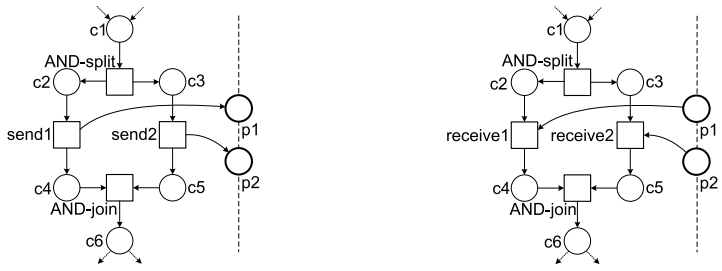
(a) SIP-4 Receive pattern (b) SIP-5 Lossy Receive pattern

Fig. 5. Patterns related to receiving a message.

Figure 5 shows two basic patterns to receive messages. Pattern SIP-4 *Receive* is the straightforward receipt of messages. The receiver blocks until the message arrives. However, the message may arrive when the receiving service is not expecting it. In this scenario, the message waits until the receiver is ready, i.e., the message is queued in the channel connected to place p. If this is not possible, the message may get lost as shown in Fig. 5(b). This is pattern SIP-5 *Lossy Receive*. Note that transition remove has an inhibitor arc connected to the input place c1 of receive. Hence, it can only be executed if receive is not enabled. If receive has multiple places, more remove transitions are needed (one for every input place). These transitions may be considered as part of the channel.

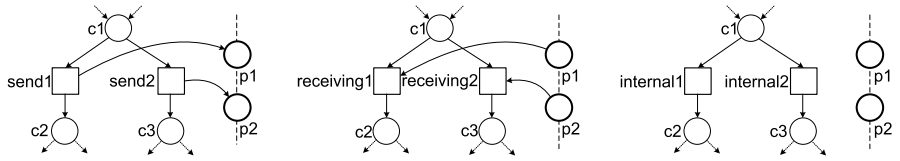
Figure 6(a) shows pattern SIP-6 *Concurrent Send* and Fig. 6(b) shows pattern SIP-7 *Concurrent Receive*. SIP-6 describes the pattern where a service can send two messages in any order: one via p1 and one via p2. SIP-7 is the logical counterpart and here the receiver can receive two messages in any order. Note that SIP-6 and SIP-7 can be combined with services that receive and send sequentially, i.e., from an interaction point these are quite "robust" unlike the choice patterns described next.

Figure 7 shows three choice patterns: SIP-8 *Sending Choice*, SIP-9 *Receiving Choice*, and SIP-10 *Internal Choice*. SIP-8 describes the situation where the choice is made within the service and communicated to the environment. Note that other services may be able to see which internal path is taken, e.g., a message via p1 reveals that send1 is executed. This pattern is used in Fig. 3(b) where the



(a) SIP-6 Concurrent Send pattern (b) SIP-7 Concurrent Receive pattern

Fig. 6. Patterns related to the concurrent sending or receiving of messages.

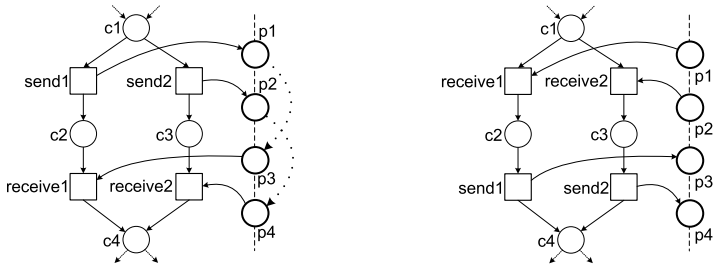


(a) SIP-8 Sending Choice pattern (b) SIP-9 Receiving Choice pattern (c) SIP-10 Internal Choice pattern

Fig. 7. Patterns related to choices in the presence or absence of communication.

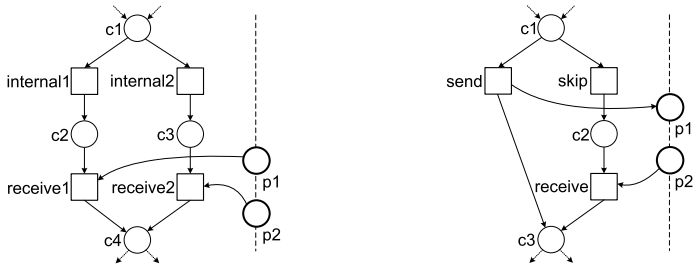
restaurant service communicates the choice to "cook" or "get drunk" by sending the food or singing Irish folk songs. SIP-9 shown in Fig. 7(b) models the pattern where the choice is influenced by the environment, i.e., the environment forces the service to take one path or another. This pattern is used in Fig. 3(b) by the guest service. Pattern SIP-10 describes a third variant where the choice is not enforced by the environment nor communicated (cf. Fig. 7(c)).

Figure 8 shows two patterns where a choice is followed by a subsequent message exchange depending on the choice. SIP-11 *Sending Choice Receiving Follow-Up* shown in Fig. 8(a) corresponds to the scenario where the service makes a choice (SIP-8) followed by the receipt of a particular message depending on the initial choice. Hence, the environment is expected to send a message via `p3` if it received a message via `p1` and it is expected to send a message via `p4` if it received a message via `p2`. This is indicated by the dotted curves in Fig. 8(a). Figure 8(b) shows the symmetrical case, i.e., pattern SIP-12 *Receiving Choice Sending Follow-Up*. In this pattern the environment takes the lead and the service follows, e.g., when receiving a message via `p1`, the service responds by sending a message via `p3`. Not shown are the patterns SIP-13 *Sending Choice Sending Follow-Up*, SIP-14 *Receiving Choice Receiving Follow-Up*, and SIP-15 *Internal Choice Sending Follow-Up*. However, given their names and the two earlier examples, their meaning is obvious. We do not consider situations



(a) SIP-11 Sending Choice Receiving Follow-Up pattern (b) SIP-12 Receiving Choice Sending Follow-Up pattern

Fig. 8. Choice with a follow-up patterns.



(a) AP-1 Internal Choice Receiving Follow-Up anti-pattern

(b) Another variant of the anti-pattern

Fig. 9. Two examples showing that "choices that matter" need to be communicated.

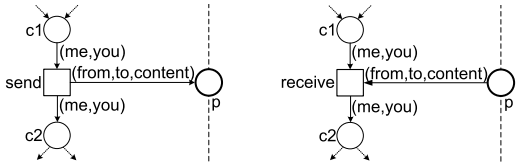
where the follow-up is an internal step, because this would not really be a follow-up related to the choice.

Figure 9(a) shows a so-called *anti-pattern*: AP-1 *Internal Choice Receiving Follow-Up*. Anti-patterns describe undesirable constructs that may introduce errors or inefficiencies. In AP-1 an internal choice is followed by a receive which should depend on the internal choice. Sometimes the service expects a message via p1 and sometimes via p2. However, the environment has no way of telling what to do, because the choice was never communicated. If one compares this with the two patterns shown in Fig. 8, it is good to see that the essential difference is that in AP-1 the environment has no way of determining an adequate strategy. By not sending a message via p1 the receiving service may deadlock and by sending a message via p1 the message may get stuck in the channel. Figure 9(b) shows a variant of the same anti-pattern. The choice to execute skip is not communicated, so the environment does not know whether it should send a message via p2 or wait forever for a message to arrive via p1.

Note that the problem in Fig. 3(a) is similar to AP-1. The restaurant service never communicates that the cook decided to get drunk, so the guest does not know whether to leave or not.

2.3 Correlation Patterns

The patterns presented thus far abstract from *correlation*, i.e., when sending a message via a channel it is assumed that it is routed to the appropriate service instance. For example, in Fig. 2(a) it is assumed that the "food message" is routed to the right instance of the guest service. If there are multiple guests, there will be multiple instances of GS_1 and RS_1. The "food message" needs to the related to the "order message" (i.e., the right dish is cooked), and the "money message" needs to be related to the two previous messages, because the price probably depends on the dish that was ordered. In this paper we define correlation as *establishing a relationship between a service instance and a message.*



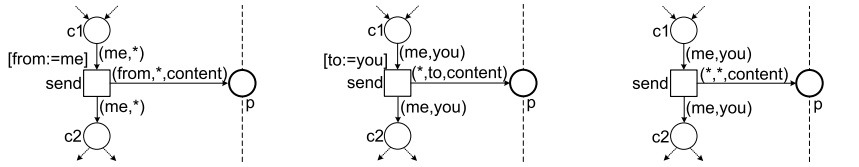
(a) Sending (b) Receiving

Fig. 10. Notation used to explain basic correlation patterns.

Correlation is a neglected topic in service interaction. Yet correlation is omnipresent. Consider for example the booking of a trip, ordering a book, reviewing papers, requesting a lab test, etc. In each of these examples multiple parties are involved while there may be many concurrent instances. To be able to link messages to service instances, so-called *correlation identifiers* are used. For example, when booking a trip a booking reference is given by the travel agency. In a hospital the patient identifier is used to route lab tests to the right department. Therefore, languages such as BPEL provide explicit mechanisms for correlation. For example, BPEL supports the concept of "correlation sets" [9]. When a message arrives for a web service which has been implemented using BPEL, the message must be delivered somewhere: either to a new or an existing instance of the BPEL process. The task of determining to which conversation (i.e., service instance) a message belongs, is supported by correlation sets. To use a correlation set, a BPEL program defines the set by enumerating the properties which comprise it, and then refers to that set from receive, reply, invoke, or pick activities (i.e., all BPEL activities involving interaction).

To illustrate the basics of correlation, consider Fig. 10. In this figure me refers to the identity of the service instance and you refers to the identity of some other service instance. Messages are described by the triplet (from,to,content). from refers to the sender of the message, to refers to the intended recipient of the message, and content refers to the message body. The variables me, you, from, and to refer to service instances and may be used for correlation. We replace me, you, from, and to by the symbol * when the variable has no value or the value does not matter. Figure 10(a) shows the notation for sending a message (from,to,content) by a service instance described by (me,you). Figure 10(b) shows the receiving counterpart. These notations will be used to illustrate correlation patterns.

Figure 11 shows two correlation patterns and one anti-pattern. All refer to sending a message. Pattern SIP-16 *Leading Correlated Send* depicted in Fig. 11(a) describes the situation where a message (from,*,content) is sent. This implies that the sender expects the other party to use the sender's identification (i.e., from = me). Therefore, we say that the sender is "leading" in SIP-16. Note that the you and to variables shown in Fig. 10(a) are all replaced by * to denote that they are irrelevant or missing. The assignment [from:=me] attached to transition send makes sure that the sender instance reveals itself appropriately. Pattern SIP-17 *Following Correlated Send* shown in Fig. 11(b) assumes that the sender

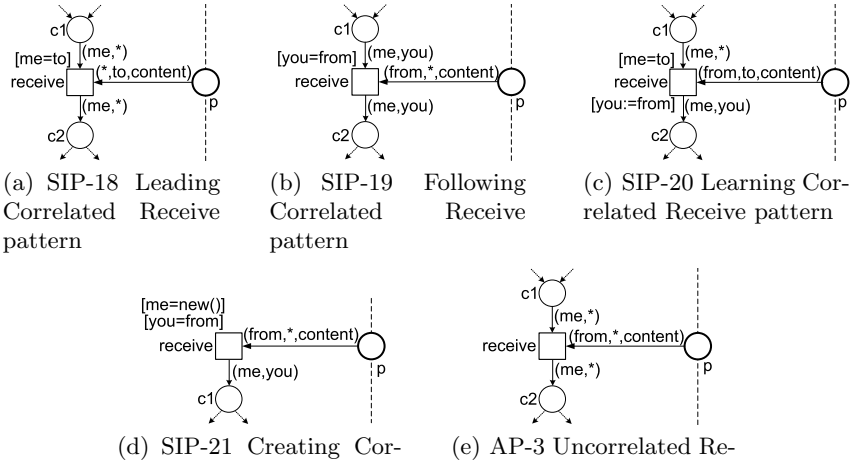


(a) SIP-16 Leading Correlated Send pattern (b) SIP-17 Following Correlated Send pattern (c) AP-2 Uncorrelated Send anti-pattern

Fig. 11. Correlation patterns related to sending a message.

is "following" and uses a correlation id set by the other party. Here the tuple (*,to,content) is sent. Therefore, the sender needs to know the identity of the receiver instance (i.e., to:=you). Figure 11(c) shows the anti-pattern AP-2 *Uncorrelated Send*. This anti-pattern refers to the situation where no explicit correlation information is given when sending the message. Other than the content of the message, there is no way in which the environment can correlate the message (*,*,content) to the instance me.

Figure 12 shows four correlation patterns and one anti-pattern. Pattern SIP-18 *Leading Correlated Receive* is the logical counterpart of SIP-17, i.e., the sender uses the correlation id of the receiver. Figure 12(a) shows the type of message (*,to,content) received and the guard [me=to] making sure that the message is routed to the right instance. Pattern SIP-19 *Following Correlated Receive* depicted in Fig. 12(b) shows the situation where the receiver "follows" the sender and needs to know its identity you. Pattern SIP-20 *Learning Correlated Receive* depicted in Fig. 12(c) shows yet another variant. Here the receiver does not know



(a) SIP-18 Leading Correlated Receive pattern (b) SIP-19 Following Correlated Receive pattern (c) SIP-20 Learning Correlated Receive pattern

(d) SIP-21 Creating Correlated Receive pattern (e) AP-3 Uncorrelated Receive anti-pattern

Fig. 12. Correlation patterns related to receiving a message.

