Operating guidelines for services

Peter Massuthe

Operating Guidelines for Services

Dissertation

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Massuthe, Peter

Operating Guidelines for Services / by Peter Massuthe.
Eindhoven: Technische Universiteit Eindhoven, 2009. Proefschrift.

ISBN 978-90-386-1702-2
NUR 933

Keywords: services / formal methods / Petri nets / operating guidelines

SIKS Dissertation Series No. 2009-12

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Printed by University Press Facilities, Eindhoven

Operating Guidelines for Services

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 21 april 2009 om 16.00 uur

door

Peter Massuthe

geboren te Bad Freienwalde, Duitsland

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. K.M. van Hee
en
prof.Dr. W. Reisig

Copromotor:
prof.Dr. K. Wolf

Operating Guidelines for Services

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(*doctor rerum naturalium*, Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II der
Humboldt-Universität zu Berlin
im Rahmen einer Doppelpromotion mit der
Technische Universiteit Eindhoven, Niederlande

von
Herrn Diplom-Informatiker

Peter Massuthe

geboren am 23. Juli 1976

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Dr. h.c. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Peter Frensch

1. Gutachter	prof.dr. Kees van Hee
2. Gutachter	Prof. Dr. Wolfgang Reisig
3. Gutachter	Prof. Dr. Karsten Wolf
eingereicht am	13. März 2009
Tag der mündlichen Prüfung	21. April 2009

Abstract

In a service-oriented world, the interaction of stateful services raises the need for formal verification of the behavioral compatibility of the services. In this thesis, we introduce a formal framework basing on Petri nets and automata for service modeling and formalize behavioral compatibility as deadlock freedom of the composition of the services.

Many other research questions, like substitutability of services and adapter generation, build on top of the behavioral compatibility question and formulate requirements for the *set* $\mathcal{R}(S)$ of behaviorally compatible services of a given service S. To efficiently characterize the set $\mathcal{R}(S)$ of S, we introduce the notion of the *operating guideline* of S. Operating guidelines can be applied to a variety of service-related analysis questions and may support service discovery, substitutability of service and the generation of adapters between behaviorally incompatible services.

All results presented in this thesis are implemented in our service analysis tool Fiona.

Kurzfassung

In der Welt zustandsbehafteter Services ist die formale Verifikation der Verhaltenskompatibilität interagierender Services von zentraler Bedeutung. In dieser Dissertation entwickeln wir einen formalen Rahmen zur Modellierung von Services mit Petrinetzen und Automaten und formalisieren die Verhaltenskompatibilität von Services als Verklemmungsfreiheit ihrer Komposition.

Viele weiterführende Analysefragen, darunter Fragen zur Austauschbarkeit von Services und die Konstruktion von Adapterservices, basieren auf der Verhaltenskompatibilität von Services und betrachten die *Menge* $\mathcal{R}(S)$ *aller* verhaltenskompatiblen Services für einen gegebenen Service S. Als effiziente Charakterisierung dieser Menge führen wir das Konzept der Bedienungsanleitung eines Services S ein. Bedienungsanleitungen können zur Entscheidung einer Vielzahl von Analysefragen im Zusammenhang mit Services angewendet und in den Bereichen der Lokalisation von Services, der Austauschbarkeit von Services sowie bei der Synthese von Adaptern für verhaltensinkompatible Services unterstützend eingesetzt werden.

Alle Ergebnisse, die in dieser Arbeit präsentiert werden, sind in unserem Analysewerkzeug für Services, Fiona, implementiert.

Table of Contents

List of Figures

List of Definitions and Notations

1 Introduction

1.1 Motivation

In recent years, enterprises more and more face the new challenges of ever faster changing business conditions and a growing number of competitors from all over the world. Consequently, functionalities are sourced out, and enterprises that may be competitors in other business areas form virtual enterprises to cooperate for a specific common business goal. That way, interorganizational business processes, distributed both logically and geographically, have become increasingly important and raised the need for a technological support of integrating heterogeneous systems within and across organizational boundaries.

In this context, *services* play an important role. As a common understanding, a service encapsulates self-contained functionalities behind a well-defined, standardized interface [ACKM03]. Services are independent of specific programming languages and operating systems and therefore help to reduce the complexity of integrating heterogenous environments.

A service can typically not be executed in isolation — services are designed for being invoked by other services, or for invoking other services themselves in order to provide more involved functionalities. The evolving paradigm of *service-oriented computing* (SOC) [Pap01, Pap07a] is concerned with the interaction of services, as well as service management and monitoring, service-oriented engineering, and many other research themes. SOC suggests technologies and standards to these concerns and therefore plays an important role in supporting interorganizational business processes.

However, the organizations involved in such an interorganizational process are essentially *autonomous* and have the freedom to create or modify their services and cooperations at any point in time. So, services must also provide high flexibility, loose coupling, distributed execution, and support the reuse of software components.

To technologically support the needed flexibility, the *service-oriented architecture* (SOA) [Got00, WCL+05, Pap07a] was proposed as an architectural style for dy-

namically composing services to complex software systems. It provides concepts and standards for publishing a service by a *service provider*, proposes *service brokers* for the registration of published services, and organizes the search for and discovery of published services by a *client*. Hence, SOA supports the dynamical binding and interaction of services. The process of publishing, finding, and binding of services is usually subsumed under the term *service brokering*. Service brokering can be seen as the key enabler for the use of services as flexible building blocks for designing highly dynamic interorganizational business processes.

In the beginning, SOC was restricted to *stateless services* performing simple request/response function calls. However, many real-world business scenarios require complex interaction patterns between multiple, possibly long-running services. Consequently, *stateful services* that communicate via *asynchronous message passing* are commonly used nowadays [Pap07a].

In this setting, a composition of services may exhibit complex structures and nontrivial interaction patterns. Therefore, small local changes can easily cause severe behavioral errors such as deadlocks or livelocks in the whole service composition (as shown in [LMSW06, LMSW08], for example). Consequently, *behavioral compatibility* of services, i.e. the absence of such behavioral errors, is an important field of research and a major concern to appropriately achieve common business goals. To decide whether a whole composition of services will interact properly is by far non-trivial [AW01, LMSW06, ALM+07, LMSW08, ALM+09]. Hence, *formal support* for checking and assuring behavioral compatibility of service interactions is crucial for the success of service brokering, and services have to be analyzed thoroughly before they are bound together and start to interact.

In this thesis, we study behavioral compatibility of service interactions on a formal basis. We abstract from other compatibility aspects such as semantics or nonfunctional properties in this thesis. In fact, these issues are orthogonal to our approach, and the respective results may complement one another. We refer to Chap. 2 for a detailed comparison.

1.2 Formal Analysis of Service Behavior

In general, a formal analysis of service behavior requires a formal modeling of services and their interaction, a formalization of the analysis question, i.e. the desired property, and formal techniques to decide the property for a given concrete model.

A well-established formal method are *Petri nets* (see [Rei85], for example). They have an intuitive graphical representation, are based on a formal operational semantics, and are supported by powerful analysis methods. The most important strength of Petri nets is their elegant representation of concurrency. For that

reason, Petri nets are used to model and analyze distributed systems in various applications.

Workflow nets [Aal98] are a special class of Petri nets which have been proven successful for the modeling and verification of business processes. The most important correctness property for workflow nets is known as *soundness* [Aal98]. Basically, soundness states that the workflow may always terminate properly and that there are no redundant activities.

For modeling *distributed* business processes, workflow nets have been refined to *workflow modules* [Kin97, Mar04], mainly by adding an interface to the workflow. A workflow module reflects the open nature of a *part* of a distributed business process and can be seen as a formal representation of a *service* in the scope of this thesis. To analyze behavioral compatibility of workflow modules, the soundness notion for workflow nets was adapted to a *weak soundness* notion of a *composition* of workflow modules, basically requiring the absence of deadlocks in the composition. Then, the major correctness notion for a workflow module in isolation was introduced as *controllability* (formerly also known as usability) [Mar04, Sch05, Wol09]. Controllability of a workflow module considers the *existence* of another workflow module such that their composition is deadlock-free. Like the soundness property for a workflow, controllability is a minimal requirement for the correctness of a service.

1.3 Problem Description and Research Goal

For more involved analysis questions regarding the behavioral correctness of service interactions, more expressive notions than controllability are needed. It turned out that *the set of all behaviorally compatible services* for a given service is of particular interest. For a service S, let $\mathcal{R}(S)$ denote this set of behaviorally compatible services for S.

In the following, we identify three important research problems in the context of behavioral compatibility of services and motivate the need for *a characterization of the set* $\mathcal{R}(S)$. The characterization of the set $\mathcal{R}(S)$ for a given service S is the main goal of this thesis. As the set $\mathcal{R}(S)$ is typically infinite, this is a non-trivial task.

Service Discovery

The discovery of published services is one of the most fundamental concepts of SOA and one of the main challenges for the practical applicability of service-orientation at the same time. From a behavioral perspective, service discovery addresses the following question. Given a client R searching for a service to

cooperate with, is there a fitting published service S such that R and S are behaviorally compatible? In our terms, service discovery searches for a published service S for which $R \in \mathcal{R}(S)$ holds.

If both R and S are formally modeled, $R \in \mathcal{R}(S)$ can in principle be answered by composing both services and *model checking* [CGP00] the composition for the desired behavioral correctness criterion (e.g. deadlock freedom). This approach, however, has two main disadvantages.

Firstly, each pair of R and S obviously has to be analyzed separately. As we assume that services are published only once, but used by a number of clients, we may expect this number of pairs — and consequently the number of required model checking tasks — to be high, potentially significantly higher than the number of published services. However, model checking typically involves state space exploration of the composition of R and S. Therefore, it is often both time and memory consuming and has typically to be performed by experts. Hence, it is not feasible for service discovery at runtime. In contrast, an approach that allows for efficient discovery of services is needed. Even if this comes at the cost of additional complexity before or during the publishing of a service, such an approach is still reasonable as publish will happen less often than discovery.

Secondly, the search for deadlocks in the composition of R and S requires detailed knowledge of the internal behaviors of both R and S. As these services are typically owned by different enterprises, neither of them wants to reveal their internal behavior to the other one in order to hide their trade secrets. Hence, privacy issues further limit the applicability of model checking.

A suggested approach to cope with this privacy issue is known as the *public view* approach [AW01, LRS02, Mar04, CTD05, ALM+07]. Basically, a public view S' of a service S is an abstracted version of S itself, hiding "enough" internals of the service S to publish S' without revealing (crucial) trade secrets of S. Then, instead of checking behavioral compatibility between R and S, behavioral compatibility between R and the public view S' of S is analyzed (e.g. by model checking). It is assumed that compatibility between R and the public view S' is sufficient to induce compatibility between R and the actual service S. That way, the internals of the service S do not need to be published. However, there are only a few concrete approaches to formalize public views [AW01, Mar04, ALM+07], and the *generation* of public views is very limited so far. In principle, public views may help to overcome the privacy issue. Nevertheless, the complexity issue of model checking for behavioral compatibility still remains.

Concluding, the discovery of behaviorally compatible services must be deemed to be unsolved. A characterization of the set $\mathcal{R}(S)$ that hides trade secrets of S and which is encompassed by decision procedures to efficiently decide $R \in \mathcal{R}(S)$ is needed to support behavioral service discovery.

Service contracts and substitutability of services

A different, currently quite common approach to realize interorganizational business processes is known as the *contract approach* [AW01, ALM+09]. Therein, the parties involved in a (future) interorganizational business process jointly specify the desired overall process and the duties of each party in that process. This overall specification serves as a *contract* between the parties. If the contract is successfully specified (and analyzed for its correctness), each party may locally implement its part of the contract. That way, the *actual* interorganizational cooperation is based on the *implementation* of the contract. Even though a contract specification itself might behave correctly, behavioral correctness of an *implementation* of the contract is non-trivial to decide [AW01, ALM+09].

Conceiving a party's share of the contract specification as a service S, the set $\mathcal{R}(S)$ contains all behaviorally compatible services of the other parties. If now the implementation of S is seen as another service S', then the inclusion relation $\mathcal{R}(S) \subseteq \mathcal{R}(S')$ states that the services of the other parties are *still behaviorally compatible* to S'. Hence, this inclusion is a sufficient condition for a correct implementation S' of S. If each party correctly implements its part, then the overall implementation is behaviorally compatible by construction [ALM+07, ALM+09]. Furthermore, each inclusion relation can be checked *locally* for each a party. That way, the approach also takes into account confidentiality issues.

This question of correct implementation of a service specification can be relaxed to a question of general *substitutability of services*. That is, under which conditions does the substitution of a service S by some other service S' impact the (possible and actual) cooperations of S? We focus again on the behavioral aspect of the substitutability of services. In this context, [SMB09] proposed several different notions of behavioral substitutability that base on the preservation of exactly or at least the hitherto existing cooperations of S by S'. Most of the decision procedures require a characterization of the sets $\mathcal{R}(S)$ and $\mathcal{R}(S')$.

Hence, the set $\mathcal{R}(S)$ of behaviorally compatible services for a service S is employed in the research areas of service contracts and substitutability of services as well. A characterization of this set allowing for any easy to use comparison of two sets $\mathcal{R}(S)$ and $\mathcal{R}(S')$ may help to decide correct implementation of a contract and many other relevant behavioral substitutability notions.

Synthesis of services

Service synthesis addresses the problem of constructing an abstract behavioral skeleton of a service R *from scratch* such that R is behaviorally compatible to a given service S. The synthesized service R can then be filled with implementation details [DKLW07, LK08]. Service synthesis can be applied in a number of scenarios like the validation of a service S before it is published [LMW07a], for test

case generation for a service S [KL09], or for the generation of adapter services mediating between behaviorally incompatible services [GMW08, Gie08].

Service synthesis is a novel research area which may profit substantially from a characterization of the set $\mathcal{R}(S)$. If this characterization is *operational*, the designer may also steer the service synthesis process such that the synthesized R has different sizes or certain (behavioral) properties.

Research Goal

In this thesis, we will address the research goal of a characterization of the set $\mathcal{R}(S)$ of all behaviorally compatible services R for a given service S. Taking into account the three research problems introduced above, such a characterization should

- be finite,
- be supported by an efficient method to decide $R \in \mathcal{R}(S)$,
- hide the internal behavior of S as much as possible,
- support the comparison of two sets $\mathcal{R}(S)$ and $\mathcal{R}(S')$, specifically the decision of the inclusion relation $\mathcal{R}(S) \subseteq \mathcal{R}(S')$, and
- be operational.

This research is novel. As the set $\mathcal{R}(S)$ is usually infinite, a characterization of this set is non-trivial. The many applications of such a characterization show its importance. This is underlined by the listing of most of these applications in the "SOC research roadmap" [PTDL08] in which the authors, all of them leading researchers in the area of services and SOC, identify the grand challenges for service-related research in the near future.

1.4 Contributions

This thesis makes the following two main contributions.

Formal modeling of service behavior. We present a variant of Petri nets, *service nets*, as a formal model for services and their interaction, and we formalize behavioral correctness of a service composition. With service automata, we introduce a formal basis for the characterization of the set $\mathcal{R}(S)$ of all behavioral compatible services R for a given service S. We show that service automata can equally be used as a formal model for services and show the equivalence of service nets and service automata. To this end, we introduce a back and forth translation between both formalisms.

Characterization of $\mathcal{R}(S)$. We introduce *Boolean annotated service automata* (*BSA*s), mainly service automata where each state is annotated by a Boolean formula. A *BSA* serves as a general concept for characterizing a set of service automata. For a given, concrete service S, we develop an algorithm to construct a special *BSA* for S, called *operating guideline* for S, OG_S, characterizing the set $\mathcal{R}(S)$ of behaviorally compatible services R for S. OG_S is finite, operational, and only reveals behavior of S that is needed to decide $R \in \mathcal{R}(S)$. We additionally provide algorithms to decide whether a service automaton R is characterized by OG_S and to compare the structure of the operating guidelines of two services S and S' in order to infer the respective inclusion relation $\mathcal{R}(S) \subseteq \mathcal{R}(S')$. Hence, OG_S pays attention to all requirements for the characterization of the set $\mathcal{R}(S)$ as motivated above.

Two further contributions of the work that has been carried out for this thesis are:

Implementation in the analysis tool Fiona. All results presented in this thesis have been prototypically implemented in our open source analysis tool Fiona (available at `http://www.service-technology.org/fiona`). Fiona can compute operating guidelines, match a service with an operating guideline (i.e. decide $R \in \mathcal{R}(S)$), and manipulate (i.e. minimize) operating guidelines. Fiona may furthermore compare the sets $\mathcal{R}(S)$ and $\mathcal{R}(S')$ of two services S and S' for equivalence or inclusion, synthesize behavioral adapters, and it provides several other functionalities. A detailed description of the implementation in Fiona is given in Chap. 7.

Published results. Based on the work presented in this thesis, the following articles have been published at reviewed workshops, conferences, or journals since September 2005.

[MS05] and [MRS05] firstly present the basic operating guidelines approach and introduce service nets (therein called open workflow nets) and service automata. Most notably, these works were restricted to deterministic and acyclic services at this time. The article [MW07] drops the first requirement and allows for the characterization and discovery of services with deterministic and non-deterministic behavior. [LMW07b] finally generalizes the operating guidelines approach to arbitrary finite-state services with only marginal restrictions.

In [MSSW08], we were able to prove that controllability is even undecidable for services with infinite state space. As controllability is a necessary prerequisite to construct operating guidelines, the restrictions made in [LMW07b] are consequently necessary for operating guidelines, too.

[LMSW06] and [LMSW08] present a broad overview of the applications of behavioral analysis for processes that stem from industrial service description languages. Operating guidelines are applied for more involved analysis of such processes. Additionally, the implementation in Fiona is firstly presented.

[LMW07a, ALM+07, ALM+09, SMB09] show the application of operating guidelines in a number of other research areas. [LMW07a] shows how to incorporate behavioral constraints to operating guidelines and presents the application for service validation. [ALM+07, ALM+09] present service contracts and how operating guidelines can be used to decide the correct implementation of a service specification. The article [SMB09] is concerned with the application of operating guidelines to decide a number of behavioral substitutability notions for services.

Besides these contributions, one main purpose of this thesis is also to lay a solid foundation for the concept of operating guidelines and to serve as a comprehensive reference for future work building upon the characterization of the set of behaviorally compatible services $\mathcal{R}(S)$ for S.

1.5 Outline of the Thesis

This thesis consists of three parts. In Part I, we introduce services and the paradigm of service-oriented computing and present our formal framework for service modeling. Part II is the main part of this thesis. Therein, we formalize the characterization of sets of services and present the construction of operating guidelines. In Part III, we exemplify the application of operating guidelines in the research areas of service discovery, substitutability of services, and adapter generation, and present the implementation of the results of Part II in the analysis tool Fiona.

Chapter 2 presents a survey of services and service-oriented computing as emerging programming concepts, and is dedicated to distinguish our focus on behavioral compatibility from other aspects of the compatibility of services.

In Chap. 3, we introduce *service nets* as our formal modeling technique for services and their interaction. We will formalize our behavioral correctness criterion of a service composition as *deadlock freedom*, and introduce *controllability* as the most fundamental correctness notion of a service in isolation. *Service automata*, a variant of communicating automata, will provide the basis of operating guidelines later on. We show that service automata and service nets are equally well suited as a formal representation for services and present a back and forth translation between service nets and service automata that preserves all relevant behavioral properties of the service.

Chapter 4 is considered with the characterization of services in general. It introduces *Boolean annotated service automata* (*BSA*s) as a means for characterizing

a set of service automata. We develop a *matching procedure* that efficiently decides whether or not a service automaton is characterized by a *BSA*. We present algorithms to equivalently transform a *BSA*, to compare the sets of characterized service automata of different *BSA*s, or to minimize *BSA*s.

In Chap. 5, we start by a formal *characterization of deadlocks* in the composition of two service automata from the *point of view of one* of these service automata. This characterization will finally enable us to derive a construction of a special *BSA*, characterizing exactly the set of behaviorally compatible service automata for a given service automaton. Hence, this *BSA* serves as operating guideline for the service.

Chapter 6 shows how operating guidelines can be applied to decide several research questions that are based on behavioral compatibility of services. Therein, we demonstrate the use of operating guidelines for service discovery, substitutability of service, and the generation of behavioral adapters.

All results presented in Chaps. 4 and 5 have been prototypically implemented. We present our analysis tool Fiona in Chap. 7. The implemented algorithms, however, often differ significantly from the theoretical ones introduced before. This is mostly based on several optimizations made to increase performance. We motivate our design decisions and describe the differences between theory and implementation.

Chapter 8 finally concludes this thesis and presents directions for future research.

Part I

Service Behavior Modeling

In this part of this thesis, we first review the research area of services and service-oriented computing in more detail and motivate the need for analyzing the behavioral compatibility of interacting services. Analysis of behavioral compatibility of services requires a formal model of the behavior of these services. To this end, we then present *service nets* as a formal modeling technique. Service nets introduce special input and output places as their interface and are therefore well suited to model services and their interaction via asynchronous message passing. Furthermore, a special class of communicating non-deterministic automata, *service automata*, are introduced. Service automata will provide the basis for operating guidelines later on.

2 Background on Services and Service-Oriented Computing

2.1 Services

The term *service* is commonly used to describe a piece of software that implements a certain encapsulated functionality accessible via a well-defined, standardized interface. The provided functionality may range from a simple computational function to a sophisticated process distributed between multiple organizations (see [PTDL08], for instance). This idea of encapsulation is not new, but has been used in a number of software engineering principles, e.g. modules, objects, and components. As the main difference to these approaches, a service is self-contained, self-explanatory, and often realizes a complete business function. Furthermore, services are required to be platform-independent, to be independent of the state or context of other services, and to support loose coupling. That way, services form a flexible infrastructure to build interorganizational business processes [Pap03, Pap07b].

Services are designed to interact with other services. During the last few years, stateful services that communicate asynchronously via message exchange have become standard. Additionally, they may employ large and complex interaction patterns in their communication with other services [BDH05].

Recent literature distinguishes between two shapes of services, *simple* and *composite* services [ACKM03, DS05, PH07, PTDL08]. A simple service provides a basic function while a composite service aggregates existing (simple or composite) services into a new service providing a more involved, combined functionality. The process of building a composite service is commonly called *service composition* [DS05]. Service composition is one of the four major research themes launched in [PTDL08]. Therein, assuring the behavioral compatibility of the composed services is stated as one open research challenge in the near future.

The most common implementation of services are *web services* (see [ACKM03, WCL+05, Pap07a], for instance). The interface of a web service is typically defined in the *Web Services Description Language* (WSDL) [CCMW01], listing the

operations that the web service can perform and the messages it accepts or produces. Web services usually communicate via the *Simple Object Access Protocol* (SOAP). The *Web Services Business Process Execution Language* (WS-BPEL, or BPEL for short) [Alv07] is an accepted industrial language to describe the internal control structure of a web service. It provides *basic activities* to communicate with other web services, to manipulate data, etc., and *structured activities* to define a causal order on the basic activities (like sequential, parallel, or repeated execution, and branching dependent on data, timeouts, or messages). A special structured activity can furthermore be used to define a scope for fault, compensation, and event handling.

2.2 Service-Oriented Computing (SOC)

Service-oriented computing (SOC) [Pap01, Pap07a] is an evolving paradigm in the context of the "programming-in-the-large" concept [DK76] for developing complex software systems using (web) services as basic building blocks. Thereby, two different approaches for constructing composite services are distinguished today.

A *service orchestration* emphasizes one particular service S and describes the internal structure of S and the logical order of the interactions between S and all other services from the perspective and under control of S. An orchestration abstracts from the internal control structures of the other services and all possible interactions between them. BPEL is currently the most prominent language to specify web service orchestrations.

A *service choreography* is used to describe the interaction of a collection of services from a global point of view. Commonly, only the message exchanges between all services are defined — the internal structures of the services are usually not represented, or they are represented only in an abstract way. Examples for languages to describe a choreography are the *Web Services Choreography Description Language* (WS-CDL) [KBR+05] and the academic choreography description language *Let's Dance* [ZBDH06].

The distinction between orchestrations and choreographies results in two possible, inherently different perspectives on one and the same concrete interaction. Seen as a choreography, the description of an interaction involving n services S_1 to S_n is focused on the message exchanges between all n services. Thereby, the global order of the conversation can be seen, and the flow of a message through the services can be tracked. By choosing one of the participating services, say S_1, the interaction can also be described as an orchestration defining the logical order of interactions between S_1 and the remaining services from the internal point of view of S_1. Only those message exchanges where S_1 is involved in can be expressed in the orchestration. Consequently, an orchestration is more detailed

with respect to one concrete service, but a choreography is more collaborative in nature [MCHP08].

However, this distinction is rather artificial and should converge into one common concept and a joint language [PTDL08]. One attempt to overcome this gap is the recently proposed language BPEL4Chor [DKLW07]. Therein, each individual (web) service is described as a BPEL process (representing the orchestration aspect), and the global interconnection is defined by a topology (representing the choreography aspect) in BPEL4Chor.

2.3 Service-Oriented Architecture (SOA)

The *service-oriented architecture* (SOA) [Got00, WCL+05, Pap07a] is a logical way to organize an infrastructure for building flexible interorganizational business processes. The flexibility is achieved by technologies to publish, to dynamically locate, and to (re-)combine services. To this end, the *basic SOA* proposes three different roles for the participants in a service interaction, *service provider*, *service requestor* (also called *client*), and *service broker*; and it defines the three basic operations *publish*, *find*, and *bind* of an SOA. The process of publishing, finding, and binding of services is usually subsumed under the term *service brokering*, schematically depicted in Fig. 2.1.

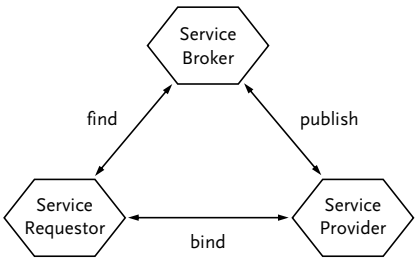


Figure 2.1: Service brokering in the basic SOA.

Service brokering in the basic SOA is organized as follows. A service provider creates (i.e. implements) or simply offers a service, defines a description of the interface and functionality of that service, and publishes this description to a trusted and publicly available *service registry*. The service broker manages the service descriptions in the registry and maintains information about the service providers (such as address and contact of the providing organization). A client may then specify certain *search criteria* about its sought-after service and searches the repository for a fitting published service. With the help of the information about the corresponding service provider, the client may then bind with the service provider and both can start to collaborate. The three roles defined by the basic

SOA are logical constructs and allow that one and the same participant can act as each of these roles (and even simultaneously) [Pap03, PTDL08]. For instance, an organization may host and distribute the description of its provided service by itself and, thereby, unifies the service provider and service broker roles.

In the context of the basic SOA, several standards for service descriptions, registry organization, and communication protocols have been proposed in order to provide technological support to automatize the search for and discovery of a published service by a client. In the end, organizing a service infrastructure according to the basic SOA promises to reduce the complexity of dynamically binding and integrating services within and across organizations.

However, the basic SOA lacks some important aspects that have become increasingly important to realize complex service-oriented applications. Firstly, it does not address the concept of composite services. By aggregating the functionality of several services into a composite service (with a new interface and a more involved functionality), an organization participating in an SOA provides a new, composite service and simultaneously requests functionality from the old, composed services. Hence, it unifies the roles of a service provider and a client. To emphasize this aggregation of services, a new SOA role is commonly distinguished, called the *service aggregator* [PH07], which has no direct representation in the basic SOA. Secondly, overarching concerns such as service administration, management, and support are not considered by the basic SOA. Consequently, the *extended SOA* (xSOA) has been proposed [Pap03, PH07] to address these concerns. The extended SOA is illustrated in Fig. 2.2.

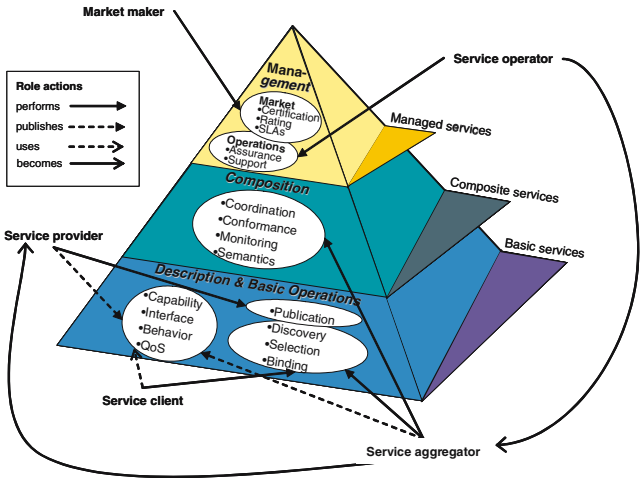


Figure 2.2: Extended SOA (graphics from [PH07]).

The extended SOA uses the basic SOA's service brokering concept as its lowest and most fundamental layer, called the *foundation layer*. The next layer of the xSOA, building upon the foundation layer, is the *composition layer*. It is devoted to the aggregation of existing services to composite services, i.e. to the service composition research theme of [PTDL08], and it enables the service brokering operations also for composite services. The uppermost layer on top of these two layers is called the *management layer*. It considers overarching concerns such as configuration, administration, and support, subsumed under the term *service management* [PH07].

Besides the new layers for service composition and management, the xSOA takes also into account the important aspects of *behavioral and non-functional properties* of a service which are also not explicitly addressed by the basic SOA. To this end, the xSOA's foundation layer refines the basic SOA's requirements concerning the published service descriptions to also include a description of the behavior of the service and its non-functional properties. That is, the service provider shall also publish relevant aspects of the behavior of its service as well as non-functional properties of the corresponding service. This enables a client to specify also quality requirements on non-functional properties as relevant search criteria, for instance, and the client may now check behavioral properties of the interaction of a published service with its own service before their actual binding. Accordingly, the xSOA also emphasizes this problem of service compatibility during the composition of services to larger units at its composition layer level (comprised under the term conformance in Fig. 2.2).

In the following section, we will elaborate on the topic of service compatibility — both for service discovery and service composition — in more detail.

2.4 Compatibility Criteria for Services

In the scope of this thesis, we make no conceptual difference between service interactions of a service provider and a client and service interactions in a composite service. In either case, the services in interaction have to be compatible to each other. There are at least four aspects of service compatibility: *syntactical compatibility*, *behavioral compatibility*, *semantical compatibility*, and *compatibility of non-functional properties* [PH07, Pap07b].

Syntactical compatibility assures the basic capability of services to exchange messages and can be seen as the most fundamental compatibility notion. It considers the different message types that are accepted or produced by a service, the service interface, and the corresponding operations that are supported by the interface (see [DHM+04], for instance). In case of a web service, this information is typically specified by a WSDL document (see Sect. 2.1). Assuring the syntactical compatibility of services is the current standard approach.

Behavioral compatibility is devoted to ensure compatibility of the control flows and message exchanges of the services. Therefore, the behavioral aspect of a service — often referred to as the *business protocol* of the service [BN08, DBN08] — has to be considered. Potential behavioral incompatibilities between business protocols include deadlocks (two services wait for a message of each other), livelocks (two services keep exchanging messages without progressing), and pending messages that have been sent but cannot be received anymore (see [DBN08] for an overview).

Semantical compatibility considers the semantical meaning (i.e. interpretation) of messages and data which are exchanged between services that possibly use a different vocabulary. To this end, the *Semantic Web* community proposes ontologies to define the commonly used terms, their meanings, as well as the relationships between different terms. It proposes to gather and publish machine-readable service descriptions with extensive semantical annotations [MBM+07, VMK+07, MGB+07] for deciding compatibility of the semantics of the services [BFM02, TP04, BHL+05, LDL08].

Non-functional compatibility addresses all quality of service (QoS) related aspects of services. To this end, mechanisms are provided to specify assurances and constraints on a service related to privacy protection, reliability and security of data transfer, performance rates, transactional features, etc. Typically, the non-functional properties of a published service are either specified as assertions by the service provider or are assessed, evaluated, and constantly monitored during the interactions with that service [HR00, DSGF06, RPD06]. The requirements of a client are specified as search criteria. When assertions and requirements fit together, the agreed-upon properties serve as a contract between provider and client. Several standards have been proposed to express and compare non-functional properties of services, such as WS-Policy [Baj06], for instance.

For a comprehensive approach to solve the service compatibility challenge (both for binding provider and client service as well as building a composite service), all four compatibility aspects have to be taken into account. Syntactical compatibility of the services is fundamentally important, but insufficient alone. Only services that are also compatible with respect to their behavior, semantics, and non-functional properties may implement the full potential of the SOC paradigm.

The aspects of behavioral, semantical, and non-functional compatibility are still subject of active research. Fortunately, these aspects complement each other and can, to a large extend, be considered separately from each other. This is reflected by mostly independent research groups considering behavioral aspects, semantical issues, and QoS related issues of services.

In this thesis, we will focus on *behavioral compatibility of services*. This compatibility aspect is more than all other aspects constrained by the fact that the services participating in an interaction are essentially autonomous and the corresponding organizations, although collaborating in the current interaction, might

still be competitors in other business areas. That is, the internals of the services are often subject to trade secrets and their owners do not intend to reveal their structure to the other participant. However, behavioral compatibility considerably depends on the interplay of the control structures of the composed services.

Furthermore, for large stateful services that communicate asynchronously, the analysis of behavioral compatibility suffers from the state explosion problem, where concurrency may cause an astronomic number of intermediate states. Accordingly, it is hard to decide whether services are behaviorally compatible and tool support is needed to assist clients to discover only behaviorally compatible services as well as to enable service aggregators to provide behaviorally sound composite services.

By focussing on behavioral compatibility, we do not consider non-functional aspects of services like performance, costs, etc., and we abstract from the semantical meaning of a message as well as message content, i.e. data values. Correspondingly, we model data-dependent choices by non-determinism and identify the meaning of a message by the name of the message channel in the rest of this thesis.

2.5 Formal Modeling of Services

To systematically approach behavioral compatibility of services, we investigate *formal models of services* instead of services specified in an industrial service description language. That way, our results are independent from the rapid evolution of industrial languages and thus durable. Furthermore, we can base on the formal semantics of the modeling technique to rigorously verify the absence of behavioral errors.

In this thesis, we consider Petri net models of services. Petri nets (see [Rei85], for example) have been proven useful in the area of business process modeling [Aal98]. By distinguishing special interface places, we derive a new class of Petri nets, service nets, which are well suited to represent services and their asynchronous communication. Suitability of service nets for service modeling has been proven by a service net semantics [Loh08] for the industrial service description language BPEL, which is the de facto standard to specify web services. A lot of effort has been spent to back BPEL with a formal semantics. While there are many partial semantics for BPEL, the semantics of [Loh08] is one of the few feature complete ones. [Loh08] also presents a formal semantics for BPEL4Chor which bases on the BPEL semantics. This formal semantics allows us to derive service models directly from a BPEL orchestration or from a service interaction specified in BPEL4Chor.

As we consider formal models of services in this thesis, we may abstract from whether the considered service is simple or composite as well as whether the

interacting services are specified as an orchestration or a choreography. In the upcoming chapter, we will introduce service nets in detail.

3 A Formal Framework for Service Modeling

In the setting of service-oriented computing (SOC), services serve as implementations of certain functionalities such that they can be invoked via asynchronous message passing. Services can be combined to gain more advanced functionalities, and they can be recombined to react on ever faster changing business conditions and regulations. Thereby, a service may not only be used as a function via a remote procedure call, where one request is sent to the service at the beginning and one answer is delivered by the service at the end of its execution. The interaction may cause nontrivial interaction behavior between them.

Hence, *behavioral compatibility* of interacting services is an important research challenge and behavioral incompatibilities between some of the participants have to be revealed *before* the services start to interact. This raises the need for a formal analysis of correct interaction of services. Therefore, a formal model of services and a formalization of correct interaction, as well as efficient algorithms to decide correct interaction of services are required.

In this chapter, we introduce our formal framework for service modeling. We will introduce two different formal models for services, *service nets* and *service automata*, as well as corresponding notions of correct interaction between services on the formal level. Service nets are a special class of Petri nets (see [Rei85], for instance), devoted to model the interaction of services via asynchronous message passing. Service automata are basically a simplification of classical I/O automata [Lyn96] with respect to the handling of asynchronous communication via message channels.

Service nets allow for an intuitive and easy to learn way of service modeling. Furthermore, they base on Petri nets and therefore are supported by several analysis methods for deciding a variety of properties on the structure of the net itself. Service net models of services can be automatically derived from industrial service description languages — like BPEL [Alv07] and BPEL4Chor [DKLW07] — by the help of implemented formal service net semantics of these languages [Loh08, Loh07].

Service automata are specifically well suited as the *behavioral* model of services and enable us to present efficient techniques and algorithms for analyzing the interaction behavior of services in the forthcoming two chapters. Service automata will provide the basis for all analysis techniques introduced later on.

This rest of this chapter is organized as follows. Section 3.1 presents some basic definitions and notations that are used in the following. In Sect. 3.2, we introduce service nets, and Sect. 3.3 is devoted to the introduction of *service automata*. Then, the relationship between service nets and service automata is studied in Sect. 3.4. We will show that both formalisms can equally be used as a formal model for services, as there exists a back and forth translation between service nets and service automata that preserves all information needed to analyze the interaction behavior of the considered services. So the service modeler may choose for his or her favorite formalism without losing available analysis methods. Furthermore, this two-fold approach eases the translation of other industrial languages into our formal framework — only one semantics is needed for a new industrial language: either a service net semantics or a service automata semantics, whatever suits best. Section 3.5 is devoted to motivate some design decisions of the definitions in Sects. 3.2 and 3.3 and explores other possible variants of the respective definitions and their implications. Then, Sect. 3.6 describes related work in detail and finally, Sect. 3.7 concludes this chapter.

3.1 Preliminaries

3.1.1 Basic Mathematical Notions

In this section we recall some basic notions of mathematics and computer science and introduce notations that will be used in the remainder of this thesis. The familiar reader may skip this section.

Let, throughout the thesis, $\mathbb{N}$ denote the set of natural numbers including 0. As usual, the *cardinality* of a set X, written $|X|$, denotes the number of elements $x \in X$ that occur in X.

Mappings and Relations

Let X and Y be some sets. A *mapping* f that assigns to each element $x \in X$ an element $y \in Y$ is denoted by $f : X \to Y$. Furthermore, the *restriction* $f_{|X'} : X' \to \mathbb{N}$ of a mapping f to a subset $X' \subseteq X$ is defined by $f_{|X'}(x) = f(x)$ for all $x \in X'$.

As usual, given a relation $rel \subseteq X \times Y$, the *inverse* of rel is the relation $rel^{-1} \subseteq Y \times X$, defined as $rel^{-1} = \{(x_2, x_1) \mid (x_1, x_2) \in rel\}$.

Multisets

Let X be some set. Then, a mapping $f : X \to \mathbb{N}$ is a *multiset* over X. Hereby, $f(x)$ is called the *multiplicity* of $x \in X$ in f and stands for the number of x elements in the multiset. We say the element x *occurs* in f, written $x \in f$, if $f(x) \geq 1$ and may denote a concrete finite multiset f by enumerating each element x that occurs in f. For example, we write $f = [x_1, x_1, x_2]$ for a multiset f with $f(x_1) = 2$, $f(x_2) = 1$, and $f(x) = 0$, for all $x \in X \setminus \{x_1, x_2\}$. The mapping f where $f(x) = 0$, for each $x \in X$, is called the *empty multiset*, denoted by $[\,]$.

Please notice the difference between the empty multiset and the special case of a multiset over an empty set $X = \emptyset$. In the latter case, the only multiset over $X = \emptyset$ is the *empty function*, denoted by $\emptyset$. For the rest of this paragraph, we assume $X \neq \emptyset$.

As multisets are special mappings, the restriction of a multiset $f : X \to \mathbb{N}$ to a subset $X^* \subseteq X$ is well-defined. Furthermore, we define the *extension* $f_{\uparrow X^*} : X^* \to Y$ of a multiset f to a superset $X^* \supseteq X$ as $f_{\uparrow X^*}(x) = f(x)$, if $x \in X$ and $f_{\uparrow X^*}(x) = 0$, otherwise.

Notation 3.1.1.
In the rest of this thesis, we do not distinguish between a multiset f and a restriction or an extension of f. To avoid confusion, we repeat the definition of the currently used f if necessary. ⌟

We will mostly apply the restriction or extension of a multiset to special multisets called *markings* when we transform a Petri net by adding or removing places (cp. Sects. 3.2 and 3.4). Thereby, a marking of the "smaller" net is a restriction of the intuitively corresponding marking(s) of the "bigger" net and a marking of the "bigger" net is an extension of a marking of the "smaller" net.

Already using this notation, we may define the *composition* $(f+g) : (X_f \cup X_g) \to \mathbb{N}$ of two multisets $f : X_f \to \mathbb{N}$ and $g : X_g \to \mathbb{N}$ as $(f + g)(x) = f(x) + g(x)$. Therefore, we can simply write $[x_1, x_2, x_2] + [x_2, x_3] = [x_1, x_2, x_2, x_2, x_3]$ without bothering whether x_3, for instance, was in the domain of the first multiset or not. The *difference* $(f - g)$ of multisets f and g is defined as

$$(f - g)(x) = \begin{cases} f(x) - g(x), & \text{if } f(x) \geq g(x) \\ 0, & \text{otherwise.} \end{cases}$$

The composition of mappings is used for the composition of markings, for instance.

As a shorthand notation, if $x \in X$, $f + x$ is an abbreviation for $f + [x]$ and stands for incrementing the multiplicity of x in f by 1 (adding an element to f). Analogously, $f - x$ stands for decrementing the multiplicity of x in f by 1 (removing an element from f). If $x \notin f$, then $(f - x)(x) = (f - [x])(x)$ is defined to be 0.

Powersets and bags

Let X be a set. Then, $\wp(X)$ denotes the *powerset* of X, i.e. $\wp(X)$ is the set of all subsets of X. Obviously, $\wp(\emptyset) = \{\emptyset\}$.

With $bags(X)$, we denote the set of all multisets over X, i.e. the set of all mappings $f : X \to \mathbb{N}$. Let furthermore, for a given *bound* $b \in \mathbb{N}$, $bags_b(X) \subseteq bags(X)$ denote the set of all those multisets f over X where $f(x) \leq b$, for all $x \in X$.

It is easy to see that $bags(\emptyset) = \{\emptyset\} = bags_b(\emptyset)$, for any bound b.

3.1.2 Petri Nets

In this section, we recall some basic definitions for classical low-level Petri nets (as introduced in [Rei85], for example) and introduce conventions for notations that are used in the following. The familiar reader may skip this section.

A *net* has two types of nodes, *places* and *transitions*, as well as a *flow relation* to connect nodes of different types. While a transition represents a dynamic element, for example an activity, a place represents a static element, such as a causality between activities or the need for a resource to perform an activity.

Definition 3.1.2 (Net).
A *net* $N = [P, T, F]$ consists of two finite, disjoint sets P of *places* and T of *transitions*, and a set $F \subseteq (P \times T) \cup (T \times P)$ of *arcs*. ⌟

An advantage of (Petri) nets over other formal modeling methods is their intuitive and easy to understand graphical notation and the compact representation of concurrency. As usual, places are graphically represented by circles, transitions are represented by boxes, and arcs are represented by directed arrows between them.

Notation 3.1.3.
We denote nets by N, N', M, etc.; possibly with indices. If not clear from the context or denoted otherwise, we refer to the parts of a net N by P_N, T_N, F_N. We often write P' instead of $P_{N'}$ or P_1 instead of P_{N_1}. ⌟

By ignoring the dots, Fig. 3.1(a) shows a net N with five places, $P_{\mathsf{N}} = \{\mathsf{p1}, \ldots, \mathsf{p5}\}$, one transition, $T_{\mathsf{N}} = \{\mathsf{t1}\}$, and five arcs, $F_{\mathsf{N}} = \{(\mathsf{p1}, \mathsf{t1}), (\mathsf{p2}, \mathsf{t1}), (\mathsf{t1}, \mathsf{p3}), (\mathsf{t1}, \mathsf{p4}), (\mathsf{t1}, \mathsf{p5})\}$.

The structural environment of a node is captured in the notions of the *preset* and the *postset* of a place or transition.

Definition 3.1.4 (Preset, postset).
For an element $x \in P \cup T$ of a net N, ${}^\bullet x = \{y \mid (y, x) \in F\}$ is the *preset* of x and $x^\bullet = \{y \mid (x, y) \in F\}$ is the *postset* of x. We canonically extend this definition to

sets of elements: if $X \subseteq P \cup T$, then ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$ and $X^\bullet = \bigcup_{x \in X} x^\bullet$ are the preset and postset of X, respectively. ⌟

The preset of the transition t1 of the net N of Fig. 3.1(a) is the set ${}^\bullet\mathsf{t1} = \{\mathsf{p1}, \mathsf{p2}\}$, the postset of t1 is $\mathsf{t1}^\bullet = \{\mathsf{p3}, \mathsf{p4}, \mathsf{p5}\}$. The postset of $\{\mathsf{p1}, \mathsf{p2}\}$, for example, is the set $\{\mathsf{p1}, \mathsf{p2}\}^\bullet = \mathsf{p1}^\bullet \cup \mathsf{p2}^\bullet = \{\mathsf{t1}\}$.

So far, a net represents a static structure only. To enable dynamics, we introduce the notions of a state of a net N and of a state change of N. A state of N is represented by a *marking* which is a distribution of *tokens* over (a subset of) the places of N.

Definition 3.1.5 (Marking).
A *marking* m of a net $N = [P, T, F]$ is a mapping $m : P \to \mathbb{N}$. ⌟

According to this definition, a marking m of a net N is a multiset over the places of N where $m(p)$ is the number of tokens on the place p at m. Hence, we may employ the earlier introduced multiset notation for markings as well and write $[p_1, p_1, p_2]$ for a marking m which marks the place p_1 with two tokens and the place p_2 with one token (and no other place). Graphically, a marking m is depicted by $m(p)$ black dots on the place p.

In Fig. 3.1(a), the marking m1 = [p1, p2, p2, p5] of the net N is depicted. Figure 3.1(b) shows the marking m2 = [p2, p3, p4, p5, p5] of N.

Figure 3.1: A net N with two different markings (a) m1 and (b) m2.

Fixing one marking of a net as the *initial marking* of the net yields a *Petri net*.

Definition 3.1.6 (Petri net).
A *Petri net* $N = [P, T, F, m_0]$ consists of a net $[P, T, F]$ and a distinguished marking m_0 of N, called the *initial marking* of N. ⌟

Accordingly, Figs. 3.1(a) and 3.1(b) show two different Petri nets. They have the same structure, i.e. the net N, but different initial markings, i.e. the initial marking m1 and m2, respectively.

With the state notion defined, we are now ready to introduce the notion of a state change for Petri nets, i.e. the *firing rule*. It is defined using the standard

Petri net semantics as introduced in [Rei85]; that is, a transition is *enabled* at a marking m if each place of its preset holds at least one token at m. An enabled transition can *fire* by consuming one token from each preset place and producing one token for each postset place, yielding a new (i.e. successor) marking. The *behavior* of a Petri net N is then defined by the states (markings) of N and the firing of transitions.