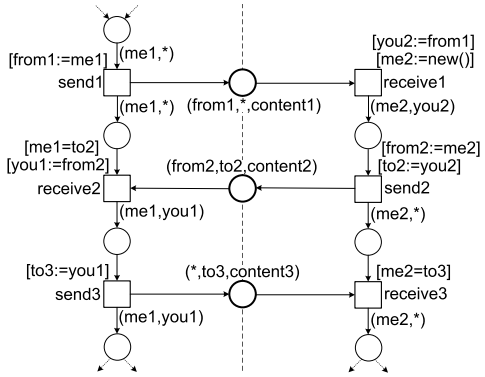


Fig. 13. SIP-22 Correlation Swap pattern.

the identity of the sender, but learns this from the incoming message. Note that the service instance is denoted by (me,you) after receiving the message while before it was denoted by (me,*). Pattern SIP-21 *Creating Correlated Receive* shown in Fig. 12(d) is similar to SIP-19 but now a new instance is created. Note that some languages provide a combination of SIP-19 and SIP-21, i.e., if the message can be correlated to an existing instance, then this is done, otherwise a new instance is created. Figure 12(e) shows the anti-pattern AP-3 *Uncorrelated Receive*. Although the sender reveals its identity, the receiver has no possibility to correlate properly (without analyzing the content of the message for "clues"). In the example of the restaurant this would correspond to preparing a dish without knowing which guest has ordered it.

Figures 11 and 12 show some of the basic correlation patterns. Clearly these can be combined to identify more complex patterns. We do not aim at providing a complete overview of such patterns. We refer to [3] for some example patterns where correlation plays a prominent role and to [17] where a more complete classification of correlation patterns is given. Moreover, to illustrate the challenges related to correlation, we show two more patterns.

Figure 13 shows the SIP-22 *Correlation Swap* pattern. Here we use the same notations as before, so the figure should be self-explanatory. The core idea of the pattern is that in the first message the correlation id of the left service instance is used while in the last message the correlation id of the right service is used. The message in the middle helps the left service instance to build up knowledge to be able to use the other party's correlation id in later interactions. Seen from the viewpoint of the left service, SIP-22 uses three basic patterns: SIP-16 (for sending the first message in a "leading role"), SIP-20 (for learning the other instance's id from the second message), and SIP-17 (for sending the third message in a "following role"). As shown in Fig. 13, the first message creates a service instance (i.e., SIP-21 is used).

Pattern SIP-23 *Correlation Broker* shown in Fig. 14 is an example of a pattern involving three services. The service in the middle acts as a mediator and is

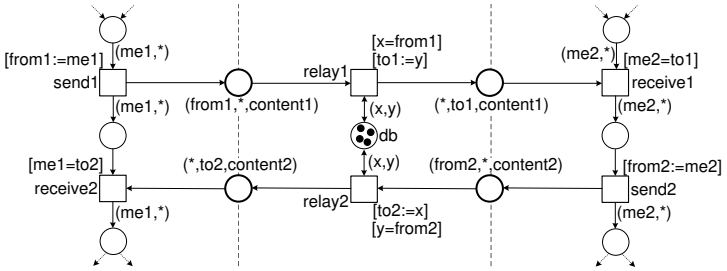


Fig. 14. SIP-23 Correlation Broker pattern.

instantiated for any connection between a service instance on the right and a service instance on the left. As a result, two services can interact without knowing each other's identity. Note that place db holds a token for each pair of service instances (we assume that there is a one-to-one correspondence). The first message is relayed to the appropriate service instance using this information. The return message is relayed in a similar way without exposing the sending service.

2.4 More Advanced Correlation Patterns

As indicated before, correlation is of the utmost importance and the patterns shown thus far only scratch the surface. For example, the patterns shown in this paper use the identity of the sending or receiving instance as a correlation id. Of course, both parties can also agree on a more neutral correlation id. Also note that languages like BPEL support multiple correlation sets [9]. The topic of correlation has many aspects and based on [3, 5, 17] we mention two dimensions showing the broad scope of this problem.

SIP-23 is the only correlation pattern considering *multilateral interaction*. All other correlation patterns consider just bilateral interactions. In realistic service choreographies multilateral correlation is needed. Consider for example the booking of a trip involving two flights, three hotels nights, two train trips, and a rental car. This requires a network of service instances involving non-binary dependencies (e.g., the hotel should be canceled if the flight is not possible and the pick-up time of the car depends on the flight data).

Another dimension is related to *multiple instances* inside a service instance or message. Thus far we assumed activities and messages to be atomic. This is often not the case. For example, consider a service handling customer orders that may consist of multiple order lines. This service needs to deal with messages and activities at the level of a customer order and at the level of individual order lines. For example, a customer places one order that is decomposed in smaller orders for specific suppliers. Moreover, for a single order line there may be multiple potential suppliers. Another example, is the organization of a conference. One instance of a conference involves multiple authors, PC members, and reviewers. There may be many papers, each paper requires multiple reviews, and papers are ranked and compared based on their reviews. One reviewer may submit multiple

reviews and each paper has multiple reviews, authors, etc. This example, shows that various types of instances interact in a complex manner.

In [5] the authors identify a patterns called *Scatter-Gather*. This is an example of a pattern that involves a variable number of service instances. The goal of the patterns is to "maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply" [5]. The Scatter-Gather pattern broadcasts a message to multiple recipients and re-aggregates the responses back into a single message. For example, one may ask a dozen car rental companies for a quote, then select the best quote, and continue interacting with the cheapest rental company.

The data intensive patterns referred to in this subsection are outside the scope of this paper. In the remainder, we abstract from correlation and restrict the scope to the patterns presented in Sect. 2.2. For analysis purposes we typically look at one instance or conversation in isolation. We will show that this is often a valid abstraction. Nevertheless, we presented several correlation patterns to stress the importance of correlation.

3 Specifying Services

Petri nets have proven to be successful for the modeling of business processes and workflows (see the work of Van der Aalst [15, 23], for instance). In this section we introduce our modeling formalism for services, viz., *open nets*, which is a refinement of Petri nets. In terms of the patterns we introduced in the previous section the focus is on the basic patterns (cf. Sect. 2.2), i.e., we only consider a single instance of each service and no correlation. We focus on service interaction, and abstract from non-functional properties, semantical information and data. We introduce the concept of open net composition and also formalize the notion of compatibility. As the formalism of open nets refines classical place/transition Petri nets, we first provide the basic definitions on Petri nets.

3.1 Basic Definitions on Petri Nets

Petri nets [12–14] consists of two kinds of nodes, *places* and *transitions*, and a *flow relation* on nodes. Graphically, a place is represented by a circle, a transition by a box, and the flow relation by directed arcs between them. 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. A *state* of the Petri net is represented by a marking, which is a distribution of tokens over the places. Graphically, a token is depicted by a black dot.

Definition 1 (Petri net). *A* Petri net $N = [P, T, F, m_0]$ *consists of*

- *two finite and disjoint sets* P *and* T *of* places *and* transitions*,*
- *a* flow relation $F \subseteq (P \times T) \cup (T \times P)$*, and*
- *an* initial marking m_0*, where a marking is a mapping* $m : P \to \mathbb{N}$.

When referring to several Petri nets we use indices, to distinguish the constituents of different Petri nets, for example, P_N refers to the set of places of Petri net N.

For the flow relation of a Petri net N we introduce the following notation to denote the pre-set and the post-set of places and transitions. Let $x \in P \cup T$ be a node of N. Then, ${}^\bullet x = \{y \mid [y, x] \in F\}$ denotes the *pre-set* of x (i.e. all nodes y that have an arc to x) and $x^\bullet = \{y \mid [x, y] \in F\}$ denotes the *post-set* of x (i.e. all nodes y with an arc from x to y).

Consider the Petri net of the guest service GS_1 in Fig. 2(a) and ignore for the moment the interface places and its adjacent arcs. This Petri net consists of the four places $\mathsf{p1}, \ldots, \mathsf{p4}$ and the three transitions place, eat, pay. Its initial marking is $m_0 = [\mathsf{p1}]$. For example, we have ${}^\bullet\mathsf{p2} = \{\mathsf{place}\}$ and $\mathsf{p2}^\bullet = \{\mathsf{eat}\}$.

The dynamics of a Petri net N is defined by the *firing rule*. The firing rule defines *enabledness* of Petri net transitions and their effects. A transition t is enabled at a marking m if there is a token on every place in its pre-set. The firing of an enabled transition t yields a new marking m', which is derived from its predecessor marking m by consuming (i.e. removing) a token from each place of t's pre-set and producing (i.e. adding) a token on each place of t's post-set. The described firing relation is denoted $m \xrightarrow{t} m'$. Thereby $m \xrightarrow{t} m'$ is a *step of* N.

The behavior of a Petri net N can be enhanced from single steps to potentially infinite transition sequences, called runs. A finite or infinite sequence $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \xrightarrow{t_3} \ldots$ is a *run of* N if and only if, for all i, $m_i \xrightarrow{t_i} m_{i+1}$ is a step of N. Let m and m' be markings of N. Then, m' is *reachable* from m if and only if there exists a finite run $m_0 \xrightarrow{t_0} m_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-2}} m_{k-1} \xrightarrow{t_{k-1}} m_k$ with $m = m_0$ and $m_k = m'$. We denote this reflexive transitive closure of the firing rule by $m \xrightarrow{*} m'$. With $R_N(m) = \{m' \mid m \xrightarrow{*} m'\}$ we denote the set of all markings that can be reached from m by firing any number of transitions.

The set $R_N(m_0)$ of reachable markings of a Petri net N contains all markings that are reachable from the initial marking m_0. That way, $R_N(m_0)$ spans a graph that has the set of reachable markings as its states and the transitions between these markings as its edges. This graph is known as the *reachability graph*, which can be represented by a *transition system*.

Consider again GS_1 in Fig. 2(a) without the interface places and their adjacent arcs. In its initial marking, [p1], only transition place is enabled. Firing of transition place yields marking [p2]. There is only one firing sequence and four reachable markings: $[\mathsf{p1}] \xrightarrow{\mathsf{place}} [\mathsf{p2}] \xrightarrow{\mathsf{eat}} [\mathsf{p3}] \xrightarrow{\mathsf{pay}} [\mathsf{p4}]$.

3.2 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*.

Asynchronous message passing means that communication is non-blocking, i.e., 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, i.e., the order in which the messages are sent is not necessarily the order in which they are received.

We model services as open nets which have been introduced as 'open workflow nets' in [24]. An open net is a Petri net as defined in the previous section and thus it can adequately model the control structure of a service. The set of final states of a service, i.e., the states in which it may successfully terminate, is modeled by a set of final markings. 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 pre-set and is used for receiving messages from a distinguished channel whereas an output place has an empty post-set and is used for sending messages via a distinguished channel.

Definition 2 (Open net). *An* open net $N = [P, T, F, I, O, m_0, \Omega]$ *consists of a Petri net* $[P, T, F, m_0]$ *together with*

- *an* interface $(I \cup O) \subseteq P$ *defined as two disjoint sets* I *of* input places *and* O *of* output places *such that* ${}^\bullet p = \emptyset$ *for any* $p \in I$ *and* $p^\bullet = \emptyset$ *for any* $p \in O$*, and*
- *a set* Ω *of* final markings*.*

We further require that in the initial and the final markings the interface places are not marked, i.e., for all $m \in \Omega \cup \{m_0\}$ *we have* $m(p) = 0$*, for all* $p \in I \cup O$*.*

Graphically, we represent an open net like a Petri net with a dashed frame around it. The interface places are depicted on the frame. Final markings have to be described separately.

We refer to an open net with an empty interface as a *closed net*. A closed net can be used to model a *service choreography*, for instance.

Definition 3 (Closed net). *An open net* N *with an empty interface, i.e.,* $I_N = \emptyset$ *and* $O_N = \emptyset$*, is a* closed net*.*

The seven nets in Figs. 2 and 3 are open nets. For example, the open net GS_1 in Fig. 2(a) has $I = \{\mathsf{food}\}$ and $O = \{\mathsf{order}, \mathsf{money}\}$. We define the final marking $\Omega = \{[\mathsf{p4}]\}$.

A closed net has finitely many states if it is *bounded*, i.e., no place can contain infinitely many tokens in any reachable marking.

Definition 4 (Boundedness). *A closed net* N *is* k-bounded *if there exists a* $k \in \mathbb{N}$ *such that for each reachable marking* $m \in R_{(N)}(m_0)$*,* $m(p) \leq k$*, for all* $p \in P_N$*.*

3.3 Composing Open Nets

The general idea of SOA is to use services as building blocks for designing complex services. To this end, services have to be composed, i.e., pairs of input and output ports of these services are connected using a channel. Communication between these services takes place by exchanging messages via these channels. Composing two open nets is modeled by fusing pairwise equally labeled input and output places. Such a fused 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 all constituents (except for the interfaces) are pairwise disjoint. This can be achieved easily by renaming. In contrast, the interfaces intentionally overlap. For a reasonable concept of composition of open nets, however, it is convenient to require that all communication is bilateral and directed, i.e., every interface place $p \in I \cup O$ has only one open net that sends into p and one open net that receives from p. Thereby the sending open net has the output place and the receiving open net has the corresponding equally labeled input place. We refer to open nets that fulfill these properties as *interface compatible*.

Definition 5 (Interface compatible open nets). *Let N_1, N_2 be two open nets with pairwise disjoint constituents except for the interfaces. If only input places of one open net overlap with output places of the other open net, i.e., $I_1 \cap I_2 = \emptyset$ and $O_1 \cap O_2 = \emptyset$, then N_1 and N_2 are* interface compatible.

As an example, each of the four pairs of open nets depicted in Figs. 2 and 3 are interface compatible open nets.

Composing two open nets means to merge their respective shared constituents. As we only define composition for interface compatible open nets, the only shared constituents are the interface places. In other words, composition corresponds to *place fusion* which is well-known in the theory of Petri nets.

Definition 6 (Composition of open nets). *Let N_1 and N_2 be two interface compatible open nets. The* composition $N = N_1 \oplus N_2$ *is the open net with the following constituents:*

- $P = P_1 \cup P_2$,
- $T = T_1 \cup T_2$,
- $F = F_1 \cup F_2$,
- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$,
- $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$,
- $m_0 = m_{01} \oplus m_{02}$, *and*
- $\Omega = \{m_1 \oplus m_2 \mid m_1 \in \Omega_1, m_2 \in \Omega_2\}$.

For markings m_1 of N_1 and m_2 of N_2 which do not mark the interface places, their composition $m = m_1 \oplus m_2$ is defined by $m(p) = m_i(p)$ if $p \in P_i$, for $i = 1, 2$.

Composition of two open nets M and N results in an open net again. Composing M and N means merging input places of M with equally labeled output

places of N (and vice versa). Therein, bilateral and directed communication between the components is guaranteed. The initial marking of the composition is the sum of the initial markings of M and N, and the set of final markings of the composition is the Cartesian product of the sets of final markings of M and N. This is reasonable, because Definition 2 ensures that the only shared constituents of M and N, the interface places, are not marked in the initial or final markings.

As an example, all pairs of open nets in Sect. 1 are interface compatible. Thus we can compose them by merging equally labeled interface places. Each resulting composition is a closed net.

To apply composition to an arbitrary number of open nets, we require these open nets to be pairwise interface compatible. This ensures bilateral communication as for a third open net N_3, a communication taking place inside the composition of open nets N_1 and N_2 is internal matter.

Open net composition is commutative and associative, i.e., for interface compatible open nets N_1, N_2 and N_3 holds $N_1 \oplus N_2 = N_2 \oplus N_1$ and $(N_1 \oplus N_2) \oplus N_3 = N_1 \oplus (N_2 \oplus N_3)$. Thus, composition of a set of open nets can be broken into recursive pairwise composition.

3.4 Behavioral Properties

We want the composition of a set of services to be *compatible*. Obviously, there is no unique definition of compatibility. A minimal requirement is, however, the absence of deadlocks in a service. A stronger criterion is the possibility of a service to terminate from every reachable state. This criterion excludes deadlocks and in addition livelocks, where a livelock is a set of reachable states of a service from which neither a deadlock nor a final state is reachable. Besides deadlocks and livelocks one may also want to exclude the existence of dead activities in a service. This criterion of compatibility coincides with the soundness notion for workflows [15]. Obviously, compatibility is only of interest for a service choreography which is modeled by a closed net, i.e., an open net with an empty interface.

In this paper we say that a closed net is compatible if it is deadlock-free, but most of the techniques can also be used for other notions of compatibility. Thereby a *deadlock* is a reachable, non-final marking m in N in which the open net N gets stuck, i.e., no transition is enabled in m.

Definition 7 (Deadlock). *Let $N = [P, T, F, I, O, m_0, \Omega]$ be a closed net. A reachable marking $m \in R_N(m_0)$ is a* deadlock *in N iff $m \notin \Omega$ and no transition $t \in T$ is enabled in m. If no such m exists in N, then N is* deadlock-free.

This definition of a deadlock differs from the standard definition in the literature, as we discriminate between terminating (final) states and non-terminating states (i.e., deadlocks).

If we assume $\Omega = \{[\mathsf{p4}, \mathsf{p8}]\}$ for all compositions in Figs. 2 and 3, then $GS_1 \oplus RS_2$ has a deadlock ($[\mathsf{p2}, \mathsf{p6}]$) and $GS_2 \oplus RS_3$ has also a deadlock ($[\mathsf{p4}, \mathsf{food}, \mathsf{p7}]$). The other two compositions, $GS_1 \oplus RS_1$ and $GS_3 \oplus RS_4$, are deadlock-free.

Given an open net N we are interested in those open nets M such that their composition $M \oplus N$ is a deadlock-free closed net.

Definition 8 (Strategy, controllability). *Let M, N be two open nets such that $I_M = O_N$ and $O_M = I_N$. Then, M is a* strategy *for N iff $M \oplus N$ is deadlock-free. With $Strat(N)$ we denote the set of all strategies for N. N is* controllable *iff its set of strategies is nonempty.*

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

In our examples, GS_1 is a strategy for RS_1 and vice versa, for instance. Hence, both open nets are controllable. Moreover, each of the seven open nets in Figs. 2 and 3 is controllable. For GS_2 and RS_3 this might be surprising at first sight. However, a restaurant service that only receives the order and then terminates is a strategy for GS_2, thus forcing the customer to leave. A strategy for RS_3 must be aware that after having ordered the cook may get drunk, in which case no food will be served. Nevertheless, the guest cannot be sure, and hence he must stay in the restaurant in order to eat the food in case it is served. This can be modeled by open net GS_1 with final markings $\Omega = \{[\mathsf{p2}], [\mathsf{p4}]\}$, i.e., after having ordered, the guest does not need to eat, but if food is served, he will eat and pay.

4 Exposing Services

For automatically selecting and composing services in a well-behaved manner, information about the services has to be exposed. In particular, this information must be sufficient to decide whether the composition of any service R with any service S is compatible. Usually, the information about some services S is stored in a repository. Selecting a service means to find for a given service R (whose behavior is given) a compatible service S in the repository. There are two ways of exposing services.

In the first approach, the *behavior of S* is exposed. Well-behavior of the composition of R and S can be verified using standard state space verification techniques [25]. However, organizations usually want to hide the trade secrets of their services and thus need to find a proper abstraction of S which is published instead of S.

The second approach does not expose the behavior of S, but a *class of services* R that is compatible with S, e.g., the set $Strat(S)$. Then the composition of R and S is compatible if $Strat(S)$ contains R. From the set of strategies it is in general not possible to derive the original service.

However, $Strat(S)$ is in general an *infinite* set of services. Hence, the challenge is to find a compact representation of this set. To this end, *operating guidelines* can be used.

In this section we only consider the latter approach of exposing services and thus recapitulate the concept of an operating guidelines [26, 27]. The operating

guidelines $OG(N)$ of an open net N is a (finite) automaton enhanced with some annotations. It represents the set $Strat(N)$ of all strategies for N.

Strictly speaking, $OG(N)$ does not characterize a set of open nets, but the *behavior* of these nets, because two (structurally) different open nets may have the same behavior. So we continue by first defining the behavior of an open net, which is a labeled transition system, and then introducing operating guidelines.

4.1 Behavior of Open Nets

The behavior of an open net N is basically the reachability graph of the inner subnet $inner(N)$ of N, which defines the Petri net that results from removing the interface places and the adjacent arcs from N. Obviously, $inner(N)$ and N coincide if N is a closed net.

Definition 9 (Inner subnet). *Let $N = [P, T, F, I, O, m_0, \Omega]$ be an open net and let $P' = P \setminus (I \cup O)$ be the set of* internal places *of N. Then, $inner(N) = [P', T, F \cap ((P' \times T) \cup (T \times P')), \emptyset, \emptyset, m_0, \Omega]$ is the* inner subnet of N.

Often we restrict ourselves to open nets where every transition is connected to at most one interface place. We refer to such open nets as *elementary communicating open nets*. This restriction is not significant, as every open net can be transformed to an equivalent elementary communicating open net [27]. All examples shown in Sect. 1 are elementary communicating open nets.

For elementary communicating open nets we define a mapping that assigns a label to each transition. We use these labels to represent the transition system of an open net N.

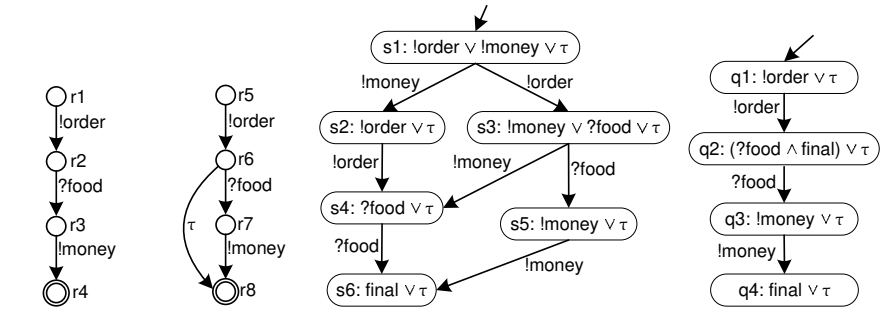
Definition 10 (Transition label of open nets). *Let $N = [P, T, F, I, O, m_0, \Omega]$ be an elementary communicating open net. The* transition labels *for N are defined by the mapping $l : T \to I \cup O \cup \{\tau\}$ ($\tau \notin I \cup O$) such that $l(t)$ is the unique interface place adjacent to $t \in T$ if one exists, and $l(t) = \tau$ if t is not adjacent to any interface place.*

In the examples we add a preceding question mark, '?', to each label of a transition connected to an input place and a preceding exclamation mark, '!', to each label of a transition connected to an output place. For example, the inner subnet of GS_2 has the labels $l(\mathsf{place}) = \mathsf{!order}, l(\mathsf{eat}) = \mathsf{?food}, l(\mathsf{pay}) = \mathsf{!money}, l(\mathsf{leave}) = \tau$.

The behavior of an open net N can now be defined by the reachability graph of the inner structure of N, where the transitions are labeled using the mapping l defined in Definition 10. Notice, the transition labels represent actions on an asynchronous channel.

Definition 11 (Behavior of open nets). *The* behavior of an open net $N = [P, T, F, I, O, m_0, \Omega]$ *is defined by the transition system $TS(N) = [Q, l, \delta, q_0, Q_F]$, where*

- *$Q = R_{inner(N)}(m_0)$ is the (nonempty) set of reachable markings of $inner(N)$,*



(a) $TS(GS_1)$ (b) $TS(GS_2)$ (c) $OG(RS_1)$ (d) $OG(RS_3)$

Fig. 15. Behavior of open nets GS_1 and GS_2 and operating guidelines (guaranteeing 1-boundedness) of open nets RS_1 and RS_3.

- *l is the labeling function,*
- *$[m, l(t), m'] \in \delta$ iff $m \xrightarrow{t} m'$, for $t \in T$, is the transition relation,*
- *$q_0 = m_0$ is the initial state, and*
- *$Q_F = \Omega$ is the set of final states.*

Figures 15(a) and 15(b) show the behavior of open nets GS_1 and GS_2, respectively. States r4 and r8 denote final states. In Fig. 15(a) the states r1, r2, r3 and r4 correspond to the markings [p1], [p2], [p3], and [p4] in GS_1, respectively.

To relate different service behaviors, we introduce the well-known weak simulation relation [28]. Weak simulation is defined for (labeled) transition systems. Since we can compute the behavior of any open net in terms of a transition system, weak simulation is also well-defined for open nets. Let τ^* denote a (possible empty) sequence of τ transitions.

Definition 12 (Weak simulation relation). *Let P and R be transition systems and let $\hat{a}$ stand for τ^* if transition label a is τ, and a otherwise. A binary relation $\varrho_{P,R} \subseteq Q_P \times Q_R$ is a* weak simulation relation of P by R *iff for every $[q_P, q_R] \in \varrho_{P,R}$, such that there is a transition $[q_P, a, q'_P] \in \delta_P$ in P, there is a transition $[q_R, \hat{a}, q'_R] \in \delta_R$ in R and $[q'_P, q'_R] \in \varrho_{P,R}$. R* weakly simulates *P iff there is a weak simulation relation $\varrho_{P,R}$ of P by R such that $[q_{0_P}, q_{0_R}] \in \varrho_{P,R}$.*

Consider again Figs. 15(a) and 15(b). $TS(GS_2)$ weakly simulates $TS(GS_1)$ using the relation $\varrho_{TS(GS_2),TS(GS_1)} = \{[\mathsf{r1}, \mathsf{r5}], [\mathsf{r2}, \mathsf{r6}], [\mathsf{r3}, \mathsf{r7}], [\mathsf{r4}, \mathsf{r8}]\}$. $TS(GS_1)$ also weakly simulates $TS(GS_2)$ using the relation $\varrho_{TS(GS_1),TS(GS_2)} = \{[\mathsf{r5}, \mathsf{r1}], [\mathsf{r6}, \mathsf{r2}], [\mathsf{r7}, \mathsf{r3}], [\mathsf{r8}, \mathsf{r4}], [\mathsf{r8}, \mathsf{r2}]\}$. So the final states do not matter for weak simulation.

4.2 Operating Guidelines

For an open net N we have the set $Strat(N)$ of all strategies for N. Since the set $Strat(N)$ is in general infinite, we need to construct a compact characterization

of this set. To this end, we introduce *operating guidelines*, a (automaton-based) representation of $Strat(N)$.