Definition 3.1.7 (Behavior of a Petri net).
A transition t of a Petri net N is *enabled* at a marking m of N if $m(p) \geq 1$, for all places $p \in {}^\bullet t$.

Let t be a transition of N that is enabled at m and let m' be the marking of N where, for all places p of N,

$$m'(p) = \begin{cases} m(p) + 1, & \text{if } p \in t^\bullet \text{ and } p \notin {}^\bullet t, \\ m(p) - 1, & \text{if } p \in {}^\bullet t \text{ and } p \notin t^\bullet, \\ m(p), & \text{otherwise.} \end{cases}$$

Then, m' is the *successor marking* of m with respect to (firing of) t; and m, t, and m' together form a *step* of N, written $m \xrightarrow{t} m'$. ⌟

It is easy to see that the transition $\mathsf{t1}$ of the Petri net N of Fig. 3.1(a) is enabled at the marking $\mathsf{m1}$. The corresponding successor marking after firing of $\mathsf{t1}$ is the marking $\mathsf{m2}$. Hence, $\mathsf{m1} \xrightarrow{\mathsf{t1}} \mathsf{m2}$ is a step of N.

A marking m' that can be constructed from a marking m by a sequence of steps is called *reachable* from m. The set of reachable markings of a Petri net represents its state space.

Definition 3.1.8 (Reachability, $R_N(m)$).
The set $R_N(m)$ of *reachable markings* from a marking m of a Petri net N is inductively defined as follows:

Basis. $m \in R_N(m)$;

Step. If $m' \in R_N(m)$ and there exists a transition t of N with $m' \xrightarrow{t} m''$, then $m'' \in R_N(m)$.

A marking m is called *reachable in* N if m is reachable from the initial marking of N, i.e. $m \in R_N(m_0)$. ⌟

Definition 3.1.9 (Bounded Petri net).
A Petri net N is *bounded* if it has only finitely many reachable markings, i.e. if the set $R_N(m_0)$ is finite. ⌟

A marking of a Petri net N that enables a transition can be left by N. It is therefore a *transient* marking. A marking that enables no transition of N is a *dead* marking.

Definition 3.1.10 (Transient, dead marking).
A marking m of a Petri net N is *transient* in N if m enables a transition of N. Otherwise, m is *dead* in N. ⌟

Because of enabledness of t1 at m1 in the Petri net N of Fig. 3.1(a), m1 is a transient marking of N. In contrast, m2 does not enable the only transition t1 of N. Hence, m2 is dead in N.

A dead marking m does not necessarily represent a design error in the Petri net model — it may represent a desired final state or it may model the waiting for some external action to occur. We will employ dead markings for both cases in the upcoming section when we introduce service nets as a representation for communicating Petri nets.

3.2 Service Modeling with Service Nets

Services communicate via *asynchronous* message passing over *message channels*. For being able to link different services, we assume a (possibly infinite) set $\mathcal{MC}$ of message channels as a *common name space* for all service models to be fixed throughout this thesis. Hereby we assume that each channel is uniquely characterized by its name: different names correspond to different channels and equal names denote the same channel. For technical reasons, we assume the special symbols τ, *final*, *true*, and *false* are not used as channel names, i.e. $\tau, \mathit{final}, \mathit{true}, \mathit{false} \notin \mathcal{MC}$.

We denote channels by lower case Latin letters, e.g. a, b, c, x, or by the meaning of a message on the channel, e.g. *login*, *invoice*, etc.

Although we assume each channel between services to be *directed*, a channel $x \in \mathcal{MC}$ does not indicate its direction by itself. The direction is fixed through the usage of the channel x by concrete services. Thereby, x is an *input channel* of one service and an *output channel* for the other one. Furthermore, we require that each channel is *bilateral*, i.e. a channel is shared by at most two services — for a third service C, a communication taking place inside the interaction of services A and B is internal matter.

To model services, we extend classical Petri nets by an *interface*, representing the asynchronous communication with other services over message channels, and *final markings* to represent successful termination. To this end, we introduce a new class of Petri nets called *service nets* in the following.

Suitability of service nets for service modeling has been proven by two feature-complete service net semantics for BPEL and BPEL4Chor. Both semantics are implemented in the compiler BPEL2oWFN [Loh07] (available at `http://service-technology.org/bpel2owfn`) which allows for the automatic translation of (1) a single BPEL process into a single service net, (2) a BPEL4Chor choreography of multiple BPEL processes into a single service net representing

the overall process, and (3) the individual translation of each party of a choreography into a single service net representing the respective part of the choreography.

3.2.1 Service Nets

Service nets (also termed *open workflow nets* in [MRS05, LMW07b] or *open nets* in [MSSW08, SMB09]) are a special class of Petri nets that introduces an *interface* to communicate with other service nets. The interface of a service net consists of a set P_{in} of *input places* for receiving messages from other service nets and a set P_{out} of *output places* for sending messages to other nets. Both P_{in} and P_{out} are finite subsets of $\mathcal{MC}$. Service nets can be *composed* yielding a new service net modeling the interaction of the represented services. This idea is based on the module concept for Petri nets which was first proposed by Kindler [Kin97].

The asynchronous nature of the message channels is reflected by a possible reordering of messages on channels. That is, although a message was sent on one channel before another message was sent on a different channel, the latter message may be received before the first one, and even two messages sent to the same channel can overtake each other. Furthermore, sending or receiving is *non-blocking*, i.e. after having sent a message, a service net can continue its execution and does not have to wait until this message is received.

For simplicity, we assume an unlimited capacity of each channel and abstract from message content, i.e. *data*. For data with *finite* domain, however, important message content can be represented in our approach. For instance, a channel receiving messages with Boolean values can be represented by its separation into two channels: one for messages with content *true* and one for messages with content *false*. For data with *infinite* domain, standard techniques like *abstract interpretation* [CC77] — known from static analysis of programs (see [NNH05] for an overview) — can be applied to identify a finite abstraction of the relevant parts of the infinite domain.

Hence, (low-level) Petri net places as a representation of interface channels are well suited to model asynchronous communication of services, and service nets provide a simple but adequate formal modeling technique to model services, still providing sufficient information to analyze the interaction behavior of services.

Definition 3.2.1 (Service net).
A *service net* $N = [P, P_{in}, P_{out}, T, F, m_0, \Omega]$ consists of

- a Petri net $[P, T, F, m_0]$,
- two disjoint sets $P_{in} \subseteq P$ of *input places* and $P_{out} \subseteq P$ of *output places*, such that $P_{in}, P_{out} \subseteq \mathcal{MC}$, ${}^\bullet P_{in} = P_{out}{}^\bullet = \emptyset$, and
- a set Ω of distinguished markings, called *final markings*.

Let $P_{io} = P_{in} \cup P_{out}$ denote the *interface* of N. We demand that neither the initial marking nor a final marking marks interface places, i.e. $m(p) = 0$, for each $p \in P_{io}$ and $m \in \{m_0\} \cup \Omega$. ⌟

The interface places of a service net correspond to the message channels. A token on an interface place represents a message that is currently pending on the corresponding channel, i.e. the message has already been sent, but not yet received.

To graphically emphasize the role of the interface places, they are positioned on a dashed frame which surrounds the service net.

Figure 3.2 depicts a service net N_{shop} modeling an online shop service. The service net has three input places, login, terms, and order, and four output places, new, known, invoice, and deliver, which are positioned on the frame around the inner of N_{shop}. The initial marking of N_{shop} is the marking [p1], and let the only final marking of N_{shop} be the marking [p9].

It is easy to see that Definitions 3.1.7 and 3.1.8, defining the behavior of Petri nets, extend canonically to service nets. So enabledness of a transition of a service net is well-defined.

A transition with an output place in its postset *sends* a message to this channel, and a transition with an input place in its preset waits for *receiving* a message from this channel.

Definition 3.2.2 (Receiving, sending, internal, interface transition).
For a transition t of a service net N, $receive(t) = {}^\bullet t \cap P_{in}$ denotes the message channels from which t receives and $send(t) = t^\bullet \cap P_{out}$ denotes the messages channels that t sends messages to.

The transition t is a *receiving transition* of N if $receive(t) \neq \emptyset$, a *sending transition* of N if $send(t) \neq \emptyset$, and an *internal transition* of N if $send(t) = receive(t) = \emptyset$. A non-internal transition of N is also called *interface transition* of N. ⌟

According to Definition 3.2.1, a transition t of some service net N can simultaneously receive or send multiple messages in N, or t may even receive *and* send messages simultaneously. In Sect. 3.4, we will introduce a special class of service nets, called *elementarily communicating service nets*, where each interface transition either receives or sends exactly one message. We will show that this class is sufficient to model services, as any service net can be transformed into a elementarily communicating service net without changing its semantics.

Notation 3.2.3 (Label of a transition).
Mostly, we label the transitions of a service net N according to their connections to interface places of N. Thereby, an interface transition is labeled with the set $receive(t) \cup send(t)$ and an internal transition of N is labeled with τ. To emphasize

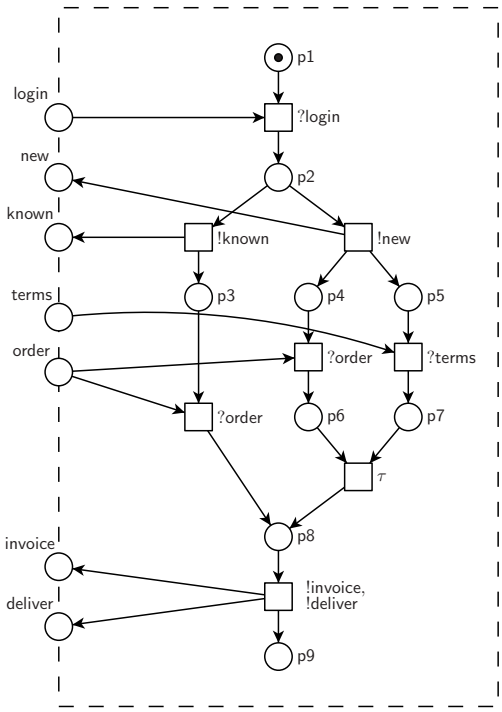


Figure 3.2: A service net N_{shop} modeling an online shop. In its initial marking [p1], the net waits for a login message from a client of the shop. If the message arrives, the transition named ?login can fire and produces a token on place p2. Then, the shop internally decides for firing transition !known, representing the successful look-up for the client in the database of known customers, or for firing transition !new, representing the first login of a newly registered client. In the first case, the shop awaits the order of the client (left hand side transition ?order) and then sends an invoice and delivers the goods to the client (transition !invoice, !deliver). In the second case, the client has to additionally accept the terms of payments once (?terms). Finally, the shop reaches its final marking [p9], and the business case is finished successfully.

the direction of the message channels, we add a preceding question mark, "?", or a preceding exclamation mark, "!", to the labels. For example, we label a (single) transition t by the label $?x, !y, !z$, if $receive(t) = \{x\}$ and $send(t) = \{y, z\}$. ⌟

In Fig. 3.2, the transitions of N_{shop} are already labeled according to this notation. As there are two transitions which are connected to the place order, N_{shop} has two equally labeled transitions ?order.[1] The transitions ?login, ?order, and ?terms of N_{shop} are receiving transitions in N_{shop} with $receive(?\mathsf{login}) = \{\mathsf{login}\}$ and

[1]Note that we omit the brackets and write ?order instead of {?order}, for instance.

$send(?\mathsf{login}) = \emptyset$, for instance. The transitions !new, !known, as well as the transition !invoice, !deliver are sending transitions of N_{shop} with $receive(!\mathsf{known}) = \emptyset$ and $send(!\mathsf{invoice}, !\mathsf{deliver}) = \{\mathsf{invoice}, \mathsf{deliver}\}$, for instance. N_{shop} has no transition which is a receiving *and* sending transition, but one transition which is neither a receiving, nor a sending, i.e. an internal transition. It is labeled by τ.

An important requirement for workflows [Aal98] is *proper termination*, i.e. the property that no "garbage" is left behind when terminating. Hence, it is also reasonable in the service domain to demand that all interface places are empty in the initial and the final markings in the definition of a service net.

Another relevant requirement for workflows, the unique sink place ω, has been dropped for service nets. The main reason is that a *set* of final markings is a more convenient and elegant way to model expected successful "goal states" of a service and avoids the need of a massaging step during the composition of service nets to achieve a structurally well-defined composed service net (compare Definition 3.2.11 and the composition of workflow modules in [Mar04]). Additionally, we relax final markings in the service domain such that a final marking of a service net N does not need to be dead in N (as required for the sink place in workflows). That is, we allow N to resume work *by itself* after having reached a final marking. This allows an easy way of modeling the return of N to its initial marking after having completed a successful interaction with some other service net, for instance. That way, the next "round" of communication can be started. A transient final marking can be compared with the approach of adding a shortcut transition t^* to a workflow net, connecting the unique sink place ω with the unique source place α of the workflow net [Aal98]. Choosing only dead markings as final markings, however, can be seen as a convention or following modeling guidelines if needed in a specific application area and is possible for service nets as well.

The set R_N of reachable markings of a service net N consists of all those markings that N can reach without interacting with another service net. For a typical service net N this set is intuitively rather small as N might expect input from its environment relatively soon. For being able to investigate the *possible* behavior of N in interaction with some environment, we consider the inner structure of N, $inner(N)$. The inner of N can be seen as the version of N in case an environment provides sufficiently many tokens at the interface of N.

Definition 3.2.4 (Inner of a service net, $inner(N)$).
The *inner* of a service net $N = [P, P_{in}, P_{out}, T, F, m_0, \Omega]$ is defined as the service net $inner(N) = [P', \emptyset, \emptyset, T, F', m_0, \Omega]$ with $P' = P \setminus P_{io}$ and $F' = F \setminus ((P_{in} \times T) \cup (T \times P_{out}))$. ⌟

The inner of a service net N is simply derived from N by removing all interface places (and all adjacent arcs) of N. Note the use of the restriction of a multiset

(see Sect. 3.1) in the previous definition for the initial and the final markings of $inner(N)$.

As an example, Fig. 3.3 shows the inner $inner(N_{\mathsf{shop}})$ of the online shop N_{shop} of Fig. 3.2. Whereas the initial marking [p1] is the only reachable marking of N_{shop}, its inner $inner(N_{\mathsf{shop}})$ has a number of reachable markings: [p1], [p2], [p3], [p4, p5], [p6, p5], [p4, p7], [p6, p7], [p8], and the final marking [p9] of $inner(N_{\mathsf{shop}})$.

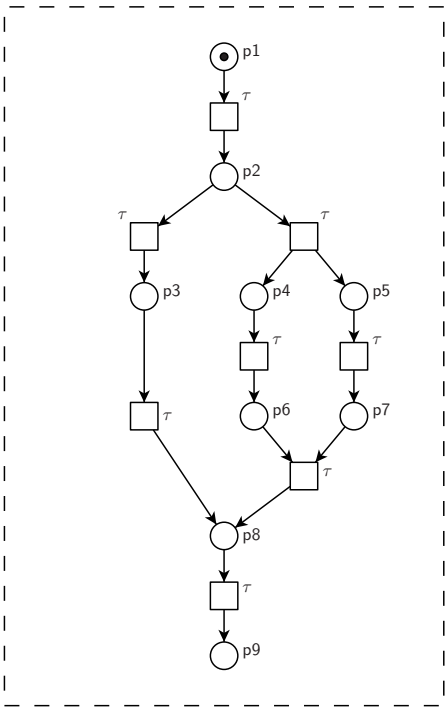


Figure 3.3: The inner of the online shop N_{shop} of Fig. 3.2, $inner(N_{\mathsf{shop}})$. As all transitions of $inner(N_{\mathsf{shop}})$ are internal transitions, they are labeled by τ in $inner(N_{\mathsf{shop}})$.

This special case of a service net with an empty interface is a *closed* service net. The usual case of a service net with a non-empty interface, in contrast, is an *open* service net.

Definition 3.2.5 (Open, closed service net).
A *service net* N with an empty interface (i.e. $P_{io\,N} = \emptyset$) is *closed*. Otherwise, it is *open*. ⌟

Obviously, the inner of any service net N is a closed service net and $N = inner(N)$, for each already closed service net N.

To relate different service nets with equal interfaces, we introduce the following definition of *interface equivalent* service nets.

Definition 3.2.6 (Interface equivalent service nets).
Two service nets N and M are *interface equivalent* if $P_{in\,N} = P_{in\,M}$ and $P_{out\,N} = P_{out\,M}$. ⌟

3.2.2 Composition of Service Nets

Now we are ready to consider the *interaction of services*. It is reflected by the *composition of the corresponding service nets* on the modeling level. Basically, composing service nets means merging uniquely named interface places, representing the shared communication channels of the services. The initial and final markings are merged accordingly. The resulting composed service net represents the corresponding services in interaction and is in general an *open* service net again. That is, it may have a non-empty interface and might be used as a component of another composition later on. That way, service nets serve as flexible building blocks to construct larger service nets.

On the other hand, the interaction of n services can easily be modeled by a sequence of $n - 1$ composition steps, each step composing two service nets. In fact, service net composition is commutative and associative. Hence, the result of the composition of n service nets is always uniquely determined by the service nets and not affected by the choice of which two services are composed in which order.

As we require that all communication channels are *bilateral* and *directed*, we have to assure that every interface place p has at most one service net that sends messages to p and at most one service net that receives from p. Hence, we introduce two compatibility notions in the following that we require to hold in order to compose two service nets — the service nets must be *internally disjoint* and *interface compatible*. The first property is only of a mere technical nature and simply assures that the composed service net is indeed a well-defined service net. The second property assures bilateral and directed communication. Then, it is sufficient to define the composition for *two* (internally disjoint and interface compatible) service nets.

Basically, two service nets are *internally disjoint* if they share at most interface places.

Definition 3.2.7 (Internally disjoint service nets).
Two service nets N and M are *internally disjoint* if the sets T_N, T_M, $(P_N \setminus P_{io\,N})$, and $(P_M \setminus P_{io\,M})$ are pairwise disjoint. ⌟

Remark 3.2.8.
Without loss of generality, we assume for the rest of this thesis that all composed service nets are internally disjoint, as any two service nets sharing an element can canonically be made internally disjoint by renaming the element accordingly. ⌟

In contrast to the internals of service nets, their interfaces often intentionally overlap.

Definition 3.2.9 (Shared, free interface place).
An interface place $p \in P_{io\,N} \cup P_{io\,M}$ of two service nets N or M is *shared* between N and M if $p \in P_{io\,N} \cap P_{io\,M}$. Otherwise, p is *free* between N and M. ⌟

To achieve a proper communication direction of a channel, we only consider the composition of service nets that are *interface compatible*. That is, if a place p is shared between N and M, then p is either an input place of N and an output place of M or vice versa.

Definition 3.2.10 (Interface compatible service nets).
Two service nets N and M are *interface compatible* if each shared interface place is an input place of one service net and an output place of the other one, i.e. $(P_{io\,N} \cap P_{io\,M}) = (P_{in\,N} \cap P_{out\,M}) \cup (P_{out\,N} \cap P_{in\,M})$. ⌟

Note that this definition is equivalent to $(P_{in\,N} \cap P_{in\,M}) = (P_{out\,N} \cap P_{out\,M}) = \emptyset$.

To compose two service nets N and M, we require that they are interface compatible. When considering the composition of n service nets, they must be *pairwise interface compatible*. That way, bilateral communication between all service nets is guaranteed.

Figure 3.4 shows an example client, N_{client}, for the online shop N_{shop} of Fig. 3.2. Let the initial marking of N_{client} be $[\mathsf{p10}]$, and let the set of final markings of N_{client} contain the single marking $[\mathsf{p15}, \mathsf{p16}]$. Note that N_{client} has a transition which simultaneously receives the new message and sends the terms message. Hence, it is labeled by ?new, !terms. Obviously, both shop and client are interface compatible.

Composition of service nets now basically means merging shared interface places. Whereas all shared interface places become internal to the composition, the free interface places form the interface of the composed service net.

Definition 3.2.11 (Composition of service nets).
The *composition* of interface compatible service nets N and M is the service net $N \oplus M = [P, P_{in}, P_{out}, T, F, m_0, \Omega]$ defined as follows:

- $P = P_N \cup P_M$,
- $P_{in} = (P_{in\,N} \setminus P_{out\,M}) \cup (P_{in\,M} \setminus P_{out\,N})$,

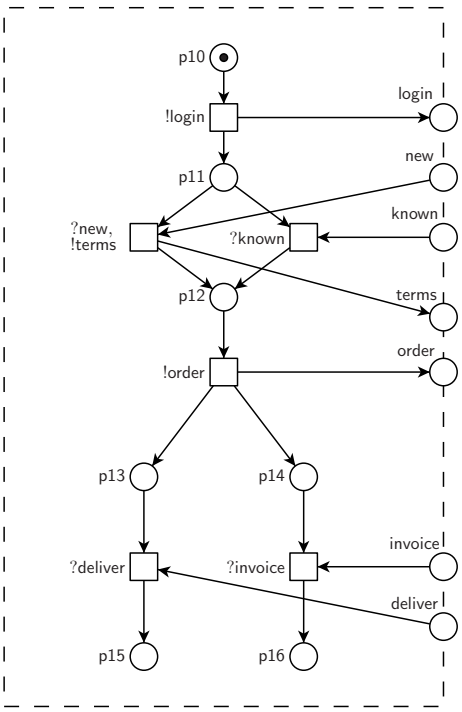


Figure 3.4: A service net N_{client} modeling a client of the online shop N_{shop} of Fig. 3.2. In its initial marking [p10], the client may send a login message to the online shop (transition !login). Afterwards, the client waits for the information from the shop whether he has to confirm the terms of payment or not (transitions ?new, !terms and ?known, respectively). In either case, the client then places his order(s) (transition !order) and finally waits concurrently for the invoice and the delivery of the ordered goods (transitions ?invoice and ?deliver). After having received both messages, the client reaches its single final marking [p15, p16].

- $P_{out} = (P_{out\,N} \setminus P_{in\,M}) \cup (P_{out\,M} \setminus P_{in\,N})$,
- $T = T_N \cup T_M$,
- $F = F_N \cup F_M$,
- $m_0 = m_{0\,N} + m_{0\,M}$, and
- $\Omega = \{m_N + m_M \mid m_N \in \Omega_N, m_M \in \Omega_M\}$. ⌟

The requirement for composing only interface compatible service nets ensures that their composition is a well-defined service net with respect to the service net definition (Definition 3.2.1) and meets all requirements for bilateral, directed, and asynchronous communication of the represented service.

As the online shop N_{shop} of Fig. 3.2 and its client N_{client} of Fig. 3.4 are interface compatible, they can be composed. The composition $N_{\mathsf{composition}} = N_{\mathsf{shop}} \oplus N_{\mathsf{client}}$ is depicted in Fig. 3.5.

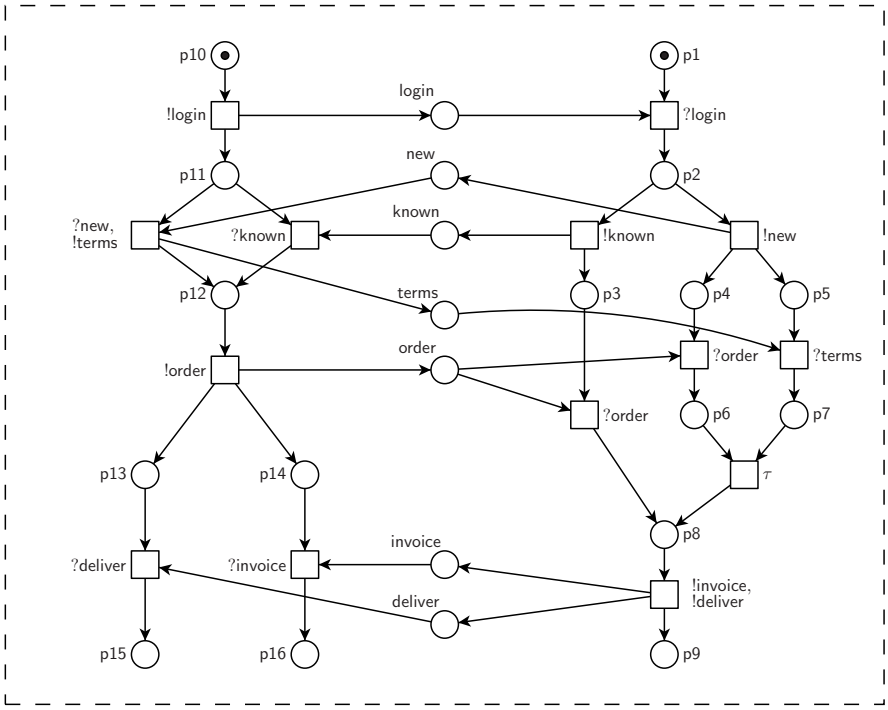


Figure 3.5: The composition $N_{\mathsf{composition}} = N_{\mathsf{shop}} \oplus N_{\mathsf{client}}$ of the online shop N_{shop} of Fig. 3.2 and its client N_{client} of Fig. 3.4. The composed initial marking is the marking $[\mathsf{p10}, \mathsf{p1}]$, and the single final marking of the composition is $[\mathsf{p15}, \mathsf{p16}, \mathsf{p9}]$. The composition is closed, i.e. has an empty interface, and all transitions of $N_{\mathsf{composition}}$ are internal in $N_{\mathsf{composition}}$. Nevertheless, the original transition labeling is kept in the graphics for convenience.

Proposition 3.2.12 (Commutativity of composition).
The composition of interface compatible service nets N and M is commutative, i.e. $N \oplus M = M \oplus N$. ⌟

So far we have defined the composition of two service nets only. To compose more than two service nets, we assume that they are pairwise interface compatible. Then, we can break down the n-fold composition into $n - 1$ compositions of two service nets. The following proposition states that the order of choosing the service nets does not matter.

Proposition 3.2.13 (Associativity of composition).
The composition of pairwise interface compatible service nets N_1, N_2, and N_3 is associative, i.e. $N_1 \oplus (N_2 \oplus N_3) = (N_1 \oplus N_2) \oplus N_3$. ⌟

Notation 3.2.14.
Because of the latter proposition, we may omit parentheses when composing service nets. ⌟

Obviously, if service nets $N_1, \ldots, N_n$ are pairwise interface compatible, then the composition of any subset of these nets is interface compatible to each of the remaining service nets N_i and even interface compatible to the single service net of their composition. Furthermore, it is easy to see that a closed service net is interface compatible to any service net, including other closed ones. However, their composition results in merely putting the service nets side by side without connection.

Notation 3.2.15.
In the remainder of this thesis, we implicitly assume that all service nets are (pairwise) interface compatible whenever we compose them. ⌟

3.2.3 Behavioral Compatibility of Service Nets

The interface compatibility notion for service nets, as introduced in the previous section, captures the *syntactical compatibility* of services. However, syntactical compatibility of services is not sufficient for the expected correct interaction of the services. Additionally, *behavioral compatibility*, a crucial criterion for the correct interaction of the services, is needed on top of their syntactical compatibility.

Two main reasons for non-expected interaction behavior of a service composition, i.e. behavioral *in*compatibility of services, can be identified. Firstly, a component of a composition may be erroneously designed itself. For instance, it may have a non-local choice, i.e. an internal decision point which influences the expected behavior of the other component, but the latter is not informed which choice has actually been made. In such a case, *no* composition with the erroneous component may behave correctly. Secondly, each component may be correct with *some* other component, but the interactional behaviors of two given services, e.g. in a concrete choreography of services, do not fit together, or even exclude each other. For example, the services may get into a situation where one component waits for a message of the other one, but that component is currently waiting for a message from the first one itself. In either case, the services are behaviorally incompatible.

As a service composition may employ nontrivial interactions between their components, it is a challenging task to decide whether the whole composition interacts

correctly. Thus, the services have to be analyzed thoroughly for their behavioral compatibility before they may start interacting.

This raises the need for a formal notion of behavioral compatibility on the level of service modeling with service nets, which is the topic of this subsection, as well as methods to decide behavioral compatibility of given service nets, as will be considered in the following Chaps. 4 and 5.

To this end, we first introduce a behavioral correctness criterion for a single service net and then derive thereof a notion of behavioral compatibility for two service nets N and M such that N and M are behaviorally compatible if their composition satisfies the correctness criterion.

Thereby, we treat closed and open single service nets differently. Behavioral correctness of a *closed* service net is captured in the notion of *well-behavior* of the net, which basically means deadlock freedom. For an *open* service net, in contrast, the behavioral correctness criterion is called *controllability*, which means the existence of another service net such that their composition is well-behaving.

From well-behavior and controllability of one service net, we may then immediately derive the corresponding behavioral compatibility notions between two service nets N and M. In case $N \oplus M$ is a closed service net, then N and M are behaviorally compatible if their composition is well-behaving (which is the proposed correctness criterion for a closed net). Then, N is called a *strategy* for M (and vice versa). In case $N \oplus M$ is an open service net, then N and M are behaviorally compatible if their composition is controllable (the proposed correctness criterion for open service nets). As the focus of this thesis is the characterization of all strategies M for N — and not the decision of controllability of a service net, we may omit the formalization of the second behavioral compatibility notion between service nets N and M, i.e. behavioral compatibility of N and M where $N \oplus M$ is controllable rather than well-behaving.

These considerations lead to the following formalizations of behavioral correctness.

Definition 3.2.16 (Deadlock).
A dead marking m of a service net N that is no final marking of N (i.e. $m \notin \Omega$) is a *deadlock* in N. ⌟

Note that this definition of a deadlock differs from the standard definition in literature as we discriminate between final and non-final dead states. Deadlock freedom is a fundamental correctness criterion for interacting services, which are represented by a single closed service net. In contrast, an open service net, representing a service in isolation, usually has deadlocks.

Definition 3.2.17 (Well-behaving service net).
A closed service net N is *well-behaving* if N has no deadlocks, i.e. for each $m \in R_N(m_0)$: m is no deadlock in N. ⌟

Then, two service nets N and M with a closed composition $N \oplus M$ are behaviorally compatible if their composition is well-behaving. In this case, M is a *strategy* for N.

Definition 3.2.18 (Strategy service net, *Strat*(N)).
A service net M is a *strategy* (service net) for a service net N if $N \oplus M$ is a closed well-behaving service net.

Let $\mathit{Strat}(N)$ denote the set of all strategies for N. ⌟

Considering our example service nets N_{shop} of Fig. 3.2 and N_{client} of Fig. 3.4, it is easy to see that their closed composition $N_{\mathsf{shop}} \oplus N_{\mathsf{client}}$ of Fig. 3.5 is well-behaving — the only reachable dead marking is $[\mathsf{p9}, \mathsf{p15}, \mathsf{p16}]$, which is the final marking of $N_{\mathsf{shop}} \oplus N_{\mathsf{client}}$. Hence, we conclude that N_{client} is a strategy for N_{shop}.

The strategy notion captures behavioral compatibility of service nets N and M. Whereas the term "behavioral compatibility" already suggests a symmetric relation, the phrase "M is a strategy for N" suggests an asymmetric relation. As already sketched, however, the strategy notion is symmetric, too:

Corollary 3.2.19 (Strategy is symmetric).
A service net M is a strategy for a service net N iff N is a strategy for M. ⌟

The main reason for this corollary is commutativity of our service net composition (cp. Proposition 3.2.12) and the formulation of well-behavior as a property of a single service net representing the composition of two service nets.

The reason for nevertheless choosing the asymmetric strategy notion is that in the upcoming Chaps. 4 and 5, we will fix one side of the composition, say N, and aim at characterizing all strategies M for N, i.e. the set $\mathit{Strat}(N)$. $\mathit{Strat}(N)$ is of particular interest as it gives a semantics of a service net N in terms of all behaviorally compatible service nets M for N.

The term strategy originates from a control-theoretic point of view (see [RW87, CL99], for instance). We may see M as a controller for N imposing well-behavior of $N \oplus M$. Therein, the receiving and sending transitions of N correspond to controllable and observable actions for M. The internal transitions of N are not observable for the controller M. In accordance with control theory, the controller M is a strategy for N if the considered correctness criterion is satisfied for the composed system.

Finally, the analogous behavioral correctness notion for an open service net N, representing a service in isolation, is *controllability*. It states the *possibility* to add (compose) a service net M to a service net N such that the resulting composition is closed and well-behaving, i.e. the *existence* of a strategy for N.

Definition 3.2.20 (Controllability).
A service net N is *controllable* if there exists a strategy for N. Otherwise, N is *uncontrollable*. ⌟

Due to the existence of the service net N_{client}, the service net N_{shop} of Fig. 3.2 is controllable. Analogously, N_{client} is controllable, too.

Obviously, an uncontrollable service is fundamentally ill-designed. Uncontrollable services represent the first one of the above introduced reasons for behaviorally incompatible services. Two controllable services that are no strategies for each other represent the second reason.

It is easy to see that the set $\mathit{Strat}(N)$ comprises infinitely many strategies for each controllable service net N. The main reason lies in the possibility for internal transitions of a strategy M for N. That is, if a service net M is a strategy for N, then a service net M', which performs n internal steps and then behaves as M, is also a strategy for N. Hence, controllable service nets have infinitely many strategies. On the other hand, obviously, $\mathit{Strat}(N) = \emptyset$ for an uncontrollable service net N.

In the forthcoming Chaps. 4 and 5 of this thesis, we aim at characterizing the set $\mathit{Strat}(N)$. The main challenge therein is to find a finite, operational description of this possibly infinite set and to provide an efficient algorithm to decide the question $M \in \mathit{Strat}(N)$. It will turn out that the *operating guideline* of N, OG_N, serves as such a characterization of $\mathit{Strat}(N)$, and a *matching procedure* can be used to decide whether a service M is characterized by OG_N.

Now we have introduced all fundamental notions and concepts for the first part of our formal framework, service nets. *Service automata*, as introduced in the following section, will complement this framework.

3.3 Service Behavior Modeling with Service Automata

The set $\mathit{Strat}(N)$ of a service net N is independent of the structure of N and depends on the interaction behavior of N only. Differently structured service nets N and N' with equal behaviors have the same strategies. Hence, it is more convenient to characterize only the *behavior* of all strategies M for a service net N, which implicitly characterizes all strategies M itself, instead of characterizing all those service nets M directly. We do not aim at *constructing* all strategies, we just want to efficiently *decide* whether two given services are behaviorally compatible. Using our approach, however, the construction of strategies is still possible — by applying standard techniques like the theory of regions [BD98], for instance.

In this section, we introduce *service automata* as a formal method for representing the *behavior* of service nets. Service automata will be used to characterize sets of services in Chapter 4 and will form the basis of operating guidelines in Chapter 5.

We will first introduce the concept of service automata in detail and then present a back and forth translation between service nets and service automata, which uses the set $\mathcal{MC}$ as a shared name space between a service net N and its corresponding service automaton A, as both the interface places of N and the interface channels of A are subsets of $\mathcal{MC}$. We will show that both formalisms are equally suited to model services and their interaction. So service *nets* can be used for service *modeling* in an intuitive and convenient way and service *automata* can be used for *state space analysis* of the interaction behavior of services.

3.3.1 Service Automata

Basically, service automata are a simplification of classical I/O automata [Lyn96] with respect to the handling of asynchronous communication via message channels. They communicate asynchronously rather than synchronously, and they do not require the explicit modeling of the states of message channels. Using I/O automata, the message channel's states would be modeled explicitly as one part of the state of an automaton. Hence, our service automaton approach leads to smaller and thus more readable automata.

Definition 3.3.1 (Service automaton).
A *service automaton* is an automaton $A = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ that consists of

- a (possibly infinite) set Q of *states*,
- two disjoint, finite sets $I_{in} \subseteq \mathcal{MC}$ of *input channels* and $I_{out} \subseteq \mathcal{MC}$ of *output channels*, with $I_{io} = I_{in} \cup I_{out}$ is the *interface* of A,
- a *transition relation* $\delta \subseteq Q \times (I_{io} \cup \{\tau\}) \times Q$,
- a distinguished state $q_0 \in Q$, called the *initial state* of A, and
- a set $\Omega \subseteq Q$ of distinguished *final states*.

For a transition $(q, x, q') \in \delta$, x is called the *label* of (q, x, q') in A. ⌟

Compared to service nets, a service automaton can basically be seen as the reachability graph of the inner of the corresponding service net where the transition labels of the service automaton correspond to the interface places of the service net. A detailed comparison of service nets and service automata can be found in the upcoming Sect. 3.4. It will turn out that service nets and service automata are equally well suited as a formal representation of services.

Notation 3.3.2.
We denote service automata by A, A', B, etc.; possibly with indices. If not clear from the context or denoted otherwise, we refer to the parts of a service automaton A by Q_A, $I_{in\,A}$, δ_A, etc. We often write Q' instead of $Q_{A'}$ or Q_1 instead of Q_{A_1}.⌟

Notation 3.3.3.
To emphasize the direction of an interface channel of a service automaton A in A's graphical representation, we represent a label $x \in I_{out}$ in the graphical representation of A by $!x$ and a label $x \in I_{in}$ by $?x$. ⌟

This transition labeling was already introduced for the transitions of a service net in the previous section (Notation 3.2.3). Therein, however, the transition of a service net could bear more than one label, which is not possible for the transition of a service automaton.

Figure 3.6 depicts a service automaton model A_{shop} of the online shop that was shown as a service net N_{shop} in Fig. 3.2. It has 11 states and 13 transitions. The initial state of A_{shop} is $\mathsf{r1}$, illustrated by an ingoing arc without source. The only final state of A_{shop} is $\mathsf{r11}$, illustrated by a double circle.

Remark 3.3.4.
The interface channels of a service automaton A are not always deducible from its graphical representation as we allow that a message channel $x \in I_{io}$ of A does not appear as a transition, i.e. x might be in the interface of A although A has no x-labeled transition. In such a case, the interface is written as an annotation to the graphics if relevant. ⌟

Notation 3.3.5 (Present transition).
An x-labeled transition is *present* at a state q of A if there is a state q' of A such that there is a transition $(q, x, q') \in \delta_A$. ⌟

An important class of service automata are *finite* service automata.

Definition 3.3.6 (Finite service automaton).
A service automaton A is *finite* if its set Q_A of states is finite. ⌟

In the following, we lift all notions defined for service nets to service automata. Some notions, however, will slightly deviate from the respective service net notion because of the different representation of the message channels in a service automaton. In such a case, we shall point out the differences when introducing the respective notion for service automata.

Transitions that are labeled by an input or an output channel are called receiving or sending transition, respectively. A τ-labeled transition is called an internal transition:

Definition 3.3.7 (Receiving, sending, internal, interface transition).
An x-labeled transition of a service automaton A is a *receiving transition* of A if $x \in I_{in}$, a *sending transition* of A if $x \in I_{out}$, and an *internal transition* of A if $x = \tau$. A non-internal transition of A is also called *interface transition* of A. ⌟

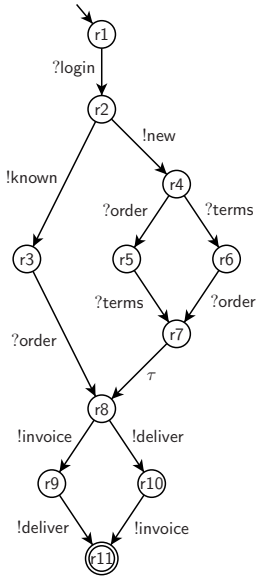


Figure 3.6: A service automaton A_{shop} modeling an online shop. A_{shop} corresponds to the service net N_{shop} of Fig. 3.2. In its initial state r1, the shop waits for a login message. After the message was received by the shop (represented by the ?login-labeled transition), it internally decides for performing either the !known-labeled or the !new-labeled transition, representing the login of a known client or the first login of a newly registered client, respectively. In the first case, the shop awaits an order (left hand side ?order-labeled transition) and then sends an invoice and delivers the goods to the client in arbitrary order (represented by the transitions !invoice and !deliver). In the second case, the client has to additionally accept the terms of payments once (transition ?terms). Finally, the shop reaches its final state r11.
In comparison with the service net N_{shop}, the concurrent reception of the order and the terms messages in case of a new client in N_{shop} is modeled by the reception of both messages in arbitrary order in A_{shop}. Accordingly, the simultaneous sending of the invoice and the deliver messages in N_{shop} is represented by sending the messages in arbitrary order in A_{shop}.

The online shop service automaton A_{shop} of Fig. 3.6 has 6 receiving transitions. One of these receiving transitions is labeled by ?login, three receiving transitions are labeled by ?order, and two receiving transitions are labeled by ?terms. Furthermore, it has 6 sending transitions, which are labeled by !known, !new, !invoice, and !deliver. The only internal transition of A_{shop} is the transition $(\mathsf{r7}, \tau, \mathsf{r8})$.

The earlier introduced communicational behavior of a service *net* (see Sect. 3.2) is explicitly represented in the net: a message channel $x \in \mathcal{MC}$ has a direct representation as an interface place and a receiving or a sending transition of the service net is directly connected to the corresponding input or output place. In

contrast, the communication capabilities of a service automaton are expressed more implicitly. The interface channels of a service automaton A appear only as transition labels of the sending and receiving transitions of A. The current state of a channel is not introduced until the service automaton participates in a composition with another automaton.

The handling of received or sent messages is one of the major differences between service nets and service automata. Whereas a transition of a service net may simultaneously receive and send multiple messages, an interface transition of a service automaton either receives exactly one message or it sends exactly one message. We will prove in Sect. 3.4 that this simplification does not restrict generality of our approach, but eases the characterization of a set of service automata in the upcoming Chapter 4.

A special class of service automata are *deterministic* service automata. Thereby, we call a service automaton A deterministic if, for each state q of A and each transition label $x \in I_{io\,A} \cup \{\tau\}$, q has at most one present x-labeled transition.

Definition 3.3.8 (Deterministic service automaton).
A state q of a service automaton A is *deterministic* if, for all states q_1, q_2 and all transitions $(q, x, q_1), (q, x, q_2)$ of A: $q_1 = q_2$.

A is a *deterministic service automaton* if each state of A is deterministic. ⌟

As the transition relation δ of a service automaton A is a set, rather than a multiset, Definition 3.3.8 in fact also excludes two different transitions $d_1 = (q, x, q')$ and $d_2 = (q, x, q')$ of A. Hence, for each $x \in \mathcal{MC} \cup \{\tau\}$ there is at most one present x-labeled transition per state of a deterministic service automaton.

This definition of determinism does not exclude internal transitions, as long as there is at most one present internal transition per state q of A. That is, we do not distinguish the label τ from any "other" message channel $x \in \mathcal{MC}$ in this definition. Note that this deviates from the classical notions of determinism as these mostly forbid internal transitions as well. However, our definition is strict enough that we are able to introduce specific, efficient analysis techniques for deterministic service automata in Sects. 4.2 and 5.2.2, for instance.

Consider again the service automaton A_{shop} of Fig. 3.6. It is deterministic, as each state of A_{shop} has at most one x-labeled present transition, for each channel $x \in \mathcal{MC} \cup \{\tau\}$. As the internal transition $(\mathsf{r7}, \tau, \mathsf{r8})$ of A_{shop} is the only present internal transition at state $\mathsf{r7}$, its presence does not violate Definition 3.3.8.

The transitions of a service automaton A connect its states. The set of connected states containing the initial state of A represents the state space of A.

Definition 3.3.9 (δ-reachable state).
The set of *δ-reachable states* from a state q of a service automaton $A = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ is inductively defined as follows:

Basis. q is δ-reachable from q in A;

Step. If q' is δ-reachable from q in A and there exists a transition $(q', x, q'') \in \delta$, then q'' is δ-reachable from q in A.

A state is *δ-reachable in A* if it is δ-reachable from q_0 in A. ⌟

We only consider states that are δ-reachable in a service automaton A. The δ-unreachable states provide no information and can be removed from A.

It is easy to see that each state of A_{shop} in Fig. 3.6 is δ-reachable in A_{shop}.

Intuitively, a state of a service automaton corresponds to a marking of a service net and a transition of a service automaton corresponds to a step of a service net. However, a δ-reachable state of a service automaton A does not correspond to a reachable marking of a service net N itself, but rather of the *inner* of N, because an x-labeled receiving transition represents only the *capability* of A to receive an x message in state q and therefore corresponds to a dead marking in N.

To derive a notion corresponding to the reachable markings of N itself, we introduce the *internally reachable states* of A in the next definition.

Definition 3.3.10 (Internally reachable state, $R_A(q)$).
The set $R_A(q)$ of *internally reachable states* from a state q in a service automaton A is inductively defined as follows:

Basis. q is internally reachable from q in A;

Step. If q' is internally reachable from q in A and there is a transition (q', x, q'') of A with $x \notin I_{in}$, then q'' is internally reachable from q in A.

A state is called *internally reachable in A* if it is internally reachable from the initial state of A. ⌟

Hence, only a state q for which there exists a sequence of sending or internal transitions leading to q is internally reachable in a service automaton. In contrast, a state that is only δ-reachable by passing a receiving transition (or not δ-reachable at all) is *not* internally reachable. This leads to the following definition of transient and stable states of a service automaton.

Definition 3.3.11 (Transient, stable state).
A state q of a service automaton A is *transient* in A if there exists a transition $(q, x, q') \in \delta_A$ with $x \notin I_{in\,A}$. Otherwise, q is *stable* in A. ⌟

A stable state q has no present transition or it only has present receiving transitions. Hence, a stable state cannot be left by A itself. It represents the possibility of an action to occur after having received a message from another service automaton.

In the service automaton A_{shop} of Fig. 3.6, the initial state r1 is already a stable state. Hence, only r1 is internally reachable in A_{shop}. From the transient state r7, for example, the states r7, r8, r9, r10, and r11 are internally reachable in A_{shop}.

Obviously, the property $R_A(q) = \{q\}$ holds for all stable states q. However, this property is not sufficient for being a stable state as there might exist an internal self-loop transition in A (i.e. a transition $(q, \tau, q) \in \delta_A$) which makes q transient.

In accordance to the notions of open and closed service nets, we define open and closed service automata. An open service automaton usually models a service in isolation, whereas a closed service automaton may be conceived as a single model for some services in interaction.

Definition 3.3.12 (Open, closed service automaton).
A *service automaton* A with an empty interface (i.e. $I_{io\,A} = \emptyset$) is *closed*. Otherwise, it is *open*. ⌟

Different service automata with equal interfaces are *interface equivalent*.

Definition 3.3.13 (Interface equivalent service automata).
Two service automata A and B are *interface equivalent* if $I_{in\,A} = I_{in\,B}$ and $I_{out\,A} = I_{out\,B}$. ⌟

Obviously, all closed service automata are interface equivalent.

The rest of this section is devoted to the interaction of services, represented by the composition of the corresponding service automata. First, we consider the composition of two service automata and then introduce the formalization of our correctness criteria for service interaction on the service automaton level.

3.3.2 Composition of Service Automata

As already motivated in Sect. 3.2.2 for service nets, we may restrict the composition of service automata to a composition of just *two* service automata as service automata composition will as well be commutative and associative. Hence, the composition of n service automata can be broken down to $n-1$ composition steps, each of them composing two service automata.

Therefore, we first introduce two compatibility notions for service automata and then define the composition of service automata. The compatibility notions for service automata correspond exactly to the two compatibility notions of service nets of Sect. 3.2.2.