An operating guidelines $OG(N)$ of an open net N characterizes the set $Match(OG(N)) = \{TS(M) \mid M \in Strat(N)\}$, i.e., the behaviors of all strategies for N and thus the set $Strat(N)$. The set $Match(OG(N))$ contains a transition system, say $TS(M^*)$, that has the least restrictions [29] and any open net M^* is called a *most permissive strategy* for N. More precisely, $TS(M^*)$ weakly simulates the behavior $TS(M)$ of each strategy M for N. The transition system $TS(M^*)$ is the first ingredient of $OG(N)$. As an example, ignore the annotations inside the states of Fig. 15(d). Apart from the final states the automaton of Fig. 15(d) is the most permissive strategy for the open net RS_3.

Unfortunately, $TS(M^*)$ also weakly simulates some transition systems, for which the corresponding open net is not a strategy for N. For example, $TS(GS_1)$ (cf. Fig. 15(a)) is weakly simulated by the most permissive strategy for RS_3 (cf. Fig. 15(d)), but GS_1 is not a strategy for RS_3. In order to exclude such transition systems, we need to specify which restrictions of the structure of $TS(M^*)$ are behaviors of strategies for N. This can be achieved by specifying which edges of $TS(M^*)$ have to be present in the weak simulation between $TS(M^*)$ and any $TS(M)$, for any strategy M for N. To this end, every state q of $TS(M^*)$ is annotated with a Boolean formula $\Phi(q)$, the second ingredient of $OG(N)$.

A *literal* of our Boolean formulae Φ is an element of the set MP of transition labels of M^* (MP stands for message ports) or one of the special literals τ and *final* (representing an internal transition and a final state, respectively). With MP^+ we denote the set $MP \cup \{final, \tau\}$. As Boolean connectors, we only need $\vee$ (Boolean *or*) and $\wedge$ (Boolean *and*). Let $\mathcal{BF}$ be the set of all such Boolean formulae over MP^+.

Thus, an operating guidelines $OG(N) = B^{\Phi}$ is a Boolean annotated service automaton that consists of a deterministic automaton B and a Boolean annotation Φ. Thereby B is the behavior $TS(M^*)$ of the most permissive strategy for N.

Definition 13 (Boolean annotated service automaton). *A* Boolean annotated service automaton *(BSA)* $B^{\Phi} = [Q, MP, \delta, q_0, \Phi]$ *consists of*

- *a nonempty set Q of states,*
- *a set MP of transition labels such that $final, \tau \notin MP$,*
- *a deterministic transition relation $\delta \subseteq Q \times MP \times Q$,*
- *an initial state q_0, and*
- *a Boolean* annotation *function $\Phi : Q \to \mathcal{BF}$.*

Figures 15(c) and 15(d) show two *BSA*s. For example, the *BSA* in Fig. 15(d) has four states $\mathsf{q1}, \ldots, \mathsf{q4}$. The initial state is $\mathsf{q1}$. The annotations are $\mathsf{!order} \vee \tau$ in state $\mathsf{q1}$, $(\mathsf{!food} \wedge \mathsf{final}) \vee \tau$ in state $\mathsf{q2}$, etc.

We use Boolean annotated service automata to represent the behavior of a set of *open nets*. Therefore, we take a *BSA* B^{Φ} and define when a service described in terms of an open net M matches with B^{Φ}. A (Boolean) *assignment* is a mapping $\beta : MP^+ \to \{true, false\}$ assigning to each literal a truth value.

Furthermore, an assignment β *satisfies* a Boolean formula $\phi \in \mathcal{BF}$, denoted by $\beta \models \phi$, if ϕ evaluates to *true* using standard propositional logic semantics. Open net M matches with B^{Φ} if

1. its *behavior* $TS(M)$ is weakly simulated by B^{Φ} and
2. for every state q_m of $TS(M)$ that is weakly simulated by a state q of B^{Φ}, the transitions leaving q_m and the fact whether q_m is a final state of $TS(M)$ constitute a satisfying assignment for $\Phi(q)$.

Definition 14 (Assignment). *Let MP be a set of message ports. An* assignment *of the behavior $TS(M) = [Q, l, \delta, q_0, Q_F]$ of an open net M assigns to each state $q \in Q$ a Boolean assignment $\beta_{TS(M)}(q) : MP^+ \to \{\text{true}, \text{false}\}$ defined by:*

$$\beta_{TS(M)}(q)(x) = \begin{cases} \text{true}, & \text{if } x \neq \textit{final} \text{ and there is a state } q' \text{ with } [q, x, q'] \in \delta, \\ \text{true}, & \text{if } x = \textit{final} \text{ and } q \in Q_F, \\ \text{false}, & \text{otherwise.} \end{cases}$$

As an example, $TS(GS_2)$ (see Fig. 15(b)) assigns in state r5 *true* to !order, in state r6 *true* to ?food and τ, in state r7 *true* to !money and in state r8 *true* to final. To all other literals in each state *false* is assigned.

With the help of the Boolean assignment β matching of an open net with a *BSA* can be defined as follows.

Definition 15 (Matching). *Let $TS(M)$ be the behavior of an open net M and let B^{Φ} be a BSA such that $TS(M)$ and B have the same transition labels. Then M* matches *with B^{Φ} iff B weakly simulates $TS(M)$ using a relation $\varrho \subseteq Q_{TS(M)} \times Q_B$ such that for each $[q_M, q_B] \in \varrho$: $\beta_{TS(M)}(q_M) \models \Phi(q_B)$. Let $Match(B^{\Phi})$ denote the set of all open nets that match with B^{Φ}.*

Consider again Fig. 15. $TS(GS_1)$ matches with the *BSA* in Fig. 15(c), i.e., Fig. 15(c) weakly simulates $TS(GS_1)$ and in each pair of states of the weak simulation relation the assignment β assigns true to sufficiently many literals such that the formula holds. As a counterexample, $TS(GS_1)$ does not match with the *BSA* in Fig. 15(d). Observe the existence of a weak simulation relation. But being in related states [r2, q2], r2 assigns only *true* to ?food yielding (*true* $\wedge$ *false*) $\vee$ *false* which is *false*. $TS(GS_2)$ matches with none of these *BSA*s. In case of Fig. 15(c), being in related states [r6, s3], a τ transition is possible in r6 yielding related states [r8, s3] in the weak simulation (note that by Definition 12 the *BSA* may execute the empty τ sequence). However, in this pair of states the annotation of s3 is violated, because r8 only assigns *true* to final. For the same reason $TS(GS_2)$ does not match with the *BSA* in Fig. 15(d). There is a pair of states [r8, q2] in the weak simulation relation, where the annotation of q2 is violated ($TS(GS_2)$ can neither receive a message food being in its final marking nor perform a τ-labeled transition).

An operating guidelines of an open net N is a *BSA* such that every matching service M is a strategy for N and every strategy for N matches with B^{Φ}. In other words, the sets $Match(B^{\phi})$ and $Strat(N)$ must be equal.

Definition 16 (Operating guidelines, *OG*). *The* operating guidelines $OG(N)$ *of an open net* N *is a BSA such that* $Match(OG(N)) = Strat(N)$.

For uncontrollable open nets N (i.e., $Strat(N) = \emptyset$) the OG consists of a single state that is annotated with *false*, assuring that *no* open net matches with this OG.

Figures 15(c) and 15(d) depict the operating guidelines of RS_1 and RS_3. Since $TS(GS_1)$ matches with $OG(RS_1)$, we conclude that GS_1 is a strategy for RS_1. $TS(GS_1)$ does not match with $OG(RS_3)$, and thus GS_1 is not a strategy for RS_3. For the same reason GS_2 is not a strategy for RS_1 nor for RS_3.

For every controllable open net N, there exists a *most permissive strategy*, i.e., a strategy M that has the least restrictions of all strategies [29]. Thus, the behavior $TS(M)$ of M corresponds exactly to the transition system of the underlying automaton of $OG(N)$. The final states of $TS(M)$ are the states of $OG(N)$ with *final* in their annotation.

Definition 17 (Most permissive strategy). *Let* $OG(N) = [Q, MP, \delta, q_0, \Phi]$ *be the operating guidelines for a controllable open net* N*. Then, an open net* M *is the most permissive strategy for* N *iff* $TS(M) = [Q, MP, \delta, q_0, \Omega]$*, where* $\Omega = \{q \mid \textit{final occurs in } \Phi(q)\}$.

So removing the annotations in the states of $OG(RS_1)$ and $OG(RS_3)$ and adding all states that contain a literal *final* to the set of final states yields the most permissive strategy for RS_1 and RS_3, respectively.

It is worthwhile mentioning that for each open net there exists an operating guidelines that only requires negation-free annotations and a deterministic structure [27]. This eases the implementation of the matching procedure. In spite of these restrictions, an operating guidelines is able to characterize even nondeterministic service models. To this end, each Boolean annotation has a disjunct τ (see Fig. 15, for instance) as otherwise a state of a transition system that can only perform a τ transition cannot satisfy the annotation of the respective state in the operating guidelines.

5 Replacing and Refining Services

In this section we consider another important application in an SOA: service replacement and service refinement. We define an accordance relation on any two services S and S' that ensures that every compatible service for S is also compatible with S', and hence S can be replaced by S. To decide accordance we present a sufficient criterion based on projection inheritance and a precise criterion based on operating guidelines. Finally, we show how to derive a service S' from a service S by using accordance-preserving transformation rules.

5.1 A Notion of Accordance

Given an open net N, it might be necessary to change or add some functionality of N by replacing it by a new version N'. Because we assume that N does not

know each service that uses N, N' must support each *compatible* service for N, i.e., all elements in $\mathit{Strat}(N)$. With accordance we demand that every compatible service for N is compatible with N' as well. An application for accordance is the upgrade of a web shop which should not affect any client. This motivates the following notion of accordance between open nets N and N'. To this end, N and N' must be interface equivalent open nets.

Definition 18 (Interface equivalent open nets). *Two open nets M and N are* interface equivalent *iff $I_M = I_N$ and $O_M = O_N$.*

Definition 19 (Accordance). *Let N and N' be two interface equivalent open nets. N'* can replace *N* under accordance *(N' accords with N, for short) iff $\mathit{Strat}(N) \subseteq \mathit{Strat}(N')$.*

Accordance guarantees that every strategy for N is a strategy for N' as well. In addition, accordance allows N' to have more compatible services. Accordance is a pre-order, i.e., it is reflexive and transitive.

As an example, the open nets RS_1 and RS_3 in Figs. 2(b) and 3(a) are interface equivalent and RS_1 accords with RS_3. In the next subsection we present a method to prove this.

Many different accordance notions—often called conformance—exists in the literature, but there are always some differences to accordance. Vogler [30] presents a deadlock-preserving equivalence for Petri nets with an interface, but he does not distinguish between deadlocks and final markings. Fournet et al. [31] also formalize the absence of deadlocks, but their pre-order is coarser than accordance (see [32]). The approaches of [33, 34] formalize a stronger termination criterion, namely the absence of deadlocks and livelocks. In addition, [34] demands only the environment to terminate, but not the service itself.

5.2 Deciding Accordance

Deciding accordance of two open nets N and N' is a nontrivial problem, because we have to compare the two possible infinite sets of strategies $\mathit{Strat}(N)$ and $\mathit{Strat}(N')$. We introduce two approaches for deciding accordance. One approach, projection inheritance, decides accordance on the net structure of N and N'. The second approach uses the operating guidelines $OG(N)$ and $OG(N')$, i.e., the compact characterizations of $\mathit{Strat}(N)$ and $\mathit{Strat}(N')$, to decide accordance.

Projection Inheritance Inheritance is one of the key concepts of object-orientation. In object-oriented design, inheritance is typically restricted to the static aspects (e.g., data and methods) of an object class. In many cases, however, the dynamics is of prime importance. Therefore, *projection inheritance* [35] focuses on the dynamics. Projection inheritance compares process models by establishing a subclass-superclass relationship. The subclass process is indeed a subclass if it inherits particular dynamic properties of its superclass.

Projection inheritance is based on *branching bisimulation* [36] (to compare the processes) and *abstraction* (to hide tasks). The assumption is that the subclass adds tasks to the superclass such that after hiding the additional tasks both are equivalent. The basic idea of projection inheritance can be characterized as follows:

> "If it is not possible to distinguish the behaviors of x and y when arbitrary methods of x are executed, but when only the effects of methods that are also present in y are considered, then x is a subclass of y" [35].

Projection inheritance was defined for workflow nets in [35], but in this definition projection inheritance refers to "methods" rather than the "sending and receiving of messages". In [37] projection inheritance has been reformulated for open nets by the following mapping: A transition that is connected to an interface place presents a method present in both the superclass and the subclass.

We continue by defining branching bisimulation for transition systems (and hence also for open nets). In order to apply this equivalence notion in our setting, branching bisimulation should guarantee that, for each final state of TS, there exists a final state in TS' and both states are related by branching bisimulation.