Definition 3.3.14 (Internally disjoint service automata).
Two service automata A and B are *internally disjoint* if A and B have disjoint sets of states. ⌟

Remark 3.3.15.
From now on, all composed services are assumed to be internally disjoint (which again can be easily achieved by renaming). ⌟

The composition of service automata A and B uses the set $\mathcal{MC}$ of message channels to link the transition labels of A and the transition labels of B. Hence, common transition labels refer to the same channel and will become internal to the composition of A and B.

Definition 3.3.16 (Shared, free channel).
A message channel $x \in I_{io\,A} \cup I_{io\,B}$ of two service automata A and B is *shared* between A and B if $x \in I_{io\,A} \cap I_{io\,B}$. Otherwise, x is *free* between A and B. ⌟

The following interface compatibility notion for service automata implements the requirements that all channels are bilateral and directed. That is, for each channel $x \in \mathcal{MC}$, at most two service automata are communicating via x and if two service automata communicate via x, then x is an input channel of one service automaton and an output channel of the other one.

Definition 3.3.17 (Interface compatible service automata).
Two service automata A and B are *interface compatible* if each shared interface channel is an input channel of one service automaton and an output channel of the other one, i.e. $(I_{io\,A} \cap I_{io\,B}) = (I_{in\,A} \cap I_{out\,B}) \cup (I_{out\,A} \cap I_{in\,B})$. ⌟

Note that this definition is equivalent to $(I_{in\,A} \cap I_{in\,B}) = (I_{out\,A} \cap I_{out\,B}) = \emptyset$.

As an example, consider the service automaton A_{client} of Fig. 3.7. It models a client of the online shop of Fig. 3.6 and can be seen as an automaton version of the service net client N_{client} of Fig. 3.4. Let the initial state of A_{client} be the state $\mathsf{s1}$, and let the only final state of A_{client} be the state $\mathsf{s8}$. Obviously, A_{client} and A_{shop} are internally disjoint and interface compatible. All channels of A_{client} and A_{shop} are shared between A_{client} and A_{shop}.

Now we are ready to consider the interaction of services on the level of (interface compatible) service automata. It is represented by the *composition* of the corresponding service automata, formalized by Definition 3.3.18.

The composition $A \oplus B$ of two service automata A and B introduces an explicit representation of the binding of A and B along their shared interface channels. As we assume asynchronous communication, such a shared channel carries all messages that have already been sent by one automaton but have not yet been received by the other one. Since more than one message of a kind may be pending, we use one multiset per composed state to represent all currently pending messages. Hence, a state of $A \oplus B$ is now a structure $[q_A, q_B, M]$ consisting of a state q_A of A, a state q_B of B, and a multiset $M \in \mathit{bags}(\mathcal{MC})$ of (internally) pending messages in the shared channels. If x is a shared channel, then any x-labeled

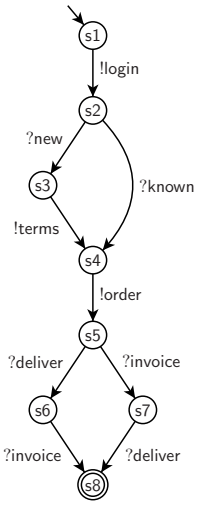


Figure 3.7: A service automaton A_{client} modeling a client of the online shop of Fig. 3.6. A_{client} intuitively corresponds to the service net N_{client} of Fig. 3.4. In comparison with the service net N_{client}, the simultaneous reception of the new message and sending of the terms message in N_{client} is modeled by first receiving and then sending the corresponding message in A_{client}. In contrast, the concurrent receiving of the invoice and the deliver messages in N_{client} is represented by their reception in arbitrary order in A_{client}.

sending or receiving transition of the components now is an *internal* transition in $A \oplus B$ that cannot be seen from the outside of the composition. Therefore, these transitions are τ-labeled in $A \oplus B$. A prior x-labeled sending transition thereby adds an x message to the multiset of pending messages, whereas a prior x-labeled receiving transition removes an x message from the multiset of pending messages.

In contrast, the free channels between A and B remain as the interface of $A \oplus B$. If x is a free channel, then $A \oplus B$ contains all x-labeled transitions of its respective component, representing remaining interaction possibilities through the interface of $A \oplus B$. A message x corresponding to a free channel is never contained in a multiset M of pending messages.

These considerations yield eight cases for transitions of the composition $A \oplus B$:

1. A sends a message x to B over a shared channel (i.e. A adds x to the multiset),
2. A receives a message x from B over a shared channel (i.e. from the multiset),
3. A sends or receives a message over a free channel (non-internal in $A \oplus B$),
4. A performs an internal transition in $A \oplus B$; and

5.-8. the mirrored four cases from the point of view of B.

The following definition is the straightforward realization of the latter considerations.

Definition 3.3.18 (Composition of service automata).
The *composition* $A \oplus B$ of two interface compatible service automata A and B is the service automaton $A \oplus B = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ with interface $I_{io} = I_{in} \cup I_{out}$ where

- $Q = Q_A \times Q_B \times bags(I_{io\,A} \cap I_{io\,B})$,
- $I_{in} = (I_{in\,A} \setminus I_{out\,B}) \cup (I_{in\,B} \setminus I_{out\,A})$,
- $I_{out} = (I_{out\,A} \setminus I_{in\,B}) \cup (I_{out\,B} \setminus I_{in\,A})$,
- $q_0 = [q_{0\,A}, q_{0\,B}, [\,]]$, and
- $\Omega = \Omega_A \times \Omega_B \times \{[\,]\}$,

and $\delta \subseteq Q \times (I_{io} \cup \{\tau\}) \times Q$ contains the following elements:

1. $([q_A, q_B, M], \tau, [q'_A, q_B, M + x])$ iff $(q_A, x, q'_A) \in \delta_A$ and $x \in I_{out\,A} \cap I_{in\,B}$;
2. $([q_A, q_B, M], \tau, [q'_A, q_B, M - x])$ iff $(q_A, x, q'_A) \in \delta_A$, $x \in I_{in\,A} \cap I_{out\,B}$, $x \in M$;
3. $([q_A, q_B, M], x, [q'_A, q_B, M])$ iff $(q_A, x, q'_A) \in \delta_A$ and $x \in I_{io}$;
4. $([q_A, q_B, M], \tau, [q'_A, q_B, M])$ iff $(q_A, \tau, q'_A) \in \delta_A$;
5. $([q_A, q_B, M], \tau, [q_A, q'_B, M + x])$ iff $(q_B, x, q'_B) \in \delta_B$ and $x \in I_{out\,B} \cap I_{in\,A}$;
6. $([q_A, q_B, M], \tau, [q_A, q'_B, M - x])$ iff $(q_B, x, q'_B) \in \delta_B$, $x \in I_{in\,B} \cap I_{out\,A}$, $x \in M$;
7. $([q_A, q_B, M], x, [q_A, q'_B, M])$ iff $(q_B, x, q'_B) \in \delta_B$, $x \in I_{io}$; and
8. $([q_A, q_B, M], \tau, [q_A, q'_B, M])$ iff $(q_B, \tau, q'_B) \in \delta_B$. ⌟

The eight items for transitions of the composed service automaton $A \oplus B$ in this definition correspond exactly to the eight cases which have been motivated beforehand. They implement an interleaving semantics: each transition of the composed service automaton $A \oplus B$ can be mapped to a prior transition of either A or B.

Remark 3.3.19.
Note that this definition of the composition technically introduces states of $A \oplus B$ that are not δ-reachable in $A \oplus B$ (which is a common, standard approach in automata theory). However, these states do not provide any information and can be removed from $A \oplus B$ for convenience. In fact, we never consider such states and skip them in the graphical representation of a (composed) service automaton, for example. ⌟

To exemplify our service automata composition, consider the composition of the example online shop A_{shop} of Fig. 3.6 and its client A_{client} of Fig. 3.7. It is the service automaton $A_{\mathsf{composition}} = A_{\mathsf{shop}} \oplus A_{\mathsf{client}}$, depicted in Fig. 3.8. According to the composition definition, the initial state of $A_{\mathsf{composition}}$ is the state $[\mathsf{r1}, \mathsf{s1}, [\,]]$, and the single final state of $A_{\mathsf{composition}}$ is $[\mathsf{r11}, \mathsf{s8}, [\,]]$. As stated before, all channels of

A_{client} and A_{shop} are shared between A_{client} and A_{shop}. Hence, $A_{\mathsf{composition}}$ is closed, i.e. has an empty interface. Therefore, all transitions of $A_{\mathsf{composition}}$ are internal transitions and labeled by τ in $A_{\mathsf{composition}}$.

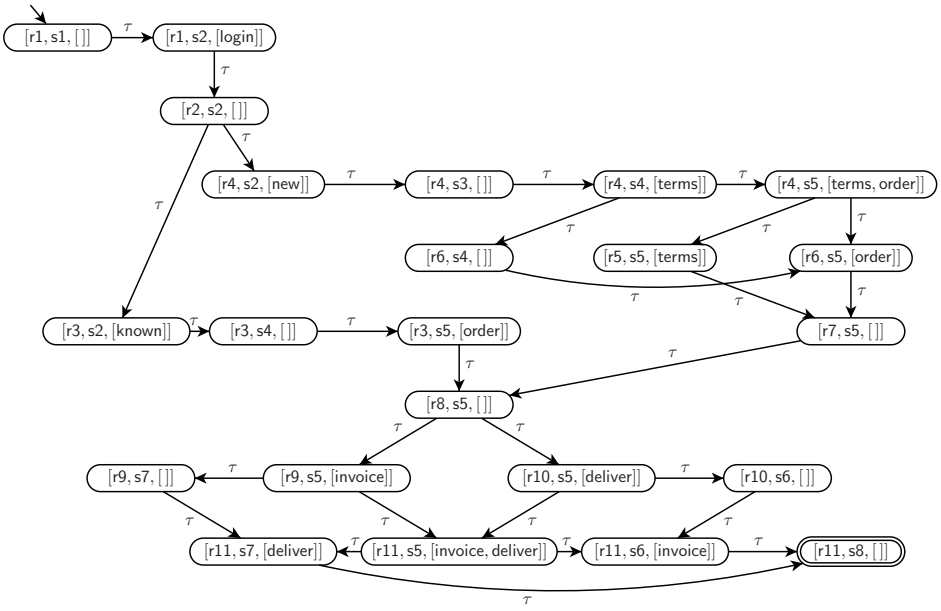
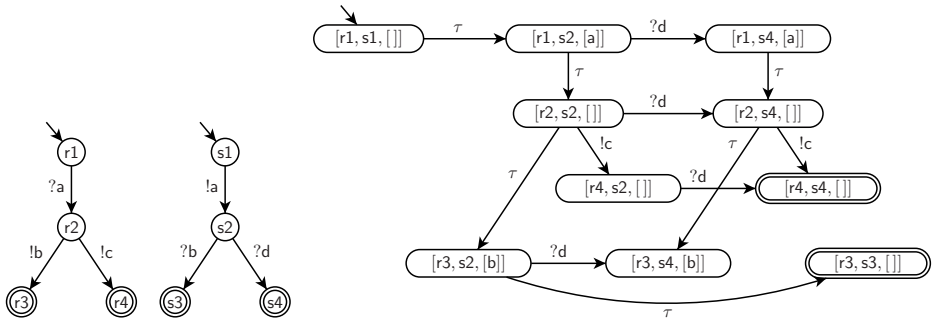


Figure 3.8: The composition $A_{\mathsf{composition}} = A_{\mathsf{shop}} \oplus A_{\mathsf{client}}$ of the online shop A_{shop} of Fig. 3.6 and its client A_{client} of Fig. 3.7. The composed initial state is the state $[\mathsf{r1}, \mathsf{s1}, [\,]]$, and the single final state of the composition is $[\mathsf{r11}, \mathsf{s8}, [\,]]$.
$A_{\mathsf{composition}}$ is a closed service automaton. Hence, all transitions are internal in $A_{\mathsf{composition}}$. Each such transition corresponds to a transition of either A_{client} or A_{shop}. Prior transitions of A_{client} are arranged horizontally, prior transitions of A_{shop} are arranged vertically. For instance, the prior transition $(\mathsf{s1}, !\mathsf{login}, \mathsf{s2})$ of A_{client} corresponds to the transition $([\mathsf{r1}, \mathsf{s1}, [\,]], \tau, [\mathsf{r1}, \mathsf{s2}, [\mathsf{login}]])$ of $A_{\mathsf{composition}}$ and is arranged horizontally.

Figure 3.9 shows two service automata A and B and their composition $\mathsf{A} \oplus \mathsf{B}$. The composed initial state of $\mathsf{A} \oplus \mathsf{B}$ is the state $[\mathsf{r1}, \mathsf{s1}, [\,]]$, and the two final states of $\mathsf{A} \oplus \mathsf{B}$ are the states $[\mathsf{r3}, \mathsf{s3}, [\,]]$ and $[\mathsf{r4}, \mathsf{s4}, [\,]]$ (cp. Definition 3.3.18). Let the interface of A consist of the channels $I_{in\,\mathsf{A}} = \{\mathsf{a}\}$ and $I_{out\,\mathsf{A}} = \{\mathsf{b}, \mathsf{c}\}$, and let the interface of B be $I_{in\,\mathsf{B}} = \{\mathsf{b}, \mathsf{d}\}$ and $I_{out\,\mathsf{B}} = \{\mathsf{a}\}$. Hence, the channels a and b are shared between A and B, whereas the channels c and d are free and remain as the interface of $\mathsf{A} \oplus \mathsf{B}$. Again, each transition of $\mathsf{A} \oplus \mathsf{B}$ corresponds to a transition of either A or B. The c-labeled and d-labeled transitions are added to $\mathsf{A} \oplus \mathsf{B}$ according to items 3. and 7. of Definition 3.3.18, for instance. The occurrence of the $!\mathsf{c}$-labeled transition at state $[\mathsf{r2}, \mathsf{s2}, [\,]]$, for example, leads to the state $[\mathsf{r4}, \mathsf{s2}, [\,]]$ and does not introduce a c message in the multiset of internally pending messages.

The production of the message c is not introduced until A ⊕ B is composed with another service automaton such that c becomes a shared channel.



(a) A (b) B (c) A ⊕ B

Figure 3.9: Two service automata A and B and their composition A ⊕ B. The initial state of A ⊕ B is the state [r1, s1, [ ]], and the two (δ-reachable) final states of the composition are the states [r3, s3, [ ]] and [r4, s4, [ ]].
A ⊕ B is an open service automaton. The remaining communication capabilities of A ⊕ B are represented by the !c-labeled and ?d-labeled transitions of A ⊕ B. Again, prior transitions of B are arranged horizontally, prior transitions of A are arranged vertically.

In accordance to the composition of service nets, service automata composition is commutative and associative. However, a state of the composed service automaton $A \oplus B$ is named in a special way, i.e. it is a structure $[q_A, q_B, M]$ of a state q_A of A, a state q_B of B, and a multiset M of pending messages in the shared channels. Hence, the corresponding state of the respective composition $B \oplus A$ is named $[q_B, q_A, M]$ instead, and hence, the compositions $A \oplus B$ and $B \oplus A$ are not equal, but equal up to *isomorphism*. That is, there exists an isomorphism between the sets of states of the service automata to cope with different state names. As we are mostly not interested in the individual names of the elements of a service automaton, but concentrate on the (graph) *structure* and the *transition labels* of the respective service automaton, we write $A \oplus B = B \oplus A$, meaning that both compositions are equal up to isomorphism.

Hence, we conclude that:

Proposition 3.3.20 (Commutativity of composition).
The composition of interface compatible service automata A and B is commutative, i.e. $A \oplus B = B \oplus A$, ⌟

and:

Proposition 3.3.21 (Associativity of composition).
The composition of pairwise interface compatible service automata A, B, and C is associative, i.e. $A \oplus (B \oplus C) = (A \oplus B) \oplus C$. ⌟

Due to the latter proposition, we may omit parentheses when composing multiple service automata. Moreover, the choice of which two service automata of a set of n service automata are composed in which order has no impact on the resulting (structure and transition labeling of the) composed service automaton.

Notation 3.3.22.
For the rest of this thesis, we assume all service automata to be pairwise interface compatible whenever they are composed. ⌟

3.3.3 Behavioral Compatibility of Service Automata

In the following, we consider the question whether or not two service automata interact *correctly*. Therefore, we introduce the notion of well-behavior of a single service automaton and define that A and B are behaviorally compatible if their closed composition $A \oplus B$ is well-behaving. For open compositions, the respective correctness notion is *controllability*.

Each definition in this subsection is a canonical translation of the corresponding definitions of Sect. 3.2.3 from service nets to service automata and bears no surprise.

Definition 3.3.23 (Deadlock).
A non-final stable state q of a service automaton A is a *deadlock* state in A. ⌟

Deadlock freedom is a fundamental, minimal correctness criterion for a closed service net, representing the interaction of multiple services. An open service automaton, in contrast, represents a single service in isolation and usually has deadlocks, as it may expect a message from its environment relatively soon.

Hence, the correctness notion for a closed single service automaton is defined as:

Definition 3.3.24 (Well-behaving service automaton).
A closed service automaton A is *well-behaving* if no internally reachable state is a deadlock, i.e. for each $q \in R_A(q_{0A})$: q is no deadlock state in A. ⌟

Please notice that the set of internally reachable states and the set of δ-reachable states coincide for closed service automata.

The derived behavioral compatibility notion for service automata A and B reads:

Definition 3.3.25 (Strategy service automaton, *Strat*(A)).
A service automaton B is a *strategy* (service automaton) for a service automaton A if $A \oplus B$ is a closed well-behaving service automaton.

Let $\mathit{Strat}(A)$ denote the set of all strategies for A. ⌟

Considering our (open) service automaton A_{shop} of Fig. 3.6, it is easy to see that already the initial state of A_{shop}, $\mathsf{r1}$, is an internally reachable deadlock state — it is a stable state which is not a final state of A_{shop}. The composed service automaton $A_{\mathsf{composition}}$ of Fig. 3.8, however, has no internally reachable deadlock — the only internally reachable stable state of $A_{\mathsf{composition}}$ is the state $[\mathsf{r11}, \mathsf{s8}, [\,]]$, which is also the final state of $A_{\mathsf{composition}}$. Hence, we conclude that $A_{\mathsf{composition}}$ is well-behaving, and thus, the client A_{client} is a strategy for the online shop service A_{shop} (and vice versa).

Again, because the composition of service automata is commutative, the strategy notion is symmetric.

Corollary 3.3.26 (Strategy is symmetric).
A service automaton B is a strategy for a service automaton A iff A is a strategy for B. ⌟

Well-behavior is only defined for closed service automata. To consider correct interaction behavior of a single, open service automaton, the notion of controllability is introduced.

Definition 3.3.27 (Controllability).
A service automaton A is *controllable* if there exists a strategy B for A. Otherwise, A is *uncontrollable*. ⌟

As in the case of service nets, if a service automaton A is controllable, then the set $\mathit{Strat}(A)$ is infinite. If A is uncontrollable, then $\mathit{Strat}(A)$ is empty.

So far, all notions have equally been defined for service nets and service automata. In the next section, we study the relationship between service nets and service automata in detail. It will turn out that both formalisms are equally well suited as a formal representation of services and that a service net can easily be translated into a service automaton and vice versa without losing relevant information. Hence, we can consider our analysis questions on both models alike.

3.4 An Equivalent Translation between Service Nets and Service Automata

To elaborate on the relationship between service nets and service automata, we present a back and forth translation between service nets and service automata

and show the equivalence of both formalisms with respect to the consideration of behavioral compatibility of services. As a preparation for the translation, we first introduce a subclass of service nets, *elementarily communicating service nets* in Sect. 3.4.1. In an elementarily communicating service net, each transition t is connected to at most one interface place, i.e. $|send(t) \cup receive(t)| \leq 1$, for all t. We show that each service net N can be transformed, i.e. *sequentialized*, into an elementarily communicating version $seq(N)$ of N without changing the set *Strat*, i.e. $Strat(N) = Strat(seq(N))$. Hence, we can assume elementarily communicating service nets for the translation without loss of generality. In Sect. 3.4.2, we then present a translation of an elementarily communicating service net N into a corresponding service automaton $SA(N)$. We show that this translation preserves all strategy relationships, i.e. a service net M is a strategy for a service net N if and only if the service automaton $SA(M)$ is a strategy for the service automaton $SA(N)$, for all (elementarily communicating) service nets N and M. This property justifies the feasibility of our translation. Finally, in Sect. 3.4.3, we show how a service automaton A can be translated back into a corresponding service net $PN(A)$ such that $A = SA(PN(A))$.

The back and forth translation allows us to decide controllability of a service net N on the level of service automata. That is, we can infer the controllability of N from the controllability of its corresponding service automaton $SA(N)$. Moreover, if we are able to construct a strategy service automaton B for the service automaton $SA(N)$, we are immediately able to construct a strategy service net M for the corresponding service net N as well — the service net $M = PN(B)$ is a strategy for N by construction.

3.4.1 Elementarily Communicating Service Nets

In this section, we define the subclass of service nets called *elementarily communicating service nets*. This class eases the translation step from a service net into a service automaton in the subsequent section. The introduction of the subclass is necessary because there is an issue caused by an arbitrary service net's transitions which are connected to more than one interface place. Such a transition is called *simultaneously communicating*. Recall that communication on the level of service automata is represented by labels. So, the translation of a service net N with a simultaneously communicating transition t into a service automaton $SA(N)$ would yield a transition of $SA(N)$ that has more than one label. Obviously, this is not allowed in our definition of service automata (cp. Definition 3.3.1). An elementarily communicating service net has no transition which is connected to more than one interface place and hence, solves this issue.

In principle, it would be possible to allow a multiset of messages as a label of a service automaton transition. However, this would cause more technicalities in the definitions of service automata and the following concept of operating guidelines

(Chaps. 4–5). Additionally, it would drastically increase the size of an operating guideline without adding any value.

First, we introduce the notion of elementarily communicating service nets and a transformation of an arbitrary service net N into an elementarily communicating version $seq(N)$. Then, we prove that no service net M can distinguish between a service net N and its sequentialization $seq(N)$, i.e. M is a strategy for N if and only if M is a strategy for $seq(N)$. Thus, for the translation of N into $SA(N)$, we can without loss of generality assume that N communicates elementarily.

As already motivated, a service net is called elementarily communicating if none of its transitions is connected to more than one interface place.

Definition 3.4.1 (Elementarily communicating service net).
An interface transition t of a service net N is *elementarily communicating* if $|send(t) \cup receive(t)| = 1$. Otherwise, t is *simultaneously communicating*.

A service net N is *elementarily communicating* if each interface transition of N is elementarily communicating. Otherwise, N is *simultaneously communicating*.⌟

Hence, a transition t of an elementarily communicating service net N is either an internal transition, then $|send(t) \cup receive(t)| = 0$, or it is an interface transition and $|send(t) \cup receive(t)| = 1$. We may call a closed service net N elementarily communicating as well because such a net introduces no trouble during the translation into a service automaton either.

Obviously, neither the online shop N_{shop} of Fig. 3.2 nor its client N_{client} of Fig. 3.4 is elementarily communicating. In N_{shop}, the transition !invoice, !deliver is simultaneously communicating, and N_{client} has the simultaneously communicating transition ?new, !terms.

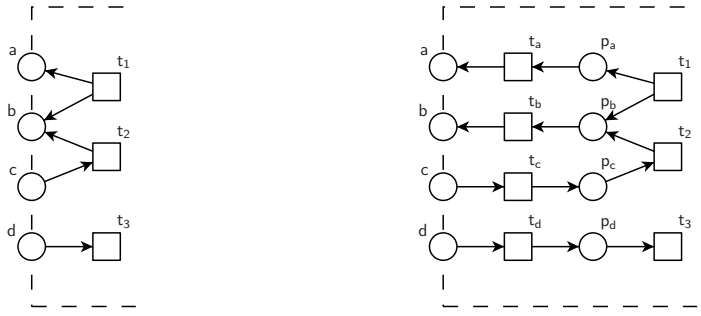
A nice property of elementarily communicating service nets is that this property is preserved when composing two elementarily communicating service nets.

Proposition 3.4.2 (Composition preserves sequential communication).
The composition $N \oplus M$ of two elementarily communicating service nets N and M is an elementarily communicating service net, too. ⌟

This proposition holds as, obviously, each interface transition t of the composed service net $N \oplus M$ has been an interface transition of either N or M. As these nets are elementarily communicating, t is still elementarily communicating in $N \oplus M$.

With the notion of elementarily communicating service nets defined, we are now ready to introduce a sequentialization of an arbitrary service net N into its elementarily communicating version $seq(N)$. Basically, the sequentialization introduces for each interface place of N a separate interface transition that handles all messages to or from the interface place in $seq(N)$. Therefore, this new transition is connected to a corresponding buffer place which takes over the role that the

interface place had in N. All prior interface transitions of N become internal in $seq(N)$ and read from or write to the respective buffer place(s) only. This construction is illustrated in Fig. 3.10.



(a) N (b) Sequentialization of N

Figure 3.10: Sequentialization of interface transitions. (a) The transitions t_1, t_2, and t_3 are connected to the interface places a, b, c, and d in N. (b) In the sequentialization $seq(\mathsf{N})$, these transitions are connected to the internal buffer places $\mathsf{p_a}$, $\mathsf{p_b}$, $\mathsf{p_c}$, and $\mathsf{p_d}$ instead. The connections to the buffers preserve all conflicts between these transitions. The new transitions $\mathsf{t_a}$, $\mathsf{t_b}$, $\mathsf{t_c}$, and $\mathsf{t_d}$ serve as forwarding transitions to (from) the real interface places of $seq(\mathsf{N})$.
A transition connected to more than one input place is sequentialized analogously to t_1.

The formalization of the sequentialization is given in the following definition. For each interface place of N, named x in the following definition, Definition 3.4.3 introduces a place p_x and a transition t_x and replaces all arcs (x,t) by a chain of arcs (x,t_x), (t_x,p_x), (p_x,t) if x is an input place of N, and each arc (t,x) is replaced by a chain of arcs (t,p_x), (p_x,t_x), (t_x,x) if x is an output place of N.

Definition 3.4.3 (Sequentialization of a service net, $seq(N)$).
Let $N = [P, P_{in}, P_{out}, T, F, m_0, \Omega]$ be a service net and let, without loss of generality, there exist new places $P^* = \{p_x \mid x \in P_{io}\}$ and new transitions $T^* = \{t_x \mid x \in P_{io}\}$ for N such that $P^* \cap P = T^* \cap T = P^* \cap T^* = \emptyset$.

Then, the *sequentialization* of the service net N is the service net $seq(N)$ defined as $seq(N) = [P', P_{in}, P_{out}, T', F', m_0, \Omega]$ with

- $P' = P \cup P^*$,
- $T' = T \cup T^*$, and
- $F' = F \setminus ((P_{in} \times T) \cup (T \times P_{out}))$
  $\cup \{(x,t_x),(t_x,p_x),(p_x,t) \mid x \in P_{in}, (x,t) \in F\}$
  $\cup \{(t,p_x),(p_x,t_x),(t_x,x) \mid x \in P_{out}, (t,x) \in F\}$.

Obviously, the sequentialization $seq(N)$ of a service net N and the net N itself are interface equivalent, and the initial and the final markings are preserved by the sequentialization (note the use of the extension of the multiset m_0, for example, to the superset P' of P in the last definition). In the special case of sequentializing a closed service net N, we obviously have $N = seq(N)$, as both P^* and T^* are empty.

Furthermore, each transition of the service net $seq(N)$ is connected to at most one interface place, i.e. $seq(N)$ is elementarily communicating. Hence, we will be able to directly apply the service automaton translation of the upcoming section to the service net $seq(N)$, as the translation will not introduce a multi-labeled transition in the corresponding service automaton.

It is easy to see that in many cases it would be sufficient to sequentialize a *subset* of interface places to remove any simultaneously communicating transition. In the example service net N of Fig. 3.10, the introduction of a buffer for the already elementarily communicating transition $\mathsf{t_3}$ of N is unnecessary, and it would be sufficient to introduce the buffering for the interface place b only to get elementarily communicating transitions $\mathsf{t_1}$ and $\mathsf{t_2}$. Even an unconnected interface place of some service net N would be sequentialized in $seq(N)$. This leaves space for optimization. However, in most cases there is no canonical subset of interface places to choose for sequentialization. For instance, if N had exactly one simultaneously communicating transition t with three interface places in $send(t) \cup receive(t)$, then sequentializing any two of these places is sufficient.

We now show that the sequentialization of a service net N preserves all its strategies.

Theorem 3.4.4 (Correctness of sequentialization for *Strat*).
A service net M is a strategy for a service net N iff M is a strategy for the service net $seq(N)$. ⌟

Proof.
From M being a strategy for N trivially follows that $N \oplus M$ is a closed service net. From $seq(N)$ being interface equivalent to N follows that $seq(N) \oplus M$ is a closed service net, too. Hence, the sequentialization according to Definition 3.4.3 corresponds to applying the standard Petri net transformation rule "fusion of series places" to get the service net $seq(N) \oplus M$ from the service net $N \oplus M$. This rule is known to preserve all relevant behavior [Mur89]. □

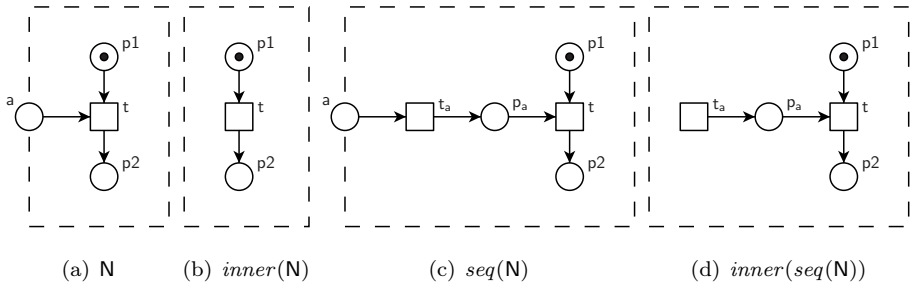
By commutativity of composition, we directly conclude that we can sequentialize both service nets without changing well-behavior of the composition.

Corollary 3.4.5 (Correctness of two sequentializations for *Strat*).
A service net M is a strategy for a service net N iff the service net $seq(M)$ is a strategy for the service net $seq(N)$. ⌟

In the special case of a closed service net N, the sequentialization $seq(N)$ preserves N's well-behavior, because $N = seq(N)$ in this case.

Proposition 3.4.6 (Correctness of sequentialization for well-behavior). A closed service net N is well-behaving iff the closed service net $seq(N)$ is well-behaving, too. ⌟

Due to Corollary 3.4.5 and Proposition 3.4.6, we have laid the basis to assume elementarily communicating service nets for the rest of this chapter without restricting generality. However, we will introduce another transformation for service nets first. This transformation will ensure that the inner $inner(N)$ of a service net N is bounded if and only if the inner $inner(seq(N))$ of its sequentialized version $seq(N)$ is bounded as well. This is not always the case with the sequentialization presented in Definition 3.4.3, as demonstrated in Fig. 3.11.



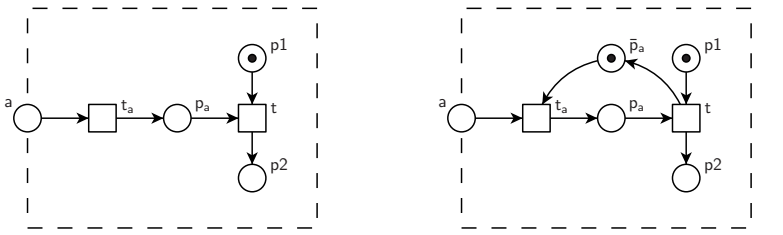
(a) N (b) *inner*(N) (c) *seq*(N) (d) *inner*(*seq*(N))

Figure 3.11: Sequentialization changes boundedness of the inner. Whereas the inner of N (in Fig. 3.11(b)) is bounded, the inner of $seq(N)$ (in Fig. 3.11(d)) is unbounded instead.

The inner of a service net will be the basis for the following translation of a service net N into a service automaton $SA(N)$. Whereas a bounded inner will result in a finite service automaton, an unbounded inner results in an infinite service automaton. Hence, the service automaton that is translated from a service net N might be substantially different from the service automaton that is translated from the sequentialization $seq(N)$ of N. This is obviously not desirable.

To overcome this problem, we informally introduce a small extension of the sequentialization according to Definition 3.4.3 in the following. Therein, for each input place x of N, we add *two* places (rather than one in Definition 3.4.3), p_x and $\bar{p}_x$, where $\bar{p}_x$ serves as a *complementary place* for p_x. Due to $\bar{p}_x$, the place p_x will never carry more than one token at the same time and $\bar{p}_x$ will be marked if and only if p_x is not marked. The construction is illustrated in Fig. 3.12.

The introduction of a complementary place is a well-known Petri net technique which preserves all relevant behavior (as it corresponds to applying the rule "fusion of series transitions", eliminating the place p_x, followed by applying the rule



(a) *seq*(N) (b) *seq*(N) with complementary place

Figure 3.12: Sequentialization with complementary place.

"eliminating self-loop places", eliminating the place $\bar{p}_x$, of [Mur89]). Hence, we may assume the introduction of complementary places without restriction of generality in the following and will still write $seq(N)$ meaning the new construction from now on. The main advantage of the new sequentialization $seq(N)$ is stated by the following proposition.

Proposition 3.4.7 (Sequentialization preserves bounded inner).
By introducing complementary places as illustrated in Fig. 3.12, the inner of a service net N, $inner(N)$, is bounded iff the inner $inner(seq(N))$ of its sequentialized version $seq(N)$ is bounded as well. ⌟

3.4.2 Translating Service Nets into Service Automata

The assumption of elementarily communicating service nets eases the following translation of a service net N into its service automaton $SA(N)$. Intuitively, $SA(N)$ is the reachability graph of the inner of N with its arc labels reflecting the changes at the interface. Because N is elementarily communicating, the labels of each transition of $SA(N)$ are uniquely determined by the corresponding service net transition. For an internal transition t of N, the resulting label is τ; for an interface transition t of N, the resulting label is the unique interface place that t is connected to.

Definition 3.4.8 (Translation $SA(N)$).
Let $N = [P, P_{in}, P_{out}, T, F, m_0, \Omega_N]$ be an elementarily communicating service net.

Then, the *corresponding service automaton* of N, $SA(N)$, is defined as $SA(N) = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ with

- $Q = \{q_m \mid m \in R_{inner(N)}(m_0)\}$,
- $I_{in} = P_{in}$,

- $I_{out} = P_{out}$,
- $\delta = \{(q_m, \tau, q_{m'}) \mid$ there is an internal transition t of N such that $m \xrightarrow{t} m'$ is a step of $inner(N)\}$
  $\cup \{(q_m, x, q_{m'}) \mid$ there is an interface transition t of N such that $m \xrightarrow{t} m'$ is a step of $inner(N)$ and $\{x\} = send(t) \cup receive(t)\}$,
- $q_0 = q_m$, with m is the initial marking of $inner(N)$, and
- $\Omega = \{q_m \mid m$ is a final marking of $inner(N)$, i.e. $m \in \Omega_{inner(N)}\}$. ⌟

The translation definition uses the set $\mathcal{MC}$ of message channels for linking the interfaces of N and $SA(N)$. As both the interface places of a service net and the interface channels of a service automaton are subsets of $\mathcal{MC}$, this connection is well-defined. That way, the transition labels of $SA(N)$ and the interface transitions of N are corresponding to each other.

The construction $SA(N)$ can already be seen as an abstraction of the behavior of N. The individual transition t responsible for a step $m \xrightarrow{t} m'$ cannot be identified by the corresponding transition $(q_m, x, q_{m'})$ as only the label x of t is preserved, not t itself. If there are transitions t_1 and t_2 with equal labeling in N and steps $m \xrightarrow{t_1} m'$ and $m \xrightarrow{t_2} m'$, then $SA(N)$ has only one respective transition from q_m to $q_{m'}$.

However, the translation preserves all relevant behavioral properties with respect to *Strat* and well-behavior. Correctness of the translation is justified by the following observations.

Lemma 3.4.9 (Correctness of $SA(N)$ for well-behavior).
A closed service net N is well-behaving iff the corresponding service automaton $SA(N)$ is well-behaving. ⌟

Proof.
A marking m is reachable in N iff the state q_m is internally reachable in $SA(N)$. Furthermore, m is a deadlock in N iff q_m is a deadlock state in $SA(N)$. Hence, by Definitions 3.2.17 and 3.3.24, we conclude that N is well-behaving iff $SA(N)$ is well-behaving. □

To prove the subsequent theorem, we consider the interplay of the SA translation and the composition operators $\oplus$ for service nets and service automata.

Lemma 3.4.10 (Relationship between different composition operators).
Let N and M be elementarily communicating service nets and $SA(N)$ and $SA(M)$ be the corresponding service automata.

Then, $SA(N \oplus M)$ is isomorphic to $SA(N) \oplus SA(M)$. ⌟

Proof.
$SA(N \oplus M)$ is constructed from the reachable markings of the inner of $N \oplus M$. For convenience in the rest of this proof, we denote such a reachable marking m of $inner(N \oplus M)$ as a sum $m_N + m_M + m_{shared}$ where m_N (m_M, resp.) is the restriction of m to the inner places of N (M, resp.) and m_{shared} is the restriction of m to the shared interface places between N and M. Furthermore, for each such marking m we write $q(m_N + m_M + m_{shared})$ meaning the state q_m of $SA(N \oplus M)$ that corresponds to m.

Analogously, $SA(N)$ and $SA(M)$ are constructed from the reachable markings m_N and m_M of the inner of N and M, respectively. We write $r(m_N)$ for the state of $SA(N)$ which corresponds to m_N, and $s(m_M)$ for the state of $SA(M)$ corresponding to m_M.

Now define a mapping $h : Q_{SA(N \oplus M)} \to Q_{SA(N) \oplus SA(M)}$ as $h(q(m_N + m_M + m_{shared})) = [r(m_N), s(m_M), m_{shared}]$. We show that h is an isomorphism.

If m_N and m_M are the initial markings of N and M, respectively, then (by definition of service net composition and construction of the inner of a service net) the marking $m_N + m_M + []$ is the initial marking of the inner of $N \oplus M$ and hence, (by definition of SA) $q^* = q(m_N + m_M + m_{shared})$ is the initial state of $SA(N \oplus M)$.

Analogously, $r(m_N)$ and $s(m_M)$ are the initial states of $SA(N)$ and $SA(M)$, respectively. By the definition of service automata composition, the state $[r(m_N), s(m_M), []] = h(q^*)$ is the initial state of $SA(N) \oplus SA(M)$. Hence, q^* is the initial state of $SA(N \oplus M)$ iff $h(q^*)$ is the initial state of $SA(N) \oplus SA(M)$.

The same argumentation holds for final markings m_N and m_M of N and M, respectively, because all interface places are empty in a final marking according to the definition of a service net.

Now assume that $m = m_N + m_M + m_{shared}$ is a reachable marking of $inner(N \oplus M)$, let x be an interface place of N and/or M, and let t be a transition of either N or M which is enabled at m. Then, there are eight possible cases for successor markings m' of m corresponding to the eight cases motivated directly before Definition 3.3.18 on page 69. We consider only one of these cases. The other cases are handled analogously.

1. x is a shared interface place and t is a sending transition of N.

Then, the successor marking of m with respect to t in $inner(N \oplus M)$ is $m' = m'_N + m_M + (m_{shared} + x)$. By the definition of SA, we have a state $q(m)$ in $SA(N \oplus M)$, the successor state $q(m')$, and an internal transition $(q(m), \tau, q(m'))$ in $SA(N \oplus M)$.

Analogously, the state $r(m_N)$ in $SA(N)$ has the successor state $r(m'_N)$ and a transition $(r(m_N), !x, r(m'_N))$ (as t is obviously also enabled in $inner(N)$). Because x is a shared channel and by the definition of service automata composition,

there is also a present internal transition at the state $[r(m_N), s(m_M), m_{shared}]$ in $SA(N) \oplus SA(M)$ leading to the state $[r(m'_N), s(m_M), (m_{shared} + x)] = h(q(m'))$.□

This lemma states that the order of applying the composition and the translation does not affect the resulting service automaton, i.e. translating the composed service net $N \oplus M$ into a service automaton on the one hand and composing the service automata $SA(N)$ and $SA(M)$ on the other hand results in one and the same service automaton.

Thus, we finally conclude:

Theorem 3.4.11 (Correctness of $SA(N)$ for *Strat*).
Let N and M be elementarily communicating service nets and $SA(N)$ and $SA(M)$ be the corresponding service automata.

Then, M is a strategy for N iff $SA(M)$ is a strategy for $SA(N)$. ⌟

Proof.
It suffices to show that $N \oplus M$ is well-behaving iff $SA(N) \oplus SA(M)$ is well-behaving.

By Lemma 3.4.9, we have $N \oplus M$ is well-behaving iff $SA(N \oplus M)$ is well-behaving. By Lemma 3.4.10, we know $SA(N \oplus M)$ is isomorphic to $SA(N) \oplus SA(M)$, which immediately gives $SA(N \oplus M)$ is well-behaving iff $SA(N) \oplus SA(M)$ is well-behaving.

Hence, $N \oplus M$ is well-behaving iff $SA(N) \oplus SA(M)$ is well-behaving. □

The value of Theorem 3.4.11 is that it justifies the whole approach of applying the analysis techniques for a service net N on the level of the corresponding service automaton $SA(N)$.

In the following, we introduce a translation of a service automaton A into a service net $PN(A)$ and show that $SA(PN(A)) = A$ for all service automata A in the following. Given a strategy service automaton B for a service automaton A, this property and Theorem 3.4.11 assure that $PN(B)$ is a strategy for $PN(A)$ on the service net level.

3.4.3 Translating Service Automata into Service Nets

We will now introduce the translation of a service automaton A into a service net $PN(A)$. Basically, each state q of A will result in a marking m_q of $PN(A)$ and a transition $d \in \delta$ of A will be a transition t_d of $PN(A)$. The label of the service automaton transition determines the connection of t_d to the interface. As we will construct a *state machine* as the underlying structure of $PN(A)$, each place of the inner of $PN(A)$, $inner(PN(A))$, exactly corresponds to a marking of $inner(PN(A))$. To achieve a well-defined Petri net $PN(A)$ (where the sets of places and transitions are always finite), we have to require that the service automaton A is finite (cp. Definition 3.3.6).

Definition 3.4.12 (Translation $PN(A)$).
Then, the *corresponding service net* $PN(A)$ of a finite service automaton A is defined as $PN(A) = [P, P_{in}, P_{out}, T, F, m_0, \Omega]$ with

- $P = \{p_q \mid q \in Q\} \cup I_{io\,A}$,
- $P_{in} = I_{in\,A}$,
- $P_{out} = I_{out\,A}$,
- $T = \{t_d \mid d \in \delta_A\}$,
- $F = \{(p_q, t_d), (t_d, p_{q'}) \mid d = (q, \tau, q') \in \delta_A\}$
  $\cup\, \{(p_q, t_d), (t_d, p_{q'}), (t_d, x) \mid x \in I_{out\,A},\ d = (q, x, q') \in \delta_A\}$
  $\cup\, \{(p_q, t_d), (t_d, p_{q'}), (x, t_d) \mid x \in I_{in\,A},\ d = (q, x, q') \in \delta_A\}$,
- $m_0 = [p_{q_0}]$, with q_0 is the initial state of A, and
- $\Omega = \{[p_q] \mid q$ is a final state of A, i.e. $q \in \Omega_A\ \}$. ⌟

Obviously, the service automaton A and the corresponding service net $PN(A)$ have equal interfaces, i.e. an element $x \in \mathcal{MC}$ is an input (output) channel of A if and only if x is an input (output) place of $PN(A)$, and we have:

Proposition 3.4.13 ($PN(A)$ is elementarily communicating).
If A is a service automaton, then $PN(A)$ is an elementarily communicating service net. ⌟

It is easy to see that a non-δ-reachable state q of A introduces place p_q of $PN(A)$ that will never get marked by a reachable marking of the inner of $PN(A)$. Hence, the translation of this service net back into a service automaton $SA(PN(A))$ filters such a place p and all transitions t with $p \in {}^\bullet t$. Thus, the property $SA(PN(A)) = A$ only holds for finite service automata A where each state is δ-reachable from the initial state.

Proposition 3.4.14 (Feasibility of translation $PN(A)$).
Let A be a (finite) service automaton where each state is δ-reachable from the initial state.

Then, $SA(PN(A)) = A$. ⌟

Proof.
Follows immediately from the constructions of $PN(A)$ and $SA(N)$. □

Another obviously possible translation of a service automaton into a Petri net is provided by the theory of regions [BD98, CKLY98]. Using the region approach, the constructed Petri nets would even be more readable as concurrency is explicitly represented.

The main correctness result for our translation $PN(A)$ is formulated in Theorem 3.4.15. It corresponds to Theorem 3.4.11 for the translation $SA(N)$.

Theorem 3.4.15 (Correctness of $PN(A)$ for *Strat*).
Let A and B be (finite) service automata where each state is δ-reachable from the respective initial state.

Then, B is a strategy for A iff $PN(B)$ is a strategy for $PN(A)$. ⌟

Proof.
By Theorem 3.4.11, $PN(B)$ is a strategy for $PN(A)$ if and only if $SA(PN(B))$ is a strategy for $SA(PN(A))$. By Proposition 3.4.14 and because each state of A and B is δ-reachable from the respective initial state, $SA(PN(B)) = B$ and $SA(PN(A)) = A$. Hence, $PN(B)$ is a strategy for $PN(A)$ iff B is a strategy for A. □

Because of the abstraction of the translation $SA(N)$ already discussed after Definition 3.4.8, we cannot always construct N exactly out of $SA(N)$:

Proposition 3.4.16.
In general, it does not hold: $PN(SA(N)) = N$. ⌟

As an example where N is different from the constructed service net $PN(SA(N))$ (i.e. translating N into a service automaton and then translating it back into a service net), assume that N has a dead transition t in the inner of N. Because t is dead, it has no representation in the corresponding service automaton $SA(N)$. Hence, it is no reintroduced when $SA(N)$ is translated back into a service net. Another example is a service net N with two different transitions with equal presets and postsets. The translation $PN(SA(N))$ will only have one such transition.

However, applying the back and forth translation once already reaches a fixed point.

Proposition 3.4.17.
For a service net N: $PN(SA(PN(SA(N)))) = PN(SA(N))$. ⌟

3.5 Possible Variants of Service Model Definitions

In the preceding sections, we introduced our formal modeling techniques of service nets and service automata, as well as behavioral compatibility notions for the interaction of services. Therein, we made several design decisions for the actual definition of some of the formal notions. In this section, we want to present and evaluate different ways for defining service nets, service automata, or related definitions and the corresponding influences to our presented results.

3.5.1 Services with Restrictions on their Final States

As introduced in Sects. 3.2.1 and 3.3.1, we allow service nets and service automata to have transient final markings or states, respectively. We motivated that this approach allows an easy way of modeling "communication rounds", i.e. the return of a service to its initial configuration after having successfully completed an interaction with some other service. That is, a transient final marking can be compared with the approach of adding a shortcut transition t^* to a workflow net, connecting the unique sink place ω with the unique source place α of the workflow net [Aal98].

However, requirements from the application domain or modeling guidelines, for example, may result in the need for stronger criteria on the final markings/states of a service. In general, such restrictions can be added to the definition of a service net (or service automaton) without any harm. We consider the implications of two prominent restrictions in the following.

Services with Non-Transient Final States

The first restriction is to require all final markings to be dead markings, or all final states to be stable states, respectively. This requirement was already made in [MRS05, LMW07b, MSSW08, SMB09], for instance. As an advantage of this approach, the notions of *final* markings/states get a better intuitive meaning.

As a technical problem, however, this stronger requirement results in a different definition of the inner of a service net (Definition 3.2.4 on page 51), because a dead final marking of a service net may become a transient final marking of its inner (see Fig. 3.13). Hence, the construction of the inner may violate such a restricted service net definition, i.e. the inner of a service net is no (restricted) service net. To overcome this problem, the inner of a service net may be defined as a variant of Petri nets with final markings, and not at all as a service net, for instance. Alternatively, the inner of a service net with restricted final states can be defined as a service net with relaxed final states, i.e. the class used in this thesis.

Services with Strictly Terminating Final States

An even more restricted variant of services is gained by requiring that all final markings/states are *strictly terminal*. Thereby, a marking m of a service net is called strictly terminal if m is a dead marking even in $inner(N)$. Correspondingly, a state q of a service automaton is strictly terminal if there is no present transition at q at all.

(a) N (b) *inner*(N)

Figure 3.13: A Service net N and its inner *inner*(N). If [p1] is a final marking of N (and hence a final marking in *inner*(N)), then *inner*(N) does not satisfy the criterion for dead final markings. Whereas the marking [p1] is dead in N, it is transient in *inner*(N).

This variant of final states is of particular interest as a service can never again perform an action after having reached a strictly terminal state once. This property is important for practical applications as it is very easy to check, but still sufficient to positively decide that a service (or a service instance) is no longer needed and, hence, can be "shut down" (and deleted from memory). In fact, all services that stem from BPEL, BPEL4Chor, or almost every other practical specification language are strictly terminating.

As in the variant without restrictions on final markings/states, we are not aware of any problems arising in the strictly terminating variant of services. Only the notion of a normal state of Boolean annotated service automata slightly changes again (see Sect. 4.6.1).

Because services without restrictions on their final markings/states are the more general class, we decided to consider that variant of services in this thesis.

3.5.2 Strategy Notion for Non-Closed Compositions

In Sects. 3.2.3 and 3.3.3, respectively, the behavioral correctness notion of *well-behavior* of a service has been defined for *closed* services only. Therein, a service is a strategy for another service only if their composition is closed. In fact, this approach was also used in [LMW07b, MSSW08, SMB09], for instance. However, in some cases, an open composition of two services may already be seen as "complete enough" to call the services strategy for each other.

In this section, we show that our distinction between open and closed compositions does not restrict generality as every pair of service nets N and M can be transformed in a way such that their composition is closed.

Transformation to Yield Closed Compositions

Assume two service nets N and M where $N \oplus M$ is an open service net. Obviously, N and M have free interface places (cp. Definition 3.2.9). Such an interface place p of N can simply be added to M as a new interface place with ${}^\bullet p = p^\bullet = \emptyset$, and vice versa. Then, the composition of the (changed) service nets N and M is closed, and well-behavior of $N \oplus M$ is well-defined.

Even if we want to assure that one side of the composition, say N, is fixed and unchanged (as it will be done in the forthcoming chapters), a closed composition $N \oplus M$ can still be achieved: all free interface places of N are added to M as described before. In contrast, each free interface place p of M is not added to N, but is removed from M. To preserve the enabledness of the transitions of M, each receiving transition of M with p in its preset is removed from M as well, as such a transition is never enabled in $N \oplus M$. Then, the composition of N and M is closed as well.

It is easy to see that this approach equally works for service automata.

Hence, our assumption of a closed composition $N \oplus M$ for the strategy relation between N and M does not restrict generality.

3.6 Related Work

Since the beginning of service-oriented computing, lots of efforts have been spent to back the industrial (web) service specification languages with formal languages to model and reason about services. Thereby, two main directions of the suggested models can be distinguished, *synchronously* and *asynchronously* communicating service models. Whereas synchronous modeling languages can be used to analyze simple request/response-based, stateless services, recent literature agrees that stateful services that communicate via asynchronous message passing are needed today [Pap07a].

[BFHS03] shows that many behavioral compatibility notions are much easier to decide in case of synchronous communication models. Hence, the results derived for synchronous communication models cannot be trivially extended to asynchronously communicating services. However, [FBS05] was able to provide sufficient conditions under which the synchronous and asynchronous communication models can be translated into each other. Basically, this is the case if the asynchronous model can get along with a message buffer of length one.

Synchronous Communication

Several authors suggest the translation of industrial service specification languages into process algebra. For instance, [BCPV04] describes the translation of conver-

sations (i.e. choreographies) of web services into the process algebra CCS. Alternatively, [Fer04] presents a formal semantics for BPEL4WS web services in the process algebra Lotos. Given a translated process algebra term of a (basic or composite) service, it can be evaluated for deadlock freedom, inducing behavioral compatibility of the services, or any other property expressible in the process algebra. Additionally, [BCPV04] sketches ideas for synthesizing an adapter (in form of a CCS term) for repairing a deadlocking conversation.

Most other formal models for synchronously communicating services are based on automata. In [Lyn96], a very basic version of such automata, called *I/O automata*, have been introduced. Most notably, an I/O automaton must be receptive towards every possible input action at every state. This is also called an *input enabled*, or "pessimistic" approach as each automaton must be able to react on each incoming message at any time. Consequently, composition of I/O automata is very simple. However, the input enabled approach results in rather inflexible and complex automata models and is not suited to model independent and loosely coupled services.

Consequently, [AH01] introduced the notion of *interface automata* which are syntactically closely related to I/O automata. They also assume a synchronization of communication actions, whereas internal actions are interleaved asynchronously. In contrast to I/O automata, interface automata follow an "optimistic" approach where some (or even all) input actions may be not accepted at a state of an interface automaton. This may lead to erroneous states, called *illegal states* in [AH01], when interface automata are composed. In an illegal state, one interface automaton wants to send a message but the other one is not able to receive this message. Hence, [AH01] introduces a compatibility notion for composed interface automata, basically stating that, at each pair of states q and q' of two interface automata, q must at least receive the messages that q' sends (and vice versa).

[BBMP06] follows a very similar approach as [AH01] and considers service models as finite-state machines. The authors also introduce a compatibility notion that requires that each sent message by one automaton must be receivable by the other automaton.

Another automaton approach to represent services can be found in [BSBM04]. Therein, the authors model services as labeled transition systems and consider different (simple) compatibility notions, namely *opposite behaviors*, *unspecified receptions*, and *deadlock-freeness*, of the transition systems. Opposite behaviors reflect the mirroring of input and output actions, unspecified receptions equals the compatibility notion of [AH01] and [BBMP06], and deadlock-freeness additionally requires reachability of a final state. The approach is limited to deterministic automata and the authors neither provide algorithms to decide the compatibility notions nor complexity results.

[YS97] provides, besides a compatibility notion, also means of correct implementation of a protocol (i.e. a service specification) and the generation of basic adapters.

However, the authors restrict themselves to deterministic services, assume a synchronous model of communication and agree that the generalization to asynchronous communication is nontrivial.

In [BCGM05], *service composition* is introduced as the question whether it is possible to orchestrate a set of available services such that a target specification is satisfied. Therein, the services as well as the target specification are given as finite-state machines. If possible, an orchestrator, called *mediator*, is synthesized. The mediator has full access to all actions of the available services and may trigger or delay actions of these services to satisfy the target specification. Due to privacy issues, this approach is obviously only feasible in practice if all services belong to one organization and full, centralized access to the actions is guaranteed (and acceptable).

Although *Petri nets* (see [Rei85], for example) usually composed via place fusion (modeling asynchronous communication), a synchronous composition mechanism (i.e. transition fusion) for Petri nets is possible as well. Synchronous communication of Petri nets is used in [Aal03, AW01] to compose *workflow nets* [Aal98], for instance. Furthermore, [Wol07, Wol09] consider service nets as introduced in this thesis and present a non-trivial generalization of our notion of strategies where service nets can employ both synchronous and asynchronous communication. Consequently, the notion of controllability of a service net N (cp. Definition 3.2.20) is generalized to the existence of a service net M that communicates synchronously and/or asynchronously with N such that N and M are behaviorally compatible. However, there is no characterization (i.e. no operating guideline) of the set of all such strategies so far.

To support more involved services than mere stateless, request/response-based services, we use service models that communicate asynchronously via message passing in this thesis.

Asynchronous Communication

Most automata dialects for asynchronously communicating are based on [BZ83] where the authors introduce *communicating finite-state machines* that exchange messages via unbounded queues. That is, each communication channel between two communicating finite-state machines is represented by one unbounded buffer, preserving the order of messages. The authors show that many properties are undecidable in case of unbounded channels, but they become decidable when considering only bounded channels.

[HBCS03] provides an overview on different views on SOC and proposes to model services as mealy machines with asynchronous communication, using bounded or unbounded queues very similar to [BZ83]. They suggest to check behavioral compatibility of services by composing the service models and checking the com-

position for the correctness criterion applying standard model checking techniques (in case of bounded channels).

In contrast to the above articles, [KP06] considers asynchronous communication between labeled transition systems such that messages can overtake each other. The authors introduce a *conformance notion* to express correct implementation of a (composite) service specification by a concrete (composite) service. To this end, they assume boundedness of the actual communication and that each sent message is eventually received. Then, conformance reduces to the existence of a simulation relation between implementation and specification.

Service automata, as introduced in this thesis, are most closely related to the labeled transition systems of [KP06]. However, we additionally introduce final states for being able to distinguish between proper and erroneous termination and an explicit notion of composition of service automata. That way, we can express behavioral compatibility of two service automata A and B as deadlock freedom of their composition. In order to check a relationship between specification and implementation, we will employ the *operating guidelines* of the services as introduced in the forthcoming Chap. 5. It will turn out that our conformance notion is much more flexible than the one introduced in [KP06].

In this chapter, we also introduced *service nets*, a special version of Petri nets, for the modeling of services. As shown in Sect. 3.4, service nets and service automata can be translated into each other without loosing behavioral properties of the service.

Petri nets with special structures, i.e. workflow nets, have been proven successful in the area of business process modeling. For *distributed* business processes, workflow nets have been enriched by interface places to *workflow modules*, enabling a workflow module to communicate asynchronously with other workflow modules [Mar04]. This idea is based on the module concept for Petri nets which was first proposed by Kindler [Kin97]. Basically, service nets are workflow modules where structural restrictions (i.e. a unique source and a unique sink place, etc.) have been dropped. [Mar04] furthermore originally introduced the notion of *usability* of a workflow module, which exactly equals the notion of controllability of a service net. However, all results in [Mar04] are restricted to services with acyclic behavior and the author does not address a characterization of all strategies of a service.

3.7 Concluding Remarks

In this chapter, we have introduced two formal modeling techniques to model services: *service nets* and *service automata*. Service nets are a special class of Petri nets with distinguished interface places to communicate with other service nets. Service automata are a version of communicating automata and can be

seen as a representation of the *behavior* of a service. We formalized behavioral compatibility of services as a strategy relationship between the services, which basically means deadlock freedom of their composition.

We were able to show that service nets and service automata can be *translated into each other* and are therefore equally well suited as a formal representation of services and their interaction. To this end, we introduced a sequentialization of service nets and proved with Corollary 3.4.5 and Proposition 3.4.6, that we can assume elementarily communicating service nets for the translation into service automata. Furthermore, we were able to show that both translations preserve our correctness criteria for behavioral compatibility. The main results in this regard are Theorem 3.4.11 and Theorem 3.4.15. These results enable us to change arbitrarily between these two formalisms without losing information about the set of strategies of a service.

An important strength of Petri nets is their nice and intuitive graphical representation, especially in the case of concurrency. This significantly eases the understanding of service models and allows for the manual design of larger case studies. Furthermore, service net models of services can be derived from various formalisms, e.g. the industrial service description language BPEL and the service choreography specification language BPEL4Chor, by applying existent formal service net semantics (like the semantics of [Loh08], for instance) of these languages. That way, our algorithms and techniques can directly be applied to real-world processes.

Service automata, however, are well suited to analyze the behavior of services and serve as the basis of our analysis techniques in the upcoming chapters. Service automata models of services can easily be derived from service nets or by automata semantics of practical service specification languages. In the forthcoming chapters, we may develop all further service analysis techniques for service automata only. Due to the existent translation, in turn, every result equally holds for service nets as well and could easily be formulated on the service net level.

In fact, the implementation of the analysis methods is based on service nets rather than service automata (cp. Chap. 7).

The formulation of all analysis questions on the formal level makes our approach independent of the evolution of real-world service description languages and thus durable. As our algorithms are computer-aided, the formal level can, however, to a large extend be hidden in real applications of our methods. Furthermore, it is easy to incorporate the support of other specific practical process description languages, such as BPMN [OMG06] and UML [OMG07], for example, or to provide methods to support further formal service modeling techniques like (other) automata dialects or process algebra descriptions of services, for example.

Part II

Analyzing the Interaction Behavior of Services

The set of all strategies of a given service, i.e. the set of all services that are behaviorally compatible to a given service, is of particular interest. If the set of strategies of a service is empty, the service is uncontrollable. Uncontrollable services are fundamentally ill-designed. In the case of controllable services, the set of strategies is infinite. For service discovery, however, efficient techniques for finding compatible services in a repository are needed. Therefore, a finite characterization of that set is important. *Operating guidelines* can be used as such a characterization.

4 Characterizing Sets of Services

In the last chapter, we have introduced our formal framework for services, i.e. service nets and service automata, and our behavioral compatibility notion of a strategy for a service. We have seen that a service S canonically induces the *set of all strategies* R for S, denoted by $\mathit{Strat}(S)$. As motivated in Sect. 1.3, our main goal of this thesis is the characterization of the set of all behaviorally compatible services R for a given service S, i.e. of the set $\mathit{Strat}(S)$ of S. Before we will introduce this particular characterization in the forthcoming Chapter 5, this chapter is devoted to present a general means for characterizing *a* set of services.

To this end, we introduce *Boolean annotated service automata* (*BSA*s) in the following. Basically, a *BSA* B^ϕ is a service automaton B where each state q of B is annotated by a Boolean formula $\phi(q)$. The purpose of a *BSA* B^ϕ is to characterize a set of service automata. Therefore, we develop a *matching procedure* to decide whether or not a concrete service automaton C is characterized by a *BSA* B^ϕ. That way, a *BSA* B^ϕ canonically induces the set $\mathit{Match}(B^\phi)$ of all service automata C that match with B^ϕ. The concept of *BSA*s will turn out to be well suited to (1) very efficiently decide $C \in \mathit{Match}(B^\phi)$ and (2) easily compare the sets $\mathit{Match}(B_1^\phi)$ and $\mathit{Match}(B_2^\psi)$ of characterized services of two *BSA*s B_1^ϕ and B_2^ψ. Hence, we already take into account the requirements for a characterization as introduced in Sect. 1.3.

In Chap. 5, we will employ the concept of *BSA*s and introduce specifically constructed *BSA*s, called *operating guideline* of a service S, that characterize exactly the set of strategies $\mathit{Strat}(S)$ of S. In other words, the operating guideline OG_S of a service S is a special *BSA* such that $\mathit{Match}(OG_S) = \mathit{Strat}(S)$. Therein, the Boolean annotations are used to express requirements on a client service R such that R satisfies these requirements if and only if R is a strategy for S. The requirements are *abstract* and express conditions for R, rather than describe the internal structure of S. Hence, a characterization of strategies by a *BSA* additionally takes into account privacy issues and hides relevant trade secrets of S.

As *BSA*s are a general concept and can be used without the purpose of characterizing exactly the set $\mathit{Strat}(S)$ of a service S, we will introduce *BSA*s as a very general concept for characterizing some set of services in this chapter, without

considering a special target service S at the moment. We will present many results that hold for arbitrary *BSA*s, rather than those *BSA*s that characterize a set of strategies.

Throughout this chapter, we will only consider the characterization of services that are modeled as service *automata* and omit the respective definitions for service *nets*. To consider a service net N, one can easily translate N into its corresponding service automaton $SA(N)$ and consider all analysis questions for $SA(N)$ instead of N directly.

The structure of this chapter is as follows. Section 4.1 introduces basic definitions and notations that are used in the following. In Sect. 4.2, we define *Boolean annotated service automata* (*BSA*s) as a finite characterization of a possibly infinite set of service automata. We introduce the matching procedure to efficiently decide which service automata are characterized by a *BSA*. The services which match with a *BSA* B^ϕ constitute the matching set $Match(B^\phi)$. Section 4.3 starts with the observation that a *BSA* may have redundant information that can be removed without changing the matching results, i.e. without changing the set *Match*. Thus, we introduce a normalization procedure for a *BSA* that deletes all such garbage, yielding a *normal BSA*. In Sect. 4.4, we show how the matching sets of two normal *BSA*s can be compared by looking at their structure only. We develop a relation $\sqsubseteq$ between the structures of *BSA*s which imposes a preorder on their matching sets. The equivalence of *BSA*s is topic of Sect. 4.5. Herein, we introduce a minimization procedure for a *BSA* that basically merges equivalent states of a *BSA* without jeopardizing its set *Match*. We will prove that the resulting minimized *BSA* is minimal and unique, i.e. there is no other *BSA* with equal or less states and transitions with the same matching set. Section 4.6 justifies some design decisions for definitions in this chapter and explores other possible variants of *BSA* definitions together with the corresponding consequences. Then, Sect. 4.7 discusses related work and Sect. 4.8 concludes this chapter.

4.1 Preliminaries

In this section, we recall basic terminology that will be used in the remainder of this chapter. The familiar reader may skip this section.

4.1.1 Strong Simulation Relation

In order to characterize sets of service automata, we need a means for comparing the behavior of two service automata. To this end, we recall the classical notion of strong simulation [Mil71, Mil89] and adapt it to the definition of service automata.

Intuitively, a service automaton A that is strongly simulated by a service automaton B has less or equal behavior than B. Technically, there must exist a strong

simulation relation ϱ between (the states of) A and B satisfying special requirements. Please note the direction: if B strongly simulates A, then $\varrho \subseteq Q_A \times Q_B$.

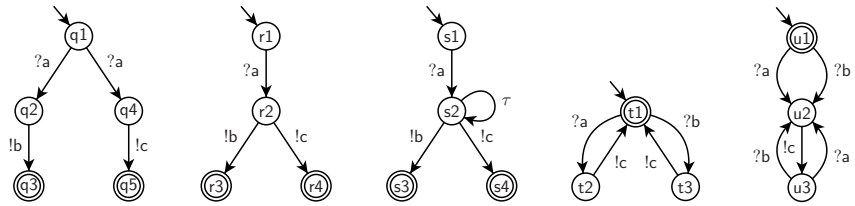
Definition 4.1.1 (Strong simulation relation).
Let $\varrho \subseteq Q_A \times Q_B$ be a relation between the states Q_A and Q_B of two service automata A and B such that, for all states $q_A \in Q_A$ of A and all states $q_B \in Q_B$ of B: if $(q_A, q_B) \in \varrho$ and there is a transition $(q_A, x, q'_A) \in \delta_A$ in A, then there is a transition $(q_B, x, q'_B) \in \delta_B$ in B with $(q'_A, q'_B) \in \varrho$, too.

Then, ϱ is a *strong simulation relation* between A and B if the initial states of A and B are related by ϱ, i.e. $(q_{0\,A}, q_{0\,B}) \in \varrho$.

If there exists a strong simulation relation ϱ between A and B, then we also say that B *strongly simulates* A. ⌟

In Definition 4.1.1, an internal, i.e. τ-labeled, transition is treated exactly like an interface (i.e. sending or receiving) transition. That is, if there is a present internal transition at a state q_A in A, then there must be a present internal transition at q_B in B for q_B simulating q_A.

Examples for strong simulation relations are given in Fig. 4.1. The figure depicts final states for the five service automata, but whether a state is a final state or not is irrelevant for checking simulation relations — only the presence or absence of transitions is important.



(a) A (b) B (c) C (d) D (e) E

Figure 4.1: (a)–(e) Service automata A, B, C, D, and E. The service automaton B strongly simulates A, and C strongly simulates B (but not vice versa, respectively), whereas the service automata D and E strongly simulate each other.

Another simulation relation known in the literature is *weak simulation* [Mil71, Mil89]. The main difference is the treatment of τ-labeled transitions: a weak simulation relation ϱ ignores internal transitions, that is, an automaton A can have an internal transition at a state q_A without enforcing the automaton B to have an internal transition itself at a state q_B with $(q_A, q_B) \in \varrho$. In the example of Fig. 4.1, the service automaton B weakly simulates C, but B does not strongly simulate C due to the τ-transition in C.

In the upcoming Chap. 5, the construction of an operating guideline of a service ensures that a strategy for this service is strongly simulated by the operating guideline. For this reason, we may consider only strong simulation relations in the following.

Notation 4.1.2.
In the rest of this thesis, we say simulation for short meaning a strong simulation relation. ⌟

Whereas Definition 4.1.1 just defines requirements of a simulation relation, there is a straightforward algorithm to construct a simulation relation between service automata A and B (if there is one): starting from the initial states $q_{0\,A}$ of A and $q_{0\,B}$ of B, it considers all transitions present at $q_{0\,A}$ in A and checks whether there are correspondingly labeled present transitions at $q_{0\,B}$ in B. The state pairs which are reached this way are added to the simulation relation. This step is iterated until no pair of states is added to ϱ anymore (cp. [Mil89], for instance).

It is well-known that this algorithm constructs a simulation relation ϱ where removing a state pair from ϱ immediately violates the requirements of Definition 4.1.1. Such a simulation relation is called a *minimal simulation relation*.

Definition 4.1.3 (Minimal simulation relation).
A simulation relation ϱ between two service automata A and B is *minimal* if there is no simulation relation ϱ' between A and B with $\varrho' \subset \varrho$. ⌟

Furthermore, a minimal simulation relation only relates reachable states.

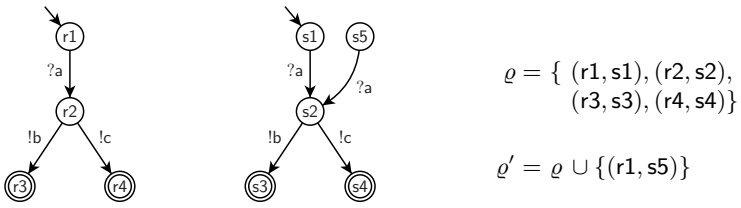
Proposition 4.1.4 (Minimal simulation relation vs. reachable states).
Let A and B be service automata such that B simulates A with minimal simulation relation $\varrho \subseteq Q_A \times Q_B$.

Then, for each state q_A of A and each state q_B of B: $(q_A, q_B) \in \varrho$ implies q_A is δ_A-reachable in A and q_B is δ_B-reachable in B. ⌟

Additionally, the minimal simulation relation between some service automaton A and a deterministic service automaton B is *unique*. In the remainder of this thesis, the service automaton B will always be deterministic.

Figure 4.2 exemplifies the existence of two different simulation relations ϱ and ϱ'. The simulation relation $\varrho = \{(\mathsf{r1}, \mathsf{s1}), (\mathsf{r2}, \mathsf{s2}), (\mathsf{r3}, \mathsf{s3}), (\mathsf{r4}, \mathsf{s4})\}$ is minimal. The relation $\varrho' = \varrho \cup \{(\mathsf{r1}, \mathsf{s5})\}$ uses the unreachable state $\mathsf{s5}$ of B' and is a simulation relation as well, but obviously not minimal. Because B' is deterministic, there is no other minimal simulation relation than ϱ.

Another simulation relation known in the literature is a *bisimulation* relation [Par81, Mil89]. A relation ϱ is a bisimulation between A and B, if ϱ is a simulation



(a) B (b) B′ (c) ϱ and ϱ'

Figure 4.2: Two service automata B and B′ with simulation relations ϱ and ϱ'.

relation between A and B and its inverse ϱ^{-1} is a simulation relation between B and A.

Hence, if *both* service automata are deterministic, then the unique minimal simulation relation between A and B is even a bisimulation.

Proposition 4.1.5 (Minimal simulation relation vs. bisimulation).
Let A and B be two deterministic service automata.

Then, if A simulates B with minimal simulation relation ϱ, then ϱ is a bisimulation between A and B. ⌟

Simulation relations are the first ingredient for service automata characterization. The next section is devoted to Boolean formulae and their evaluation which will be the second ingredient.

4.1.2 Boolean Formulae

In the following, we will employ a positive, i.e. negation-free, variant of propositional logics, called *Boolean formulae*. These formulae are used as annotations to the states of a service automaton. That way, the annotated service automaton can be used to characterize a set of services. The atomic propositions of such a Boolean formula are elements of the set $\mathcal{MC}$ of message channels together with the special elements τ, representing an internal action, and *final*, representing final states. As the only Boolean connectors, we use $\vee$ (Boolean *or*) and $\wedge$ (Boolean *and*). Let, for the rest of this thesis, $\mathcal{MC}^+$ denote the set $\mathcal{MC} \cup \{\mathit{final}, \tau\}$.

Definition 4.1.6 (Boolean formula over $\mathcal{MC}^+$, $\mathcal{BF}$).
The set $\mathcal{BF}$ of *Boolean formulae* over $\mathcal{MC}^+$ is inductively defined as follows:

Basis. $x \in \mathcal{MC}^+$ is a Boolean formula, as well as the special Boolean formulae *true* and *false*;

Step. If ϕ and ψ are Boolean formulae, then their *disjunction*, $(\phi \vee \psi)$, and their *conjunction*, $(\phi \wedge \psi)$, are Boolean formulae. ⌟

As a syntactical simplification, we omit parentheses in case no confusion may arise, writing e.g. $a \vee b \vee c$ for $((a \vee b) \vee c)$.

Notation 4.1.7.
Boolean formulae are denoted by lower case Greek letters, e.g. ϕ, ψ, or χ. An element $x \in \mathcal{MC}^+$ is called a *literal*. We write $x \in \phi$ if the literal x occurs in ϕ.⌟

To evaluate Boolean formulae, we use standard propositional logic semantics:

Definition 4.1.8 (Truth value, Boolean assignment, satisfaction, β, $\models$).
As usual, we fix the *truth values true* and *false*.

A *Boolean assignment* β (assignment for short) is a mapping $\beta : \mathcal{MC}^+ \to \{true, false\}$ assigning to each literal a truth value.

Furthermore, an assignment β *satisfies* a Boolean formula ϕ, denoted $\beta \models \phi$, if ϕ evaluates to *true* under β using standard propositional logic semantics. Thereby, the formulae *true* and *false* are always evaluated to the truth value *true* and *false*, respectively. ⌟

Boolean assignments can be compared by comparing the truth values for each literal separately. As usual, the truth value *true* is considered to be greater than the truth value *false*.

Definition 4.1.9 (Domination of assignments, $\beta \leq \beta'$).
An assignment β is *dominated* by an assignment β', denoted $\beta \leq \beta'$, if $\beta(x) = true$ implies $\beta'(x) = true$, for each $x \in \mathcal{MC}^+$. ⌟

As we do not allow negation in Boolean formulae, we can immediately conclude:

Corollary 4.1.10 (Monotonicity of Boolean formulae).
For each Boolean formula ϕ and all assignments β, β': if $\beta(x) \models \phi$ and $\beta \leq \beta'$, then $\beta'(x) \models \phi$. ⌟

Hence, a Boolean formula ϕ which is evaluated to *true* under an assignment β is also evaluated to *true* by any greater assignment. Consequently, if ϕ is evaluated to *false* by an assignment, then ϕ is evaluated to *false* by any smaller assignment. This property of negation-free Boolean formulae will be exploited to derive a normal form of Boolean annotated service automata in Sects. 4.2 and 4.3. The normal form of such Boolean annotated automata is crucial for comparing the sets of characterized services in Sect. 4.4 and is a prerequisite for the canonical characterization of services in Sect. 4.5.

In the following, we fix some further notations for Boolean formulae.

Notation 4.1.11.
If each satisfying assignment of a Boolean formula ϕ satisfies a Boolean formula ψ, then ϕ *implies* ψ, denoted by $\phi \Rightarrow \psi$. If ϕ and ψ are satisfied by the same assignments, they are *equivalent*, denoted by $\phi \equiv \psi$.

A Boolean formula ϕ that is satisfied by any assignment is a *tautology*, denoted by $\phi \equiv \mathit{true}$. A Boolean formula ϕ that is satisfied by no assignment is a *contradiction*, denoted by $\phi \equiv \mathit{false}$.

If $\mathcal{F} \subseteq \mathcal{BF}$ is a finite set of Boolean formulae $\mathcal{F} = \{\phi_1, \ldots, \phi_n\}$, the conjunction $\phi_1 \wedge \ldots \wedge \phi_n$ is abbreviated by $\bigwedge_{\phi \in \mathcal{F}} \phi$. If $\mathcal{F} = \emptyset$, then the *empty conjunction* $\bigwedge_{\phi \in \emptyset} \phi$ is defined to be *true*. The disjunction $\bigvee_{\phi \in \mathcal{F}} \phi$ over a finite set of Boolean formulae $\mathcal{F}$ is defined analogously; and the *empty disjunction* is defined to be *false*. ⌟

Remark 4.1.12 (Syntactical simplification of Boolean formulae).
We recall some basic simplifications for Boolean formulae:

- $(\phi \wedge \mathit{false}) \equiv \mathit{false}$,
- $(\phi \vee \mathit{false}) \equiv \phi$,
- $(\phi \wedge (\phi \vee \psi) \equiv \phi$,
- $(\phi \vee (\phi \wedge \psi)) \equiv \phi$,

which we will apply when normalizing annotations in Sect. 4.3. ⌟

Boolean formulae are used as annotations to the states of a service automaton in the following. With the help of the notion of a simulation relation and the evaluation of the Boolean formulae, such an annotated service automaton can be used to characterize a set of service automata.

4.2 Boolean Annotated Service Automata

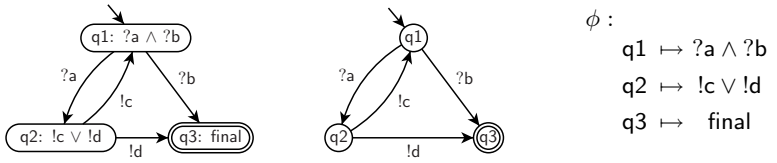
A *Boolean annotated service automaton* (*BSA*) B^ϕ, as introduced in this section, is used to characterize a set of service automata. To this end, each state q of the service automaton B is annotated by a Boolean formula $\phi(q)$. Because deterministic service automata provide all information needed later on while easing most subsequent decision procedures, we restrict *BSA*s to deterministic structures.

Definition 4.2.1 (Boolean annotated service automaton, *BSA*).
A *Boolean annotated service automaton* (*BSA*) $B^\phi = [B, \phi]$ consists of

- a *deterministic* service automaton $B = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ and
- a mapping $\phi : Q \to \mathcal{BF}$, called (Boolean) *annotation*. ⌟

We refer to B as the *underlying service automaton* of a *BSA* B^ϕ. According to Definition 4.2.1, the underlying service automaton B is deterministic (cp. Definition 3.3.8 on page 64); that is, B may have τ-labeled transitions, but for all $x \in \mathcal{MC} \cup \{\tau\}$, there is at most one present x-labeled transition per state, i.e. $(q, x, q'), (q, x, q'') \in \delta_B$ implies $q' = q''$, for all $q \in Q$.

An example *BSA* B^ϕ is depicted in Fig. 4.3(a). Its underlying service automaton B consists of three states und four transitions and is shown in Fig. 4.3(b). In B^ϕ, each state q of B is annotated with a Boolean formula $\phi(q)$; the complete mapping ϕ is listed in Fig. 4.3(c).



(a) B^ϕ (b) B (c) ϕ

Figure 4.3: A Boolean annotated service automaton B^ϕ with its constituents.

Notation 4.2.2.
If we consider more than one *BSA*, we employ indices or primed letters, e.g. B_1^ϕ or B'^ϕ. ⌟

Notation 4.2.3.
To emphasize the connection between a literal and the corresponding message channel, we also denote a literal $x \in \mathcal{MC}$ in a Boolean formula $\phi(q)$ of a *BSA* B^ϕ by $?x$ if x is an input channel of B, or by $!x$ if x is an output channel of B.

Furthermore, we lift the notion of interface equivalence of service automata (Definition 3.3.13) to *BSA*s and call two *BSA*s B_1^ϕ and B_2^ψ *interface equivalent* iff the underlying service automata B_1 and B_2 are interface equivalent. ⌟

The purpose of a *BSA* is to characterize service automata. To this end, we introduce the notion of the matching of a service automaton with a *BSA*.

4.2.1 Matching

Intuitively, a service automaton C matches with a *BSA* B^ϕ if B simulates C and the states of C provide assignments that satisfy all formulae of ϕ.

Thereby, the labels of present transitions at a state q of a service automaton C constitute an assignment: a present x-labeled transition sets the corresponding

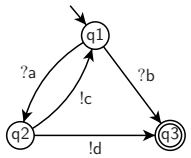
literal x to *true* at q. If there is no x-labeled transition at q, then the literal x is set to *false*. The special literal *final* is set to *true* at state q if and only if q is a final state of C. This assignment is used to evaluate the annotations of the corresponding states of a *BSA* later on.

Definition 4.2.4 (Assignment of a service automaton, β_C).
An *assignment* β_C *of a service automaton* $C = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ is a set $\{\beta_C(q) \mid q \in Q\}$ of Boolean assignments $\beta_C(q) : \mathcal{MC}^+ \to \{true, false\}$ defined as

$$\beta_C(q)(x) = \begin{cases} true, & \text{if } x \in \mathcal{MC}^+ \setminus \{final\} \text{ and there is a } q' \text{ with } (q, x, q') \in \delta, \\ true, & \text{if } x = final \text{ and } q \in \Omega, \\ false, & \text{otherwise.} \end{cases}$$

⌟

To show an example, reconsider the service automaton B of Fig. 4.3(b), depicted again in Fig. 4.4(a). The corresponding assignments are listed in Fig. 4.4(b). A literal $x \in \mathcal{MC}^+$ not listed in Fig. 4.4(b) is set to *false* by the corresponding assignment.



(a) B

$\beta_B(q1)$:
?a ↦ *true*
?b ↦ *true*

$\beta_B(q2)$:
!c ↦ *true*
!d ↦ *true*

$\beta_B(q3)$:
final ↦ *true*

(b) β_B

Figure 4.4: The service automaton B and its assignment β_B. A literal $x \in \mathcal{MC}^+$ not listed in Fig. 4.4(b) is set to *false* by the corresponding assignment.

The intuitively formulated characterization of a set of services by a single *BSA* is now formalized by the following matching definition. The matching of a service automaton C with a *BSA* B^ϕ requires C to be simulated by B and that C satisfies all annotations of states of B that are used in the simulation relation.

Definition 4.2.5 (Matching, *Match*(B^ϕ)).
Let C be a service automaton and let B^ϕ be a *BSA* such that C and B are interface equivalent.

Then, C *matches* with B^ϕ if there exists a strong simulation relation $\varrho \subseteq Q_C \times Q_B$ such that

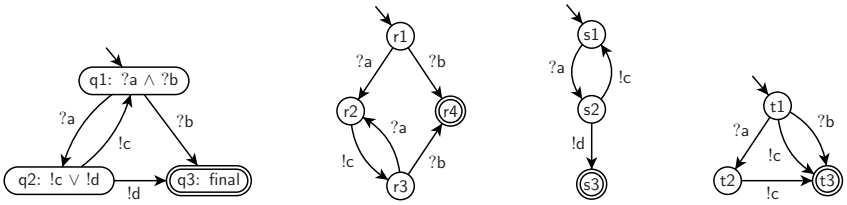
- B simulates C (i.e. $(q_{0C}, q_{0B}) \in \varrho$) and
- for all $(q_C, q_B) \in \varrho$: $\beta_C(q_C) \models \phi(q_B)$.

Let $Match(B^\phi)$ denote the set of all service automata that match with B^ϕ. ⌟

The matching provides a mechanism for characterizing a set of service automata by a *BSA*. Given a *BSA* B^ϕ, we have canonically given the corresponding set $Match(B^\phi)$ of characterized services. Therefore, the set $Match(B^\phi)$ can be seen as the *semantics* of a *BSA* B^ϕ. More precisely, *BSA*s can be seen as syntax, sets of service automata can be seen as the semantical model, and the matching mechanism is the semantics, i.e. the mapping of *BSA*s into the semantical model.

To demonstrate the matching, reconsider the example *BSA* B^ϕ of Fig. 4.3(a) in Fig. 4.5(a) and the service automata C, D, and E in Figs. 4.5(b)–4.5(d). The service automaton C matches with B^ϕ. The simulation relation between C and the underlying service automaton B of B^ϕ contains the elements $(\mathsf{r1}, \mathsf{q1})$, $(\mathsf{r2}, \mathsf{q2})$, $(\mathsf{r3}, \mathsf{q1})$, and $(\mathsf{r4}, \mathsf{q3})$. Furthermore, each annotation of a state q of B^ϕ is satisfied by a related state r of C. For instance, the assignment $\beta_\mathsf{C}(\mathsf{r1})$ assigns *true* to the literals ?a and ?b (because both transitions are present at the state r1), satisfying the annotation $?\mathsf{a} \wedge ?\mathsf{b}$, i.e. $\beta_\mathsf{C}(\mathsf{r1}) \models ?\mathsf{a} \wedge ?\mathsf{b}$.

However, D and E do not match with B^ϕ: the state s1 of D does not satisfy the annotation of state q1 of B^ϕ; and the !c-labeled transition present at state t1 of E causes B to not simulate E.



(a) B^ϕ (b) C (c) D (d) E

Figure 4.5: (a) A *BSA* B^ϕ. The annotation $\phi(q)$ is depicted inside the state q. (b)–(d) Three service automata C, D, and E. C matches with B^ϕ, but D and E do not match with B^ϕ.

Let B^ϕ be an arbitrary *BSA*. Then, from interface equivalence of B and each $C \in Match(B^\phi)$ (as required by Definition 4.2.5) immediately follows that all elements of $Match(B^\phi)$ are (pairwise) interface equivalent.

To consider the matching in Fig. 4.5, let this property also hold for the service automata B, C, D, and E of Fig. 4.5. That is, let !d be in the interface of C, for instance, although there is no !d-labeled transition in C.

4.2.2 Properties of Matching

In the following, we want to convey a better feeling for *BSA*s and the matching procedure. We will consider the matching of non-deterministic service automata

with *BSA*s and will have a closer look on the special literal *final*. Afterwards, we will consider *BSA*s where *no* service automaton can match with and *BSA*s with *infinite* semantics.

In the upcoming Sect. 4.3, we will then characterize a class of *BSA*s called *normal BSA*s, for which there exist efficient algorithms, e.g. to compare the corresponding semantics.

Matching for Non-Deterministic Service Automata

Whereas the definition of a *BSA* B^ϕ requires its underlying service automaton B to be deterministic, the definition of the matching between a service automaton C with B^ϕ does not assume determinism of C. That is, we allow the matching of both deterministic and non-deterministic service automata C with a *BSA* B^ϕ and thus, the semantics of a *BSA* comprises deterministic and non-deterministic service automata.

In the matching of a non-deterministic service automaton C with a *BSA* B^ϕ, the assignment of C at a state q_C of C sets the special literal $\tau \in \mathcal{MC}^+$ to *true* if and only if q_C has a present internal transition in C. That way, an annotation $\phi(q_B)$ of a state q_B of B with $\tau \in \phi(q_B)$ may be satisfied due to this internal transition of C. B^ϕ may also strongly simulate such a service automaton C as B is allowed to have one present internal transition per state.

Let $(q_C, q_B) \in \varrho$. If C has two present x-labeled transitions (q_C, x, q'_C) and (q_C, x, q''_C) at the state q_C, then both states q'_C and q''_C must be in relation with the unique successor q'_B that is reached following the x-labeled transition from q_B in B (if there is one). Then, according to Definition 4.2.5, both q'_C and q''_C must satisfy the annotation $\phi(q'_B)$ of q'_B in B^ϕ.

Hence, it is sufficient to consider *BSA*s with deterministic structure only without restricting generality.

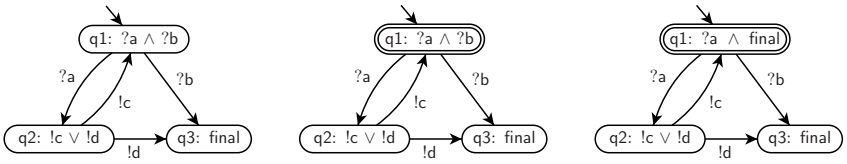
The Relation between Final States and the Literal *final*

In a *BSA*, we distinguish final states and states that are annotated with *final* (i.e. a state q with *final* $\in \phi(q)$). In the example *BSA* B^ϕ of Fig. 4.5(a), these states coincide: the corresponding final state is also annotated with *final*. This, however, is not necessary according to the definition of *BSA*s.

Figure 4.6 shows examples where final states are different from states annotated with *final*. The first two *BSA*s, B'^ϕ of Fig. 4.6(a) and B''^ϕ of Fig. 4.6(b), are derived from B^ϕ of Fig. 4.5(a) by changing its set of final states. However, it is easy to see that the corresponding underlying service automata B, B', and B'' simulate each other and therefore, B^ϕ, B'^ϕ, and B''^ϕ characterize the same set of strategies — disregarding the different final states.

Hence, the set Ω of final states of a *BSA* B^ϕ is completely irrelevant for its semantics $Match(B^\phi)$.

In the third *BSA*, $\mathsf{B}''^{\phi'}$ of Fig. 4.6(c), in contrast, the annotation of the initial state has been changed additionally. The different annotation ϕ' results in a different *Match* set of $\mathsf{B}''^{\phi'}$: for instance, the service automaton C of Fig. 4.5(b) matches with B''^{ϕ}, but does not match with $\mathsf{B}''^{\phi'}$ (because its initial state $\mathsf{r1}$ is no final state and hence violates the annotation $?\mathsf{a} \wedge \mathit{final}$ of the initial state of $\mathsf{B}''^{\phi'}$). Furthermore, it is easy to find a service automaton that matches with $\mathsf{B}''^{\phi'}$, but not with B''^{ϕ} (e.g. the service automaton D which is derived from B'' by removing the ?b-labeled transition).



(a) B'^{ϕ} (b) B''^{ϕ} (c) $\mathsf{B}''^{\phi'}$

Figure 4.6: Variants of the *BSA* B^ϕ of Fig. 4.5(a).

However, we often prefer a *BSA* B^ϕ where its underlying service automaton B itself matches with B^ϕ. Thereby, the set of final states of B plays an important role as it determines the assignment of the *final* literal in the assignment β_B of the underlying service automaton B of B^ϕ. So in most example *BSA*s B^ϕ in the following, a state q of B^ϕ with $\mathit{final} \in \phi(q)$ will also be a final state of B^ϕ.

Empty and Infinite *BSA* Semantics

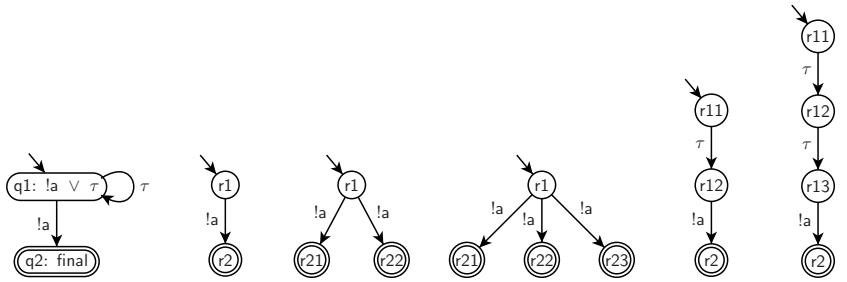
A special case of Boolean annotated service automata is a *BSA* B^ϕ that *no* service automaton can match with. Such a *BSA* is called *empty BSA*.

Definition 4.2.6 (Empty *BSA*).
A *BSA* B^ϕ with $Match(B^\phi) = \emptyset$ is an *empty BSA*. ⌟

An example for an empty *BSA* is a *BSA* with only one state which is annotated with *false*, e.g. the *BSA* $B^\phi = [\{q_0\}, I_{in}, I_{out}, \delta = \emptyset, q_0, \Omega = \{q_0\}, \phi : q_0 \mapsto \mathit{false}]$ for some arbitrary sets of input and output channels I_{in} and I_{out}.

On the other hand it is easy to see that if a non-empty *BSA* B^ϕ characterizes a service automaton C with at least one transition, then B^ϕ characterizes infinitely many other service automata as well. The main reason for this is the potential non-deterministic nature of C. C is allowed to have arbitrarily many different

successor states that are reached by equally labeled transitions. Hence, B^ϕ has infinite semantics. If B^ϕ is cyclic, another dimension for infinitely many matching service automata is established by unrolling or repeating the cycle in the matching service automata.



(a) F^ϕ (b) G_1 (c) G_2 (d) G_3 (e) G^1 (f) G^2

Figure 4.7: A *BSA* F^ϕ with infinite semantics $\mathit{Match}(\mathsf{F}^\phi)$: the service automata G_i for $i \in \mathbb{N}$ have i different $!\mathsf{a}$-successors. The service automata G^j for $j \in \mathbb{N}$ perform j internal transitions before sending an a.

An example is depicted in Fig. 4.7. It illustrates two different dimensions to achieve infinitely many matching service automata: (1) arbitrarily many different states reached by equally labeled transitions (as seen in Figs. 4.7(b)–4.7(d)) and (2) n-fold unrolling of a loop in the *BSA* (illustrated in Figs. 4.7(e) and 4.7(f)). The second dimension is independent of the label of the unrolled transition, i.e. internal as well as interface transitions can be unrolled.

Remark 4.2.7.
In case a non-empty *BSA* B^ϕ has a single state only (i.e. $Q_B = \{q_0\}$) and no transition, it technically still has an infinite semantics as a matching service automaton C is allowed to have an unbounded, even infinite, number of states which are unreachable from C's initial state. However, we do not consider unconnected services. ⌟

In either case, the matching procedure gives us a means to decide for a given *BSA* B^ϕ and a given service automaton C whether or not C is characterized by B^ϕ.

4.3 Normal Boolean Annotated Service Automata

In the previous section, we introduced *BSA*s and a matching procedure to decide whether or not a service automaton C is characterized by a *BSA* B^ϕ. The matching requires C to be simulated by B and that C satisfies all annotations of states of B that are used in the simulation relation.

In this section, we show that these two requirements can restrict or even exclude each other. In such a case, the *BSA* B^ϕ is inflated by "garbage" that adds no information. Therefore, we develop the notion of a *normal BSA*. A normal *BSA* has no such "garbage" and thus provides a sound and intuitive basis for characterizing services. Furthermore, given two normal *BSA*s B_1^ϕ and B_2^ψ, it is possible to decide whether or not B_1^ϕ comprises the semantics of B_2^ψ on the structures of B_1^ϕ and B_2^ψ only.