Definition 20 (Branching bisimulation). *Two labeled transition systems TS and TS' are* branching bisimular *iff there exists a symmetric relation ϱ_{bb} such that $[q_0, q_0'] \in \varrho_{bb}$ and, for all q_1, q_1' holds: If $[q_1, q_1'] \in \varrho_{bb}$ and $q_1 \xrightarrow{\alpha} q_2$, then either*

- *$\alpha = \tau$ and $[q_2, q_1'] \in \varrho_{bb}$ or*
- *there are q_2', q_3' such that $q_1' \xrightarrow{\tau^*} q_2' \xrightarrow{\alpha} q_3'$, $[q_1, q_2'] \in \varrho_{bb}$, and $[q_2, q_3'] \in \varrho_{bb}$.*

Furthermore, for each final marking $q \in Q_F$ holds: if $[q, q'] \in \varrho_{bb}$, then either $q' \in Q'_F$ or there exists a transition sequence τ^* starting from q' that contains a state $q_1' \in Q'_F$ with $[q, q_1'] \in \varrho_{bb}$.

To decide whether two open nets are related by projection inheritance, it is sufficient to check if their behaviors are branching bisimular. In contrast to [35], we do not need to define an abstraction operator. In our mapping, the comparison of the two open nets is restricted to the transitions that are connected to an interface place. We abstract from all other transitions by labeling them with τ. The labeling, however, is fixed in Definition 10 (transition label) and thus no additional definition of an abstraction is necessary. Consequently, we can define projection inheritance of two open nets as follows.

Definition 21 (Projection inheritance). *Two open nets N and N' are related by* projection inheritance *iff their behaviors are branching bisimular.*

Note that projection inheritance is an equivalence.

As an example, consider $TS(GS_1)$ and $TS(GS_2)$ in Figs. 15(a) and 15(b), respectively. Although $TS(GS_2)$ simulates $TS(GS_1)$ they are not branching bisimular. The reason is that the τ transition in state r6 yields a relation between states

r2 and r8 which obviously violates branching bisimulation. Thus, GS_1 and GS_2 are not related under projection inheritance.

In [37] we have proven that the accordance notion is more liberal than projection inheritance, i.e., projection inheritance implies accordance (in both directions). This gives a sufficient criterion for deciding accordance.

Theorem 1 (Projection inheritance implies accordance [37]). *Let N and N' be two open nets. If N and N' are related by projection inheritance, then N' accords with N and N accords with N'.*

Although the notion of projection inheritance preserves all strategies, it turns out that in practice it is too restrictive. In other words, N accords with N' and N' accords with N does in general not imply that N and N' are related by projection inheritance. This is mainly caused by the fact that projection inheritance looks at the structure of the nets rather than the exchange of messages. For example, when messages are sent, their order does not really matter. This is caused by the fact that we consider asynchronous message passing, i.e., messages may be consumed in a different order than they were produced. Nevertheless, projection inheritance will differentiate between different orderings of sending messages. As another example, open nets RS_1 and RS_3 are not branching bisimular, but RS_1 accords with RS_3.

Checking Accordance with Operating Guidelines We consider now a more liberal refinement notion that is necessary *and* sufficient.

Remember that we need to compare the sets $\mathit{Strat}(N)$ and $\mathit{Strat}(N')$ in order to decide accordance of N and N'. The problem is that the set *Strat* may correspond to an infinite set of open nets. With the operating guidelines of N and N' we have, however, a compact representation of $\mathit{Strat}(N)$ and $\mathit{Strat}(N')$ which can be used to decide accordance. To this end, we define a refinement relation $\sqsubseteq$ for operating guidelines. Informally, $OG(N) \sqsubseteq OG(N')$, i.e., $OG(N')$ refines $OG(N)$, if and only if there is a simulation relation between the states of $OG(N)$ and $OG(N')$ such that the annotations in $OG(N)$ imply the annotations in $OG(N')$. Here we need a (strong) simulation relation. However, operating guidelines are deterministic (see Definition 13) and for deterministic transition systems the notions of (strong) and weak simulation are equivalent.

Definition 22 (Refinement of OGs). *Let N and N' be interface equivalent open nets and let $OG(N) = [Q, MP, \delta, q_0, \Phi]$ and $OG(N') = [Q', MP', \delta', q_0', \Phi']$ be the corresponding operating guidelines. Then, $OG(N) \sqsubseteq OG(N')$ (i.e., $OG(N')$ refines $OG(N)$) iff there is a simulation relation $\xi \subseteq Q \times Q'$ such that for all $[q, q'] \in \xi$, the formula $\Phi(q) \Rightarrow \Phi'(q')$ is a tautology.*

As an example, consider the two operating guidelines $OG(RS_1)$ and $OG(RS_3)$ in Fig. 15. RS_1 and RS_3 are interface equivalent and $OG(RS_1)$ simulates $OG(RS_3)$, i.e., each step in $OG(RS_3)$ can be mimicked in $OG(RS_1)$. Furthermore, the annotations of $OG(RS_3)$ imply the annotations in $OG(RS_1)$. For example, $\mathsf{!order} \vee \tau$

implies !order$\vee$!money$\vee\tau$ in $[\mathsf{q1}, \mathsf{s1}] \in \xi$, (?food$\wedge$final)$\vee\tau$ implies !money$\vee$?food$\vee\tau$ in $[\mathsf{q2}, \mathsf{s3}] \in \xi$, etc. Consequently, we have $OG(RS_3) \sqsubseteq OG(RS_1)$. It is easy to observe that $OG(RS_1) \sqsubseteq OG(RS_3)$ does not hold, because $OG(RS_3)$ does not simulate $OG(RS_1)$.

The relation $\sqsubseteq$ is a pre-order. With the help of the next theorem we show that $OG(N')$ refines $OG(N)$ iff N' accords with N and thus it can be used to decide accordance of N and N'. This result has been first introduced in [37] for acyclic open nets and has been extended to cyclic open nets in [32].

Theorem 2 (Checking accordance [32]). *Let N and N' be two open nets and let $OG(N)$ and $OG(N')$ be the corresponding operating guidelines. Then, $OG(N) \sqsubseteq OG(N')$ iff $Strat(N) \subseteq Strat(N')$.*

Based on the above consideration we conclude that $Strat(RS_3) \subseteq Strat(RS_1)$, and hence RS_1 accords with RS_3.

The value of Theorem 1 and Theorem 2 is that accordance can be checked independently of the services that use N, and only N and N' have to be known to decide accordance.

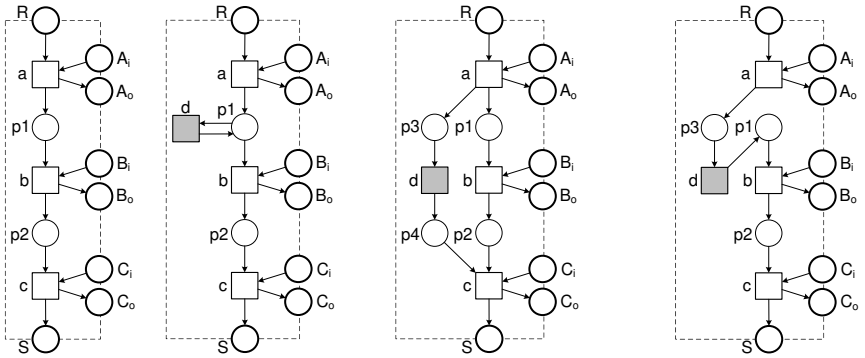
5.3 Refining Services

In the previous section we have presented an algorithm to decide for two given open nets N and N' whether N' accords with N and thus can replace N without violating any strategy for N. However, designing N' is a nontrivial and error-prone task even for experienced service designers. In order to support service designers, we introduce an approach to refine open nets. Given an open net N we want to incrementally transform N to an open net N' such that every transformation step preserves accordance. To this end, fragments of N are incrementally replaced by other fragments. In this approach, a fragment M of N is replaced by another fragment M' yielding the open net N'. We prove that if M' accords with M, then N' accords with N. The results we are going to present in this section have been published in [37].

An open net M is a fragment of an open net N if there is an open net N_{rest} and the composition of M and N_{rest} is the open net N. The set of interface places of M 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 M is the empty marking and the set of final markings is the singleton set with the empty marking.

Definition 23 (Fragment). *Let M be an open net with $m_0 = \underline{0}$ and $\Omega = \{\underline{0}\}$. Open net M is a* fragment *of an open net N iff there exists an open net N_{rest} such that $N = M \oplus N_{rest}$.*

As an example, consider the open net GS_1 in Fig. 2(a). A possible fragment M would be the open net with $P_M = \{\mathsf{p2}, \mathsf{p3}, \mathsf{food}\}$, $T_M = \{\mathsf{eat}\}$ and the adjacent arcs. In this case $R_M = \{\mathsf{p2}\}$ and $S_M = \{\mathsf{p3}\}$.



(a) M_0. (b) M_1: Adding a loop to M_0. (c) M_2: Putting transition d in parallel to b. (d) M_3: Inserting transition d in-between a and b.

Fig. 16. Accordance-preserving transformation rules based on projection inheritance.

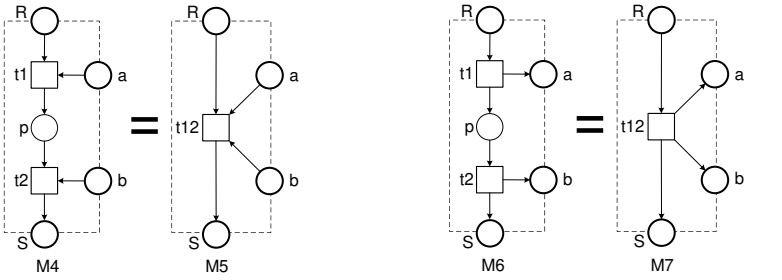
The next theorem states that if an open net N has a fragment M and there is another fragment M' that accords with M, then we can replace M by M' without affecting any strategy for N. Such transformations can be applied incrementally and thus refine a service specification to an implementation by applying transformation steps. The resulting implementation is correct by construction, i.e., it preserves all strategies of the specification.

Theorem 3 (Justification of transformation rules [37]). *Let $N_1 \oplus N_2$ be a deadlock-free open net composition. Let M be a fragment of N_1, and let N_{rest} be an open net such that $N_1 = M \oplus N_{rest}$. For any open net M' that accords with M, the composition $(M' \oplus N_{rest}) \oplus N_2$ is deadlock-free.*

Inheritance-preserving Transformation Rules Based on the notion of projection inheritance, three *inheritance-preserving transformation rules* have been defined in [35]. These rules correspond to design patterns for extending a superclass to incorporate new behavior: (1) adding an internal loop (2) put a new internal transition in parallel with existing transitions, and (3) insert an internal transition in-between existing transitions.

We exemplify these rules in Fig. 16. Figure 16(a) represents a fragment M_0 of an open net N. M_0 contains transitions a, b and c. By Definition 23, there are no other connections of a, b, c, p1 and p2 than those shown in Fig. 16(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. Note that $\mathsf{A_i}$, $\mathsf{A_o}$, $\mathsf{B_i}$, $\mathsf{B_o}$, $\mathsf{C_i}$, $\mathsf{C_o}$ 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 M_1, M_2 and M_3. For example, M_1 is obtained by adding transition d to M_0.

If one considers the behavior of these open nets, then M_0, M_1, M_2 and M_3 are branching bisimular. Hence each pair of these four fragments is related by pro-



(a) Rule 1: $Strat(M_4) = Strat(M_5)$ (b) Rule 2: $Strat(M_6) = Strat(M_7)$

Fig. 17. Rule 1 and Rule 2.

jection inheritance. From Theorem 1 we conclude that the three transformation rules depicted in Fig. 16 preserve accordance in both directions.

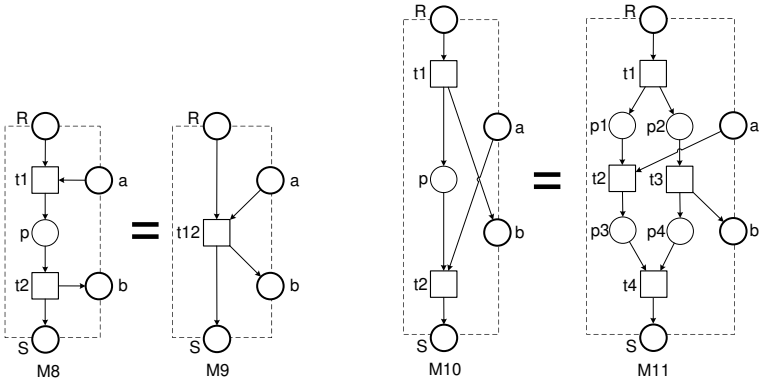
Inheritance-preserving transformation rules only change internal transitions of an open net. Next we present transformation rules that affect transitions that are adjacent to an interface place.

Accordance-preserving Transformation Rules We present five accordance-preserving transformation rules. Four of these rules preserve accordance in both directions and one rule preserves accordance only in one direction. Although these transformation rules are sufficient, they are not complete, meaning they do not cover all possible service implementations. Given an open net N, each transformation rule specifies a fragment M of N (see Definition 23) which can be replaced by another open net M' yielding an implementation of N. Theorem 3 justifies that this replacement preserves all strategies for N. As in case of the inheritance-preserving transformation rules, the rules are only informally described and illustrated by help of some figures.

Rule 1 is depicted in Fig. 17(a) and specifies that a sequence of receiving transitions can be merged, and the messages can be sent simultaneously. Rule 1 preserves accordance in both directions. Thus, we can derive that a sequence of receiving transitions can also be reordered or can be executed concurrently. Reordering of receiving transitions and executing receiving transitions concurrently preserve accordance in both directions. The same holds for a sequence of sending transitions. The corresponding rule (Rule 2) is depicted in Fig. 17(b).