To decide this property also for non-normal *BSA*s, a *normalization procedure* is needed that transforms an arbitrary *BSA* into a normal one. We will develop such a normalization in the following and prove that the normalization of a *BSA* preserves its semantics.

We start by formalizing the above-mentioned interplay of the simulation relation and the annotations. Then, we introduce the normalization of annotations, followed by the normalization of states of a *BSA*. Finally, both normalizations are combined to derive normal *BSA*s.

4.3.1 The Interplay of the Matching Conditions

Intuitively, the two conditions of the matching definition, i.e. the simulation relation and the annotation satisfaction, provide opposite directives for matching services. Whereas the simulation relation allows a service automaton C to have *at most* the transitions which are present at the corresponding state q of the *BSA*, the annotation of q constitutes a requirement of *minimal* present transitions in C. Hence, as a rule of thumb, the more transitions are present at q in the *BSA*, the more services match with it; but the fewer satisfying assignments $\phi(q)$ has, the fewer services match with the *BSA*. However, both requirements — present transitions and annotations — clearly interact each other.

Basically, this interplay of present transitions and annotations in a *BSA* can have the following shapes:

1. The annotation of a state q considers exactly these literals $x \in \mathcal{MC} \cup \{\tau\}$ for which an x-labeled transition is present at q.
2. There is an x-labeled transition present at q, but x does not occur in the annotation of q.
3. The annotation considers some literal $x \in \mathcal{MC} \cup \{\tau\}$, but there is no present x-labeled transition at q.

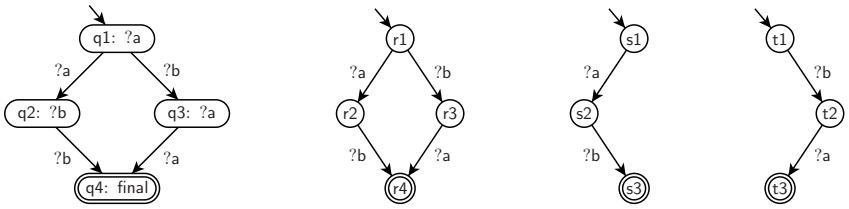
In fact, all depicted *BSA*s so far correspond to the first shape of a *BSA*.

Figure 4.8(a) shows an example of a *BSA* C^ϕ corresponding to the second shape: there is a present ?b-labeled transition at the state q1 of C^ϕ but the annotation

$\phi(\mathsf{q1}) = ?\mathsf{a}$ of this state does not use the corresponding literal ?b. The two service automata D and D' of Figs. 4.8(b) and 4.8(c), respectively, match with C^{ϕ}, but the service automaton D'' of Fig. 4.8(d) does not match with C^{ϕ}, because the state $\mathsf{t1}$ of D'' violates the annotation $\phi(\mathsf{q1}) = ?\mathsf{a}$.

If we changed the annotation $\phi(\mathsf{q1}) = ?\mathsf{a}$ into $\phi'(\mathsf{q1}) = ?\mathsf{a} \vee ?\mathsf{b}$, $\mathsf{C}^{\phi'}$ would correspond to the first shape now. However, this change influences the semantics of C^{ϕ}: the service automaton D'' now matches with $\mathsf{C}^{\phi'}$. Changing the formula into $\phi''(\mathsf{q1}) = ?\mathsf{a} \wedge ?\mathsf{b}$ yields another different semantics: only the service automaton D matches with this *BSA* $\mathsf{C}^{\phi''}$.

Thus we conclude that the first two shapes are truly different, but both shapes can be used to express useful requirements, i.e. they represent the "normal" case without "garbage".



(a) C^{ϕ} (b) D (c) D' (d) D''

Figure 4.8: (a) A *BSA* C^{ϕ}. (b)–(d) Three service automata D, D', and D''. D and D' match with C^{ϕ}, but D'' does not match with C^{ϕ}.

The third shape, however, causes a *BSA* to contain unnecessary redundancies.

Figure 4.9 shows a fragment of a *BSA* E^{ϕ} with a state q that corresponds to the third shape: the annotation $\phi(\mathsf{q}) = \mathsf{b} \vee \mathsf{c}$ references two non-existent transitions (i.e. transitions labeled with b and c, respectively). In fact, E^{ϕ} corresponds to the second shape, too: there is a present a-labeled transition at q, but there is no literal $\mathsf{a} \in \phi(\mathsf{q})$. Obviously, there is no matching service automaton C that can use this state q because it is not possible to (1) neither have a b-labeled nor a c-labeled transition (as required by the simulation relation) and (2) still satisfy $\phi(\mathsf{q}) = \mathsf{b} \vee \mathsf{c}$. Thus, there must not be a state q_C of C for which $(q_C, \mathsf{q}) \in \varrho$ as otherwise q_C inevitably violates at least one of the matching requirements and therefore C does not match with E^{ϕ}. This, again, means that q indeed is "garbage" in E^{ϕ}.

The rest of the section is devoted to remove such redundancies. We first formalize the connection of the two matching requirements and then introduce a normalization that removes those redundancies.

To this end, we basically check whether "B^{ϕ} satisfies its own annotations". There-

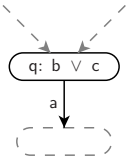


Figure 4.9: Fragment of a *BSA* E^{ϕ} showing a state q with totally unrelated annotation and present transitions. This state is "garbage" in the *BSA* and should be removed.

fore, we apply the matching procedure and match the underlying service automaton B of B^{ϕ} with the *BSA* B^{ϕ} itself. That is, we employ the assignment $\beta_B(q)$ of each state q of B^{ϕ} and check $\beta_B(q) \models \phi(q)$. Thereby, we collect which literals of $\phi(q)$ are set to *false* by $\beta_B(q)$ and for which states q the assignment $\beta_B(q)$ violates $\phi(q)$. Such literals or states are redundant and can be removed from B^{ϕ}.

Taking the original matching algorithm, however, we would remove too many literals or states. The main reason is that a state q_C of C which is simulated by a state q_B of B can set more literals to *true* than q_B does by itself. In fact, *one* literal has to be treated differently: the literal *final*.

4.3.2 The Maximal Assignment

We start with the observation that a service automaton C matching with a *BSA* B^{ϕ} can assign *true* only to a limited subset of literals of a formula $\phi(q)$ of a state q of B^{ϕ}. This subset is determined by the structure of B^{ϕ} itself. The literal *final*, however, can be set to *true* by C independently of q.

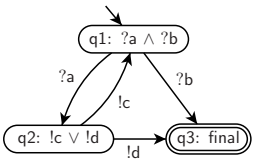
Definition 4.3.1 (Maximal assignment, β_B^+).
Let B^{ϕ} be a *BSA*, let $q \in Q$ be a state of B^{ϕ}, and let $\beta_B(q)$ be the assignment of B at state q.

Then, the *maximal assignment* $\beta_B^+(q)$ at q is defined as $\beta_B^+(q)(x) = \mathit{true}$ if $x = \mathit{final}$, and $\beta_B^+(q)(x) = \beta_B(q)(x)$, otherwise.

The *maximal assignment of* B is the set $\beta_B^+ = \{\beta_B^+(q) \mid q \in Q\}$. ⌟

Given a *BSA* B^{ϕ} with state q and a matching service automaton C with state q_C such that $(q_C, q) \in \varrho$, the maximal assignment $\beta_B^+(q)$ at q takes into account which literals the assignment $\beta_C(q_C)$ can at most set to *true*. As q_C must be simulated by q, q_C can have at most the transitions which are present at q in B. On the other hand, q_C may be a final state of C. As $\mathit{final} \in \phi(q)$ does not necessarily imply that q is a final state of B^{ϕ}, $\beta_C(q_C)$ can set the literal *final* to *true* whereas $\beta_B(q)$ sets *final* to *false*. Hence, the literal *final* is always set to *true* by $\beta_B^+(q)$.

As an example, we recall the *BSA* B^ϕ of Fig. 4.3(a) and Fig. 4.5(a) in Fig. 4.10 and show the maximal assignments of B^ϕ at its states q1 and q3 in Fig. 4.10(c). Whereas $\beta^+_\mathsf{B}(\mathsf{q1})$ changes the value of the *final* literal compared to $\beta_\mathsf{B}(\mathsf{q1})$, $\beta^+_\mathsf{B}(\mathsf{q3})$ is equal to $\beta_\mathsf{B}(\mathsf{q3})$ because q3 is already a final state of B^ϕ.



(a) B^ϕ

$\beta_\mathsf{B}(\mathsf{q1})$:
?a $\mapsto$ *true*
?b $\mapsto$ *true*

$\beta_\mathsf{B}(\mathsf{q3})$:
final $\mapsto$ *true*

(b) β_B

$\beta^+_\mathsf{B}(\mathsf{q1})$:
?a $\mapsto$ *true*
?b $\mapsto$ *true*
final $\mapsto$ *true*

$\beta^+_\mathsf{B}(\mathsf{q3})$:
final $\mapsto$ *true*

(c) β^+_B

Figure 4.10: (a) and (b): The *BSA* B^ϕ of Fig. 4.3(a) with its assignment β_B of Fig. 4.4(b) recalled. (c): The maximal assignments of B^ϕ at states q1 and q3. Again, a literal $x \in \mathcal{MC}^+$ not listed is set to *false* by the corresponding assignment.

As another example, consider again the state q of the fragment of the *BSA* E^ϕ depicted in Fig. 4.9. The corresponding maximal assignment $\beta^+_\mathsf{E}(\mathsf{q})$ sets the literals final and a to *true* only — all other literals, e.g. b and c, are set to *false*.

The following lemma states that at most literals x with $\beta^+_B(q_B)(x) = \mathit{true}$ can be set to *true* by an assignment β_C of a matching service automaton C, i.e. $\beta^+_B(q_B)$ dominates $\beta_C(q_C)$ for all $C \in \mathit{Match}(B^\phi)$ and all $(q_C, q_B) \in \varrho$. In this sense, $\beta^+_B(q_B)$ constitutes a maximum with respect to the assignments $\beta_C(q_C)$ that an arbitrary matching C can provide to evaluate $\phi(q_B)$.

Lemma 4.3.2 (β^+_B dominates all assignments of matching C's).
Let B^ϕ be a *BSA*, let $C \in \mathit{Match}(B^\phi)$ be a service automaton, and let ϱ be the simulation relation between C and B.

Then, for each $(q_C, q_B) \in \varrho$: $\beta_C(q_C) \leq \beta^+_B(q_B)$. ⌟

Proof.
Let $(q_C, q_B) \in \varrho$. By matching we know that q_B simulates q_C. Thus, for each present x-labeled transition q_C in C there is a present x-labeled transition at q_B in B. Therefore, $\beta_C(q_C)(x) = \mathit{true}$ implies $\beta_B(q_B)(x) = \mathit{true}$, for all $x \in \mathcal{MC} \cup \{\tau\}$ and therefore, $\beta_C(q_C)(x) = \mathit{true}$ implies $\beta^+_B(q_B)(x) = \mathit{true}$, for all $x \in \mathcal{MC}^+$. □

With Lemma 4.3.2 and Corollary 4.1.10 (on page 100) we immediately get that if $\phi(q_B)$ evaluates to *false* under $\beta^+_B(q_B)$, then $\phi(q_B)$ evaluates to *false* under any $\beta_C(q_C)$. This means that no matching service automaton can satisfy $\phi(q_B)$. Therefore, we will apply the maximal assignment $\beta^+_B(q_B)$ to normalize the annotation $\phi(q_B)$ of a state q_B (in Sect. 4.3.3) as well as to decide whether q_B can

be completely removed from B^ϕ (in Sect. 4.3.4) without changing the semantics of B^ϕ. Finally, the annotation normalization and the state normalization are combined to normalize *BSA*s in Sect. 4.3.5.

4.3.3 Annotation Normalization of a Boolean Annotated Service Automaton

Lemma 4.3.2 justifies the following notion of normal annotations. If $\phi(q)$ is a normal annotation, then each literal $x \in \phi(q)$ can be set to *true* by a matching service automaton C (if q occurs in the respective simulation relation ϱ).

Definition 4.3.3 (Normal annotation).
Let B^ϕ be a *BSA* and let q be a state of B^ϕ with annotation $\phi(q)$.

The Boolean formula $\phi(q)$ is *normal* at q if there is no $x \in \phi(q)$ with $\beta_B^+(q)(x) = \mathit{false}$.

B^ϕ has *normal annotations* if, for each state q of B^ϕ: $\phi(q)$ is normal at q. ⌟

Informally, the definition of normal annotations excludes the third shape of annotations as introduced in Sect. 4.3.1 on page 108.

As an example, Figs. 4.11(a) and 4.11(c) show two service automata, F^ϕ and F^ψ, with non-normal annotations. In both cases, a literal $\mathsf{!b}$ occurs in the annotation of the respective state $\mathsf{q1}$, but $\beta_{\mathsf{F}}^+(\mathsf{q1})(\mathsf{!b})$ is *false*. Figures 4.11(b) and 4.11(d) show two service automata with normal annotations.

Note that the special Boolean formulae *true* and *false* are trivially normal at any state q as there is no $x \in \mathit{true}$ or $x \in \mathit{false}$ that might violate the criterion of Definition 4.3.3.

We now introduce a normalization construction that transforms a *BSA* B^ϕ with arbitrary annotations into a *BSA* $\mathit{normal}_\phi(B^\phi)$ with only normal annotations. Thereby, literals violating the criterion for a normal annotation (i.e. all literals $x \in \mathcal{MC}^+$ with $\beta_B^+(q)(x) = \mathit{false}$) are removed from $\phi(q)$. The subsequent Lemma 4.3.5 states that the semantics of the *BSA* is not changed during this transformation.

Definition 4.3.4 (Annotation normalization, $\mathit{normal}_\phi(B^\phi)$).
The *normalization* of an annotation $\phi(q)$ of a state q of a *BSA* B^ϕ, $\mathit{normal}(\phi(q))$, is defined as the Boolean formula which is derived from $\phi(q)$ by replacing each occurrence of each literal $x \in \phi(q)$ where $\beta_B^+(q)(x) = \mathit{false}$ by the Boolean formula *false*.

The *annotation normalization* of a *BSA* B^ϕ, $\mathit{normal}_\phi(B^\phi)$, is defined as the *BSA* $\mathit{normal}_\phi(B^\phi) = B^{\phi'}$ where $\phi'(q) = \mathit{normal}(\phi(q))$ for each state $q \in Q$. ⌟

Obviously, the *BSA* $normal_\phi(B^\phi)$ has normal annotations.

For further optimization it seems reasonable to apply reduction rules on the formulae as described in Remark 4.1.12 (on page 101).

The normalization of annotations is demonstrated in Fig. 4.11. The annotation of the initial state of F^ϕ of Fig. 4.11(a), $\phi(\mathsf{q1}) = \mathsf{!a} \vee \mathsf{!b}$, is normalized to $normal(\phi(\mathsf{q1})) = \mathsf{!a}$ in Fig. 4.11(b) because $\beta^+_{\mathsf{F}}(\mathsf{q1})(\mathsf{!b}) = \mathit{false}$ and $\mathsf{!a} \vee \mathit{false}$ is equivalent to $\mathsf{!a}$. As another example, the annotation of the initial state of F^ψ of Fig. 4.11(c), $\psi(\mathsf{q1}) = \mathsf{!a} \wedge \mathsf{!b}$, is normalized to $normal(\psi(\mathsf{q1})) = \mathit{false}$ in Fig. 4.11(d) because $\mathsf{!a} \wedge \mathit{false}$ is equivalent to false.

(a) F^ϕ (b) $normal_\phi(\mathsf{F}^\phi)$ (c) F^ψ (d) $normal_\psi(\mathsf{F}^\psi)$

Figure 4.11: Two *BSA*s F^ϕ and F^ψ with non-normal annotations and their annotation normalized versions $normal_\phi(\mathsf{F}^\phi)$ and $normal_\psi(\mathsf{F}^\psi)$, respectively.

The normalization of annotations according to Definition 4.3.4 normalizes both non-normal as well as already normal annotations. As only literals which cause a formula to be non-normal are replaced, normalizing a normal annotation has no effect, i.e. $normal(\phi(q)) = \phi(q)$ for normal annotations $\phi(q)$. This specifically holds for the special Boolean formulae *true* and *false*, i.e. $normal(\mathit{true}) = \mathit{true}$ and $normal(\mathit{false}) = \mathit{false}$.

The annotation normalization can be applied both to empty and non-empty *BSA*s. If B^ϕ is an empty *BSA*, then it will always have a state q with $\phi(q) = \mathit{false}$, but q is not necessarily the initial state. The *BSA* $normal_\psi(\mathsf{F}^\psi)$ of Fig. 4.11(d) is an empty *BSA* with normal annotations.

We now show that the normalization of the annotations of a *BSA* B^ϕ preserves the semantics of B^ϕ.

Lemma 4.3.5 (Existence of *BSA* with normal annotations).
For each *BSA* B^ϕ there exists a *BSA* $B^{\phi'}$ with normal annotations and the same semantics as B^ϕ, i.e. $Match(B^\phi) = Match(B^{\phi'})$. ⌟

Proof.
Let $B^{\phi'} = normal_\phi(B^\phi)$. We show that $Match(B^\phi) = Match(normal_\phi(B^\phi))$. Therefore, let C be an arbitrary service automaton that is simulated by B and let ϱ be the simulation relation. It suffices to show that, for each $(q_C, q_B) \in \varrho$: $\beta_C(q_C) \models \phi(q_B)$ iff $\beta_C(q_C) \models normal(\phi(q_B))$.

By Lemma 4.3.2 we know that $\beta_C(q_C)(x) = \mathit{false}$ for each literal x of $\phi(q_B)$ that was replaced by *false* in $\mathit{normal}_\phi(B^\phi)$. As, for any Boolean formula ψ, $(\psi \wedge \mathit{false}) \equiv (\psi \wedge x)$ if $\beta_C(q_C)(x) = \mathit{false}$; and $(\psi \vee \mathit{false}) \equiv (\psi \vee x)$ if $\beta_C(q_C)(x) = \mathit{false}$, we conclude that $\beta_C(q_C) \models \phi(q_B)$ iff $\beta_C(q_C) \models \mathit{normal}(\phi(q_B))$. □

By applying the annotation normalization of Definition 4.3.4, we now have *BSA*s with "garbage-free" Boolean annotations. However, this is only the first step towards normal *BSA*s. The next section considers states that can be completely removed from a *BSA* without changing its semantics. This results in a *state normal BSA*. The state normalization is independent of the annotation normalization; that is, it can be applied to a *BSA* with non-normal annotations as well.

4.3.4 State Normalization of a Boolean Annotated Service Automaton

Consider again the (non-initial) state q of the *BSA* E^ϕ of Fig. 4.9: the a-labeled transition is the only present transition at q, but the annotation of q is $\phi(\mathsf{q}) = \mathsf{b} \vee \mathsf{c}$. The normalized annotation of q is $\mathit{normal}(\phi(\mathsf{q})) = \mathit{false} \vee \mathit{false}$ which is equivalent to *false*. It is easy to see that no matching service automaton C can "use" q during the matching (i.e. there is no state q_C of C with $(q_C, \mathsf{q}) \in \varrho$) because no assignment of C can satisfy $\mathit{normal}(\phi(\mathsf{q}))$. Therefore, q is a garbage state and should be removed from E^ϕ.

Intuitively, a state q of a *BSA* is normal, if the (sub-)*BSA* with q as its initial state is a non-empty *BSA*, i.e. it is possible for some service automaton to match with the *BSA* starting at q. Otherwise, q is non-normal in B^ϕ. This characterization of normal states, however, is a semantic one. Using the maximal assignment β^+ instead, we have a criterion which can be checked locally for each state and on the structure of the *BSA* only.

Definition 4.3.6 (Normal state).
A state $q \neq q_0$ of a *BSA* B^ϕ is *normal* in B^ϕ if $\beta_B^+(q) \models \phi(q)$.

The initial state q_0 of B^ϕ is *normal* in B^ϕ if $\beta_B^+(q_0) \models \phi(q_0)$ or if q_0 is the only state of B^ϕ and B^ϕ has no transitions (i.e. $Q_B = \{q_0\}$ and $\delta_B = \emptyset$).

B^ϕ is a *normal state BSA* if each state q of B^ϕ is normal in B^ϕ. ⌟

According to this definition, a normal state $q \neq q_0$ satisfies its own annotation (up to *final*) and therefore enables a service automaton to match with q. The initial state q_0 is normal if it satisfies its own annotation (up to *final*), too. In case q_0 *violates* its annotation, it is still normal if q_0 is the only state (without any transitions) — then, this single state *BSA* B^ϕ is an empty *BSA* and is normal as well.

A non-normal state provides no information because no service automaton C can distinguish a present x-labeled transition leading to a non-normal state in a *BSA* B^ϕ from a non-present x-labeled transition in B^ϕ. In both cases, C is not allowed to have an x-labeled transition itself at its corresponding state.

Recall the four *BSA*s of Fig. 4.11. All states of the first three *BSA*s, i.e. F^ϕ, $normal_\phi(\mathsf{F}^\phi)$, and F^ψ, are normal in the corresponding *BSA* because each state satisfies its own annotation. In contrast, the initial state $\mathsf{q1}$ of the empty *BSA* $normal_\psi(\mathsf{F}^\psi)$ of Fig. 4.11(d) is *non*-normal because it violates its annotation but is not the only state of $normal_\psi(\mathsf{F}^\psi)$. If, however, the state $\mathsf{q2}$ was removed from $normal_\psi(\mathsf{F}^\psi)$, the resulting *BSA* would still be an empty *BSA*, but the state $\mathsf{q1}$ would now be normal in the new *BSA*. Another example of a non-normal state is the (non-initial) state q of the *BSA* E^ϕ of Fig. 4.9.

The next corollary shows that each state which is used by the matching must be normal. This means that a non-normal state of a *BSA* can never be used by a matching service automaton C.

Corollary 4.3.7 (Normal states sufficient for matching).
Let B^ϕ be a non-empty *BSA*, let C be a service automaton with $C \in Match(B^\phi)$, and let $\varrho \subseteq Q_C \times Q_B$ be the minimal simulation relation, such that B simulates C.

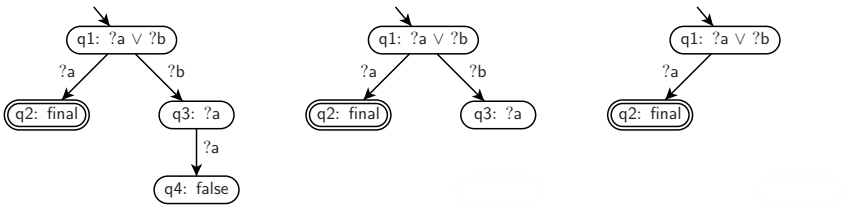
Then, for all $(q_C, q_B) \in \varrho$: $\beta_B^+(q_B) \models \phi(q_B)$. ⌟

Proof.
Let $(q_C, q_B) \in \varrho$ be an arbitrary pair of matching states. By assumption $C \in Match(B^\phi)$, we know $\beta_C(q_C) \models \phi(q_B)$; and by Lemma 4.3.2, we have $\beta_C(q_C) \leq \beta_B^+(q_B)$. Together with Corollary 4.1.10 this yields $\beta_B^+(q_B) \models \phi(q_B)$. □

This corollary motivates the idea to simply remove non-normal states.

Unfortunately, a normal state q ($q \neq q_0$) with a non-normal successor q' can become non-normal itself when removing q' and all transitions (q, x, q'). For instance, consider the *BSA* G^χ of Fig. 4.12(a). The state $\mathsf{q3}$ with annotation $\chi(\mathsf{q3}) = ?\mathsf{a}$ is normal in G^χ, but the state $\mathsf{q4}$ with annotation $\chi(\mathsf{q4}) = \mathsf{false}$ is non-normal in G^χ. If, however $\mathsf{q4}$ is removed (yielding $\mathsf{G}'^{\chi'}$ of Fig. 4.12(b)), the annotation of $\mathsf{q3}$ is no longer satisfied by the assignment $\beta^+_{\mathsf{G}'}(\mathsf{q3})$ (which assigns *true* only to the literal *final*). Hence, $\mathsf{q3}$ is now non-normal in $\mathsf{G}'^{\chi'}$. Applying Corollary 4.3.7 again, we can conclude that this state can be removed as well without changing the set *Match*. This yields the normal state *BSA* $\mathsf{G}''^{\chi''}$ of Fig. 4.12(c).

This step-by-step state removal provides the basis for a state normalization procedure for *BSA*s. As a non-normal initial state may become normal by the removal of its last successor, it is explicitly preserved during the removal steps.



(a) G^{χ} (b) $\mathsf{G}'^{\chi'}$ (c) $\mathsf{G}''^{\chi''}$

Figure 4.12: A *BSA* G^{χ} with non-normal state $\mathsf{q4}$. If this state is removed, then the previously normal state $\mathsf{q3}$ becomes non-normal in $\mathsf{G}'^{\chi'}$ and can be removed as well. $\mathsf{G}''^{\chi''} = normal_Q(\mathsf{G}^{\chi})$ has normal states only and therefore is a normal state *BSA*.

Definition 4.3.8 (State normalization, $normal_Q(B^{\phi})$).
Let $B^{\phi} = [Q, I_{in}, I_{out}, \delta, q_0, \Omega, \phi]$ be a *BSA* and let $(B^{\phi})^i$, $i = 0, 1, \ldots$, be a sequence of *BSA*s $(B^{\phi})^i = [Q^i, I_{in}, I_{out}, \delta^i, q_0, \Omega^i, \phi^i]$ inductively defined as follows:

Basis. $(B^{\phi})^0 = [Q^0, I_{in}, I_{out}, \delta^0, q_0, \Omega^0, \phi^0] = B^{\phi}$,

Step. $(B^{\phi})^{i+1} = [Q^{i+1}, I_{in}, I_{out}, \delta^{i+1}, q_0, \Omega^{i+1}, \phi^{i+1}]$ with

- $Q^{i+1} = \{q_0\} \cup \{q \in Q^i \mid q \text{ is normal and } \delta^i\text{-reachable in } (B^{\phi})^i\}$,
- $\delta^{i+1} = \{(q, x, q') \in \delta^i \mid q, q' \text{ are normal and } \delta^i\text{-reachable in } (B^{\phi})^i\}$,
- $\Omega^{i+1} = \Omega^i \cap Q^{i+1}$, and
- $\phi^{i+1} = \phi_{|Q^{i+1}}$.

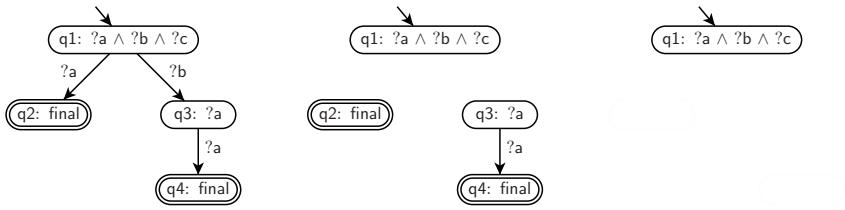
Then, the *state normalization* of B^{ϕ} is the *BSA* $normal_Q(B^{\phi}) = (B^{\phi})^i$ for the smallest i with $(B^{\phi})^i = (B^{\phi})^{i+1}$. ⌟

This state normalization procedure iteratively removes non-normal states from the intermediate *BSA*s $(B^{\phi})^i$. By removing a non-normal state, a normal state can become non-normal and is removed in the next iteration. Additionally, a state can become unreachable from the initial state. Such states also do not add information and are removed as well. Because a minimal simulation relation does not use such unreachable states (cp. Proposition 4.1.4 on page 98), a service automaton matches with the *BSA* with or without such a state. As the initial state is never removed (due to the first argument of the union construction of Q^{i+1} in Definition 4.3.8), each $(B^{\phi})^i$ contains at least one state, and is therefore a well-defined *BSA*.

In case B^{ϕ} has been a non-empty *BSA*, each state of $normal_Q(B^{\phi})$ satisfies its own annotation (up to *final*), and therefore each state is normal in $normal_Q(B^{\phi})$.

In case B^{ϕ} has been an empty *BSA*, then $normal_Q(B^{\phi})$ is a single state empty *BSA* by construction (see also Fig. 4.13). If an empty *BSA* B^{ϕ} has had more than one state or at least one transition, then the initial state q_0 of some $(B^{\phi})^i$

eventually becomes non-normal. Then, the construction in Definition 4.3.8 assures that q_0 and all its adjacent transitions are removed (construction of δ^{i+1} and second argument of the union construction of Q^{i+1}), but q_0 will be preserved (first argument of the union). This step assures that all potential other states are not reachable from q_0 in $(B^\phi)^{i+1}$ anymore. This, again, results in their removal in the next iteration. That way, $normal_Q(B^\phi)$ is a single state empty *BSA*.



(a) $(\mathsf{H}^\phi)^0$ (b) $(\mathsf{H}^\phi)^1$ (c) $(\mathsf{H}^\phi)^2 = (\mathsf{H}^\phi)^3$

Figure 4.13: (a) An empty *BSA* $(\mathsf{H}^\phi)^0$ with a non-normal initial state `q1` (due to a missing !c-labeled transition). (b) All present transitions at the non-normal state `q1` are removed in $(\mathsf{H}^\phi)^1$. As all other states are normal in $(\mathsf{H}^\phi)^0$, they are preserved in $(\mathsf{H}^\phi)^1$ (but become unreachable). (c) All unreachable states of $(\mathsf{H}^\phi)^1$ are removed in $(\mathsf{H}^\phi)^2$. The initial state `q1` was non-normal in $(\mathsf{H}^\phi)^1$, but is preserved in $(\mathsf{H}^\phi)^2$. As `q1` is now normal in $(\mathsf{H}^\phi)^2$, nothing is removed from $(\mathsf{H}^\phi)^2$, i.e. $(\mathsf{H}^\phi)^2 = (\mathsf{H}^\phi)^3$, and $(\mathsf{H}^\phi)^2$ is the state normalization of $(\mathsf{H}^\phi)^0$.

Hence, the *BSA* $normal_Q(B^\phi)$ is a normal state *BSA* in either case.

The preservation of the semantics of a *BSA* during the state normalization is formalized in the following lemma.

Lemma 4.3.9 (Existence of normal state *BSA*).
For each *BSA* B^ϕ there exists a normal state *BSA* $B'^{\phi'}$ with the same semantics, i.e. $Match(B^\phi) = Match(B'^{\phi'})$. ⌟

Proof.
Let $B'^{\phi'} = normal_Q(B^\phi)$. We show $Match(B^\phi) = Match(normal_Q(B^\phi))$ by showing both $Match(B^\phi) \subseteq Match(normal_Q(B^\phi))$ and $Match(normal_Q(B^\phi)) \subseteq Match(B^\phi)$.

$Match(B^\phi) \subseteq Match(normal_Q(B^\phi))$:

Let C be an arbitrary service automaton with $C \in Match(B^\phi)$. We show that $C \in Match(normal_Q(B^\phi))$.

To this end, let ϱ be the minimal simulation relation between C and B. By Lemma 4.1.4 and Corollary 4.3.7 we have that for all $(q_C, q_B) \in \varrho$: q_B is δ-reachable in B and $\beta_B^+(q_B) \models \phi(q_B)$. Because the construction $normal_Q(B^\phi)$

only removes states which are not δ-reachable or states with $\beta_B^+(q_B) \not\models \phi(q_B)$, we conclude that for all $(q_C, q_B) \in \varrho$: q_B is a state of $normal_Q(B^\phi)$. Because all transitions between preserved states are preserved, too, this implies that ϱ is a minimal simulation relation between C and $normal_Q(B^\phi)$ as well.

Let q_B be an arbitrary state of $normal_Q(B^\phi)$ with $(q_C, q_B) \in \varrho$. Let q'_B be a successor state of q_B in B^ϕ (i.e. there is some transition $(q_B, x, q'_B) \in \delta_B$). If q'_B is not a state of $normal_Q(B^\phi)$, then, by construction of $normal_Q(B^\phi)$ we know that q'_B violated its annotation, i.e. $\beta_B^+(q'_B) \not\models \phi(q'_B)$, in some iteration. Applying Corollary 4.3.7 we conclude that q'_B was not used by ϱ. Therefore, q_C has no present x-labeled transition itself, and, hence, $\beta_C(q_C)(x)$ is *false*. Because $\beta_C(q_C) \models \phi(q_B)$ in B^ϕ, this implies that $\beta_C(q_C)$ still satisfies $\phi(q_B)$ in $normal_Q(B^\phi)$.

$Match(normal_Q(B^\phi)) \subseteq Match(B^\phi)$: Let C be an arbitrary service automaton with $C \in Match(normal_Q(B^\phi))$. The simulation relation of C with $normal_Q(B^\phi)$ is a simulation relation of C with B^ϕ as well because B has at most more states and transitions than $normal_Q(B^\phi)$. Because the annotations of the original states of $normal_Q(B^\phi)$ are not changed in B^ϕ, annotation satisfaction is also preserved.□

By applying the state normalization of Definition 4.3.8, we now have *BSA*s without "garbage states". Besides the annotation normalization, this is the second step towards normal *BSA*s. Both steps are independent of each other so far and can be applied separately. Now, we are going to combine both notions to get *normal BSA*s.

4.3.5 Normalization of a Boolean Annotated Service Automaton

Finally, we are ready to formalize the intuitively motivated notion of normal *BSA*s: a *BSA* B^ϕ is *normal* if each state and each annotation of B^ϕ is normal.

Definition 4.3.10 (Normal *BSA*).
A *BSA* B^ϕ is *normal* if B^ϕ is a normal state *BSA* and B^ϕ has normal annotations.⌟

Accordingly, the *normalization of a BSA* combines the normalization of annotations and the normalization of states.

Definition 4.3.11 (Normalization of a *BSA*, $normal(B^\phi)$).
The *normalization* of a *BSA* B^ϕ is the *BSA* $normal(B^\phi)$ defined as $normal(B^\phi) = normal_\phi(normal_Q(B^\phi))$. ⌟

Because the removal of a state q from B^ϕ may make the annotation of a predecessor of q non-normal, the state normalization must precede the annotation normalization. As none of the two normalizations changes the semantics of B^ϕ (Lemmas 4.3.5 and 4.3.9) we directly conclude:

Theorem 4.3.12 (Expressiveness of normal *BSA*s).
For each *BSA* B^{ϕ} there exists a normal *BSA* $B'^{\phi'} = normal(B^{\phi})$ with the same semantics, i.e. $Match(B^{\phi}) = Match(B'^{\phi'})$. ⌟

Normal *BSA*s provide a sound and intuitive basis for the characterization of services. There is no "garbage" in a normal *BSA* — all states and all literals in the annotations of a normal *BSA* carry information and can be used by a matching service automaton. Removing a state or a literal from a normal *BSA* results in a change of the semantics of the *BSA*.

In the rest of this chapter, we aim at being as general as possible and will formalize all newly introduced notions for arbitrary, i.e. normal as well as non-normal, *BSA*s (if applicable). Most of the decision procedures (in form of lemmas and theorems), however, will assume normal *BSA*s in the following. Together with Theorem 4.3.12, though, this assumption does not restrict generality as each non-normal *BSA* could be normalized before applying the corresponding decision procedure.

4.4 A Preorder on Boolean Annotated Service Automata

With normal (i.e. garbage-free) Boolean annotated service automata we are now able to compare the semantics, i.e. the sets of matching service automata, of two *BSA*s by looking at the structures of the *BSA*s only. Consequently, normal *BSA*s will enable us to efficiently decide equivalence of *BSA*s (in the sense of equal semantics).

We start by defining a structural smaller relation on *BSA*s. We will then show that this relation imposes an inclusion relation of the corresponding semantics for normal *BSA*s and prove it to be a preorder on normal *BSA*s.

Definition 4.4.1 (Smaller relation, $\sqsubseteq$, on *BSA*s).
Let B_1^{ϕ} and B_2^{ψ} be interface equivalent *BSA*s.

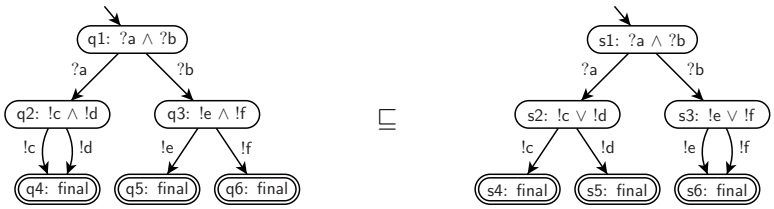
Then, B_1^{ϕ} is *smaller* than B_2^{ψ}, denoted $B_1^{\phi} \sqsubseteq B_2^{\psi}$, if
- there is a simulation relation $\varrho \subseteq Q_{B_1} \times Q_{B_2}$ such that
- for all $(q_{B_1}, q_{B_2}) \in \varrho$: $\phi(q_{B_1}) \Rightarrow \psi(q_{B_2})$. ⌟

According to this definition, B_1^{ϕ} is smaller than B_2^{ψ} if (1) B_2 simulates B_1 and (2) the annotation of a state q_{B_1} of B_1 implies the annotation of each state q_{B_2} of B_2 which simulates q_{B_1}. Note that the implication $\phi \Rightarrow \psi$ of two Boolean formulae ϕ and ψ is no Boolean formula itself, but means that each satisfying assignment β for ϕ is also a satisfying assignment for ψ (cp. Notation 4.1.11).

The basic idea of these two requirements of Definition 4.4.1 is quite similar to the matching definition (Definition 4.2.5 on page 103). As the main difference, the smaller relation requires an implication of the annotations in the second item rather than the formulae satisfaction requirement of the matching definition. In other words, the matching definition requires the annotations of B_2 to be satisfied by *one specific* assignment which is determined by the structure of the matching service automaton C. The smaller relation, instead, requires the annotations of B_2 to be satisfied by *any* assignment that satisfies a corresponding annotation of B_1.

The smaller relation will be applied to compare the semantics of normal *BSA*s. As the underlying service automaton B of a *BSA* B^ϕ as well as all elements of $Match(B^\phi)$ are (pairwise) interface equivalent, it is feasible to define this relation only for interface equivalent *BSA*s.

To exemplify the smaller relation, consider the two *BSA*s B_1^ϕ and B_2^ψ with $\mathsf{B}_1^\phi \sqsubseteq \mathsf{B}_2^\psi$ in Fig. 4.14. The simulation relation is $\varrho = \{(\mathsf{q1}, \mathsf{s1}), (\mathsf{q2}, \mathsf{s2}), (\mathsf{q3}, \mathsf{s3}), (\mathsf{q4}, \mathsf{s4}), (\mathsf{q4}, \mathsf{s5}), (\mathsf{q5}, \mathsf{s6}), (\mathsf{q6}, \mathsf{s6})\}$, and the annotation implication is satisfied for each pair. For instance, $\phi(\mathsf{q2}) = \mathsf{!c} \wedge \mathsf{!d}$ implies $\mathsf{!c} \vee \mathsf{!d} = \psi(\mathsf{s2})$. It is easy to see that the smaller relation does not hold in the other direction, i.e. $\mathsf{B}_2^\psi \not\sqsubseteq \mathsf{B}_1^\phi$. The corresponding simulation relation is $\varrho' = \{(q, q') \mid (q', q) \in \varrho\}$, but obviously $\psi(\mathsf{s2}) = \mathsf{!c} \vee \mathsf{!d}$ does *not* imply $\mathsf{!c} \wedge \mathsf{!d} = \phi(\mathsf{q2})$, which violates the second requirement of Definition 4.4.1.



(a) B_1^ϕ (b) B_2^ψ

Figure 4.14: Two *BSA*s B_1^ϕ and B_2^ψ with $\mathsf{B}_1^\phi \sqsubseteq \mathsf{B}_2^\psi$, but $\mathsf{B}_2^\psi \not\sqsubseteq \mathsf{B}_1^\phi$.

The following theorem justifies the "smaller" relation: it states that a smaller *BSA* has a semantics which is included in the greater *BSA*'s semantics, and if one semantics is included in another one, there is a smaller relation between the corresponding *BSA*s. Because the criteria of the relation $\sqsubseteq$ consider the structure of the compared *BSA*s only, we have to assume that the *BSA*s do not contain garbage, i.e. we assume normal *BSA*s for the following theorem. Together with Theorem 4.3.12, however, this assumption does not restrict generality.

It is easy to see that if B_1^ϕ is a normal, empty *BSA*, then $B_1^\phi \sqsubseteq B_2^\psi$ for each (interface equivalent) *BSA* B_2^ψ.

Theorem 4.4.2 (Smaller relation vs. matching).
Let B_1^ϕ and B_2^ψ be normal, interface equivalent *BSA*s.

Then, $B_1^\phi \sqsubseteq B_2^\psi$ iff $Match(B_1^\phi) \subseteq Match(B_2^\psi)$. ⌟

Proof.
($\Rightarrow$): Let $B_1^\phi \sqsubseteq B_2^\psi$ and let $C \in Match(B_1^\phi)$ be arbitrary. We show that $C \in Match(B_2^\psi)$.

Because of $C \in Match(B_1^\phi)$ and Definition 4.2.5 (Matching) there is a simulation relation $\varrho_{(C,B_1)} \subseteq Q_C \times Q_{B_1}$ such that B_1^ϕ simulates C. Because of $B_1^\phi \sqsubseteq B_2^\psi$ and Definition 4.4.1 (Smaller relation) there is a simulation relation $\varrho_{(B_1,B_2)} \subseteq Q_{B_1} \times Q_{B_2}$ such that B_2 simulates B_1. Define a relation $\varrho_{(C,B_2)}$ between the states of C and B_2^ψ as follows: $\varrho_{(C,B_2)} = \{(q_C, q_{B_2}) \mid (q_C, q_{B_1}) \in \varrho_{(C,B_1)}$ and $(q_{B_1}, q_{B_2}) \in \varrho_{(B_1,B_2)}\}$.

It suffices to show that (1) $\varrho_{(C,B_2)}$ is a simulation relation such that B_2 simulates C and that (2) for each $(q_C, q_{B_2}) \in \varrho_{(C,B_2)} : \beta_C(q_C) \models \psi(q_{B_2})$.

(1): Obviously, the initial state of C is in relation $\varrho_{(C,B_1)}$ with the initial state of B_1 and the initial state of B_1 is in relation $\varrho_{(B_1,B_2)}$ with the initial state of B_2. So, by construction of $\varrho_{(C,B_2)}$, the initial state of C is in relation $\varrho_{(C,B_2)}$ with the initial state of B_2.

Let q_C, q_{B_1}, and q_{B_2} be arbitrary states of C, B_1, and B_2, respectively, with $(q_C, q_{B_1}) \in \varrho_{(C,B_1)}$ and $(q_{B_1}, q_{B_2}) \in \varrho_{(B_1,B_2)}$. Because q_{B_2} simulates q_{B_1} and q_{B_1} simulates q_C, q_{B_2} also simulates q_C.

(2): Let again q_C, q_{B_1}, and q_{B_2} be arbitrary states of C, B_1, and B_2, respectively, with $(q_C, q_{B_1}) \in \varrho_{(C,B_1)}$ and $(q_{B_1}, q_{B_2}) \in \varrho_{(B_1,B_2)}$. Because of $C \in Match(B_1^\phi)$, we have $\beta_C(q_C) \models \phi(q_{B_1})$ and because of $B_1^\phi \sqsubseteq B_2^\psi$, we know $\phi(q_{B_1}) \Rightarrow \psi(q_{B_2})$. Hence, $\beta_C(q_C) \models \psi(q_{B_2})$.

($\Leftarrow$): Let $Match(B_1^\phi) \subseteq Match(B_2^\psi)$. Let C be a service automaton that is isomorphic to B_1, but let each state q of C be a final state in C where $\mathit{final} \in \phi(q)$ in B_1. We prove $B_1^\phi \sqsubseteq B_2^\psi$ by showing (1) there is a simulation relation between B_1 and B_2 and (2) each satisfying assignment of a formula of B_1 satisfies the corresponding formula of B_2.

(1): C being isomorphic to B_1 implies that there is a simulation relation between the states of C and B_1. From the construction of C we have $\beta_C(q) = \beta^+_{B_1}(q)$ for each state q of C and B_1, i.e. C provides the maximal assignment at state q. From B_1^ϕ being normal (i.e. $\beta^+_{B_1}(q) \models \phi(q)$) we further conclude that $\beta_C(q) \models \phi(q)$. Hence, C matches with B_1^ϕ, and hence, by assumption, C matches with B_2^ϕ. By Definition 4.2.5 (Matching), there is a simulation relation between C and B_2. Together with the isomorphism of C and B_1 it follows that there is a simulation relation between B_1 and B_2, too.

(2): Let this simulation relation be ϱ and let $(q_1, q_2) \in \varrho$ be arbitrary. Let furthermore β be an arbitrary assignment with $\beta \models \phi(q_1)$. Construct from C a new service automaton C' by removing from q_1 in C all present transitions (q_1, x, q_1') with $\beta(x) = \mathit{false}$ and let q_1 be a final state of C' iff $\beta(\mathit{final}) = \mathit{true}$. Obviously, $\beta_{C'}(q_1)(x) = \beta(q_1)(x)$, for all $x \in \phi(q)$ and therefore $\beta_{C'}(q_1) \leq \beta(q_2)$.

As only transitions with $\beta(x) = \mathit{false}$ where removed from C, we know with Definition 4.2.5 (Matching) that C' still matches with B_1^ϕ and therefore, by assumption, C' matches with B_2^ψ. Again by Definition 4.2.5, we conclude that $\beta_{C'}(q_1) \models \psi(q_2)$, too, and with $\beta_{C'}(q_1) \leq \beta(q_2)$ we have $\beta(q_2) \models \psi(q_2)$. Hence, all satisfying assignments of $\phi(q_1)$ satisfy $\psi(q_2)$, and therefore $\phi(q_{B_1}) \Rightarrow \psi(q_{B_2})$.□

According to Theorem 4.4.2, checking an inclusion relation between the semantics of two normal *BSA*s B_1^ϕ and B_2^ψ reduces to a check for the existence of a smaller relation between B_1^ϕ and B_2^ψ. This, again, is a mere check for the existence of a simulation relation and all relevant annotation implications. Hence, an inclusion relation between the semantics of two *BSA*s can be decided by a simple comparison of the structures of the *BSA*s.

Applying the theorem to the example of Fig. 4.14 of the two normal *BSA*s B_1^ϕ and B_2^ψ with $\mathsf{B}_1^\phi \sqsubseteq \mathsf{B}_2^\psi$ shows that B_1^ϕ has a semantics that is included in the semantics of B_2^ψ, i.e. $\mathit{Match}(\mathsf{B}_1^\phi) \subseteq \mathit{Match}(\mathsf{B}_2^\psi)$. With $\mathsf{B}_2^\psi \not\sqsubseteq \mathsf{B}_1^\phi$ (and hence $\mathit{Match}(\mathsf{B}_2^\psi) \not\subseteq \mathit{Match}(\mathsf{B}_1^\phi)$) we conclude $\mathit{Match}(\mathsf{B}_1^\phi) \subset \mathit{Match}(\mathsf{B}_2^\psi)$. An example for a service automaton matching with B_2^ψ but not with B_1^ϕ is the service automaton C (not depicted) that is derived from B_2 by removing state $\mathsf{s5}$. C would still satisfy the formula $\mathsf{!c} \vee \mathsf{!d}$ but would violate $\mathsf{!c} \wedge \mathsf{!d}$ due to the missing $\mathsf{!d}$-labeled transition at the state $\mathsf{s2}$.

The assumption of B_1^ϕ and B_2^ψ being normal was only used in the replication direction of the proof of Theorem 4.4.2. Hence, a weaker version of this theorem, considering only an implication, even holds for non-normal *BSA*s, i.e. $B_1^\phi \sqsubseteq B_2^\psi$ implies $\mathit{Match}(B_1^\phi) \subseteq \mathit{Match}(B_2^\psi)$ for arbitrary *BSA*s. Therein, a non-normal state q of the smaller *BSA* can be in relation with a normal or non-normal state q' of the greater *BSA*. As q is never used by a service automaton C matching with the smaller *BSA*, the state q' is also not used if C is matched with the greater *BSA*. For the other replication direction, however, it is easy to see that $\mathit{Match}(B_1^\phi) \subseteq \mathit{Match}(B_2^\psi)$ does not necessarily imply $B_1^\phi \sqsubseteq B_2^\psi$ for arbitrary *BSA*s. A simple example are the *BSA*s G^χ and $\mathsf{G}'^{\chi'}$ of Fig. 4.12 (on page 116). Both *BSA*s have the same semantics, and hence $\mathit{Match}(\mathsf{G}^\chi) \subseteq \mathit{Match}(\mathsf{G}'^{\chi'})$, but G' does not simulate G, and thus $\mathsf{G}^\chi \not\sqsubseteq \mathsf{G}'^{\chi'}$.

We are now ready to show that the smaller relation $\sqsubseteq$ induces a preorder on normal, interface equivalent *BSA*s. Again, the interface equivalence requirement is motivated by the fact that all elements of any *Match* set are (pairwise) interface equivalent themselves.

Lemma 4.4.3 ($\sqsubseteq$ is a preorder relation).
The smaller relation $\sqsubseteq$ is a preorder on normal, interface equivalent *BSA*s:

Reflexivity: $B_1^\phi \sqsubseteq B_1^\phi$.
Transitivity: if $B_1^\phi \sqsubseteq B_2^\psi$ and $B_2^\psi \sqsubseteq B_3^\chi$, then $B_1^\phi \sqsubseteq B_3^\chi$. ⌟

Proof.
Let B_1^ϕ, B_2^ψ, and B_3^χ be normal, interface equivalent *BSA*s and let $\sqsubseteq$ be as defined above.

Reflexivity: From $Match(B_1^\phi) \subseteq Match(B_1^\phi)$ and Theorem 4.4.2 immediately follows $B_1^\phi \sqsubseteq B_1^\phi$.

Transitivity: From $Match(B_1^\phi) \subseteq Match(B_2^\psi)$ and $Match(B_2^\psi) \subseteq Match(B_3^\chi)$ follows $Match(B_1^\phi) \subseteq Match(B_3^\chi)$. Together with Theorem 4.4.2 again, this implies $B_1^\phi \sqsubseteq B_3^\chi$. □

The smaller relation $\sqsubseteq$ is, however, no partial order relation as from $Match(B_1^\phi) = Match(B_2^\psi)$ does not follow $B_1^\phi = B_2^\psi$. As an example, consider again the two *BSA*s B_1^ϕ and B_2^ψ of Fig. 4.14. If the annotations of the states q2 and s2 and the annotations of the states q3 and s3 were equal, both *BSA*s would characterize the same set of service automata. However, B_1 and B_2 are definitely structurally different.

In the next section, we will apply the smaller relation to derive an equivalence notion for *BSA*s and to decide equivalence of *BSA*s.

Furthermore, the preorder relation can be applied to decide *accordance* [ALM+09, ALM+07, SMB09], an important notion in the research area of *substitutability of services*, i.e. the question whether a service S can be substituted by a new service S' under preservation of certain properties. Accordance between S' and S basically demands that the substituting service S' preserves all strategies of the substituted service S. Representing the strategies of S and S' by two *BSA*s B_1^ϕ and B_2^ψ, checking the existence of a smaller relation between B_1^ϕ and B_2^ψ can be used to decide accordance of S' with S.

4.5 The Canonical Representative of a Boolean Annotated Service Automaton

As already mentioned in the last section, there are different *BSA*s that nevertheless have the same semantics, i.e. characterize the same set of services. In this sense, those *BSA*s are *equivalent*. This section is devoted to such equivalent *BSA*s in general. The main result will be the characterization of a canonical representative in each set of equivalent *BSA*s. This representative is minimal with respect to the

number of states. That is, there is no equivalent *BSA* which has a fewer number of states than the representative. Furthermore, we develop a minimization of a *BSA* that transforms an arbitrary normal *BSA* into the respective representative.

The representative of a *BSA* is particularly important as given two representatives, checking semantical equivalence of these *BSA*s reduces to a graph isomorphism problem and a check for annotation equivalence of isomorphic states. These checks are even more efficient than checking for a bisimulation relation between the *BSA*s and checking two formula implications. Additionally, there is the obvious advantage of reduced capacity requirements for storing a representative of a *BSA* instead of the *BSA* itself.

We continue as follows. In Sect. 4.5.1, we derive a notion of equivalence of *BSA*s B_1^ϕ and B_2^ψ which can be decided on the structures of B_1^ϕ and B_2^ψ (if both *BSA*s are normal). To this end, we recall the smaller relation $\sqsubseteq$ and apply it to decide the equivalence of normal *BSA*s. From the equivalence of *BSA*s we then derive an equivalence relation on the states of a single *BSA* in Sect. 4.5.2. With the help of these equivalence relations, we will then be able to present a minimization for normal *BSA*s in Sect. 4.5.3. The minimization basically merges equivalent states. We will show that the minimized *BSA* is minimal in the corresponding set of all equivalent *BSA*s and therefore serves as the representative.

4.5.1 Equivalence of Boolean Annotated Service Automata

Intuitively, two *BSA*s are equivalent if they characterize the same set of service automata, i.e. if they have the same matching sets.

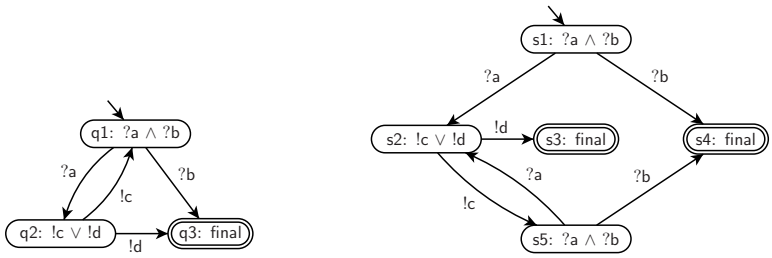
In the following, we will formalize this equivalence notion on *BSA*s. As *Match* can be seen as a function assigning to each *BSA* its set of characterized service automata, this equivalence notion is canonically derived and bears no surprise.

Definition 4.5.1 (Equivalent *BSA*s, $\equiv$).
Two interface equivalent *BSA*s B_1^ϕ and B_2^ψ are (semantically) *equivalent*, denoted $B_1^\phi \equiv B_2^\psi$, if $Match(B_1^\phi) = Match(B_2^\phi)$. ⌟

Figure 4.15 shows two equivalent *BSA*s B_1^ϕ and B_2^ψ. B_1^ϕ is the already known *BSA* from Fig. 4.5(a) (on page 104). It is easy to see that the service automaton C of Fig. 4.5(b), for instance, matches with both B_1^ϕ and B_2^ψ.

In order to decide the semantically defined equivalence of two *BSA*s, we obviously can apply the structurally defined smaller relation $\sqsubseteq$ on *BSA*s which was introduced in the last section. As mentioned above, we have to assume normal *BSA*s for this decision procedure.

Corollary 4.5.2 (Deciding the equivalence of *BSA*s).
Let B_1^ϕ and B_2^ψ be normal, interface equivalent *BSA*s.



(a) B_1^ϕ (b) B_2^ψ

Figure 4.15: Two equivalent *BSA*s B_1^ϕ and B_2^ψ. For instance, the service automaton C of Fig. 4.5(b) matches with both *BSA*s.

Then, $B_1^\phi \equiv B_2^\psi$ iff $B_1^\phi \sqsubseteq B_2^\psi$ and $B_2^\psi \sqsubseteq B_1^\phi$. ⌟

Proof.
The corollary follows immediately from Theorem 4.4.2 and Definition 4.5.1: $B_1^\phi \equiv B_2^\psi$ iff $Match(B_1^\phi) = Match(B_2^\psi)$ iff $Match(B_1^\phi) \subseteq Match(B_2^\psi)$ and $Match(B_2^\psi) \subseteq Match(B_1^\phi)$ iff $B_1^\phi \sqsubseteq B_2^\psi$ and $B_2^\psi \sqsubseteq B_1^\phi$. □

In order to decide equivalence of two *non-normal BSA*s B_1^ϕ and B_2^ψ, one can easily transform both *BSA*s into their corresponding normal forms $normal(B_1^\phi)$ and $normal(B_2^\psi)$ (cp. Definition 4.3.11 on page 118) and then apply Corollary 4.5.2 for $normal(B_1^\phi)$ and $normal(B_2^\psi)$.

As the smaller relation $\sqsubseteq$ is defined on the structure of the compared *BSA*s (cp. Definition 4.4.1), the latter corollary provides a mechanism to check the equivalence notion $\equiv$ of *BSA*s on the structures of the *BSA*s, too. This means we do not have to compare the two infinite sets of matching service automata, but just have to find two simulation relations and check all relevant annotation implications. This check can easily be performed during one depth-first traversal of both *BSA*s.

Our example *BSA*s B_1^ϕ and B_2^ψ of Fig. 4.15 are both normal. Hence, we easily check equivalence of B_1^ϕ and B_2^ψ by checking $\mathsf{B}_1^\phi \sqsubseteq \mathsf{B}_2^\psi$ and $\mathsf{B}_2^\psi \sqsubseteq \mathsf{B}_1^\phi$. The two corresponding simulation relations are $\varrho = \{(\mathsf{q1}, \mathsf{s1}), (\mathsf{q2}, \mathsf{s2}), (\mathsf{q3}, \mathsf{s3}), (\mathsf{q3}, \mathsf{s4}), (\mathsf{q1}, \mathsf{s5}), (\mathsf{q3}, \mathsf{s4})\}$ and $\varrho' = \varrho^{-1}$. As the annotations of each pair $(q, s) \in \varrho$ and $(s, q) \in \varrho'$ are equal, we trivially have both implications. Thus, we conclude $Match(\mathsf{B}_1^\phi) = Match(\mathsf{B}_2^\psi)$ and therefore $\mathsf{B}_1^\phi \equiv \mathsf{B}_2^\psi$.

To justify the "equivalence" notion, we prove the following lemma which states that $\equiv$ is indeed an equivalence relation. It equally holds for normal as well as non-normal *BSA*s.

Lemma 4.5.3 ($\equiv$ is an equivalence relation).
The relation $\equiv$ is an equivalence relation on interface equivalent *BSA*s. ⌟

Proof.
Let B_1^ϕ, B_2^ψ, and B_3^χ be (pairwise) interface equivalent *BSA*s. It suffices to show:
Reflexivity, $B_1^\phi \equiv B_1^\phi$.
Symmetry, $B_1^\phi \equiv B_2^\psi$ iff $B_2^\psi \equiv B_1^\phi$.
Transitivity, if $B_1^\phi \equiv B_2^\psi$ and $B_2^\psi \equiv B_3^\chi$, then $B_1^\phi \equiv B_3^\chi$.

All three items follow from reflexivity, symmetry, and transitivity of set equality and Definition 4.5.1. □

As usual, the equivalence relation $\equiv$ on *BSA*s is used to derive the canonical equivalence class of a *BSA* B^ϕ.

Definition 4.5.4 (Equivalence class of a *BSA*, $[B^\phi]$).
The *equivalence class* $[B_1^\phi]$ of a *BSA* B_1^ϕ is defined as $[B_1^\phi] = \{B_2^\psi \mid B_2^\psi \equiv B_1^\phi\}$. ⌟

Obviously, the example *BSA*s B_1^ϕ and B_2^ψ of Fig. 4.15 are in the same equivalence class.

The set $[B^\phi]$ represents all *BSA*s which have the same semantics as B^ϕ. As $[B^\phi]$ comprises normal and non-normal *BSA*s, it is an infinite set even for an empty *BSA* B^ϕ. In other words, for each *BSA* there are infinitely many different *BSA*s that have the same semantics. Even when restricting the equivalence class of a *BSA* B^ϕ to normal *BSA*s, there is still a dimension to yield an infinite class: a loop of B^ϕ can be unrolled arbitrarily often.

Therefore, there is the need for a canonical representative of the class $[B^\phi]$.

4.5.2 Equivalence of *BSA* states

In order to present a minimization procedure that transforms an arbitrary *BSA* into the corresponding representative in Sect. 4.5.3, this subsection is devoted to derive an equivalence relation $\simeq$ on the states of a single *BSA* from the equivalence relation $\equiv$ on *BSA*s. Then, the minimization basically merges equivalent states such that the semantics of the *BSA* is not changed.

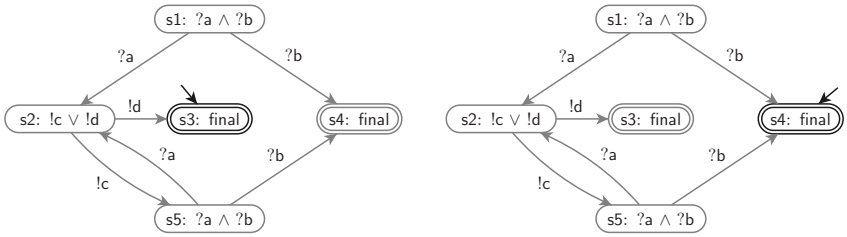
Again, this equivalence notion of states, as well as the notion of an equivalence class of a *BSA* state, follow canonically from the equivalence notion for *BSA*s.

Intuitively, two states q_1 and q_2 are equivalent in a *BSA* B^ϕ if the (sub-)*BSA*s starting at q_1 and q_2, respectively, characterize the same services. Therefore, we first formalize the notion of the q-starting *BSA* of B^ϕ and then employ the equivalence relation $\equiv$ of the q_1-starting and q_2-starting *BSA*s of B^ϕ to characterize the equivalence of the states q_1 and q_2 in B^ϕ.

Definition 4.5.5 (q-starting *BSA* of B^ϕ, B^ϕ_q).
Let $B^\phi = [Q, I_{in}, I_{out}, \delta, q_0, \Omega, \phi]$ be a *BSA* and $q \in Q$ a state of B^ϕ.
Then, the *q-starting BSA* B^ϕ_q of B^ϕ is defined as $B^\phi_q = [Q, I_{in}, I_{out}, \delta, q, \Omega, \phi]$. ⌟

The q-starting *BSA* of B^ϕ is a variant of B^ϕ where the initial state is set to q instead of q_0.

As an example, Fig. 4.16 shows the s3-starting and s4-starting *BSA*s of the *BSA* B^ψ_2 of Fig. 4.15(b). It is easy to see that both *BSA*s characterize the same set of services, i.e. they are equivalent.



(a) s3-starting *BSA* of B^ψ_2 (b) s4-starting *BSA* of B^ψ_2

Figure 4.16: The s3-starting and s4-starting *BSA*s of the *BSA* B^ψ_2 of Fig. 4.15(b). Unreachable states are grayed out. Obviously, both *BSA*s are equivalent.

Now we are ready to define equivalence of states of a *BSA*.

Definition 4.5.6 (Equivalent states of a *BSA*, $\simeq$).
Let B^ϕ be a *BSA*, q_1, q_2 be states of B^ϕ, and $B^\phi_{q_1}$ and $B^\phi_{q_2}$ be the q_1-starting and q_2-starting *BSA*s of B^ϕ, respectively.
Then, q_1 and q_2 are *equivalent* in B^ϕ, denoted $q_1 \simeq_{B^\phi} q_2$, if $B^\phi_{q_1} \equiv B^\phi_{q_2}$. ⌟

We omit the index and write $q_1 \simeq q_2$ if the considered *BSA* is clear from the context.

Corollary 4.5.7 ($\simeq$ is an equivalence relation).
The relation $\simeq_{B^\phi}$ is an equivalence relation on the states of a *BSA* B^ϕ. ⌟

Proof.
Reflexivity, symmetry, and transitivity of $\simeq$ directly follow from reflexivity, symmetry, and transitivity of $\equiv$ and Definition 4.5.6. □

It is easily possible to generalize the state equivalence $\simeq$ of one *BSA* to a state equivalence between states of different *BSA*s. Then, two states of different *BSA*s

are equivalent if their corresponding q-starting *BSA*s are equivalent. However, we will apply the state equivalence only for the minimization of a *BSA* by merging equivalent states of this *BSA*. Hence, we may restrict the state equivalence to the states of one *BSA* only.

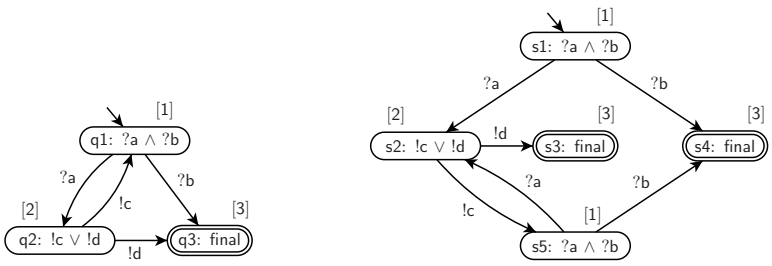
In analogy to the equivalence $\equiv$ on *BSA*s, the equivalence relation $\simeq$ on the states of a *BSA* determines the canonical equivalence class of a state q of a *BSA*.

Definition 4.5.8 (Equivalence class of a *BSA* state, $[q]$).
Let B^ϕ be a *BSA* with set Q of states and let $q \in Q$ be a state of B^ϕ.

Then, the *equivalence class* $[q]$ of q is defined as $[q] = \{q' \mid q' \in Q, q' \simeq q\}$. ⌟

The equivalence $\simeq$ of states is defined for normal and non-normal states of a *BSA* B^ϕ. It is easy to see that any two non-normal states q_1 and q_2 of B^ϕ are equivalent to each other because the corresponding q-starting *BSA*s $B^\phi_{q_1}$ and $B^\phi_{q_2}$ (cp. Definition 4.5.5) both are empty *BSA*s, and therefore $B^\phi_{q_1}$ and $B^\phi_{q_2}$ are equivalent. Hence, all non-normal states of B^ϕ are in one equivalence class.

To illustrate the equivalence of states, Fig. 4.17 recalls the *BSA*s B_1^ϕ and B_2^ψ of Fig. 4.15. In Fig. 4.17, a number is assigned to each state of B_1^ϕ and B_2^ψ. All states of one *BSA* with equal numbers are equivalent in the respective *BSA* and the set of states with equal numbers constitute the equivalence class of these states in the respective *BSA*. In B_1^ϕ, no two states are equivalent to each other, whereas in B_2^ψ, the states s1 and s5 are equivalent, as well as the states s3 and s4.



(a) B_1^ϕ (b) B_2^ψ

Figure 4.17: The two *BSA*s B_1^ϕ and B_2^ψ of Fig. 4.15. The number attached to a state denotes the corresponding equivalence class of the state. That is, states with equal numbers are equivalent in the respective *BSA*. Whereas B_2^ψ has equally numbered, i.e. equivalent, states, no two states of B_1^ϕ are equivalent to each other.

In the following, we will present a minimization procedure for a *BSA* which basically merges equivalent states. Beforehand, we prove two lemmas on properties of

equivalent states of normal *BSA*s that will ease the minimization procedure later on.

First, we show that equivalent (normal) states have equivalent annotations.

Lemma 4.5.9 (Equivalent states have equivalent annotations).
For all states $q_1 \simeq q_2$ of a normal *BSA* B^ϕ: $\phi(q_1) \equiv \phi(q_2)$. ⌟

Proof.
Let $B^\phi_{q_1} = [Q, I_{in}, I_{out}, \delta, q_1, \Omega, \phi]$ and $B^\phi_{q_2} = [Q, I_{in}, I_{out}, \delta, q_2, \Omega, \phi]$ be the respective q-starting *BSA*s of B^ϕ.

We have $q_1 \simeq q_2$ iff $B^\phi_{q_1} \equiv B^\phi_{q_2}$ (by Definition 4.5.6) and hence, $B^\phi_{q_1} \sqsubseteq B^\phi_{q_2}$ and $B^\phi_{q_2} \sqsubseteq B^\phi_{q_1}$ (by Corollary 4.5.2). Thus, we know that $\phi(q_1) \Rightarrow \phi(q_2)$ and $\phi(q_2) \Rightarrow \phi(q_1)$ (by Definition 4.4.1), which yields $\phi(q_1) \equiv \phi(q_2)$. □

Next, we can prove that equivalent states simulate each other.

Lemma 4.5.10 (Equivalent states simulate each other).
For all states $q_1 \simeq q_2$ of a normal *BSA* B^ϕ: q_1 simulates q_2 and q_2 simulates q_1.⌟

Proof.
Let $B^\phi = [Q, I_{in}, I_{out}, \delta, q_0, \Omega, \phi]$ be a normal *BSA*, let $q_1, q_2 \in Q$ be arbitrary with $q_1 \simeq q_2$, and let $B^\phi_{q_1}$ and $B^\phi_{q_2}$ be the respective q-starting *BSA*s of B^ϕ.

By $q_1 \simeq q_2$ and Corollary 4.5.2, we have $B^\phi_{q_1} \sqsubseteq B^\phi_{q_2}$ and $B^\phi_{q_2} \sqsubseteq B^\phi_{q_1}$. This implies that q_1 (of $B^\phi_{q_1}$) simulates q_2 (of $B^\phi_{q_2}$) and vice versa. As $B^\phi_{q_1}$ and $B^\phi_{q_2}$ are isomorphic to B^ϕ, this implies q_1 simulates q_2 and q_2 simulates q_1 in B^ϕ. □

Obviously, both lemmas do not hold if B^ϕ is non-normal, because two non-normal states may be equivalent but have different present transitions or annotations which are not equivalent.

4.5.3 Minimization of a Boolean Annotated Service Automaton

Using these equivalence notions, we are finally ready to approach the main goal of this section, the canonical representative of a *BSA*.

To this end, we define the minimization of a *BSA* B^ϕ. As already sketched, the minimization basically merges equivalent states. Technically, each state of the minimized *BSA* $minimal(B^\phi)$ is an equivalence class of the states of B^ϕ. A transition of B^ϕ is now represented by a transition between the corresponding equivalence classes in $minimal(B^\phi)$.

Definition 4.5.11 (Minimization of a *BSA*, *minimal*(B^ϕ)).
The *minimization* of a *BSA* $B^\phi = [Q, I_{in}, I_{out}, \delta, q_0, \Omega, \phi]$ is defined as the *BSA* $minimal(B^\phi) = [Q', I_{in}, I_{out}, \delta', q_0', \Omega', \phi']$ with

- $Q' = \{[q] \mid q \in Q\}$,
- $\delta' = \{([q], x, [q']) \mid (q, x, q') \in \delta\}$,
- $q_0' = [q_0]$,
- $\Omega' = \{[q] \mid q \in \Omega\}$,

and $\phi' : Q' \to \mathcal{BF}$ is defined as $\phi'([q]) = \phi(q)$, for all $[q] \in Q'$. ⌟

In Fig. 4.17, the *BSA* B_1^ϕ is (isomorphic to) the minimization of the *BSA* B_2^ψ. The state q1 corresponds to the equivalence class $[1] = \{\mathsf{s1}, \mathsf{s5}\}$, q2 corresponds to the equivalence class $[2] = \{\mathsf{s2}\}$, and q3 corresponds to the equivalence class $[3] = \{\mathsf{s3}, \mathsf{s4}\}$. Because the initial state s1 of B_2^ψ is in class [1], q1 is the initial state of B_1^ϕ. As there is a final state in class [3], q3 is a final state of B_1^ϕ. The annotations of the states q1, q2, and q3 are arbitrarily chosen from the annotation of some state in the corresponding class.

The minimization procedure of Definition 4.5.11 works for normal and non-normal *BSA*s. As all non-normal states of a *BSA* B^ϕ are equivalent to each other, the minimized *BSA* $minimal(B^\phi)$ has at most one non-normal state. Unfortunately, merging two non-normal states may result in a normal state of $minimal(B^\phi)$. This, in general, causes a change of the semantics of B^ϕ. In this sense, the minimization relies on the fact that equivalent *normal* states have equivalent annotations and equally labeled transitions. Hence, the following Theorem 4.5.14 states that the semantics of B^ϕ is not changed by the minimization, i.e. $Match(B^\phi) = Match(minimal(B^\phi))$, only if B^ϕ is normal.

To ease the argumentation in the proof of Theorem 4.5.14, we first derive two corollaries from Lemma 4.5.9 and Lemma 4.5.10.

Corollary 4.5.12 (Annotation of equivalence class).
Let B be a normal *BSA* and let $minimal(B^\phi) = [B', \phi']$ be its minimization. Let q be a state of B^ϕ and $[q]$ be the corresponding state of $minimal(B^\phi)$.

Then, $\phi(q) \equiv \phi'([q])$. ⌟

Proof.
Follows from Lemma 4.5.9 and the construction of ϕ' in Definition 4.5.11. □

Corollary 4.5.13 (Equivalence class and simulation).
Let B be a normal *BSA* and let $minimal(B^\phi)$ be its minimization. Let q be a state of B^ϕ and $[q]$ be the corresponding state of $minimal(B^\phi)$.

Then, q simulates $[q]$ and $[q]$ simulates q. ⌟

Proof.
Follows from Lemma 4.5.10 and the construction of δ' in Definition 4.5.11. □

By combining the results of these corollaries, we are finally able to prove that the minimization procedure of Definition 4.5.11 does not change the semantics of a normal *BSA*:

Theorem 4.5.14 (Minimization preserves semantics).
For each normal *BSA* B^ϕ, $Match(B^\phi) = Match(minimal(B^\phi))$. ⌟

An obvious way to prove Theorem 4.5.14 is to show $B^\phi \sqsubseteq minimal(B^\phi)$ and $minimal(B^\phi) \sqsubseteq B^\phi$. With Theorem 4.4.2 this immediately implies $Match(B^\phi) \subseteq Match(minimal(B^\phi))$ and $Match(minimal(B^\phi)) \subseteq Match(B^\phi)$. However, we will show both inclusions directly.

Proof (of Theorem 4.5.14).
We show $Match(B^\phi) = Match(minimal(B^\phi))$ by showing both $Match(B^\phi) \subseteq Match(minimal(B^\phi))$ and $Match(minimal(B^\phi)) \subseteq Match(B^\phi)$.

$Match(B^\phi) \subseteq Match(minimal(B^\phi))$: Let $C \in Match(B^\phi)$, let ϱ be the corresponding simulation relation, and define a relation ϱ' between (the states of) C and (the states of) $minimal(B^\phi)$ as $\varrho' = \{(q_C, [q_B]) \mid (q_C, q_B) \in \varrho\}$. It suffices to show ϱ' is a simulation relation and for all $(q_C, [q_B]) \in \varrho'$: $\beta_C(q_C)$ satisfies the annotation of $\phi'([q_B])$.

To this end, let $(q_C, q_B) \in \varrho$ be arbitrary. As $C \in Match(B^\phi)$, we know q_B simulates q_C and q_C satisfies $\phi(q_B)$.

By the construction of ϱ', we have $(q_C, [q_B])$ in ϱ'. According to Corollary 4.5.13, $[q_B]$ simulates q_B, and according to Corollary 4.5.12, $\phi'([q_B]) \equiv \phi(q_B)$.

Hence, $[q_B]$ simulates q_C and q_C satisfies $\phi'([q_B])$, too, and we can conclude that $C \in Match(minimal(B^\phi))$ and therefore $Match(B^\phi) \subseteq Match(minimal(B^\phi))$.

$Match(minimal(B^\phi)) \subseteq Match(B^\phi)$: Let $C \in Match(minimal(B^\phi))$, let ϱ' be the corresponding simulation relation, and define a relation ϱ between (the states of) C and B^ϕ as $\varrho = \{(q_C, q_B) \mid (q_C, [q]) \in \varrho', q_B \in [q]\}$. It suffices to show ϱ is a simulation relation and for all $(q_C, q_B) \in \varrho$: $\beta_C(q_C)$ satisfies the annotation $\phi(q_B)$.

To this end, let $(q_C, [q]) \in \varrho'$ be arbitrary. As $C \in Match(minimal(B^\phi))$, we know $[q]$ simulates q_C and q_C satisfies $\phi'([q])$.

We have for all $q_B \in [q]$:

- (q_C, q_B) in ϱ (by the construction of ϱ),
- q_B simulates $[q]$ (by Corollary 4.5.13), and
- $\phi(q_B) \equiv \phi'([q])$ (by Corollary 4.5.12).

Hence, ϱ is a simulation relation as requested and q_C satisfies all $\phi(q_B)$, too. Thus, we conclude $C \in Match(B^\phi)$ and therefore $Match(minimal(B^\phi)) \subseteq Match(B^\phi)$.□

In other words, B^ϕ and $minimal(B^\phi)$ are in the same equivalence class of *BSA*s, i.e. $[B^\phi] = [minimal(B^\phi)]$.

Furthermore, the Corollaries 4.5.12 and 4.5.13 immediately imply:

Corollary 4.5.15 (Minimization preserves normal form).
If B^ϕ is a normal *BSA*, then $minimal(B^\phi)$ is a normal *BSA*, too. ⌟

The rest of this section is devoted to show that the minimization $minimal(B^\phi)$ of a *BSA* B^ϕ may indeed serve as the canonical representative of the corresponding equivalence class.

As $minimal(B^\phi)$ is derived from B^ϕ by merging states, $minimal(B^\phi)$ has at most a smaller number of states than B^ϕ. Next, we prove that a minimal *BSA* is indeed *minimal*, i.e. there is no *BSA* B'^ψ such that B'^ψ is equivalent to $minimal(B^\phi)$ and B'^ψ has less states than $minimal(B^\phi)$. Thus, $minimal(B^\phi)$ is a suitable candidate to serve as the representative of its equivalence class.

Definition 4.5.16 (Minimal *BSA*).
A *BSA* B^ϕ is *minimal* in its equivalence class $[B^\phi]$ if there is no *BSA* B'^ψ such that $B'^\psi \in [B^\phi]$ and $|Q_{B'}| < |Q_B|$. ⌟

Theorem 4.5.17 (Minimality of $minimal(B^\phi)$).
For each normal *BSA* B^ϕ, the *BSA* $minimal(B^\phi)$ is minimal in the equivalence class $[B^\phi]$. ⌟

For the proof of this theorem, we first prove two lemmas which help us to derive from the equivalence of two *BSA*s B^ϕ and B'^ψ an equivalence of states q and q' of a state q of B^ϕ and q' of B'^ψ.

Lemma 4.5.18.
Let B^ϕ and B'^ψ be normal, interface equivalent *BSA*s. Let $B^\phi \sqsubseteq B'^\psi$ and let ϱ be the corresponding simulation relation between states of B and B'.
Then, for each $(q, q') \in \varrho$: $B^\phi_q \sqsubseteq B'^\psi_{q'}$. ⌟

Proof.
The simulation relation ϱ between B^ϕ and B'^ψ is a simulation relation between B^ϕ_q and $B'^\psi_{q'}$ as well. As $\phi(q) \Rightarrow \psi(q')$ for all $(q, q') \in \varrho$, we conclude $B^\phi_q \sqsubseteq B'^\psi_{q'}$.□

Lemma 4.5.19.
Let B^ϕ and B'^ψ be normal, interface equivalent *BSA*s. Let $B^\phi \equiv B'^\psi$ and let ϱ be the corresponding simulation relation between the states of B^ϕ and B'^ψ.
Then, for each $(q, q') \in \varrho$: $B^\phi_q \equiv B'^\psi_{q'}$. ⌟

Proof.
By Corollary 4.5.2, $B^\phi \equiv B'^\psi$ iff $B^\phi \sqsubseteq B'^\psi$ and $B'^\psi \sqsubseteq B^\phi$. Let ϱ and ϱ' be the respective simulation relations.

By Lemma 4.5.18, $B^\phi \sqsubseteq B'^\psi$ implies $B^\phi_q \sqsubseteq B'^\psi_{q'}$ for all $(q, q') \in \varrho$; and $B'^\psi \sqsubseteq B^\phi$ implies $B'^\psi_{q'} \sqsubseteq B^\phi_q$ for all $(q', q) \in \varrho'$. As (by Proposition 4.1.5 on page 99), $\varrho' = \varrho^{-1}$, we conclude: $B^\phi_q \sqsubseteq B'^\psi_{q'}$ and $B'^\psi_{q'} \sqsubseteq B^\phi_q$ for all $(q, q') \in \varrho$. Applying Corollary 4.5.2 again, this yields $B^\phi_q \equiv B'^\psi_{q'}$ for all $(q, q') \in \varrho$. □

Proof (of Theorem 4.5.17).
Let B^ϕ be the minimization of a normal *BSA* as constructed in Definition 4.5.11 and let Q be the set of states of B^ϕ. We show that, for all *BSA* $C^\psi \in [B^\phi]$: $|Q_C| \geq |Q|$.

To this end, let $C^\psi \in [B^\phi]$ be arbitrary and assume $|Q_C| < |Q|$. We show that this assumption leads to a contradiction.

From $C^\psi \in [B^\phi]$ we know $C^\psi \equiv B^\phi$ (by Definition 4.5.4). This implies $B^\phi \sqsubseteq C^\psi$ (by Corollary 4.5.2). Let ϱ be the corresponding simulation relation.

Because of the assumption $|Q_C| < |Q|$, there must exist states $q_C \in Q_C$ and $q_B, q'_B \in Q$ such that $(q_B, q_C) \in \varrho$, $(q'_B, q_C) \in \varrho$, and $q_B \neq q'_B$. From $C^\psi \equiv B^\phi$ and Lemma 4.5.19, we know $B^\phi_{q_B} \equiv C^\psi_{q_C}$ and $B^\phi_{q'_B} \equiv C^\psi_{q_C}$. With $\equiv$ being an equivalence relation (and therefore being transitive), this yields $B^\phi_{q_B} \equiv B^\phi_{q'_B}$, which again means $q_B \simeq q'_B$. But $q_B \neq q'_B$ and $q_B \simeq q'_B$ violates the precondition that B^ϕ is a minimal *BSA* as constructed in Definition 4.5.11. Hence, the assumption $|Q_C| < |Q|$ must be wrong and, therefore, B^ϕ is indeed minimal in $[B^\phi]$. □

The minimality of $\mathit{minimal}(B^\phi)$ in the equivalence class $[B^\phi]$ of a normal *BSA* B^ϕ justifies the use of $\mathit{minimal}(B^\phi)$ as the canonical representative of B^ϕ. According to Theorem 4.5.17, there is no equivalent *BSA* for a *BSA* $\mathit{minimal}(B^\phi)$ with less states than $\mathit{minimal}(B^\phi)$. Together with Lemma 4.5.19 this also means that there is no *BSA* with an equal number of states but less transitions (as otherwise the states cannot be equivalent). Hence we conclude:

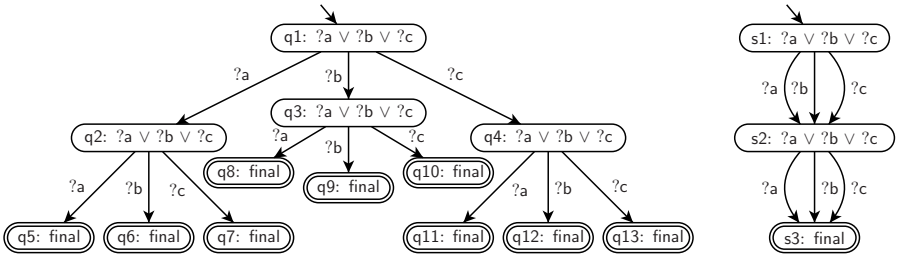
Corollary 4.5.20 (Canonical representative of a *BSA*).
The *BSA* $\mathit{minimal}(B^\phi)$ of a normal *BSA* B^ϕ is uniquely defined (up to isomorphism) and thus called the *canonical representative* of B^ϕ. ⌟

Therefore, it is possible to decide equivalence of two *BSA*s by minimizing both *BSA*s and then checking isomorphism of the resulting minimizations and annotation equivalence of isomorphic states.

Another, particularly important advantage of the minimization procedure is the reduced capacity requirement for storing a representative of a *BSA* instead of the original *BSA*.

Although all *BSA*s have a deterministic structure, the effect of the minimization can drastically reduce the number of states (and transitions) of a *BSA*.

Figure 4.18 illustrates one such case. The *BSA* B^{ϕ}, depicted in Fig. 4.18(a), has three present transitions at the states q1, q2, q3, and q4, respectively, each transition leading to a different state. In total, B^{ϕ} has 13 states and 12 transitions. It can easily be seen, however, that the states q2, q3, and q4 are equivalent to each other; as well as the states q5 to q13. Hence, the minimized *BSA* $\mathit{minimal}(\mathsf{B}^{\phi})$ of Fig. 4.18(b) has only three states, s1, s2, and s3. The state s1 represents the state q1; the state s2 represents q2, q3, and q4; and s3 represents the states q5 to q13. If B^{ϕ} had even more transitions per state or more than two "communication steps" before ending in a final state, the reduction would be even stronger.



(a) B^{ϕ} (b) $\mathit{minimal}(\mathsf{B}^{\phi})$

Figure 4.18: Minimization of a *BSA* with drastic reduction.

In Chapter 5, we will see that a *BSA* such as the one depicted in Fig. 4.18(a) can indeed be computed as the operating guideline OG_A of a service automaton A. That is, B^{ϕ} is no artificial example and practically relevant. This emphasizes the importance of our minimization procedure, as we obviously do not want to store B^{ϕ}, if we could store the equivalent *BSA* $\mathit{minimal}(\mathsf{B}^{\phi})$.

4.6 Possible Variants of *BSA* Definitions

In this section, we want to analyze possible different ways for defining Boolean annotated automata and the corresponding influences to other definitions or existent results.

4.6.1 Services with Restrictions on their Final States

As already motivated in Sect. 3.5.1, there are different possible restrictions for the choice of final states of a service net or service automaton. Such a restriction

reduces the number of service automata C which are possibly matched with a *BSA* B^ϕ. Hence, additional constraints have to be considered in the normalization of a *BSA*.

Services with Non-Transient Final States

If final states are required to be non-transient, then the state normalization of a *BSA* B^ϕ must also remove states q with annotation $\phi(q) = x \wedge \mathit{final}$ if x is an output channel of B^ϕ. No service automaton C's assignment β_C can satisfy such a formula, because only a state q_C of C which is a final state in C assigns *true* to the literal *final*, i.e. $\beta_C(q_C)(\mathit{final}) = \mathit{true}$, but as x is also an output channel of C in this case, q_C cannot have an x-labeled outgoing transition in such a final state. So $\beta_C(q_C)(x) = \mathit{false}$ and hence $\beta_C(q_C) \not\models \phi(q)$.

Services with Strictly Terminating Final States

In the case of strictly terminating services, the normalization of a *BSA* B^ϕ must remove all states q with annotation $\phi(q) = x \wedge \mathit{final}$ for arbitrary message channels $x \in \mathcal{MC}$. A matching service automaton C can assign *true* to the literal *final* at a state q_C only if q_C is a final state of C. As C is strictly terminating, q_C then has no present transitions at all and thus $\beta_C(q_C)(x) = \mathit{false}$ for each $x \in \mathcal{MC}$. Hence, $\beta_C(q_C) \not\models \phi(q)$ and q must be removed from B^ϕ.

4.6.2 *BSA*s with Different Underlying Structures

*BSA*s without Internal Transitions

A *BSA* B^ϕ as defined in Definition 4.2.1 (on page 101) has an underlying *deterministic* service automaton B that *can have internal transitions* (according to Definition 3.3.8 on page 64). As already stated, this is a rather uncommon definition of determinism. In this subsection, we show that it is possible to restrict the definition of *BSA*s to deterministic structures in the classical sense, i.e. it is possible to disallow internal transitions for *BSA*s.

Then, however, the matching procedure of a service automaton C and a *BSA* B^ϕ has to be adjusted such that it "tolerates" internal transitions in C. This can easily be achieved by preserving τ-literals in the annotations (and therefore retain classical annotation satisfaction) but relaxing the simulation relation to a *weak* simulation relation. Then, however, an inconvenient level of indirection is introduced in most proofs that rely on the simulation relation ϱ.

Furthermore, the notion of a normal state *BSA* B^ϕ has to be slightly adjusted to preserve the result $\mathit{Match}(B^\phi) = \mathit{Match}(\mathit{normal}(B^\phi))$: a present literal τ in an

annotation $\phi(q)$ has to be set to *true* by the maximal assignment $\beta^+(q)$ although the state q of the *BSA* has no present τ-labeled transition. In other words, the τ literal must now be treated like the *final* literal and not like a literal $x \in \mathcal{MC}$. Consequently, all proofs relying on the maximal assignment $\beta_B^+(q)$ have to be adjusted.

In summary, disallowing internal transitions in a *BSA* would not change any result, but would result in different definitions and proofs. In fact, the implementation in FIONA (see Chap. 7) does not store internal transitions.

Non-deterministic *BSA*s

On the other hand it is also possible to relax Boolean annotated service automata to a fully non-deterministic version. This mainly results in the possibility that there are several different simulation relations for the matching of a service automaton C with a *BSA* B^ϕ.

Therefore, the matching procedure must be adjusted such that it is sufficient that *one* of these simulations fulfills the annotation satisfaction requirement. Analogously, the requirements of the smaller relation $\sqsubseteq$ between *BSA*s (cp. Sect. 4.4) have to be adjusted such that the annotation implication holds for at least one simulation relation.

However, *finding* the "correct" simulation relation is far more complex than finding the unique simulation relation in the deterministic case (with or without deterministic internal transitions) and therefore raises the complexity of the matching and the check for $\sqsubseteq$. Hence, a non-deterministic *BSA* is not suited to efficiently decide matching.

4.6.3 Negation in Boolean Formulae

Allowing the negation of a literal in a Boolean formula enhances the expressiveness of *BSA*s. For instance, consider a state q with annotation $\phi(q) = \neg(a \wedge b)$ and two present transitions at q labeled with a and b, respectively. A *BSA* with such a state would allow for the characterization of a set of service automata that cannot be characterized without negation, because the formula $\neg(a \wedge b)$ cannot be expressed without negation.

Additionally, many results of this chapter would be lost. For instance, the domination of assignments is no longer a sufficient criterion for the satisfaction of Boolean formulae. This considerably complicates the normalization of *BSA*s. Furthermore, the structural relation $\sqsubseteq$ induces no longer a relationship between the semantics of the considered *BSA*s and we do not see an obvious way to repair this theorem.

4.7 Related Work

Boolean annotated automata similar to *BSA*s as introduced in this chapter have already been developed in [WFMN04]. Therein, the authors suggest to annotate each state q of an automaton A by a Boolean formula $\phi(q)$. This Boolean formula is then used to decide behavioral compatibility of A with another (annotated) automaton B.

In contrast to our service automata, the automata in [WFMN04] communicate synchronously, i.e. the composition of annotated automata A and B according to [WFMN04] is their synchronous product $A \otimes B$. Thereby, $A \otimes B$ has a common x-labeled transition leaving a state (q, q') of $A \otimes B$ if and only if A has an $!x$-labeled transition at q and B has a corresponding $?x$-labeled transition at q' (or vice versa). During the composition, the annotation of a composed state (q, q') is the conjunction of the annotations of q and q'. That way, the composed state (q, q') can have less leaving transitions than q or q' and the local states q and q' may satisfy their respective formulae[1] whereas the composed state (q, q') may violate the composed formula. In this case, the state (q, q') represents an erroneous state (comparable to non-normal states of *BSA*s in this thesis). The corresponding compatibility notion of [WFMN04] is then the existence of at least one transition sequence over non-erroneous states of the composition $A \otimes B$ leading to a final state.

Compared to our *BSA* approach, we see several disadvantages of the annotated automata of [WFMN04]. First of all, the automata of [WFMN04] are based on a synchronous communication model, simplifying the correctness verification to an emptiness question of the accepted language of the composed automaton $A \otimes B$. Secondly, their compatibility notion does not capture deadlock freedom of the composition but only the existence of one successful interaction. Finally, the authors propose their annotations as a *specification* technique. That is, the designer has to manually annotate the automata for being able to detect errors of the composition later on. In contrast, we will present a method to *generate* the Boolean annotations in the forthcoming Chap. 5 and thus allow for a fully automatic characterization of compatible services.

In [MWF05], the authors build upon their results of [WFMN04] and introduce an abstraction technique of *infinite* accepted transition sequences of an annotated automaton to *finite* transition sequences such that the abstraction preserves the compatibility relationship to other annotated automata. With the help of the finite abstraction, standard indexing algorithms can be applied for being able to efficiently find a behaviorally compatible service in a (possibly large) set of published services. A case study shows the feasibility of the approach for a set containing up to 822 services. It is interesting future work to check the applicability of this approach to our *BSA*s.

[1] Meaning $\beta_A(q) \models \phi(q)$ and $\beta_B(q') \models \phi(q')$ in our terms.

4.8 Concluding Remarks

In this chapter, we have introduced Boolean annotated service automata (*BSA*s) as a means to *characterize a set of services*. To this end, a *BSA* is a special service automaton whose states are annotated by Boolean formulae. That is, such a *BSA* B^ϕ consists of a service automaton B and a mapping ϕ assigning to each state q of B a Boolean formula $\phi(q)$. Then, a *matching procedure* is used to decide whether or not another service automaton C is characterized by B^ϕ. The set of service automata characterized by a *BSA* B^ϕ, i.e. the set $Match(B^\phi)$, forms the *semantics* of B^ϕ.

We then introduced a *normal form* of *BSA*s to remove redundant information from the *BSA*. By the help of a *normalization procedure*, an arbitrary *BSA* can be transformed into its normal form while preserving the semantics of the *BSA*. The main result in this regard is Theorem 4.3.12. Using normal *BSA*s, we were then able to relate the semantics of two *BSA* by comparing the structure of the *BSA*s. That way, we can also efficiently decide *equivalence* of *BSA*s (Theorem 4.4.2 and Corollary 4.5.2). Finally, we derived the *canonical representative* of a *BSA* and a minimization procedure to transform an arbitrary normal *BSA* B^ϕ into its canonical representative $minimal(B^\phi)$. In Theorem 4.5.14, we could prove that this minimization also preserves the semantics of a *BSA*. Furthermore, we showed that the minimization $minimal(B^\phi)$ is *minimal* and *unique*. That is, there is no other equivalent *BSA* with a smaller or equal number of states or transitions.

*BSA*s will be employed in the upcoming Chapter 5, where we will introduce the notion of an *operating guideline* OG_A of a service automaton A. OG_A will turn out to be a specially constructed *BSA* such that its semantics is equal to the set of strategies of A, i.e. $Match(OG_A) = Strat(A)$.