Rule 3 in Fig. 18(a) combines sending and receiving transitions. A receiving transition followed by a sending transition can be executed simultaneously while preserving accordance in both directions. Due to Rules 1 and 2, Rule 3 can be generalized to a sequence of receiving transitions followed by a sequence of sending transitions.

So far, we excluded the possibility that a sending transitions is followed by a receiving transitions. Rule 4, depicted in Fig. 18(b), specifies that first sending

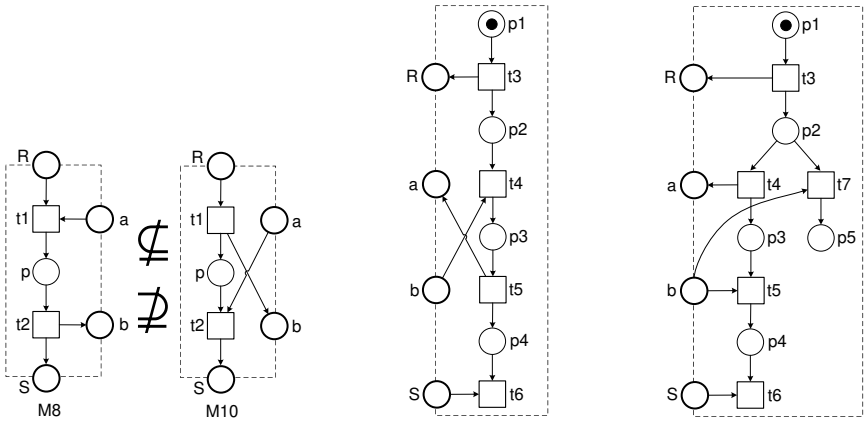


(a) Rule 3: $Strat(M_8) = Strat(M_9)$ (b) Rule 4: $Strat(M_{10}) = Strat(M_{11})$

Fig. 18. Rule 3 and Rule 4.

and then receiving a message can also be executed concurrently and vice versa. Rule 4 preserves accordance in both directions, too.

Figure 19(a) shows that first sending and then receiving cannot be reordered in general: M_{10} does not accord with M_8 and M_8 does not accord with M_{10}. Suppose the final markings to be equivalent to the singleton set with the empty marking. Then, the open net depicted in Fig. 19(b) is a strategy for M_{10}, but no strategy for M_8, and the open net depicted in Fig. 19(c) is a strategy for M_8 but not for M_{10}.



(a) Anti-pattern: $Strat(M_8) \neq Strat(M_{10})$

(b) Strategy for M_{10} but not for M_8.

(c) Strategy for M_8 but not for M_{10}.

Fig. 19. Counterexamples.

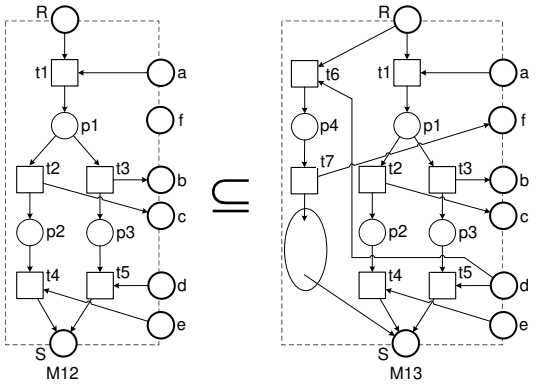


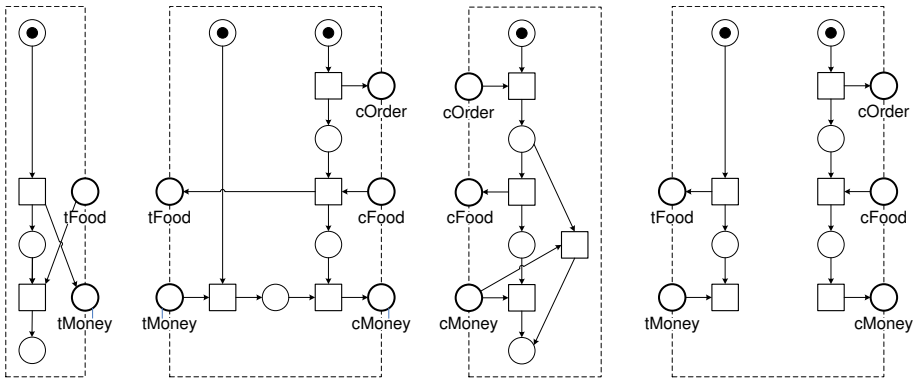
Fig. 20. Rule 5 (adding an alternative branch): $Strat(M_{12}) \subseteq Strat(M_{13})$.

Figure 19(a) can be seen as an anti-pattern, however, not in the sense of the anti-patterns mentioned in Sect. 2. The main difference is that Fig. 19(a) refers to a problematic modification while the earlier anti-patterns refer to problematic service definitions.

From the anti-pattern shown in Figure 19(a) it follows that first receiving and then sending (cf. M_8) cannot be transformed to a fragment that sends and receives concurrently (M_{11}), because we could transform the latter net to M_{10} by applying Rule 4. Consequently, first receiving then sending does not accord to sending and receiving concurrently and vice versa. Analogously, first sending then receiving (M_{10}) cannot be transformed to sending and receiving simultaneously (M_9), because the latter can be transformed to M_8 by applying Rule 3. Thus, first sending then receiving does not accord with sending and receiving simultaneously and vice versa.

Rule 5 specifies a way to add an alternative branch to a fragment M_{12} depicted on the left hand side of Fig. 20. The fragment M_{12} first receives a and then enters either the left or the right branch. In the left (right) branch, message b (c) is sent, and then message d (e) is received. The fragment M_{12} can be transformed to M_{13} by adding an alternative branch. In this branch, d is received, and then a message f is sent. Afterwards, this branch can be arbitrary, i.e., there can be any continuation (including direct continuation in S) of this net as illustrated by the ellipse. Rule 5 preserves accordance in one direction only. The intuition behind this rule is that a strategy of M_{12} has to wait for the decision of M_{12} which branch it will enter. Otherwise, it could happen that an environment sends d, but M_{12} enters the left branch and waits for message e.

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 [13] (known for general Petri nets) 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 compat-



(a) Tourist (b) Adapter (c) Cook (d) Adapter?

Fig. 21. Running example for adapter generation

ibility of the whole net is studied in [30]. These results could be applied in our setting.

6 Integrating Services Using Adapters

Service-oriented computing aims to create complex services by composing less-complex services. As services are often developed independently, upon composition they may turn out to be incompatible. In this section we discuss some sources of incompatibility and some ways to resolve them. This section is primarily based on [38].

Figure 21 contains the running example for this section. The tourist modeled in Fig. 21(a) enters a restaurant in a foreign country. The tourist noticed something like a special offer on the door, but he does not understand the local language. So he just places the required fee on the table and waits for food. The local cook modeled in Fig. 21(c), however, insists on an order before preparing any meal. Moreover, if the cook already gets some money before serving the food, he may immediately stop cooking.

When integrating some services that have been developed independently, some typical kinds of incompatibilities are:

- names of the message types;
- encoding of similar message types;
- semantics of similar message types;
- order in which messages are expected or transmitted.

It is clear that the open nets in Fig. 21(a) and Fig. 21(c) have different sets of interface places, even if we ignore the 't' and 'c' prefixes. Moreover, if we try to compose them by fusing the obvious combinations of interface places, and hide (i.e., make them internal) the other interface places, then the result contains a

deadlock: the cook waits for an order, while the tourist gives some money and waits for food.

In this section we focus on the last kind of incompatibility, which we call *behavioral incompatibility*; however, we will not ignore the other ones. For simplicity reasons, we assume that the name of each message port coincides with the name of the message type that can be transmitted over the channel.

If the services to be composed are incompatible, there are a few options:

- replace some of the services by similar services that are compatible;
- change the implementation of some of the services;
- introduce an *adapter service* that bridges the incompatibilities.

In terms of the running example, these options can correspond to, respectively, going to another restaurant with tourist-friendly personnel, attending a language-and-culture course, or hiring a tour guide.

In this section we focus on the situation where the services have already been selected, and their implementation cannot be changed; in this case, adapters are the most obvious solution. To be able to discuss adapters as an additional service in between the given services, we assume that the interfaces of the given services are disjoint; this can be achieved by renaming. In the running example this has been achieved through the 't' and 'c' prefixes for the names of the interface places.

In the remainder of this section, we first discuss the ingredients of an adapter specification, and a specific language for it. Then we show how it can be used to automatically generate an adapter, including a discussion on some of the design decisions to be made.

6.1 Adapter Specification

In this section we discuss the contents of an adapter specification. Behavioral incompatibilities typically manifest itself in deadlocks of the composed system. Therefore the first ingredient of the adapter specification is a behavioral property, in our case deadlock freedom, on the composed system (which, by definition, guarantees that the composed system is closed). To be able to check whether an adapter establishes deadlock freedom, the adapter specification should also include models of the given services, say, open nets N_1 and N_2. For simplicity reasons, we only discuss the integration of two given services, although it can easily be scaled up to any number of services.

For the running example, an adapter service that establishes deadlock freedom is modeled in Fig. 21(b). It gives a default order to the cook, and then passes on the food when it arrives. In the mean time it accepts the money from the tourist, but it only forwards the money once the cook has actually served the food. The composition of Fig. 21(a), Fig. 21(b) and Fig. 21(c) is indeed a closed net and it is deadlock-free.

On the other hand, the service from Fig. 21(d) also establishes this property, but is this a proper adapter? Such an example illustrates that the specification so

far admits adapters that are the composition of two unconnected components A_1 and A_2 such that both $N_1 \oplus A_1$ and $N_2 \oplus A_2$ are deadlock-free, but independently of each other. Hence it admits adapters that can arbitrarily create and delete messages, including real goods like food and money, which is not very realistic.

Apart from the requirement on the composed system, a requirement on the internals of the adapter is needed such that it can actually be implemented. To this end, we extend the adapter specification with the set of elementary activities (from a semantical perspective) that can be used in the adapter.

Thus the adapter specification consist of the following three parts:

- models of the services to be composed;
- behavioral property to be established by the composed system;
- elementary activities for the adapter.

6.2 Elementary Adapter Activities

In this section we explore the typical elementary activities for an adapter, and we describe a way to specify them. As the given services have an asynchronous communication interface, the basic activities of an adapter are receiving a message from an interface port, and sending a message to an interface port. As these are separate tasks, it also possible to delay the forwarding of messages.

Internally, the activities of an adapter include ways to deal with messages; thus reflecting semantic dependencies between certain message types. Most approaches [39–44] agree that the activities of an adapter should include the following activities:

- **Create a message**: This is possible for simple control messages, and messages with a default value. However, it is impossible for messages containing important data such as passwords, and personal data of a user.
- **Copy a message**: This is possible for most electronic messages, although it could be inappropriate for single-use data such as transaction numbers. It is also inappropriate for messages that represent real goods.
- **Delete a message**: This is possible for most electronic data, while it is inappropriate for real goods.
- **Transform/Split/Merge some messages**: This is possible if the underlying transformation routine is provided, e.g., calculating a metric measure from an imperial one, or deriving a city name from a zip code.

Based on these example activities, it becomes clear that the applicability of an activity to particular message types strongly depends on semantic considerations that depend on the message types. As a result, we conclude that the possible activities of an adapter must be specified per message type.

We specify the capabilities of an adapter using a *Specification of the Elementary Activities* (SEA). Given a set of message types MT, an SEA is a set of transformation rules on these message types. The set MT contains at least the names of the interface ports of the given services, but it may also contain auxiliary message types.

Table 1. Examples of elementary activities in terms of transformation rules

Elementary activity	Possible transformation rule
Create a	$\mapsto\ a$
Copy a	$a\ \mapsto\ a, a$
Delete a	$a\ \mapsto$
Transform a, b, c into d, e	$a, b, c\ \mapsto\ d, e$ or $a, b, c\ \mapsto\ a, b, c, d, e$
Split a into b, c, d	$a\ \mapsto\ b, c, d$ or $a\ \mapsto\ a, b, c, d$
Merge a, b, c into d	$a, b, c\ \mapsto\ d$ or $a, b, c\ \mapsto\ a, b, c, d$

Definition 24 (Specification of the Elementary Activities (SEA)). *Given a set of message types MT. An SEA over the message types MT is a set of transformation rules of the shape*

$$X\ \overset{Z}{\mapsto}\ Y$$

where X and Y are bags (multi sets) over the set MT, and where Z is a total function from messages of the types X to messages of the types Y.

Such a rule denotes that, using the transformation Z, a message of each type in X is consumed, and a message of each type in Y is produced. After receiving some messages, the adapter can apply several transformations to the internally available messages before sending any messages. Synthesizing an adapter then boils down to applying these rules in a right order, and sending and receiving messages to and from the interface at a right moment.

For the synthesis of an adapter, we can largely abstract from the actual data transformations Z. Therefore we often omit Z in the transformation rules, but in Sect. 7 we will discuss how Z can be integrated in the synthesized adapters.

Table 1 shows some examples of activities in terms of transformation rules. For some rules, we give two versions: one for real items and one for electronic items. Some more-complicated patterns would require multiple rules: a typical "collapse" pattern, where an arbitrary series of a messages has to be merged into one message b, could be modeled using the two rules $\mapsto\ b$ (create an empty 'b') and $a, b\ \mapsto\ b$ (add a single 'a' to an existing 'b').