5 Operating Guidelines for Services

In Chap. 3, we have introduced our formal framework for service modeling and motivated the need for a formal analysis of correct interaction of services. We have developed the notion of a strategy B for a service automaton A, capturing behavioral compatibility of A and B. The set $\mathit{Strat}(A)$, representing all strategies for A, is of particular interest as it gives a semantics for a service A in terms of all behaviorally compatible services B for A.

In the subsequent Chap. 4, we have seen how to characterize *some* set of services with the help of a Boolean annotated service automaton (*BSA*) B^ϕ. To this end, we have introduced a matching procedure to efficiently decide $C \in \mathit{Match}(B^\phi)$.

In this chapter, we will combine both notions and present a construction of a special *BSA* for a given service A that characterizes exactly the set of strategies for A. Such a *BSA* is called *operating guideline* of A.

Definition 5.0.1 (Operating guideline, OG_A).
A *BSA* B^ϕ is called *operating guideline* for a service automaton A, denoted OG_A, iff $\mathit{Match}(B^\phi) = \mathit{Strat}(A)$. ⌟

As already seen in the previous chapter, different *BSA*s may characterize the same set of services. Correspondingly, there is no notion of *the* operating guideline of a service. However, the minimization procedure for *BSA*s can also be applied to operating guidelines in order to derive the canonical representative of all operating guidelines of a service.

The rest of this chapter is organized as follows. We start with a characterization of deadlocks between arbitrary (interface compatible) service automata A and B from the point of view of B in Sect. 5.1. To decide deadlock freedom of A and B, we investigate the *knowledge* of B about A, basically a projection of the composition $A \oplus B$ to the states of B. Then, Sect. 5.2 is devoted to completely break the symmetric characterization of deadlocks into an asymmetric characterization of strategies. Therefore, we will construct a special service automaton $\mathcal{F}_A$

for A such that the knowledge of $\mathcal{F}_A$ about A and conditions on the simulation relation between B and $\mathcal{F}_A$ are sufficient to decide whether B is a strategy for A or not. Unfortunately, $\mathcal{F}_A$ allows for almost no abstraction from the internals of A and, even worse, $\mathcal{F}_A$ is usually infinite. Consequently, the subsequent sections are devoted to these issues. We will first restrict our strategy characterization to finite strategies B for A by a finite part of $\mathcal{F}_A$ in Sect. 5.3, and then formulate all conditions on the knowledge of this finite part of $\mathcal{F}_A$ as Boolean annotations to the states of $\mathcal{F}_A$ in Sect. 5.4. That way, we derive a *BSA* characterizing the set of finite strategies B for A, i.e. an operating guideline of A. Again, we will explore different design decisions for operating guidelines and their induced consequences in Sect. 5.5. Section 5.6 describes related work in detail and finally, Sect. 5.7 concludes this chapter.

In the forthcoming Chap. 7, we will finally present an implementation of the operating guidelines approach in our analysis tool Fiona. We furthermore show a case study justifying (1) the computability of operating guidelines for real-world services and (2) the feasibility of using operating guidelines as an artifact to efficiently decide behavioral compatibility of services.

5.1 A Characterization of Deadlocks

In Chapter 3, a deadlock was defined as a behavioral property of a single service. Based on this property, a service is a strategy for another service if and only if their interaction — represented by the single service of their composition — is deadlock-free. In this sense, we did not distinguish between the participants of a composition for deciding their behavioral compatibility.

This section is devoted to lay the basis for breaking this symmetry: we will fix one service, always called A in the following, and want to characterize deadlocks that may occur in the interaction of some service B with A from the point of view of B. To this end, we will develop certain conditions that are necessary and sufficient to decide the existence (or absence) of deadlocks in the composition $A \oplus B$. The basic idea is sketched in Fig. 5.1.

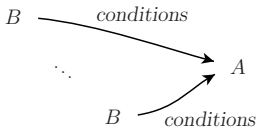


Figure 5.1: Schematic overview of deadlock characterization between a service automaton A and several service automata B. We formulate conditions from the point of view of B that are sufficient and necessary for $B \in \mathit{Strat}(A)$.

The presented characterization will be applicable to arbitrary (interface compat-

ible) service automata A and B and allows for open compositions $A \oplus B$ in this section. Section 5.2 will then consider *strategy* relationships only, i.e. closed compositions $A \oplus B$.

5.1.1 Situations and Knowledge

Consider a state $[q_A, q_B, M]$ of the composition $A \oplus B$ of two service automata A and B. It consists of a state q_A of A, a state q_B of B, and a multiset M of currently pending messages. In order to decompose such a composed state, we employ the notion of a *situation* of a service automaton. A situation of A comprises its state q_A and the message multiset M. That is, the situation of A corresponding to the state $[q_A, q_B, M]$ is $[q_A, M]$. Accordingly, the corresponding situation of B is $[q_B, M]$.

The situation $[q_A, M]$ of A reflects the point of view of a service automaton B on the state $[q_A, q_B, M]$ of the composition $A \oplus B$ if B is in its state q_B. That way, we are able to consider a composition $A \oplus B$ from the service automaton B's viewpoint. To this end, we first introduce the general concept of a situation and then formalize the point of view of B in the notion of the *knowledge* of B about the service automaton A.

Definition 5.1.1 (Situation, *situations*(A)).
Let A be a service automaton, let $q_A \in Q_A$ be a state of A, and let $M \in \mathit{bags}(I_{io\,A})$ be a multiset of messages.

Then, $[q_A, M]$ is a *situation* of A. Furthermore, let $\mathit{situations}(A) = Q_A \times \mathit{bags}(I_{io\,A})$ denote the set of all situations of A. ⌟

Figure 5.2 shows two service automata A and B. The set of situations of A comprises, among infinitely many others, the situations [r1, [a]], [r1, [a, a, a, a, a, a]], [r2, [b, b]], and [r3, [a, b, c, c]], for instance, as r1, r2, and r3 are states of A, and a, b, and c are in the interface of A. Analogously, [s1, [a]], [s1, [a, a, a, a, a, a]], [s2, [b, b]], and [s3, [a, b, c, c]], for instance, are situations of B.

Situations are used to represent the point of view of one service automaton participating in a composition $A \oplus B$ on a state $[q_A, q_B, M]$ of the composition. As composition is symmetric, we can equally consider the viewpoint of any of the two service automata A and B.

Corollary 5.1.2 (Mutual situations).
Let A and B be two service automata and let $[q_A, q_B, M]$ be a state of their composition $A \oplus B$.

Then, $[q_A, M]$ is a situation of A and $[q_B, M]$ is a situation of B. ⌟

(a) A (b) B

Figure 5.2: Two interface compatible service automata A and B. It is easy to see that B is a strategy for A.

Proof.
From $[q_A, q_B, M]$ being a state of $A \oplus B$ and by the definition of service automata composition (Definition 3.3.18) immediately follows that each $x \in M$ is a shared channel between A and B, i.e. $x \in I_{io\,A} \cap I_{io\,B}$, for all $x \in M$. Hence, each $x \in M$ is an interface channel of B. By the definition of a situation of B this directly implies that $[q_B, M]$ is a situation of B. □

In the following, we will fix the service automaton A and always consider deadlocks from the point of view of some interface compatible service automaton B for A, i.e. we consider deadlocks from the viewpoint of B. This idea has already been illustrated in Fig. 5.1.

So far, the notion of a situation of a service automaton A is very general and the set $\mathit{situations}(A)$ contains combinations of a state of A and messages M that can never be part of a state of a composition $A \oplus B$ for any B.

For instance, the service automaton A of Fig. 5.2(a) sends at most one message b. Hence, all situations of A with more than one pending b, e.g. the situation $[\mathsf{r2}, [\mathsf{b}, \mathsf{b}]]$ of A, can never "occur" in a composition with A.

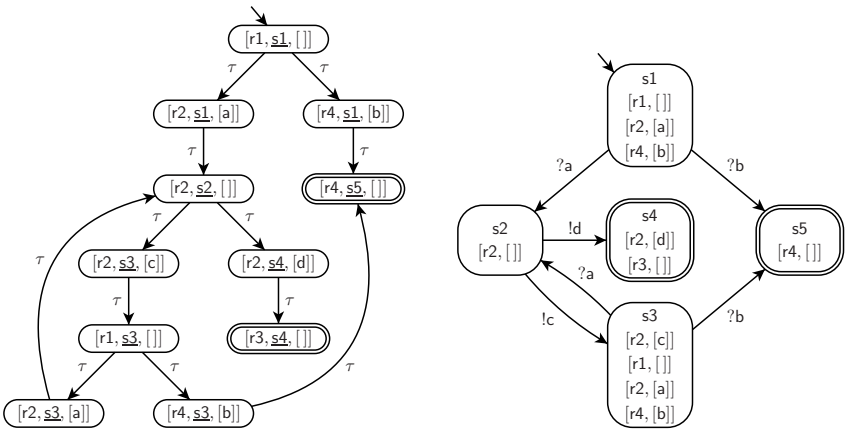
To link the situations of a service automaton to the actual participants of a composition, we introduce the notion of the *knowledge* of one service automaton about the other one. Intuitively, the knowledge of B about A is the set of situations of A that A might be in while B is in state q_B. More precisely, the knowledge of a state q_B of a service automaton B collects all situations $[q_A, M]$ of A such that $[q_A, q_B, M]$ is an internally reachable state of the composition $A \oplus B$. Hence, it reflects B's point of view of the composition $A \oplus B$ at its state q_B.

Definition 5.1.3 (Knowledge, $k(q)$).
Let A and B be service automata and let $A \oplus B$ be their composition.

Then, the *knowledge* of B about A is a mapping $k_{(B,A)} : Q_B \to \wp(\mathit{situations}(A))$ defined as $k_{(B,A)}(q_B) = \{[q_A, M] \mid \text{there exists a state } q_B \in Q_B \text{ such that } [q_A, q_B, M] \in Q_{A \oplus B} \text{ and } [q_A, q_B, M] \text{ is internally reachable in } A \oplus B\}$. ⌟

We omit the index of the knowledge $k_{(B,A)}(q_B)$ and write $k(q_B)$ if the considered service automata and the direction are clear from the context. Furthermore, we often refer to $k(q_B)$ as the *knowledge set* (of B) at q_B and the situations in $k(q_B)$ as *knowledge values* at q_B.

To exemplify the knowledge notion, consider again the service automata A and B of Fig. 5.2. The composition of these interface compatible service automata, A⊕B, is depicted in Fig. 5.3(a). Figure 5.3(b) shows the service automaton B with all its knowledge values at a state q depicted inside q. For instance, the knowledge of B at its state s1, $k_{(\mathsf{B},\mathsf{A})}(\mathsf{s1})$, contains each situation $[q_A, M]$ of A that occurs with s1 in the composition in Fig. 5.3(a). Considering the initial state $[\mathsf{r1}, \mathsf{s1}, [\,]]$ of $\mathsf{A} \oplus \mathsf{B}$, we easily see that the situation $[\mathsf{r1}, [\,]]$ is an element of $k_{(\mathsf{B},\mathsf{A})}(\mathsf{s1})$. Collecting all states of A⊕B with second element s1, we get $k_{(\mathsf{B},\mathsf{A})}(\mathsf{s1}) = \{[\mathsf{r1}, [\,]], [\mathsf{r2}, [\mathsf{a}]], [\mathsf{r4}, [\mathsf{b}]]\}$. Hence, as long as B is in state s1, A is in one of the situations $[\mathsf{r1}, [\,]]$, $[\mathsf{r2}, [\mathsf{a}]]$, or $[\mathsf{r4}, [\mathsf{b}]]$.



(a) A ⊕ B (b) B with $k_{(\mathsf{B},\mathsf{A})}$

Figure 5.3: (a) Composition A ⊕ B of the two interface compatible service automata A and B of Figs. 5.2(a) and 5.2(b) with the states of B highlighted. (b) The service automaton B with its knowledge about A depicted inside the corresponding state of B.

From the point of view of B, the knowledge of B about A at a state q_B of B represents the set of all states that A can be in by receiving (already present) messages or sending messages on its own. Assuming that B knows the structure of A (e.g. the sets Q_A of states and δ_A of transitions of A) and the history of sent and received messages only, B does not know the actual state that A is currently in. From its information, however, B can deduce, for each of its states q_B, a *set of situations* of A which *contains* the current situation of A. Thus, the knowledge $k_{(B,A)}(q_B)$ can be seen as a *hypothesis* of B with respect to the actual state of A

and the actual state of the message channels.

In the knowledge sets of Fig. 5.3(b), we can see that there is no state of B with a situation $[\mathsf{r5}, M]$ (for some M) as knowledge value in $k_{(\mathsf{B},\mathsf{A})}$. That is, the state r5 of A does never occur in the composition with B. This can also be seen in the composition $\mathsf{A} \oplus \mathsf{B}$ in Fig. 5.3(a). There is no composed state with r5 as the first component. If we construct the knowledge values for the opposite direction, i.e. the knowledge $k_{(\mathsf{A},\mathsf{B})}$ of A about B, we end up with a special knowledge set at this state r5, i.e. $k_{(\mathsf{A},\mathsf{B})}(\mathsf{r5}) = \emptyset$.

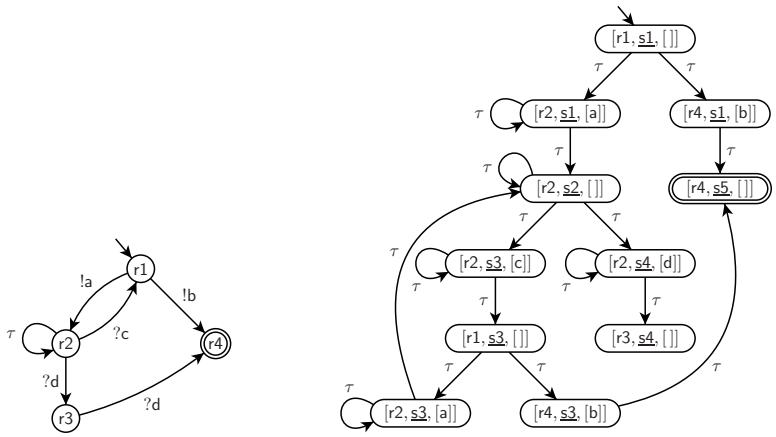
Such a knowledge set $k(q) = \emptyset$ represents the fact that there is no internally reachable state in the composition "using" q. It is usually called *empty knowledge*. The empty knowledge is one of the reasons that the knowledge k of a service automaton B about some service automaton A is already an (albeit small) abstraction of the composed service automaton $A \oplus B$. That is, it is in general not possible to fully reconstruct A from B and $k_{(B,A)}$ alone, as states of A that do not occur in the knowledge of B about A cannot be reconstructed from the knowledge. Furthermore, no information about final states of A is preserved by the knowledge construction according to Definition 5.1.3.

Figure 5.4 illustrates the abstraction. It shows another service automaton, A', and its composition $\mathsf{A}' \oplus \mathsf{B}$ with the service automaton B of Fig. 5.2(b). The most important difference between the service automaton A of Fig. 5.2(a) and A' is that the respective state r3 is a final state in A, but not in A'. Hence, the state $[\mathsf{r3}, \mathsf{s4}, [\,]]$ is a deadlock in $\mathsf{A}' \oplus \mathsf{B}$, but not in $\mathsf{A} \oplus \mathsf{B}$. Furthermore, A' has no unused state, i.e. for each state q of A', there is a multiset M of messages and a state q_B of B with $[q, M]$ is a knowledge value at q_B. However, the knowledge of B about A and the knowledge of B about A' are equal, i.e. A and A' are indistinguishable by B. For instance, the situation $[\mathsf{r3}, [\,]]$ of A is a knowledge value of the state s4, just as the situation $[\mathsf{r3}, [\,]]$ of A'.

As deadlocks are specifically important for the characterization of strategies, we present a classification of the situations of a service automaton in the following and distinguish, in analogy to the notions of transient and stable states of a service automaton (cp. Definition 3.3.11), *transient* and *stable situations*.

Definition 5.1.4 (Transient, stable situation).
A situation $[q, M]$ of a service automaton A is *stable* in A if for all transitions $(q, x, q') \in \delta_A$: $x \in I_{in\,A}$ and $x \notin M$. Otherwise, $[q, M]$ is *transient* in A. ⌟

Considering again the service automaton A of Fig. 5.2(a), the situations $[\mathsf{r1}, [\mathsf{a}]]$ and $[\mathsf{r1}, [\mathsf{a}, \mathsf{a}, \mathsf{a}, \mathsf{a}, \mathsf{a}]]$, for instance, are transient (as there is a present sending transition at r1), but the situations $[\mathsf{r2}, [\mathsf{b}, \mathsf{b}]]$ and $[\mathsf{r3}, [\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{c}]]$, for instance, are stable in A (as there is no present b-labeled receiving transition at the state r2 or no present transition at all at the state r3, respectively). In the service automaton B of Fig. 5.2(b), the situation $[\mathsf{s1}, [\mathsf{a}]]$, for instance, is transient (as there is a present



(a) A' (b) $\mathsf{A}' \oplus \mathsf{B}$

Figure 5.4: (a) A service automaton A' which is a structurally different version of the service automaton A of Fig. 5.2(a). (b) Consequently, the composition $\mathsf{A}' \oplus \mathsf{B}$ is different from the composition $\mathsf{A} \oplus \mathsf{B}$ of Fig. 5.3(a). In particular, the state $[\mathsf{r3}, \mathsf{s4}, [\,]]$ is a deadlock of $\mathsf{A}' \oplus \mathsf{B}$, whereas $\mathsf{A} \oplus \mathsf{B}$ has no deadlock.
Hence, B is a strategy for A, but B is no strategy for A'. However, in each state of B, the knowledge of B about A' is equal to its knowledge about A, i.e. $k_{(\mathsf{B},\mathsf{A})}(q_B) = k_{(\mathsf{B},\mathsf{A}')}(q_B)$ for each state q_B of B.

a-labeled receiving transition at the state s1 and there is an a pending in this situation).

5.1.2 A Characterization of Deadlocks by Knowledge

The knowledge of a service automaton B about some service automaton A, together with the information about final states of A and the distinction between transient and stable situations, is sufficient to characterize internally reachable deadlocks in $A \oplus B$ from the point of view of B.

According to the following lemma, a state $[q_A, q_B, M]$ of $A \oplus B$ is an internally reachable deadlock if and only if both situations $[q_A, M]$ and $[q_B, M]$ are stable in the respective service automaton and $[q_A, q_B, M]$ is no final state of $A \oplus B$. That is, neither A nor B can make a "move" on its own, but $A \oplus B$ is not yet in one of its final states.

Lemma 5.1.5 (Characterization of deadlocks).
Let A and B be service automata, $A \oplus B$ their composition, and let k be the knowledge of B about A.

Then, $A \oplus B$ has an internally reachable deadlock iff there is a state q_B of B and a situation $[q_A, M]$ of A with $[q_A, M] \in k(q_B)$ such that all of the following conditions hold:

(i) the situation $[q_A, M]$ of A is stable in A and

(ii) the situation $[q_B, M]$ of B is stable in B and

(iii) $q_A \notin \Omega_A$ or $q_B \notin \Omega_B$ or $M \neq [\,]$. ⌟

Proof.
($\Rightarrow$): Let $[q_A, q_B, M]$ be an internally reachable deadlock of $A \oplus B$. By Definition 3.3.18 (composition) and Definition 5.1.3 (knowledge), q_B is a state of B with $[q_A, M] \in k(q_B)$. We show (i), (ii), and (iii).

By Definition 3.3.23 (deadlock) and Definition 3.3.11 (stable state), there is no sending transition and no internal transition in $A \oplus B$ starting at $[q_A, q_B, M]$.

Hence, by the construction of $A \oplus B$, this implies that there is no sending and no internal transition in neither A nor B, as well as there is no x-labeled receiving transition in neither A nor B with $x \in M$. Hence, we conclude (i) and (ii).

Furthermore, $[q_A, q_B, M]$ being a deadlock of $A \oplus B$ implies that $[q_A, q_B, M]$ is no final state of $A \oplus B$. Again by the construction of $A \oplus B$, this implies (iii).

($\Leftarrow$): Let q_B be a state of B, $[q_A, M]$ be a situation of A with $[q_A, M] \in k(q_B)$, and let (i), (ii), and (iii) hold. By Definition 3.3.18 (composition) and Definition 5.1.3 (knowledge), $[q_A, q_B, M]$ is an internally reachable state of $A \oplus B$. We show $[q_A, q_B, M]$ is a deadlock of $A \oplus B$.

From (i), (ii), and the construction of $A \oplus B$, we conclude that there is no transition in $A \oplus B$ starting at $[q_A, q_B, M]$. From (iii) we know that $[q_A, q_B, M]$ is no final state of $A \oplus B$. Hence, $[q_A, q_B, M]$ is a deadlock in $A \oplus B$. □

With the help of Lemma 5.1.5 we are able to decide whether or not the composition $A \oplus B$ of service automata A and B has an internally reachable deadlock by checking the three conditions for each situation of A in the knowledge of each state of B. If all three conditions are fulfilled by a situation $[q_A, M]$ in the knowledge of a state q_B, then the state $[q_A, q_B, M]$ is an internally reachable deadlock of $A \oplus B$. Hence, Lemma 5.1.5 provides conditions whose check is sufficient and necessary for behavioral *in*compatibility of two service automata A and B.

A positive reformulation of the latter lemma is given in Corollary 5.1.6. It provides a direct characterization when the composition $A \oplus B$ is deadlock-*free*, i.e. has no internally reachable deadlock. In this case, A and B are behaviorally compatible.

Corollary 5.1.6 (Characterization of deadlock freedom).
Let A and B be service automata, let $A \oplus B$ be their composition, and let k be the knowledge of B about A.

Then, $A \oplus B$ has no internally reachable deadlock iff for all states q_B of B and all situations $[q_A, M]$ of A with $[q_A, M] \in k(q_B)$, at least one of the following conditions is fulfilled:

(i) $[q_A, M]$ is transient in A or

(ii) the situation $[q_B, M]$ of B is transient in B or

(iii) $q_A \in \Omega_A$ and $q_B \in \Omega_B$ and $M = [\,]$. ⌟

Proof.
The corollary is an equivalent reformulation of Lemma 5.1.5 by negating both sides of the lemma. □

According to Corollary 5.1.6, the composition of service automata A and B is deadlock-free if and only if for each pair q_B and $[q_A, M] \in k(q_B)$, at least one of the conditions of Corollary 5.1.6 is fulfilled.

As an example, we apply Corollary 5.1.6 to the service automata A and B of Fig. 5.2, and easily verify deadlock freedom of the composition $\mathsf{A} \oplus \mathsf{B}$ shown in Fig. 5.3(a). The only stable situations of A occurring in the knowledge $k_{(\mathsf{B},\mathsf{A})}$ (see Fig. 5.3(b)) are:

- $[\mathsf{r2}, [\mathsf{a}]]$ and $[\mathsf{r4}, [\mathsf{b}]]$ in the knowledge of $\mathsf{s1}$, but both $[\mathsf{s1}, [\mathsf{a}]]$ and $[\mathsf{s1}, [\mathsf{b}]]$ are transient situations in B;
- $[\mathsf{r2}, [\,]]$ in the knowledge of $\mathsf{s2}$, but $[\mathsf{s2}, [\,]]$ is transient in B;
- $[\mathsf{r2}, [\mathsf{a}]]$ and $[\mathsf{r4}, [\mathsf{b}]]$ in the knowledge of $\mathsf{s3}$, but both $[\mathsf{s3}, [\mathsf{a}]]$ and $[\mathsf{s3}, [\mathsf{b}]]$ are transient in B;
- $[\mathsf{r3}, [\,]]$ in the knowledge of $\mathsf{s4}$, but $\mathsf{r3}$ is a final state of A, $\mathsf{s4}$ is a final state of B, and there are no messages pending; and
- $[\mathsf{r4}, [\,]]$ in the knowledge of $\mathsf{s5}$, but $\mathsf{r4}$ is a final state of A, $\mathsf{s5}$ is a final state of B, and there are no messages pending.

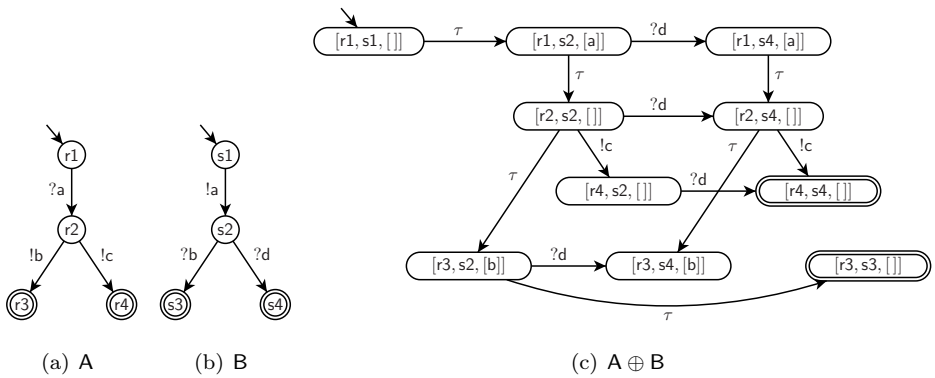
That is, at least one condition of Corollary 5.1.6 is fulfilled for each state q of B and each situation in $k_{(\mathsf{B},\mathsf{A})}(q)$. Hence, $\mathsf{A} \oplus \mathsf{B}$ has no internally reachable deadlock. As additionally $\mathsf{A} \oplus \mathsf{B}$ is closed, we conclude that B is a strategy for A.

In contrast, $\mathsf{A}' \oplus \mathsf{B}$ of Fig. 5.4(b) *has* a deadlock. This can be easily verified with Corollary 5.1.6, too. The situation $[\mathsf{r3}, [\,]]$ of A' is in the knowledge of the state $\mathsf{s4}$ of B, too, but neither $[\mathsf{r3}, [\,]]$ nor $[\mathsf{s4}, [\,]]$ is transient (violating conditions (i) and (ii) of the corollary), and the state $\mathsf{r3}$ is no final state in A' (violating condition (iii) of the corollary). Hence, all conditions of Corollary 5.1.6 are violated for the state $\mathsf{s4}$ of B and the situation $[\mathsf{r3}, [\,]] \in k_{(\mathsf{B},\mathsf{A})}(\mathsf{s4})$. Hence, B is no strategy for A'.

It is worth to mention that Corollary 5.1.6 equally holds for open and closed compositions $A \oplus B$. That is, we can verify deadlock freedom also for service automata A and B, where $A \oplus B$ is an open service automaton.

As an example, reconsider the service automata A and B and their open composition A ⊕ B of Fig. 3.9, recalled in Fig. 5.5. We apply Corollary 5.1.6 to reveal all deadlocks of A ⊕ B.

- Due to the composed state [r4, s2, [ ]], the knowledge of B about A at B's state s2 contains the stable situation [r4, [ ]] of A. As [s2, [ ]] is stable in B as well, but s2 is no final state of B, all conditions of Corollary 5.1.6 are violated. Hence, [r4, s2, [ ]] is a deadlock of A ⊕ B.
- Furthermore, due to the composed state [r3, s4, [b]], the knowledge of B about A at state s4 contains the stable situation [r3, [b]] of A. As [s4, [b]] is stable in B, but the multiset of pending messages is non-empty, all conditions of Corollary 5.1.6 are violated, and [r3, s4, [b]] is another deadlock of A ⊕ B.



(a) A (b) B (c) A ⊕ B

Figure 5.5: The two interface compatible service automata A and B of Fig. 3.9 and their open composition A ⊕ B. Corollary 5.1.6 can be used to reveal the two (internally reachable) deadlocks [r4, s2, [ ]] and [r3, s4, [b]] of A ⊕ B.

5.2 An Asymmetric Characterization of Strategies

So far we have seen how to decide whether the composition $A \oplus B$ of arbitrary interface compatible service automata A and B is deadlock-free. Therefore, we made use of the knowledge of B about A and formulated in Corollary 5.1.6 conditions which are sufficient and necessary to decide deadlock freedom of $A \oplus B$. The major drawback of this approach, however, is that the knowledge of B about A has to be computed for each considered B. That is, to decide deadlock freedom of $A \oplus B$ for a number of B's, the respective knowledge of B about A has to be computed separately for each B in order to check the conditions of Corollary 5.1.6 (cp. Fig. 5.1).

This section is devoted to speed up the check for several B's significantly. Therefore, we completely break the symmetry of the deadlock characterization by introducing a specific interface compatible service automaton $\mathcal{F}_A$ for a given service automaton A such that the knowledge of $\mathcal{F}_A$ about A and simple conditions between a service automaton B and $\mathcal{F}_A$ can be used to decide deadlock freedom of B's interaction with A.

For the construction of $\mathcal{F}_A$, we will restrict ourselves to the decision of the *strategy* relationship between a service automaton B and a given service automaton A only. That is, we consider only those service automata B for A where $A \oplus B$ is a *closed* service automaton. In other words, we restrict ourselves to decide the question $B \in \mathit{Strat}(A)$.

For service automata B such that $A \oplus B$ is closed, the knowledge of $\mathcal{F}_A$ about A and simple conditions on the simulation relation between the respective B and $\mathcal{F}_A$ suffice to decide whether or not B is a strategy for A. That way, the knowledge of the B's about A is not needed; only the knowledge of $\mathcal{F}_A$ about A has to be computed once and only the simulation relation between each B and $\mathcal{F}_A$ has to be checked for the conditions. The use of $\mathcal{F}_A$ to characterize the strategies for a given service automaton A is illustrated in Fig. 5.6. We claim that this is more efficient than checking deadlock freedom of each B with A according to Corollary 5.1.6 as illustrated in Fig. 5.1.

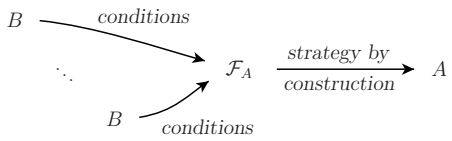


Figure 5.6: Schematic overview of an asymmetric strategy characterization between a service automaton A and several service automata B using the automaton $\mathcal{F}_A$ of A. Checking conditions on the simulation relation between B and $\mathcal{F}_A$ suffices to decide $B \in \mathit{Strat}(A)$.

However, it will turn out that $\mathcal{F}_A$ has two major drawbacks. Firstly, $\mathcal{F}_A$ is infinite for most service automata A. Secondly, the conditions which have to be checked to decide behavioral compatibility of B and A still use the knowledge of $\mathcal{F}_A$ about A and thus require detailed knowledge about the internal structure of A. Hence, $\mathcal{F}_A$ is not suited to be published as behavioral description by a service provider. The upcoming Sects. 5.3 and 5.4 are devoted to these issues. We will identify a finite part of $\mathcal{F}_A$ characterizing finite strategies B for A and present a translation of the conditions for verifying the strategy relationship into Boolean annotations to (the states of) $\mathcal{F}_A$ which will replace the knowledge of $\mathcal{F}_A$ about A. This finally results in a Boolean annotated service automaton (*BSA*) as introduced in Chap. 4, characterizing the set of (finite) strategies B for A. Thus, this *BSA* constitutes an operating guideline of A.

5.2.1 Operations on Knowledge Sets

For the construction of $\mathcal{F}_A$, we first introduce two operations on sets of situations of A. As these operations are applied to derive the knowledge $k_{(\mathcal{F}_A,A)}$ of $\mathcal{F}_A$ about A later on, we denote a set of situations by K in the following.

Definition 5.2.1 (Closure, *closure*(K)).
Let A be a service automaton and $K \subseteq \mathit{situations}(A)$ be a set of situations of A.

Then, the *closure* of K, $\mathit{closure}(K)$, is inductively defined as follows:

Basis. $K \subseteq \mathit{closure}(K)$;

Step. If $[q, M] \in \mathit{closure}(K)$ and $(q, x, q') \in \delta_A$, then
- $[q', M + x] \in \mathit{closure}(K)$ if $x \in I_{out\,A}$;
- $[q', M - x] \in \mathit{closure}(K)$ if $x \in I_{in\,A}$ and $x \in M$; and
- $[q', M] \in \mathit{closure}(K)$ if $x = \tau$. ⌟

Intuitively, the closure of a set of situations K of A considers each transient situation of A in K and adds all those situations of A to K that can be "reached" from the transient situation by A without input of B. Therefore, each sending transition of A, each x-labeled receiving transition where $x \in M$, and each internal transition of A is considered, and the respectively reached situation of A is added to $\mathit{closure}(K)$. It is important to notice that a message x is added to M by the closure construction only if x is an *output* channel of A.

Proposition 5.2.2 (Closure yields situations).
Let A be a service automaton and $K \subseteq \mathit{situations}(A)$ be a set of situations of A.

Then, $\mathit{closure}(K) \subseteq \mathit{situations}(A)$. ⌟

Obviously, $\mathit{closure}(\mathit{closure}(K)) = \mathit{closure}(K)$, for all K.

To exemplify the closure construction, consider again the service automaton A of Fig. 5.2(a). The closure of the singleton set $K = \{[\mathsf{r1}, [\,]]\}$ of situations of A is the set $\mathit{closure}(K) = \{[\mathsf{r1}, [\,]], [\mathsf{r2}, [\mathsf{a}]], [\mathsf{r4}, [\mathsf{b}]]\}$. This set $\mathit{closure}(K)$ can be found in Fig. 5.3(b) as the knowledge of B about A at state $\mathsf{s1}$. It is easy to see in Fig. 5.3(b) that each knowledge set $k_{(\mathsf{B},\mathsf{A})}(q)$ of a state q of B is the closure of any of its subsets, i.e. $k_{(\mathsf{B},\mathsf{A})}(q) = \mathit{closure}(K)$ for each $K \subseteq k_{(\mathsf{B},\mathsf{A})}(q)$.

The closure operation is the first operation to construct $\mathcal{F}_A$. The second operation is called *x-event*. Whereas closure is devoted to the construction of the knowledge values of $\mathcal{F}_A$, the x-event operation is needed to represent the effects of a communication action of $\mathcal{F}_A$ to a set of situations of A.

Definition 5.2.3 (Event, *event*(K, x)).
Let A be a service automaton, $K \subseteq \mathit{situations}(A)$ be a set of situations of A, and let $x \in I_{io\,A} \cup \{\tau\}$.

Then, the x-*event* of K in A, $event(K, x)$, is defined as

$$event(K,x) = \begin{cases} \{[q, M+x] \mid [q,M] \in K\}, & \text{if } x \in I_{in\,A}, \\ \{[q, M-x] \mid [q,M] \in K, x \in M\}, & \text{if } x \in I_{out\,A}, \\ K, & \text{otherwise, i.e. if } x = \tau. \end{cases}$$ ⌟

In contrast to the definition of closure, a message x is added to M by an x-event only if x is an *input* channel of A. Hence, the notion of an x-event expresses the effect of a strategy of A on a set of situations of A. For this reason, we will consider the ?a-event in A if a is an *output* channel of a service automaton A and the !b-event in A if b is an *input* channel of A.

Proposition 5.2.4 (Event yields situations).
Let A be a service automaton and $K \subseteq situations(A)$ be a set of situations of A. Then, $event(K, x) \subseteq situations(A)$, for each $x \in I_{io\,A} \cup \{\tau\}$. ⌟

To exemplify the event construction, consider the set $K = \{[\mathsf{r1}, [\,]], [\mathsf{r2}, [\mathsf{a}]], [\mathsf{r4}, [\mathsf{b}]]\}$ of situations of our example service automaton A of Fig. 5.2(a). The ?a-event of K in A is the set $K' = event(K, ?\mathsf{a}) = \{[\mathsf{r2}, [\,]]\}$, because a is an output channel of A, the only situation in K with $\mathsf{a} \in M$ is $[\mathsf{r2}, [\mathsf{a}]]$, and $[\mathsf{a}] - \mathsf{a} = [\,]$. It is easy to see in Fig. 5.3(b), that K is the knowledge of B about A at state s1, and K' is the knowledge of B about A at state s2, which is reached from s1 by the ?a-labeled transition of B. Furthermore, the !c-event of $K' = \{[\mathsf{r2}, [\,]]\}$ is $K'' = event(K', !\mathsf{c}) = \{[\mathsf{r2}, [\mathsf{c}]]\}$, which is a subset of the knowledge of B about A at state s3 in Fig. 5.3(b).

As another example, the !c-event of $K = \{[\mathsf{r1}, [\,]], [\mathsf{r2}, [\mathsf{a}]], [\mathsf{r4}, [\mathsf{b}]]\}$ in A is the set $event(K, !\mathsf{c}) = \{[\mathsf{r1}, [\mathsf{c}]], [\mathsf{r2}, [\mathsf{a}, \mathsf{c}]], [\mathsf{r4}, [\mathsf{b}, \mathsf{c}]]\}$, because c is an input channel of A and therefore c is added to each multiset M of a situation $[q, M]$ of K. The set $event(K, !\mathsf{c})$ is not depicted in Fig. 5.3(b) because there is no present !c-labeled transition in our example service automaton B at state s1 (see also Fig. 5.2(b)).

5.2.2 The Overapproximation $\mathcal{F}$ of Strategies

Having defined these two operations, we are now able to construct $\mathcal{F}_A$, representing an overapproximation of all strategies B for A. In fact, $\mathcal{F}_A$ represents any interface compatible service automaton B such that $A \oplus B$ is a closed service automaton, i.e. any *potential* strategy B for A.

Definition 5.2.5 (Overapproximation of strategies, $\mathcal{F}$).
The *overapproximation of strategies* for a service automaton A is defined as the service automaton $\mathcal{F}_A = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ with

- $Q = \{q_K \mid K \subseteq \mathit{situations}(A)\}$,
- $I_{in} = I_{out\,A}$,
- $I_{out} = I_{in\,A}$,
- $\delta = \{(q_K, x, q_{K'}) \mid K, K' \subseteq \mathit{situations}(A), x \in I_{io\,A} \cup \{\tau\}, K' = \mathit{closure}(\mathit{event}(K, x))\}$,
- $q_0 = q_{K_0}$ with $K_0 = \mathit{closure}(\{[q_{0\,A}, [\,]]\})$, and
- $\Omega = \{q_K \mid K \subseteq \mathit{situations}(A), [q, [\,]] \in K, q \in \Omega_A\}$. ⌟

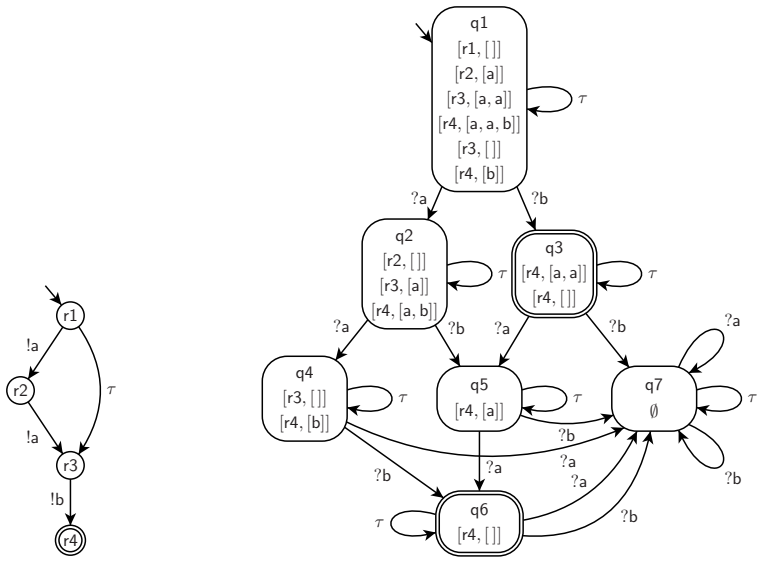
Notation 5.2.6.
We omit the index and write $\mathcal{F}$ instead of $\mathcal{F}_A$ if the considered service automaton A is clear from the context. ⌟

Obviously, the overapproximation $\mathcal{F}$ of strategies for A is interface compatible to A. Hence, we could immediately apply our strategy check according to Corollary 5.1.6 of the previous section to decide whether or not $\mathcal{F}$ itself is a strategy for A by computing the knowledge of $\mathcal{F}$ about A. As motivated above, however, we will employ $\mathcal{F}$ to classify (other) service automata B according to whether or not B is a strategy for A, i.e. to decide $B \in \mathit{Strat}(A)$. As the composition $A \oplus \mathcal{F}$ is even a closed service automaton, we will consider those service automata B that are interface equivalent to $\mathcal{F}$ (and hence $A \oplus B$ is closed as well). We will show that the knowledge of $\mathcal{F}$ about A and conditions on the simulation relation between the respective service automaton B and $\mathcal{F}$ can be used to decide $B \in \mathit{Strat}(A)$ without having to compute the knowledge of B about A.

It is easy to see that $\mathcal{F}$ usually has infinitely many states, as the set $\mathit{situations}(A)$ is usually infinite. Hence, there may be infinitely many sets $K \subseteq \mathit{situations}(A)$ and thus infinitely many states q_K. Even the set of δ-reachable states of $\mathcal{F}$ is usually infinite. If A has at least one input channel x, the corresponding $\mathcal{F}$ may perform the respective $!x$-event arbitrarily often, always incrementing the number of x elements in the set M of the situation $[q_{0\,A}, M]$ of A, for instance. The forthcoming Sect. 5.3 will be devoted to this issue. Therein, we identify a relevant finite part of $\mathcal{F}$ characterizing all finite strategies B for A.

To exemplify the overapproximation of strategies, Fig. 5.7(a) shows a new service automaton A. As A has only finitely many states and no input channel, the δ-reachable part of its overapproximation $\mathcal{F}_\mathsf{A}$ of strategies for A is finite. It is depicted in Fig. 5.7(b). $\mathcal{F}_\mathsf{A}$ is constructed according to Definition 5.2.5. The set K of a state q_K of $\mathcal{F}_\mathsf{A}$ is depicted inside the respective state q_K. Additionally, we assigned a new (i.e. short) name to each state q_K. For instance, the initial state q_{K_0} with $K_0 = \mathit{closure}(\{[q_{0\mathsf{A}}, [\,]]\})$ now bears the name $\mathsf{q1}$. Note that the τ-event at a state q_K of $\mathcal{F}_\mathsf{A}$ leads to the state q_K again, as $\mathit{event}(K, \tau) = K$, for all K.

Another special characteristic of $\mathcal{F}_\mathsf{A}$ can be seen at state $\mathsf{q7}$ in Fig. 5.7(b). As there is no situation $[q, M]$ with $\mathsf{b} \in M$ in the set K of e.g. the state $\mathsf{q3}$ (or $\mathsf{q5}$ or $\mathsf{q6}$),



(a) A (b) $\mathcal{F}_{\mathsf{A}}$

Figure 5.7: (a) A service automaton A and (b) its overapproximation $\mathcal{F}_{\mathsf{A}}$ of strategies for A. The set $K \subseteq \mathit{situations}(A)$ of a state q_K of $\mathcal{F}_{\mathsf{A}}$ is depicted inside the state q_K. According to Definition 5.2.5, the states q3 and q6 are final states of $\mathcal{F}_{\mathsf{A}}$, as the situation [r4, [ ]] is a knowledge value of the respective set K.

the respective ?b-event yields the empty set of situations (cp. Definition 5.2.3). As $\emptyset \subseteq \mathit{situations}(A)$, this *empty state* $q_\emptyset$ is a well-defined state of $\mathcal{F}_{\mathsf{A}}$ according to Definition 5.2.5.

We record that the empty state can only be reached in $\mathcal{F}$ by a receiving transition.

Proposition 5.2.7 (The empty state $q_\emptyset$ of $\mathcal{F}$).
Let $\mathcal{F}$ be the overapproximation of strategies for some service automaton A.

Then, the special state $q_\emptyset$, called the *empty state* of $\mathcal{F}$, is δ-reachable in $\mathcal{F}$ iff A has at least one sending transition. ⌟

By the definitions of closure and event, we furthermore conclude:

Proposition 5.2.8 (The empty state $q_\emptyset$ of $\mathcal{F}$ is persistent).
Let $\mathcal{F}$ be the overapproximation of strategies for some service automaton A, and let $q_\emptyset$ be the empty state of $\mathcal{F}$.

Then, each present transition at $q_\emptyset$ leads to $q_\emptyset$ again, i.e. for each transition $(q_\emptyset, x, q) \in \delta_{\mathcal{F}}$: $q = q_\emptyset$. ⌟

Before introducing the characterization of strategies by $\mathcal{F}$, we first collect some further important properties of $\mathcal{F}$ needed for the simulation relations later on. Afterwards we show that the knowledge of $\mathcal{F}$ about A is determined by the operations closure and event. Then, in the upcoming Sect. 5.2.3, we will show how to characterize strategies B for A with the help of the knowledge of $\mathcal{F}$ about A and the simulation relation between B and $\mathcal{F}$.

Lemma 5.2.9 ($\mathcal{F}$ is complete).
Let A be a service automaton and let $\mathcal{F}$ be its overapproximation of strategies.

Then, for each state q of $\mathcal{F}$ and each message channel $x \in I_{io\mathcal{F}} \cup \{\tau\}$ there is exactly one state q' of $\mathcal{F}$ with $(q, x, q') \in \delta_{\mathcal{F}}$. ⌟

Proof.
Let $K \subseteq \mathit{situations}(A)$ be such that $q = q_K$ and let $x \in I_{io\mathcal{F}} \cup \{\tau\}$ be arbitrary. By Propositions 5.2.2 and 5.2.4, $K' = \mathit{closure}(\mathit{event}(K, x)) \subseteq \mathit{situations}(A)$. Hence, by the construction of $\mathcal{F}$, there is a state $q_{K'}$ of $\mathcal{F}$ and a transition $(q_K, x, q_{K'}) \in \delta_{\mathcal{F}}$. Again by the construction of $\mathcal{F}$ and because K' is uniquely defined by K and x, there is no other present x-labeled transition at q_K. □

Remark 5.2.10.
The letter $\mathcal{F}$, used to denote the overapproximation of strategies for a service automaton A, already suggests the *full* usage of message channels at each state of $\mathcal{F}$. ⌟

Lemma 5.2.9 immediately implies:

Corollary 5.2.11 ($\mathcal{F}$ simulates all B).
Let A be a service automaton and let $\mathcal{F}$ be its overapproximation of strategies.

Then, $\mathcal{F}$ simulates each service automaton B such that $A \oplus B$ is closed. ⌟

That is, the structure of $\mathcal{F}$ is rich enough to comprise any strategy B for A, expressed by the respective simulation relation between B and $\mathcal{F}$. This simulation relation provides the basis for the characterization of strategies for A in the upcoming section.

Furthermore, Lemma 5.2.9 yields:

Corollary 5.2.12 ($\mathcal{F}$ is deterministic).
The overapproximation $\mathcal{F}$ of strategies for a service automaton A is deterministic. ⌟

Determinism of $\mathcal{F}$ is crucial for an efficient computation of simulation relations of some other service automaton B with $\mathcal{F}$.

The following lemma states that the knowledge of $\mathcal{F}$ about A is directly correlated to the construction of the states of $\mathcal{F}$ by the sets $K \subseteq \mathit{situations}(A)$ of situations of A, i.e. for each state q_K of $\mathcal{F}$, the knowledge of $\mathcal{F}$ at q_K is the set K itself.