Many languages have been proposed that are similar to the SEA rules, but there are subtle but essential differences. Languages like in [43, 44] do not support multiple alternative rules, like two SEA rules $A \mapsto B$ and $A \mapsto C$ that specify that at any time the adapter can choose which rule to apply. The rules in languages like in [39–41] have no direction [45], while the SEA rules are asymmetric.

As an SEA represents semantical dependencies, it may be possible to use techniques related to semantic web and ontologies to construct an SEA. As these are research areas on their own, we only focus on using a given SEA. The approach of [44] is interesting as it presents an interactive approach to find and refine SEA-like specifications for adapters using mismatch trees.

For the running example, a possible SEA is the one in Table 2. Note that the adapter from Fig. 21(d) can violate the last two transformation rules. The

adapter in Fig. 21(b) obeys the rules; moreover, it uses each rule exactly once, but this is a coincidence (see, e.g., the running example from [38]).

Table 2. Running example: SEA

	$\mapsto$ cOrder
cFood	$\mapsto$ tFood
tMoney	$\mapsto$ cMoney

6.3 Adapter Generation

In this section we discuss how to generate an adapter. The adapter specification consists of the given open nets N_1 and N_2, the deadlock-freedom property, and an SEA. An adapter is an open net such that its composition with the given open nets N_1 and N_2 is closed and deadlock-free. Furthermore, to ensure implementability, an adapter may only be constructed from the elementary activities described by the SEA.

Adapter generation without SEA Let us first ignore the SEA, and explore the basic construction of an adapter. Such an adapter is defined as an open net A such that $(N_1 \oplus N_2) \oplus A$ is deadlock-free. That is, A is a strategy for the composition $N_1 \oplus N_2$. Such an adapter can be computed as a witness for controllability of $N_1 \oplus N_2$.

Adapter generation with an SEA The SEA imposes additional restrictions on the adapter. Most approaches to adapter generation modify the computation of a strategy with (on-the-fly) removal of the branches that violate this requirement. The result is typically a complex custom algorithm [39–41]. In [46, 47] it is shown that there exists a dual approach that first integrates the SEA restrictions with the given open nets, such that afterwards every computed strategy is an adapter.

A conceptually simpler approach [38] is to immediately translate the SEA rules into a new open net that is part of the adapter. We call this part of the adapter the engine, and the remainder of the adapter the controller. This results in a two-piece adapter that separates *data* (implementability) and *control* (behavioral property).

The *engine* E is an open net that encodes all the elementary activities from the SEA. It has an interface with the given open nets N_1 and N_2, and hence it can ensure that all outgoing messages are obtained from the incoming messages using the SEA rules only. The engine E has an additional interface to a *controller* C. This interface allows the controller to decide in which order the elementary activities are performed. Figure 22 shows a schematic representation of this structure.

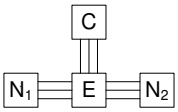


Fig. 22. Conceptual structure

Before showing an example encoding of an SEA in terms of an engine, we first determine how it can be used to generate an adapter. Formally, given an engine E for the SEA, we want to construct a controller C such that $(N_1 \oplus N_2) \oplus (E \oplus C)$ is deadlock-free. Such a controller can be computed as a witness for controllability of $(N_1 \oplus N_2) \oplus E$. As every C yields an SEA-based open net $E \oplus C$, every witness C for controllability of the open net $(N_1 \oplus N_2) \oplus E$, yields an SEA-based adapter $E \oplus C$. So deadlock freedom is guaranteed independently of any specifics of the used engine.

Lemma 1 (Two-piece behavioral adapter [38]). *Given any open nets N_1, N_2, and E. For every strategy C for the composed open net $(N_1 \oplus N_2) \oplus E$, the composed open net $E \oplus C$ is an adapter for the open nets N_1 and N_2.*

So, at a conceptual level, this adapter-synthesis approach consists of the following steps, given the open nets N_1 and N_2 and an SEA:

1. generate an engine E from the SEA and the interface of the open net $N_1 \oplus N_2$;
2. synthesize a controller C as a strategy for the open net $(N_1 \oplus N_2) \oplus E$;
3. compose engine E and controller C to obtain the final adapter $A = E \oplus C$.

Note that the engine is used twice: for generating a controller, and as part of the final adapter. The constraint-oriented approach from [48, 49] uses a "Store" that contains part of the functionality of an engine, but it is not part of the final adapter.

Regarding the running example, the adapter from Fig. 21(b) has been generated in this way. In what follows we first discuss the encoding of an engine and then the selection of a controller.

6.4 Encoding an SEA as an Engine

In this section we show how an SEA can be encoded as an engine E modeled by an open net. For simplicity of presentation, we assume that for each SEA rule, the bags before and after the $\mapsto$ are sets; the general case follows analogously using open nets with arc multiplicities.

Let N be the composition $N_1 \oplus N_2$. Let I_N and O_N denote the (disjoint sets of) input and output places, respectively, of N. Let K be a set such that the SEA consists of the rules $X_k \mapsto Y_k$, for any $k \in K$. Let MT be a set of message types, containing (the types of) the sets of places I_N and O_N, and the set of message types used in K. The SEA may contain auxiliary message types (ones that do not occur in the given open nets), and hence we have $I_N \cup O_N \subseteq MT$.

For defining the open net E, we use names from the space $(MT \cup K) \times \{e, n, c, r, s\}$, where e, n, c, r, and s denote fresh names that do not occur in the given open nets. Moreover, we assume that the sets MT and K are disjoint.

The interface of open net E consists of output places I_N, input places O_N (i.e., the interfaces of the given open nets in opposite orientation), and an interface (defined later on) for interaction with the controller (cf. Fig. 22). For each message type $m \in MT$, we introduce in the open net E an internal place (m, c), where c refers to "conceptual". In the initial and final markings, the internal places are empty.

The open net E has three kinds of transitions. For every input place $o \in O_N$, there is a transition (o, r), where r refers to "receive", to move arriving messages from interface place o to their internal place (o, c). For every transformation rule $X_k \mapsto Y_k$, where $k \in K$, there is a transition (k, c) to perform the actual transformation in terms of the internal places. Finally, for every output place $i \in I_N$, there is a transition (i, s), where s refers to "send", to move messages from their internal place (i, c) to interface place i.

What remains is to describe the interface of the engine E with the controller C. For every transition (o, r), where $o \in O_N$, there is an output place (o, n) that notifies an arrived message o. For every transition (k, c), where $k \in K$, there is an input place (k, e) that enables transformation rule k, and an output place (k, n) that notifies an execution of transformation rule k. Finally, for every transition (i, s), where $i \in I_N$, there is an input place (i, e) that enables the delivery of a message i. Thus this engine is formally defined as:

Definition 25 (Engine). *Let I, O, MT, K be as introduced before. The engine E is defined as an open net with the following constituents:*

$$P = (MT \times \{c\}) \ \cup \ I \ \cup \ O; \quad m_0 = \underline{0}; \quad \Omega = \{\underline{0}\};$$
$$I = O_N \ \cup \ (K \times \{e\}) \ \cup \ (I_N \times \{e\}); \quad O = I_N \ \cup \ (K \times \{n\}) \ \cup \ (O_N \times \{n\});$$
$$T = (O_N \times \{r\}) \ \cup \ (K \times \{c\}) \ \cup \ (I_N \times \{s\});$$

$$F = F_r \cup F_c \cup F_s;$$
$$F_r = \bigcup_{o \in O_N} \ \{ \ [o, (o, r)], \ [(o, r), (o, n)], \ [(o, r), (o, c)] \ \};$$
$$F_c = \bigcup_{k \in K} \ (\ \{[(m, c), (k, c)] \mid m : \ m \in X_k\} \ \cup \ \{[(k, e), (k, c)]\} \ \cup \ \{[(k, c), (k, n)]\} \ \cup \ \{[(k, c), (m, c)] \mid m : \ m \in Y_k\} \ );$$
$$F_s = \bigcup_{i \in I_N} \ \{ \ [(i, c), (i, s)], \ [(i, e), (i, s)], \ [(i, s), i] \ \}.$$

Figure 23 models the engine for the running example. The left and the right interfaces are for the tourist and the cook, respectively, while the top interface is for a controller; in this figure we use simplified names for the interface with the controller. As far as the transfer of money is concerned, the engine looks as follows. When there is a token on the interface place tMoney, transition (tMoney,r) transfers it to the internal place (tMoney,c) and sends a notification to the controller. Afterwards, transition (money,c) can transform a token from place (tMoney,c) into a token in place (cMoney,c); this is only possible if the controller has enabled this transition, and then the controller is notified. Fi-

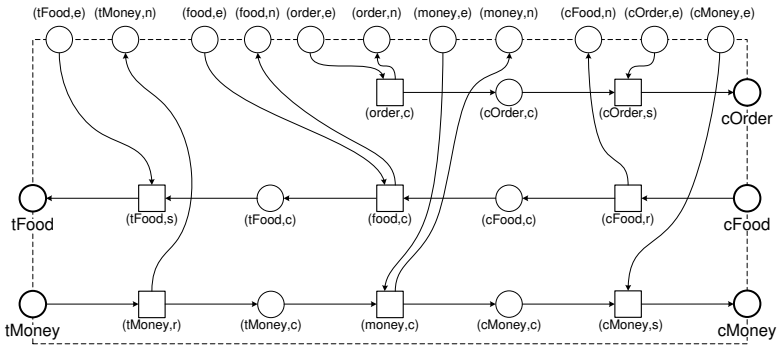


Fig. 23. Running example: engine

nally, transition (cMoney,s) transfers a token from internal place (cMoney,c) to interface place cMoney, but only if the controller has enabled this transition.

So, each internal place (and each interface place with the given open nets) is associated with a particular message type. In terms of these places, each single transition either follows the SEA rules, or transfers tokens from places that are associated with the same message type. The interface places with the controller only restrict the order in which the transitions of the engine can fire. Thus the engine guarantees that the generated adapter adheres to the SEA, independently of any specifics of the controller.

However, engines are not unique. For example, in some cases some of the interface places between the engine and the controller can be removed without changing the adapters that can be generated. In [46] techniques are presented to compare different engines in terms of the resulting adapters.

6.5 Selecting a Controller

In this section we consider the selection of a controller for a two-piece adapter. In general, the open net $(N_1 \oplus N_2) \oplus E$ has several strategies, and every strategy can be used as a controller for an adapter (see Lemma 1).

A particularly interesting strategy is the most-permissive strategy (see Definition 17), as it represents somehow the largest behavior that can enforce the behavioral property to be established. In this way, it causes the smallest constraints on the interface of the controller. A potential drawback of a most-permissive strategy is its size, but it exhibits a tremendous amount of non-determinism, which, in many cases, results in nice concurrency in terms of open nets.

On the other hand, there are usually many strategies that are smaller than the most-permissive one. Such strategies often restrict the interaction with the given open nets, and, in particular, reduce concurrency.

To sum up, both most-permissive strategies and arbitrary small strategies have specific advantages and disadvantages which more or less complement each other. This gives an opportunity to make a trade-off between the complexity of

adapter synthesis and the quality of the resulting adapter (in terms of its size and run-time behavior).

This is also related to two kinds of application scenario's for adapters. In the first one, a set of services is carefully selected, and then as a final engineering step an adapter is calculated. In the second one, a user at run-time selects some services, and an adapter is required to make these services work together. In the first scenario's larger run-times are permissable than in the second scenario, but also a higher quality is expected.

7 Tool Support

In this section we sketch how the techniques from the previous sections have been implemented in research tools. All described tools are available at

`http://www.service-technology.org/tools/`

7.1 Translating Services to Open Nets

In practice, services are not modeled by formalisms such as Petri nets. Instead, a number of service description languages have been proposed by several industrial consortiums. The most-prominent language is BPEL.

For BPEL, there exists a feature-complete open net semantics [50] and a compiler, BPEL2oWFN, to translate a BPEL process to an open net. This semantics is feature-complete in the sense that it supports all concepts of BPEL including control flow, data flow, message flow, exception handling, and compensation handling.

Since there is also a tool, oWFN2BPEL, to translate open nets to BPEL (using abstract processes) [51], a complete tool chain for translations between BPEL and open nets is available. Hence, all analysis methods for open nets can be used for BPEL processes.

7.2 Operating Guidelines

In Sect. 4 we have introduced the notion of operating guidelines as a compact characterization of all strategies for an open net N. Since the algorithm to compute operating guidelines explores all reachable states of N, in an implementation we have to restrict ourselves to *finite state services*. Such services can still have infinitely many strategies.

On the modeling level an open net has finitely many states if its inner structure $inner(N)$ is bounded. The composition of two bounded open nets may, however, result in an unbounded open net, because tokens may accumulate on the former interface places. To achieve a bounded open net composition, we have to restrict the number of tokens at those interface places. To this end, we need a notion of boundedness for interface places, which has been introduced in [27] as *k-limited communication*.

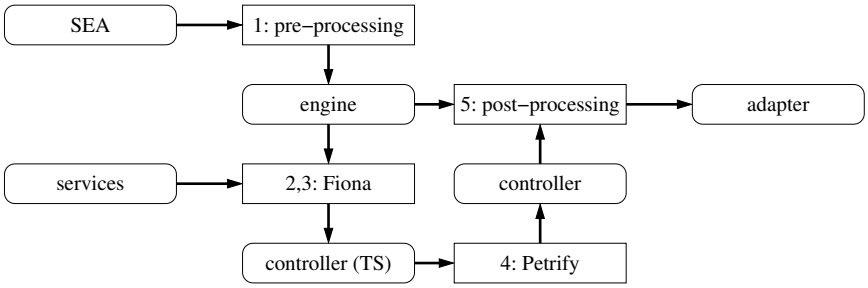


Fig. 24. Tool chain for adapter generation

For unbounded open nets, controllability has been proven to be undecidable [52]. For the implementation of our algorithms we require open nets to be bounded and to satisfy k-limited communication, for some k. Since services in practice are finite-state services, this restriction does not harm our approach.

In [27] an algorithm has been presented to compute an operating guidelines of a bounded open net. The OG construction algorithm first computes the most permissive strategy. Therefore, it starts with an over-approximation of compatible behavior of any strategy and then iteratively removes all states which cause violations of the deadlock-freedom property. Finally, the annotations are derived from information collected during the computation. If the service is uncontrollable, the algorithm eventually removes all states. The algorithm is implemented in the service analysis tool Fiona [50].

Besides computing the operating guidelines of an open net, Fiona can also be used to

- decide matching of an open net with an operating guidelines;
- decide accordance of two open nets using Theorem 2;
- compute some strategy of an open net.

7.3 Adapter Generation

In this section we discuss a Fiona-based implementation [38] of the adapter generation approach described in Sect. 6. It turns out that in the engine, the activities for message creation can easily lead to unbounded places, including interface places. To be able to compute an operating guidelines, we impose artificial bounds on these places. Using the techniques in [46], it can be shown that the resulting adapter is also an adapter for the given services without the artificial bounds. Moreover, every finite-state adapter can be synthesized if the bounds are chosen sufficiently large.

We use the tool-chain described in Fig. 24. The inputs are an open net model of each given service, and an SEA; the output is an open net model of the adapter. In what follows we briefly describe the various steps:

1. **Create an engine model from the SEA** The procedure as described in Sect. 6.4, including the required bounds, has been implemented in Fiona. By construction, all outputs to the given services have been obtained from the inputs of these services using the SEA transformation rules only.
2. **Compose the service models and the engine model** The composition of service models is supported by Fiona. Afterwards we apply structural Petri-net reduction, which consists of local graph-transformations in an open net. It preserves the interface behavior of the transformed net, but it may significantly reduce the number of reachable internal states. It is inspired by classical Petri-net reduction (like [13]). We apply it for the purpose of reducing complexity in subsequent steps.
3. **Synthesize a controller as a transition system** A controller is a strategy, and strategy synthesis is the core functionality of Fiona. The resulting controller is represented as a transition system rather than an open net.
4. **Transform this transition system into an open net** Petrify [53] is an external tool that translates a transition system into an equivalent Petri net. The resulting Petri net tends to exhibit a large degree of concurrency, and tends to be significantly more compact than the original transition system.
5. **Compose the engine and the controller into an adapter** Like before, the composition is supported by Fiona, and we apply structural Petri-net reduction afterwards. However, this time the reduction aims at simplifying the resulting structures, thus leading to a more compact Petri-net. In particular, the reduction may iron-out dead parts in the adapter (like SEA transitions that are not used) or collapse a sequence of transitions into a single transition. As such a sequence may consist of a transition that stems from the controller and another one that stems from the engine, the interface between them may become invisible in the resulting adapter.

An optional last step is to translate the open net for the adapter into an executable language. The tool oWFN2BPEL can generate an abstract BPEL process that includes an opaque activity for each transition of the open net. Remembering that SEA rules can be annotated with actual transformations (for instance in XSL), and SEA rules correspond to transitions in the engine, we can fill the opaque activities with actual code and turn them into executable activities.

Currently, we are developing engines with a synchronous interface to the controller, for which the first results show that these are more efficient. A most-permissive strategy as controller for a synchronous engine turns out to perform better than an arbitrary strategy (or a most-permissive strategy) as controller for an asynchronous engine. This applies to both the run-time of the adapter generator, and the size of the generated adapter in terms of open nets. The example adapter in Fig. 21(b) was actually generated in this way.

8 Conclusions

The shift towards service orientation was initially intended to mainly support cross-organizational processes. However, the wide adoption of service-oriented architectures shows that this paradigm shift is also important for intra-organizational processes. Monolithic information systems can now be decomposed into several smaller services. Service orientation leads to systems that can be viewed as *interacting services*. Therefore, it is vital to understand service interaction in all its aspects.

This paper studies service interaction from various angles. First of all, the paper provides a collection of *service interaction patterns*. This provides an overview of the challenges in this domain and aids in a better understanding of the important concepts. Moreover, by presenting a few anti-patterns we reveal typical pitfalls in the design of services.

Secondly, the paper *formalizes essential concepts* such as strategies, controllability, and accordance. This is done in the setting of open nets. Finally, the core of the paper focusses on three important challenges: *Exposing services* (Sect. 4), *Replacing and refining services* (Sect. 5), and *Integrating services using adapters* (Sect. 6). These challenges are non-trivial. However, the body of work centering around open nets provides a solid basis for addressing these challenges. This is illustrated by the availability of analysis tools that support all three challenges and that can also work with industrial languages such as BPEL.

Acknowledgements

Van der Aalst and Mooij participate in the Poseidon project at Thales under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

References

1. Dumas, M., Aalst, W., Hofstede, A.: Process-Aware Information Systems: Bridging People and Software through Process Technology. Wiley & Sons (2005)
2. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services Concepts, Architectures and Applications. Springer-Verlag, Berlin (2004)
3. Barros, A., Dumas, M., Hofstede, A.: Service Interaction Patterns. In Aalst, W., Benatallah, B., Casati, F., Curbera, F., eds.: International Conference on Business Process Management (BPM 2005). Volume 3649 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2005) 302–318
4. Decker, G., Puhlmann, F., Weske, M.: Formalizing Service Interactions. In Dustdar, S., Faideiro, J., Sheth, A., eds.: International Conference on Business Process Management (BPM 2006). Volume 4102 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2006) 414–419
5. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley Professional, Reading, MA (2003)

6. Mulyar, N., Aldred, L., Aalst, W.: The Conceptualization of a Configurable Multi-party Multi-message Request-Reply Conversation. In Felber, P., Pu, C., Moorsel, A., eds.: Proceedings of the OTM Conference on Distributed Objects and Applications (DOA 2007). Volume 4803 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2007) 735–753
7. Wegner, P.: Why interaction is more powerful than algorithms. Communications of the ACM **40**(5) (1997) 80–91
8. Zaha, J., Dumas, M., Hofstede, A., Barros, A., Decker, G.: Service Interaction Modeling: Bridging Global and Local Views. In: International Enterprise Distributed Object Computing Conference (EDOC 2006), IEEE Computer Society (2006) 45–55
9. Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., Guzar, A., Kartha, N., Liu, C., Khalaf, R., Koenig, D., Marin, M., Mehta, V., Thatte, S., Rijn, D., Yendluri, P., Yiu, A.: Web Services Business Process Execution Language Version 2.0 (OASIS Standard). WS-BPEL TC OASIS, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (2007)
10. Aalst, W., Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow Patterns. Distributed and Parallel Databases **14**(1) (2003) 5–51
11. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3) (2005) 35–43
12. Desel, J., Esparza, J.: Free Choice Petri Nets. Volume 40 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK (1995)
13. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE **77**(4) (1989) 541–580
14. Reisig, W.: Petri Nets: An Introduction. Volume 4 of EATCS Monographs in Theoretical Computer Science. Springer-Verlag, Berlin (1985)
15. Aalst, W.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison Wesley, Reading, MA, USA (1995)
17. Mulyar, N.: Patterns for Process-Aware Information Systems: An Approach Based on Colored Petri Nets. PhD thesis, Eindhoven University of Technology, Eindhoven (2009)
18. Russell, N., Aalst, W., Hofstede, A., Edmond, D.: Workflow Resource Patterns: Identification, Representation and Tool Support. In Pastor, O., Falcao e Cunha, J., eds.: Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05). Volume 3520 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2005) 216–232
19. Russell, N., Hofstede, A., Edmond, D., Aalst, W.: Workflow Data Patterns: Identification, Representation and Tool Support. In Delcambre, L., Kop, C., Mayr, H., Mylopoulos, J., Pastor, O., eds.: 24nd International Conference on Conceptual Modeling (ER 2005). Volume 3716 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2005) 353–368
20. Alexander, C.: A Pattern Language: Towns, Building and Construction. Oxford University Press (1977)
21. Russell, N., Hofstede, A., Aalst, W., Mulyar, N.: Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22, BPMcenter.org (2006)

22. Russell, N., Aalst, W., Hofstede, A.: Workflow Exception Patterns. In Dubois, E., Pohl, K., eds.: Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06). Volume 4001 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2006) 288–302
23. Aalst, W., Hee, K.: Workflow Management: Models, Methods, and Systems. MIT press, Cambridge, MA (2004)
24. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. In: Proceedings of the 2nd South-East European Workshop on Formal Methods 2005 (SEEFM05), Ohrid, Republic of Macedonia (2005)
25. Clarke, E., Grumberg, O., Peled, D.: Model Checking. The MIT Press, Cambridge, Massachusetts and London, UK (1999)
26. Massuthe, P., Schmidt, K.: Operating Guidelines - an Automata-Theoretic Foundation for the Service-Oriented Architecture. In Cai, K., Ohnishi, A., Lau, M., eds.: Proceedings of the Fifth International Conference on Quality Software (QSIC 2005), Melbourne, Australia, IEEE Computer Society (2005) 452–457
27. Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In Kleijn, J., Yakovlev, A., eds.: 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings. Volume 4546 of Lecture Notes in Computer Science., Springer (2007) 321–341
28. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc. (1989)
29. Wolf, K.: Does my service have partners? LNCS ToPNoC **II**(5460) (2008) 152–171
30. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. Volume 625 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1992)
31. Fournet, C., Hoare, C.A.R., Rajamani, S.K., Rehof, J.: Stuck-Free Conformance. In Alur, R., Peled, D., eds.: CAV 2004. Volume 3114 of LNCS., Springer-Verlag (2004) 242–254
32. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding substitutability of services with operating guidelines. LNCS ToPNoC **II**(5460) (2008) 172–191
33. Bravetti, M., Zavattaro, G.: Contract Based Multi-party Service Composition. In Arbab, F., Sirjani, M., eds.: FSEN 2007. Volume 4767 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2007) 207–222
34. Castagna, G., Gesbert, N., Padovani, L.: A Theory of Contracts for Web Services. SIGPLAN Not. **43**(1) (2008) 261–272
35. Basten, T., Aalst, W.: Inheritance of Behavior. Journal of Logic and Algebraic Programming **47**(2) (2001) 47–145
36. Glabbeek, R., Weijland, W.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM **43**(3) (1996) 555–600
37. Aalst, W., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: From Public Views to Private Views: Correctness-by-Design for Services. In Dumas, M., Heckel, H., eds.: Proceedings of the 4th International Workshop on Web Services and Formal Methods (WS-FM 2007). Volume 4937 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2008) 139–153
38. Gierds, C., Mooij, A., Wolf, K.: Specifying and generating behavioral service adapters based on transformation rules. Preprints CS-02-08, Institut fur Informatik, Universitat Rostock (2008)
39. Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H.R., Toumani, F.: Developing Adapters for Web Services Integration. In: Proc. CAiSE. Volume 3520 of LNCS. (2005) 415–429
40. Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. Journal of Systems and Software **74**(1) (2005) 45–54

41. Brogi, A., Canal, C., Pimentel, E., Vallecillo, A.: Formalizing Web Service Choreographies. Electr. Notes Theor. Comput. Sci. **105** (2004) 73–94
42. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: Proc. ICSOC. Volume 4294 of LNCS. (2006) 27–39
43. Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In: Proc. BPM. Volume 4102 of LNCS., Springer (2006) 65–80
44. Motahari Nezhad, H., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proc. WWW. (2007) 993–1002
45. Brogi, A., Canal, C., Pimentel, E.: On the semantics of software adaptation. Science of Computer Programming **61** (2006) 136–151
46. Mooij, A., Voorhoeve, M.: Proof techniques for adapter generation. In: Proc. WS-FM. (2008)
47. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: Proceedings of Business Process Management. Volume 4714 of Lecture Notes in Computer Science., Springer-Verlag, Berlin (2007) 271–287
48. Canal, C., Poizat, P., Salaün, G.: Model-based adaptation of behavioral mismatching components. IEEE Transactions on Software Engineering **34**(4) (2008) 546–563
49. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In: Proc. ICSOC. (2008) 84–99
50. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting WS-BPEL processes using flexible model generation. Data & Knowledge Engineering **64**(1) (2008) 38–54
51. Lohmann, N., Kleine, J.: Fully-automatic Translation of Open Workflow Net Models into Human-readable Abstract BPEL Processes. In: Proc. Modellierung. Volume P-127 of Lecture Notes in Informatics (LNI). (2008) 57–72
52. Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Undecidablity of partner existence for open nets. Information Processing Letters **108**(6) (2008) 374–378
53. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Logic synthesis of asynchronous controllers and interfaces. Advanced Microelectronics. Springer-Verlag (2002)