Lemma 5.2.13 (Knowledge of $\mathcal{F}$ determined by closure and event).
Let A be a service automaton and let $\mathcal{F}$ be its overapproximation of strategies.

Then, for each δ-reachable state q_K of $\mathcal{F}$: $k_{(\mathcal{F},A)}(q_K) = K$. ⌟

Proof.
We abbreviate $k_{(\mathcal{F},A)}$ by k and show $k(q_K) = K$ by structural induction on δ in $\mathcal{F}$. Please notice that $A \oplus \mathcal{F}$ is closed, i.e. $A \oplus \mathcal{F}$ has only internal transitions. Hence, a state $[q_A, q_K, M]$ is δ-reachable in $A \oplus \mathcal{F}$ iff $[q_A, M] \in k(q_K)$.

Basis. We show $k(q_{K_0}) = K_0$ for the initial state q_{K_0} of $\mathcal{F}$. By definition of $k(q_{K_0})$ and because $K_0 = \mathit{closure}(\{[q_{0A}, [\,]]\})$, it is sufficient to show $[q_A, q_{K_0}, M] \in Q_{A\oplus\mathcal{F}}$ iff $[q_A, M] \in \mathit{closure}(\{[q_{0A}, [\,]]\})$.
This obviously holds for the initial state $[q_{0A}, q_{K_0}, [\,]]$ of $A \oplus \mathcal{F}$, as (by definition of closure) $[q_{0A}, [\,]] \in \mathit{closure}(\{[q_{0A}, [\,]]\})$. As (by definition of $\mathcal{F}$) $\mathcal{F}$ and A have no free channels, the cases of adding a situation to $\mathit{closure}(\{[q_{0A}, [\,]]\})$ correspond exactly to the (remaining) three cases for transitions in $A \oplus \mathcal{F}$ (cp. Definition 3.3.18).

Step. Let $k(q_K) = K$ and let $(q_K, x, q_{K'}) \in \delta$. We show $k(q_{K'}) = K'$ by showing $k(q_{K'}) \subseteq K'$ and $k(q_{K'}) \supseteq K'$.

($\subseteq$): Let $[q_A, M] \in k(q_{K'})$ be an arbitrary situation of A. Hence, $[q_A, q_{K'}, M] \in Q_{A\oplus\mathcal{F}}$ is a δ-reachable state in $A \oplus \mathcal{F}$. By the construction of $A \oplus \mathcal{F}$ there exists a δ-reachable state $[q_A^*, q_K, M^*]$ in $A \oplus \mathcal{F}$ such that $[q_A, q_{K'}, M]$ is δ-reachable from $[q_A^*, q_K, M^*]$ in $A \oplus \mathcal{F}$ by following a transition $([q_A^*, q_K, M^*], \tau, [q_A^*, q_K', M^{**}])$ in $A \oplus \mathcal{F}$ which corresponds to the transition $(q_K, x, q_{K'})$ in $\mathcal{F}$, followed by a sequence of transitions that all correspond to A from $[q_A^*, q_K', M^{**}]$ to $[q_A, q_{K'}, M]$.

From $[q_A^*, q_K, M^*]$ being δ-reachable in $A \oplus \mathcal{F}$, we have $[q_A^*, M^*] \in k(q_K)$; and by the assumption $k(q_K) = K$ we know $[q_A^*, M^*] \in K$.
By the definition of event, we derive $[q_A^*, M^{**}] \in \mathit{event}(K, x)$. By the definition of closure we get $[q_A, M] \in \mathit{closure}(\mathit{event}(K, x))$. Finally, with $K' = \mathit{closure}(\mathit{event}(K, x))$, we know $[q_A, M] \in K'$.

($\supseteq$): Let $[q_A, M] \in K'$ be an arbitrary situation of A. By $(q_K, x, q_{K'}) \in \delta$ (from the assumption) and the construction of $\mathcal{F}$, we know that $K' = \mathit{closure}(\mathit{event}(K, x))$. Hence, there must exist situations $[q_A^*, M^*]$ and $[q_A^*, M^{**}]$ of A with $[q_A^*, M^*] \in K$, $[q_A^*, M^{**}] \in \mathit{event}(\{[q_A^*, M^*]\}, x)$, and $[q_A, M] \in \mathit{closure}(\{[q_A^*, M^{**}]\})$.
From $[q_A^*, M^*] \in K$ follows (with the assumption $k(q_K) = K$) that $[q_A^*, q_K, M^*] \in Q_{A\oplus\mathcal{F}}$. By the construction of event, we hence conclude that $[q_A^*, q_K', M^{**}] \in Q_{A\oplus\mathcal{F}}$. By the construction of closure, we finally derive $[q_A, q_K', M] \in Q_{A\oplus\mathcal{F}}$. □

The main value of this lemma is that we do not have to construct the composition $A \oplus \mathcal{F}$ to derive the knowledge of $\mathcal{F}$ about A — the constructions closure and event

are sufficient for deriving all knowledge sets. Furthermore, the construction of a successor state $q_{K'}$ of q_K by the property $K' = \mathit{closure}(\mathit{event}(K, x))$ suggests the successive computation of (the internally reachable part of) $\mathcal{F}$ starting from q_{K_0} with $K_0 = \mathit{closure}(\{[q_{0\,A}, [\,]]\})$, instead of first generating all states of $\mathcal{F}$ and then connecting all those pairs of states q_K, $q_{K'}$ by a transition $(q_K, x, q_{K'})$ which fulfill the property $K' = \mathit{closure}(\mathit{event}(K, x))$.

Lemma 5.2.13 can easily be verified for the example overapproximation $\mathcal{F}_{\mathsf{A}}$ of strategies for the service automaton A of Fig. 5.7(a), depicted in Fig. 5.7(b). Recall that the set K of a state q_K of $\mathcal{F}_{\mathsf{A}}$ is depicted inside the state q_K. For each state q_K, the set K is equal to the knowledge of $\mathcal{F}_{\mathsf{A}}$ about A at q_K. The construction of the composition $\mathsf{A} \oplus \mathcal{F}_{\mathsf{A}}$ of A and $\mathcal{F}_{\mathsf{A}}$, however, is left as an exercise for the interested reader.

The property of Lemma 5.2.13 justifies the graphical representation of the set K of a state q_K inside the respective state in Fig. 5.7 as it was already done earlier for the knowledge at the state.

By Lemma 5.2.13, together with the efficient computation of the simulation relation between a service automaton B and $\mathcal{F}$ due to the deterministic structure of $\mathcal{F}$, we conclude that $\mathcal{F}$ is a well-suited candidate to decide $B \in \mathit{Strat}(A)$.

5.2.3 A Characterization of Strategies by $\mathcal{F}$

In the following, we show how the overapproximation $\mathcal{F}$ of strategies for a service automaton A can be used to decide whether or not $B \in \mathit{Strat}(A)$ for arbitrary B.

To this end, the following lemma will generalize the result of Lemma 5.2.13 and basically says that the knowledge $k_{(\mathcal{F},A)}$ of $\mathcal{F}$ about A can be used to deduce the knowledge $k_{(B,A)}$ of another service automaton B about A. Therefore, only the simulation relation between B and $\mathcal{F}$ has to be computed. This means that we can apply Corollary 5.1.6 also for deciding whether or not B is a strategy for A without actually computing $k_{(B,A)}$ from the composition $A \oplus B$.

Lemma 5.2.14 (Knowledge of B is union of knowledge of $\mathcal{F}$).
Let $\mathcal{F}$ be the overapproximation of strategies for a service automaton A, let B be a service automaton that is interface equivalent to $\mathcal{F}$, and let $\varrho \subseteq Q_B \times Q_{\mathcal{F}}$ be the minimal simulation relation between B and $\mathcal{F}$.

Then, $k_{(B,A)}(q_B) = \bigcup_{(q_B, q_{\mathcal{F}}) \in \varrho} k_{(\mathcal{F},A)}(q_{\mathcal{F}})$, for each state q_B of B. ⌟

Proof.
We abbreviate $k_{(B,A)}$ by k_B and $k_{(\mathcal{F},A)}$ by $k_{\mathcal{F}}$. Notice that both $A \oplus B$ and $A \oplus \mathcal{F}$ are closed. We show $k_B(q_B) = \bigcup_{(q_B, q_{\mathcal{F}}) \in \varrho} k_{\mathcal{F}}(q_{\mathcal{F}})$ by showing $k_B(q_B) \subseteq \bigcup_{(q_B, q_{\mathcal{F}}) \in \varrho} k_{\mathcal{F}}(q_{\mathcal{F}})$ and $\bigcup_{(q_B, q_{\mathcal{F}}) \in \varrho} k_{\mathcal{F}}(q_{\mathcal{F}}) \subseteq k_B(q_B)$.

($\subseteq$): We show $k_B(q_B) \subseteq \bigcup_{(q_B,q_\mathcal{F})\in\varrho} k_\mathcal{F}(q_\mathcal{F})$. Let therefore q_B be an arbitrary state of B and let $[q_A, M] \in k_B(q_B)$. It suffices to show that there exists a state $q_\mathcal{F}$ of $\mathcal{F}$ with $(q_B, q_\mathcal{F}) \in \varrho$ such that $[q_A, M] \in k_\mathcal{F}(q_\mathcal{F})$.

From $[q_A, M] \in k_B(q_B)$ we know that the state $[q_A, q_B, M]$ of $A \oplus B$ is reachable in $A \oplus B$. Let σ be an arbitrary sequence of (all internal) transitions of $A \oplus B$ such that $[q_A, q_B, M]$ is reached from the initial state of $A \oplus B$ by σ. Consider now the transitions $d_1, \ldots, d_n$ in σ that correspond to transitions of B and let the corresponding labels of these transitions in B be $x_1, \ldots, x_n$.

Because $\mathcal{F}$ is full (according to Lemma 5.2.9), it can perform any x-labeled transition at any of its states. Hence, we can construct a sequence σ' in the composition $A \oplus \mathcal{F}$ by replacing each transition d_i of B in σ by an equally labeled transition of $\mathcal{F}$. By the definition of service automata composition, following σ' in $A \oplus \mathcal{F}$ reaches a state $[q_A, q_\mathcal{F}, M]$ of $A \oplus \mathcal{F}$. Hence, $[q_A, M] \in k_\mathcal{F}(q_\mathcal{F})$. Due to the construction of σ', we further know that $(q_B, q_\mathcal{F}) \in \varrho$.

($\supseteq$): We show $\bigcup_{(q_B,q_\mathcal{F})\in\varrho} k_\mathcal{F}(q_\mathcal{F}) \subseteq k_B(q_B)$. Let therefore $q_\mathcal{F}$ be an arbitrary state of $\mathcal{F}$ such that $(q_B, q_\mathcal{F}) \in \varrho$ and let $[q_A, M] \in k_\mathcal{F}(q_\mathcal{F})$ be arbitrary. It suffices to show $[q_A, M] \in k_B(q_B)$.

As $(q_B, q_\mathcal{F}) \in \varrho$ and because ϱ is minimal, there exists a sequence of transition labels $x_1, \ldots, x_n$ such that q_B ($q_\mathcal{F}$, respectively) is reached in B ($\mathcal{F}$, respectively) from the initial state of B ($\mathcal{F}$, respectively) by following correspondingly labeled transitions in B ($\mathcal{F}$, respectively).

From $[q_A, M] \in k_\mathcal{F}(q_\mathcal{F})$ and Lemma 5.2.13, we know that $[q_A, M]$ can be constructed by applying the operations closure and event. By the definition of closure and event, their application defines a sequence σ of transitions of $A \oplus \mathcal{F}$ such that the state $[q_A, q_\mathcal{F}, M]$ of $A \oplus \mathcal{F}$ is reached from the initial state of $A \oplus \mathcal{F}$. Thereby, the application of closure corresponds to following transitions of A, and the application of event corresponds to following transitions of $\mathcal{F}$. As the state $q_\mathcal{F}$ can be reached in $\mathcal{F}$ by a sequence of transitions labeled $x_1, \ldots, x_n$, there even exists such a sequence σ in $A \oplus \mathcal{F}$, where the event operations are performed with message channels $x_1, \ldots, x_n$. Hence, the transitions that corresponding to $\mathcal{F}$ in this sequence σ are labeled $x_1, \ldots, x_n$ in $\mathcal{F}$.

As the state q_B of B can be reached by following transitions labeled $x_1, \ldots, x_n$ in B as well, we can (by the definition of service automata composition) replace each transition of $\mathcal{F}$ in σ by an equally labeled transition of B and yield a transition sequence σ' in $A \oplus B$ reaching $[q_A, q_B, M]$ in $A \oplus B$. Thus, $[q_A, M] \in k_B(q_B)$. □

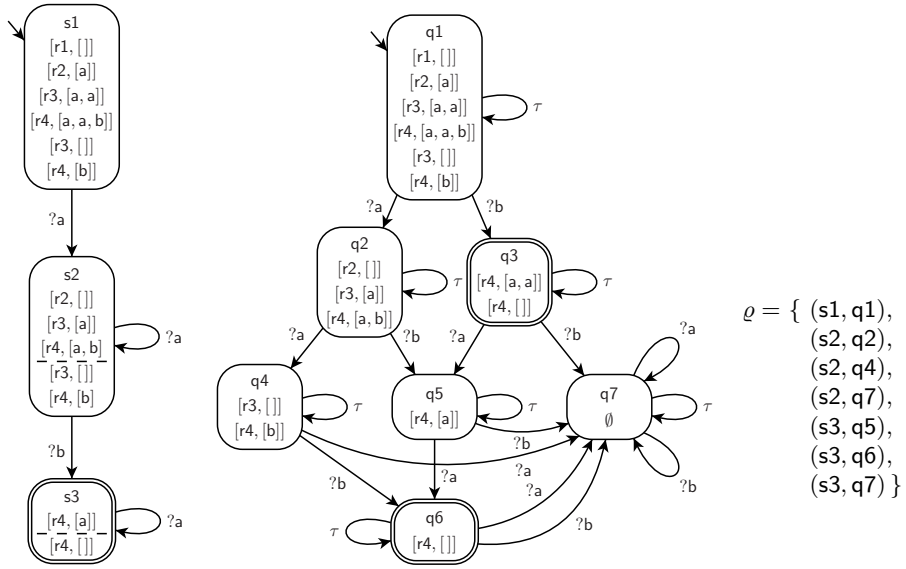
That is, the knowledge of a state q of B is equal to the union of the knowledge values of all those states $q_\mathcal{F}$ of $\mathcal{F}$ which are used in the simulation relation between B and $\mathcal{F}$ (under the assumption that B is interface equivalent to $\mathcal{F}$).

As an example, consider the service automaton B of Fig. 5.8(a) with its knowledge about the service automaton A of Fig. 5.7(a). Figure 5.8(c) shows the

minimal simulation relation ϱ between (the states of) B and $\mathcal{F}_\mathsf{A}$. According to Lemma 5.2.14, the knowledge of B about A at a state q_B of B is the union of all states q of $\mathcal{F}_\mathsf{A}$ where $(q_\mathsf{B}, q) \in \varrho$. We verify this property for the state s2 of B. From Fig. 5.8(c), we have $(\mathsf{s2}, \mathsf{q2})$, $(\mathsf{s2}, \mathsf{q4})$, and $(\mathsf{s2}, \mathsf{q7})$ in ϱ. Hence:

$$k_{(\mathsf{B},\mathsf{A})}(\mathsf{s2}) =$$
$$k_{(\mathcal{F}_\mathsf{A},\mathsf{A})}(\mathsf{q2}) \cup k_{(\mathcal{F}_\mathsf{A},\mathsf{A})}(\mathsf{q4}) \cup k_{(\mathcal{F}_\mathsf{A},\mathsf{A})}(\mathsf{q7}) =$$
$$\{[\mathsf{r2}, [\,]], [\mathsf{r3}, [\mathsf{a}]], [\mathsf{r4}, [\mathsf{a}, \mathsf{b}]]\} \cup \{[\mathsf{r3}, [\,]], [\mathsf{r4}, [\mathsf{b}]]\} \cup \emptyset =$$
$$\{[\mathsf{r2}, [\,]], [\mathsf{r3}, [\mathsf{a}]], [\mathsf{r4}, [\mathsf{a}, \mathsf{b}]], [\mathsf{r3}, [\,]], [\mathsf{r4}, [\mathsf{b}]]\},$$

which exactly corresponds to the knowledge depicted inside the state s2 of B in Fig. 5.8(a).



(a) B with knowledge about A

(b) $\mathcal{F}_\mathsf{A}$

(c) ϱ

Figure 5.8: (a) A service automaton B with its knowledge about the service automaton A of Fig. 5.7(a). (b) The overapproximation $\mathcal{F}_\mathsf{A}$ of strategies for A, taken from Fig. 5.7(b). (c) The minimal simulation relation between B and $\mathcal{F}_\mathsf{A}$ such that $\mathcal{F}_\mathsf{A}$ simulates B. The dashed line inside the knowledge of a state in Fig. 5.8(a) visualizes the union of the knowledge sets of the different states of $\mathcal{F}_\mathsf{A}$.

Figure 5.8 also points out another advantage of the construction of $\mathcal{F}$. While the knowledge of $\mathcal{F}$ about A can be computed by the simple operations closure and event, the knowledge of some B about A can (in general) not be derived that

simple. For example, the knowledge of the state s2 of the service automaton B of Fig. 5.8(a) is *not* equal to the closure of the ?a-event of the knowledge of its state s1, i.e. $k_{(\mathsf{B},\mathsf{A})}(\mathsf{s2}) \supset \mathit{closure}(\mathit{event}(k_{(\mathsf{B},\mathsf{A})}(\mathsf{s1}), ?\mathsf{a}))$. However, the knowledge at s2 can be deduced from the simulation relation between B and $\mathcal{F}_\mathsf{A}$ and the knowledge of $\mathcal{F}_\mathsf{A}$.

Unfortunately, Lemma 5.2.14 does, in general, not hold for open compositions. For instance, assume a slightly changed service automaton B' of our example service automaton B of Fig. 5.8(a) having only the states s1 and s2, and let the message channel b be not in the interface of B'. Hence, b is free between A and B' and is never added to the multiset M of pending messages in $\mathsf{A} \oplus \mathsf{B}'$ (cp. Definition 3.3.18). Thus, the knowledge of B' at its state s2 does not contain the situations $[\mathsf{r4}, [\mathsf{a}, \mathsf{b}]]$ and $[\mathsf{r4}, [\mathsf{b}]]$, but the situations $[\mathsf{r4}, [\mathsf{a}]]$ and $[\mathsf{r4}, [\,]]$ instead. Hence, it is *not* the union of the knowledge sets of $\mathcal{F}_\mathsf{A}$ at its states q2, q4, and q7.

The result of Lemma 5.2.14 for closed compositions $A \oplus B$, however, is sufficient to formalize the characterization of strategies for a service automaton A by using the knowledge of $\mathcal{F}$ about A.

Theorem 5.2.15 (Characterization of strategies for A by $\mathcal{F}$).
Let A be a service automaton and $\mathcal{F}$ be the overapproximation of strategies for A. Let furthermore B be a service automaton which is interface equivalent to $\mathcal{F}$ and $\varrho \subseteq Q_B \times Q_\mathcal{F}$ be the minimal simulation relation between B and $\mathcal{F}$.

Then, B is a strategy for A iff for all states q_B of B, all $(q_B, q_\mathcal{F}) \in \varrho$, and all situations $[q_A, M]$ of A with $[q_A, M] \in k_{(\mathcal{F},A)}(q_\mathcal{F})$, at least one of the following conditions is fulfilled:

(i) $[q_A, M]$ is transient in A or

(ii) the situation $[q_B, M]$ of B is transient in B or

(iii) $q_A \in \Omega_A$ and $q_B \in \Omega_B$ and $M = [\,]$. ⌟

Proof.
The theorem is mainly a reformulation of Corollary 5.1.6 using the property of Lemma 5.2.14.

Since, according to Lemma 5.2.14, $[q_A, M] \in k_{(B,A)}(q_B)$ iff $[q_A, M] \in k_{(\mathcal{F},A)}(q_\mathcal{F})$ for some $(q_B, q_\mathcal{F}) \in \varrho$, we can immediately apply Corollary 5.1.6 to infer deadlock freedom of $A \oplus B$.

From the assumption that B and $\mathcal{F}$ are interface equivalent, we conclude that $A \oplus B$ is a closed service automaton. Hence, B is a strategy for A iff at least one condition is fulfilled for each situation of A in the knowledge sets of $\mathcal{F}$. □

Due to the construction of the overapproximation $\mathcal{F}$ for A, the interface equivalence requirement between B and $\mathcal{F}$ in the latter theorem implies that the composition of A and B is closed. Hence, Theorem 5.2.15 is a characterization of

strategies B for A by using the overapproximation $\mathcal{F}$ of strategies for A. That is, to decide well-behavior of $A \oplus B$, we only have to check conditions between B and $\mathcal{F}$, A is not needed anymore once $\mathcal{F}$ is computed.

Without this theorem, the knowledge of B about A was needed to decide $B \in \mathit{Strat}(A)$, which simply means that we had to construct the composition $A \oplus B$. With this theorem, the knowledge of B about A can be derived from the knowledge of $\mathcal{F}$ about A and the simulation relation between B and $\mathcal{F}$ — without constructing $A \oplus B$. Because the knowledge of $\mathcal{F}$ about A can be derived by the simple operations closure and event, and the simulation relation between B and $\mathcal{F}$ can be computed efficiently due to the deterministic structure of $\mathcal{F}$, this strategy characterization is feasible.

For the characterization of strategies according to Theorem 5.2.15, the final states of $\mathcal{F}$ are not considered and therefore have no relevance. We have seen a similar approach already in Sect. 4.2.2, where the final states of a *BSA* B^{ϕ} are completely irrelevant for the matching of a service automaton C with B^{ϕ}. According to Definition 5.2.5, a state q_K of $\mathcal{F}$ is final if the corresponding set of situations K contains a situation $[q_A, [\,]]$ of A representing a final state q_A of A and empty message channels. That way, the composed state $[q_A, q_K, [\,]]$ is a final state of $A \oplus \mathcal{F}$. This, however, is only a beauty constraint and not relevant for deciding $B \in \mathit{Strat}(A)$.

As Theorem 5.2.15 is based on Lemma 5.2.14, it inherits the restriction of the lemma to closed compositions $A \oplus B$ (assured by the condition that B is interface equivalent to $\mathcal{F}$). This restriction results in the characterization of *strategies* for a service automaton A only, instead of characterizing arbitrary service automata B such that $A \oplus B$ is deadlock-free in Theorem 5.2.15. However, as already shown in Sect. 3.5.2, this does not restrict generality of our approach as every pair of services can be transformed in a way such that their composition is closed.

For a practical application of our strategy characterization for a service according to Theorem 5.2.15, two other issues with $\mathcal{F}$ have to be considered. Firstly, $\mathcal{F}$ may still have infinitely many δ-reachable states. Hence, an implementation of the construction of $\mathcal{F}$ according to Definition 5.2.5 would not terminate in such a case and thus, the decision procedure introduced by the latter theorem is useless for practical purposes. Secondly, the knowledge of $\mathcal{F}$ is no real abstraction of the structure of A, and most of the states and transitions of A can usually be reconstructed from $\mathcal{F}$ (see again the example of Fig. 5.4 and the corresponding discussion).

The upcoming Sects. 5.3 and 5.4 are devoted to these issues. In Sect. 5.3, we will restrict our strategy characterization to finite-state services. Mainly, this restriction will enable us to represent the conditions of Theorem 5.2.15 by Boolean formulae in Sect. 5.4. Therefore, we can attach to each state q of $\mathcal{F}$ a Boolean formula $\phi(q)$, which expresses the requirements that a state q_B of a service automaton B must satisfy for B being a strategy for A. The formulae replace the knowledge

of $\mathcal{F}$, which is discarded once the annotations have been derived. That way, we get a Boolean annotated service automaton (*BSA*) $\mathcal{F}^\phi$ as introduced in Chap. 4 such that $\mathcal{F}^\phi$ characterizes the set of strategies of A, i.e. $Match(\mathcal{F}^\phi) = Strat(A)$. Hence, this *BSA* serves as an operating guideline OG_A of A.

5.3 Restriction to Finite-State Services

The results of the previous section enable us to characterize strategies B for a service automaton A with the help of the overapproximation $\mathcal{F}$ of strategies for A. However, the presented decision procedure of Theorem 5.2.15, which is based on checking special conditions for the simulation relation between B and $\mathcal{F}$, is only of theoretical nature. The fundamental construct used in this theorem, the overapproximation $\mathcal{F}$, is usually infinite and the need of its construction is therefore obviously not feasible for practical purposes.

Furthermore, we were able to show that controllability is even *undecidable* for services with infinite state space [MSSW08]. However, we considered a stronger correctness criterion, *weak termination*, in [MSSW08] than the one, i.e. deadlock freedom, used in this thesis.

In the rest of this thesis, we follow the approach of [LMW07b] and restrict ourselves to the characterization of those strategies B for A where the composition $A \oplus B$ is a finite service automaton (cp. Definition 3.3.6), which we claim to be a realistic assumption in practice. To this end, we assume that all considered service automata A and B are finite. Unfortunately, the composition $A \oplus B$ of two finite service automata A and B is not necessarily finite itself, because infinitely many pending messages may be accumulated between A and B. Hence, we additionally introduce a *message bound* b, basically limiting the number of pending messages that each message channel $x \in \mathcal{MC}$ can carry, and demand *b-bounded communication* between A and B in the following. That way, the composition $A \oplus B$ will be guaranteed to have only finitely many reachable states, i.e. be finite as well. Then, we may canonically derive the notion of a *b-strategy* for a service. In the upcoming Sect. 5.4, we will finally show that there is a finite part of $\mathcal{F}$ characterizing all b-strategies B for A — which therefore constitutes the basis of the operating guideline OG_A of a service automaton A.

Recall that $bags_b(\mathcal{MC})$ denotes the set of all those multisets over $\mathcal{MC}$ where each $x \in \mathcal{MC}$ occurs at most b times (cp. Sect. 3.1).

Definition 5.3.1 (*b*-bounded communication, message bound *b*).
Let A and B be two service automata and let $b \in \mathbb{N}$ be a given *message bound*.

Then, A and B *communicate b-bounded* if $Q_{A \oplus B} \subseteq Q_A \times Q_B \times bags_b(\mathcal{MC})$. ⌟

Please notice that b-bounded communication between A and B alone is also not sufficient to ensure a finite composition $A \oplus B$. Only the combination of requiring finite service automata A and B and b-bounded communication results in finiteness of $A \oplus B$.

Proposition 5.3.2 (Finite composition).
If two service automata A and B are finite and communicate b-bounded, for some message bound $b \in \mathbb{N}$, then their composition $A \oplus B$ is finite. ⌟

The reason for the latter proposition is as follows. By Q_A and Q_B being finite and $Q_{A \oplus B} \subseteq Q_A \times Q_B \times \mathit{bags}_b(\mathcal{MC})$ due to b-bounded communication, we know that $Q_{A \oplus B}$ is obviously finite as well. Hence, we conclude that $A \oplus B$ is a finite service automaton.

Remark 5.3.3.
Technically, the latter proposition holds in both directions, i.e. it also holds: if a composition $A \oplus B$ is finite, then the two service automata A and B are finite and communicate b-bounded, for some message bound $b \in \mathbb{N}$. The reason for this property lies in the composition definition according to Definition 3.3.18, where we defined the set of states of the composition $A \oplus B$ of A and B to be *equal* to $Q_A \times Q_B \times \mathit{bags}(I_{io\,A} \cap I_{io\,B})$.

However, it is possible that the set of internally reachable states of $A \oplus B$ may be finite although the sets of states of A or B are infinite, i.e. it is possible that the infinite behavior of A and B is restricted by their communication. Nevertheless, requiring finite service automata is reasonable in practice. ⌟

We assume the message bound b to be given a priori. This number can stem from practical considerations like limited capacity of the physical message channels, it can be agreed upon between the parties of a cooperation before they start to interact, it may be calculated by prior statical analysis of the communication capabilities of the services, or the number can just be chosen arbitrarily.

With the notions of finite service automata and b-bounded communication, we may now canonically derive the notion of a b-strategy B for A.

Definition 5.3.4 (b-strategy, $\mathit{Strat}_b(A)$).
A service automaton B is a *b-strategy* for a service automaton A if

- A and B are finite,
- A and B communicate b-bounded, and
- $A \oplus B$ is a closed well-behaving service automaton.

Let $\mathit{Strat}_b(A)$ denote the set of all b-strategies for A. ⌟

Analogously, the notion of controllability of a service automaton A extends to the notion of b-controllability of A.

Definition 5.3.5 (*b*-controllability).
A service automaton A is *b-controllable* for some given message bound $b \in \mathbb{N}$ if there exists a b-strategy B for A. ⌟

Obviously, 0-controllability forbids any message exchange and results in merely putting A and B side by side without any connection.

It is important to notice that each b-strategy B for a service A is a $(b+1)$-strategy for A as well. Hence, a b-controllable service is also b'-controllable, for any $b' > b$. On the other hand, for every message bound $b \geq 1$, there are services which are b-controllable, but not $(b-1)$-controllable. As an example, consider a service A which sends b many messages x and then rests in its final state. It is b-controllable by a mirrored service B receiving all x messages. However, as A sends b many messages already by itself, A is not $(b-1)$-controllable.

There are even services which are controllable but not b-controllable, for any b. As an example, consider another service automaton A that has one (initial and final) state q and one sending transition $(q, !x, q)$. Hence, A sends an unbounded number of x's by itself. Again, the mirrored service B, which has one (initial and final) state s and one transition $(s, ?x, s)$, is a strategy for A as their composition is obviously deadlock-free. However, for each number b, there is a reachable state of the composition with more than b many x messages pending.

Just as for arbitrary strategies, the set $Strat_b(A)$ is infinite if and only if A is b-controllable, even in the restricted setting of considering only b-strategies. Analogously, $Strat_b(A)$ is empty if and only if A is b-uncontrollable.

All presented service automata in this thesis, as well as their respective compositions are finite. For example, the service automata A and B of Fig. 5.2 communicate 1-bounded, as can be seen in their composition in Fig. 5.3(a). Hence, A is b-controllable, for each $b \geq 1$.

The service automata A of Fig. 5.7(a) and B of Fig. 5.8(a) communicate 2-bounded, which can be seen in the knowledge of the state s1 of B where two a messages are pending between A and B. Hence, A is b-controllable, for each $b \geq 2$. As A sends two a messages already by its own, it is obviously not 1-controllable.

Recall the definition of an operating guideline of a service automaton A given at the beginning of this chapter in Definition 5.0.1. With the introduction of the message bound b we derive the notion of b-operating guidelines, characterizing all b-strategies, for a given message bound b.

Definition 5.3.6 (b-operating guideline, OG_A^b).
A *BSA* B^ϕ is called *b-operating guideline* for a service automaton A, denoted OG_A^b, if $Match(B^\phi) = Strat_b(A)$. ⌟

As done before, we may omit the index A and write OG^b if the considered service automaton A is clear from the context. Furthermore, we will always assume some

message bound $b \in \mathbb{N}$ to be given in the following. Consequently, we may also omit the bound b and write OG_A for an b-operating guideline of A for *some* message bound b.

Before continuing with the characterization of all b-strategies for a service, we derive the canonical notion of a *b-situation* of a service automaton.

Definition 5.3.7 (b-situation, $situations_b(A)$).
Let A be a service automaton, let q_A be a state of A, and let $M \in bags_b(I_{io\,A})$ be a multiset of messages where each $x \in I_{io\,A}$ occurs at most b times.

Then, $[q_A, M]$ is a *b-situation* of A. Furthermore, let $situations_b(A) = Q_A \times bags_b(I_{io\,A})$ denote the set of all b-situations of A. ⌟

The set of b-situations of A will be employed in the characterization of b-strategies in the upcoming section.

5.4 An Operating Guideline Characterization of Strategies

In this section, we build upon the characterization of (arbitrary) strategies according to Theorem 5.2.15 and aim at characterizing all finite strategies that communicate b-bounded with a given service automaton A (for some given message bound b). To this end, we will show that a specific part of $\mathcal{F}$ is sufficient to characterize all b-strategies. We will denote this part of $\mathcal{F}$ by $\mathcal{F}^b$ and show that $\mathcal{F}^b$ is always finite, i.e. it has a finite set of states and, for each state q_K of $\mathcal{F}^b$, K is finite. For this reason, the restriction to b-strategies enables us to translate the conditions of Theorem 5.2.15 into Boolean formulae, expressing the requirements that some B has to fulfill to be a b-strategy for A. That way, we get a Boolean annotated service automaton B^ϕ (as introduced in Chap. 4), characterizing all b-strategies for A, i.e. $Match(B^\phi) = Strat_b(A)$. It is thus called *$b$-operating guideline* OG^b_A for A. By introducing OG^b_A as an abstraction from the conditions in Fig. 5.6, we derive the operating guideline approach to decide $B \in Strat_b(A)$ for a service automaton A by matching B with OG^b_A as depicted in Fig. 5.9.

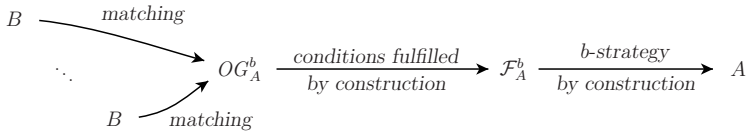


Figure 5.9: Schematic overview of a strategy characterization using the b-operating guideline OG^b_A of A. Matching B with OG^b_A according to the *BSA* matching of Sect. 4.2 suffices to decide $B \in Strat_b(A)$.

5.4.1 A Characterization of b-Strategies by $\mathcal{F}^b$

We start with presenting a slightly changed definition of the overapproximation $\mathcal{F}$ of strategies for a service automaton A, causing $\mathcal{F}$ to have finitely many states and finite knowledge at each state. Beforehand, we motivate a precondition for the changed construction of $\mathcal{F}$ in the following lemma. It provides a sufficient criterion for b-uncontrollability of a service automaton A.

Lemma 5.4.1 (Sufficient criterion for b-uncontrollability).
A service automaton A with $\mathit{closure}(\{[q_{0\,A}, [\,]]\}) \not\subseteq \mathit{situations}_b(A)$ is b-uncontrollable. ⌟

Proof.
Let $\mathit{closure}(\{[q_{0\,A}, [\,]]\}) \not\subseteq \mathit{situations}_b(A)$, let B be an arbitrary service automaton which is interface compatible to A, and let $q_{0\,B}$ be the initial state of B. From the composition definition we know that $[q, q_{0\,B}, M]$ is an internally reachable state of $A \oplus B$, for each $[q, M] \in \mathit{closure}(\{[q_{0\,A}, [\,]]\})$. By assumption, there is a situation in $\mathit{closure}(\{[q_{0\,A}, [\,]]\})$ violating the message bound. Hence, there is an internally reachable state of the composition $A \oplus B$, violating the message bound, too. Hence, B is no b-strategy for A. □

Applying this lemma, we easily prove that the service automata A of Fig. 5.7(a) (on page 153) is not 1-controllable. Because the initial state $\mathsf{q1}$'s knowledge K_0 of the overapproximation $\mathcal{F}_{\mathsf{A}}$ of Fig. 5.7(b) contains the situation $[\mathsf{r3}, [\mathsf{a}, \mathsf{a}]]$, it violates the message bound 1. Hence, $K_0 = \mathit{closure}(\{[\mathsf{r1}, [\,]]\}) \not\subseteq \mathit{situations}_1(\mathsf{A})$ and thus, A is not 1-controllable according to Lemma 5.4.1.

Because we want to characterize b-strategies using $\mathcal{F}$ in the following, we can assume $\mathit{closure}(\{[q_{0\,A}, [\,]]\}) \subseteq \mathit{situations}_b(A)$ for the construction of $\mathcal{F}$ in the following definition.

Definition 5.4.2 (Overapproximation of b-strategies, $\mathcal{F}^b$).
Let A be a finite service automaton where $\mathit{closure}(\{[q_{0\,A}, [\,]]\}) \subseteq \mathit{situations}_b(A)$.

Then, the *overapproximation of b-strategies* for A, $\mathcal{F}^b$, is defined as the service automaton $\mathcal{F}^b = [Q, I_{in}, I_{out}, \delta, q_0, \Omega]$ with

- $Q = \{q_K \mid K \subseteq \mathit{situations}_b(A)\}$,
- $I_{in} = I_{out\,A}$,
- $I_{out} = I_{in\,A}$,
- $\delta = \{(q_K, x, q_{K'}) \mid K, K' \subseteq \mathit{situations}_b(A), x \in I_{io\,A} \cup \{\tau\},$
  $K' = \mathit{closure}(\mathit{event}(K, x))\}$,
- $q_0 = q_{K_0}$ with $K_0 = \mathit{closure}(\{[q_{0\,A}, [\,]]\})$, and
- $\Omega = \{q_K \mid K \subseteq \mathit{situations}_b(A), [q, [\,]] \in K, q \in \Omega_A\}$. ⌟

In comparison with the definition of $\mathcal{F}$ in Definition 5.2.5 (on page 151), the only change in the new definition of $\mathcal{F}^b$ is the introduction of the message bound limit for the considered sets K of situations of A and the additional precondition $\mathit{closure}(\{[q_{0\,A}, [\,]]\}) \subseteq \mathit{situations}_b(A)$. Mainly, the precondition assures that q_{K_0} is a well-defined state according to the definition of the set of states Q of $\mathcal{F}^b$.

Together with the additional assumption of a *finite* service automaton A, we assure finiteness of all components of $\mathcal{F}^b$, as formalized in the following lemma.

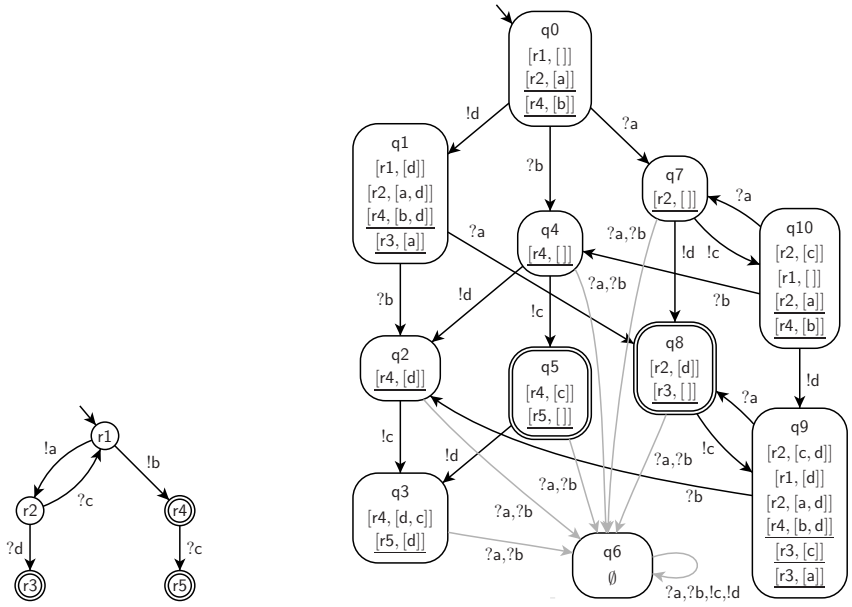
Lemma 5.4.3 (Finiteness of $\mathcal{F}^b$ and its knowledge).
For each service automaton A, the overapproximation $\mathcal{F}^b$ of b-strategies for A is a finite service automaton and for each state q_K of $\mathcal{F}^b$, the knowledge K is finite.⌟

Proof.
By the assumption of A being finite in Definition 5.4.2, we know that Q_A is finite. Together with $I_{io\,A}$, and thus $\mathit{bags}_b(I_{io\,A})$, being finite, the set $\mathit{situations}_b(A)$ (cp. Definition 5.3.7) is finite. This directly implies that $\mathcal{F}^b$ is finite as well. From $K \subseteq \mathit{situations}_b(A)$, we further know that each knowledge set K is finite, too. □

In order to exemplify the overapproximation of b-strategies according to Definition 5.4.2, consider again the overapproximation $\mathcal{F}_{\mathsf{A}}$ of (arbitrary) strategies of the service automaton A of Fig. 5.7(a) as depicted in Fig. 5.7(b). As mentioned above, A is not 1-controllable. Hence, $\mathcal{F}^1_{\mathsf{A}}$ is undefined. As A is 2-controllable, the corresponding overapproximation $\mathcal{F}^2_{\mathsf{A}}$ is defined. In fact, it is equal to $\mathcal{F}_{\mathsf{A}}$, i.e. $\mathcal{F}^2_{\mathsf{A}} = \mathcal{F}_{\mathsf{A}}$. Moreover, $\mathcal{F}_{\mathsf{A}}$ equals $\mathcal{F}^b_{\mathsf{A}}$, for all $b \geq 2$. The main reason for this is that A has no input channels and itself sends at most 2 messages a (and only one b).

As an example where only the introduction of a message bound b makes the overapproximation $\mathcal{F}^b$ finite, we recall the service automaton A of Fig. 5.2(a) in Fig. 5.10(a). The input channel c (or d) of A allows its overapproximation $\mathcal{F}_{\mathsf{A}}$ of A (not depicted) to perform arbitrarily many $!\mathsf{c}$-events (or $!\mathsf{d}$-events), causing $\mathcal{F}_{\mathsf{A}}$ to have infinitely many δ-reachable states. The corresponding overapproximation $\mathcal{F}^1_{\mathsf{A}}$ of 1-strategies for A, in contrast, allows for at most one simultaneously pending c (or d) message and is therefore finite. It is depicted in Fig. 5.10(b). The figure shows all (δ-reachable) states q_K of $\mathcal{F}^1_{\mathsf{A}}$ with $K \subseteq \mathit{situations}_1(\mathsf{A})$, as well as each x-labeled transition between states q_K and $q_{K'}$ such that $K' = \mathit{closure}(\mathit{event}(K, x))$, for $x \neq \tau$.

Please notice that $\mathcal{F}^b$ is not complete anymore (cp. Lemma 5.2.9), i.e. there may exist a message channel x and a state q of $\mathcal{F}^b$ such that q has no present x-labeled transition. Due to the construction of $\mathcal{F}^b$, however, we know that the x-successor of q would violate the message bound b in this case. More precisely, only states q_K where K violates the message bound are not part of the overapproximation $\mathcal{F}^b$ of b-strategies compared to the overapproximation $\mathcal{F}$ of arbitrary strategies. Hence, we "lose" only such service automata B which do not communicate b-bounded with A. This is formalized in the following corollary.



(a) A (b) $\mathcal{F}^1_{\mathsf{A}}$

Figure 5.10: (a) The service automaton A of Fig. 5.2(a). (b) The overapproximation $\mathcal{F}^1_{\mathsf{A}}$ of 1-strategies for A. For better readability, we omit all τ-loops. However, they are part of $\mathcal{F}^1_{\mathsf{A}}$ (as shown in Fig. 5.7(b), for instance). Furthermore, transitions leading to the empty state $q_\emptyset$ of $\mathcal{F}^1_{\mathsf{A}}$ are grayed out, and multiple arc labels denote multiple transitions between two states. Moreover, underlined situations in a knowledge set K illustrate stable situations of A in $\mathcal{F}^1_{\mathsf{A}}$.

Corollary 5.4.4 ($\mathcal{F}^b$ simulates all b-bounded communicating B).
Let A be a service automaton and let $\mathcal{F}^b$ be its overapproximation of b-strategies.

Then, $\mathcal{F}^b$ simulates each service automaton B such that A and B communicate b-bounded and $A \oplus B$ is closed. ⌟

That is, the structure of $\mathcal{F}^b$ is rich enough to comprise any b-strategy B for A, expressed by the respective simulation relation between B and $\mathcal{F}^b$.

Hence, we may apply the conditions of Theorem 5.2.15 to decide $B \in \mathit{Strat}_b(A)$ using $\mathcal{F}^b$ instead of $\mathcal{F}$ and conclude the following corollary.

Corollary 5.4.5 (Characterization of b-strategies for A by $\mathcal{F}^b$).
Let A be a service automaton and let $\mathcal{F}^b$ be its overapproximation of b-strategies.

Then, a service automaton B is a b-strategy for A iff B satisfies the conditions of Theorem 5.2.15 using $\mathcal{F}^b$ instead of $\mathcal{F}_A$. ⌟

The advantage of $\mathcal{F}^b$ over $\mathcal{F}$ is justified by Lemma 5.4.3. Whereas $\mathcal{F}$ is usually infinite, $\mathcal{F}^b$ is always finite. $\mathcal{F}^b$ can still be constructed using the operations closure and event. Whenever this construction introduces more than b messages of the same channel, it can be stopped and the currently computed set K can be skipped, as q_K is not needed to characterize b-strategies. That way, we can assure that the construction of $\mathcal{F}^b$ terminates, and $\mathcal{F}^b$ is therefore feasible in practice.

However, the second issue motivated at the end of Sect. 5.2.3, i.e. the weak abstraction of A by its overapproximation of strategies, still remains. That is, the usage of $\mathcal{F}^b$ enables us to reconstruct A from the knowledge of $\mathcal{F}^b$ about A to a large extend. For this reason, we will translate the conditions on the situations expressed by Theorem 5.2.15 into a *Boolean annotation* of the states of $\mathcal{F}^b$ in the following section. Due to this translation, the knowledge of $\mathcal{F}^b$ about A is not needed anymore and can be removed from $\mathcal{F}^b$. Only the structure of $\mathcal{F}^b$, i.e. its states and transitions, and the annotations are used in the future characterization. That way, the reason *why* B satisfies or violates Theorem 5.2.15 will be hidden — only the satisfaction or violation *as such* will be preserved.

5.4.2 A Characterization of b-Strategies by Operating Guidelines

Recall the three conditions that our strategy characterization according to Theorem 5.2.15 requires for a service automaton B to be a b-strategy for a service automaton A. For each situation $[q_A, M]$ in all knowledge sets of $\mathcal{F}^b$ "used" by a state q_B of B, fulfillment of one of the conditions is sufficient to verify $B \in \mathit{Strat}_b(A)$:

(i) $[q_A, M]$ is transient in A or

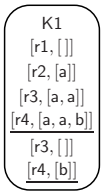
(ii) the situation $[q_B, M]$ of B is transient in B or

(iii) $q_A \in \Omega_A$ and $q_B \in \Omega_B$ and $M = [\,]$.

However, if we fix a service automaton A and check the three conditions for several service automata B, we can easily see that condition (i) is equally evaluated for every B — only A itself has influence on the fulfillment of this first condition. Furthermore, considering the fulfillment of condition (ii), $\mathcal{F}_A$ "knows" each situation $[q_A, M]$ of A for which the situation $[q_B, M]$ of B is checked for being transient. It is easy to see that only those transitions of B at the state q_B can make the situation $[q_B, M]$ transient in B that are either internal (i.e. τ-labeled) or sending transitions in B, or that are receiving transitions labeled by some $x \in M$. Finally, the first and the third part of condition (iii) are fixed by A as well, and the second part can as well be easily expressed as a Boolean value of the state q_B of B.

As a motivating example, consider the knowledge sets $\mathsf{K1}$ and $\mathsf{K2}$ about the service automaton A of Fig. 5.7(a) as depicted in Fig. 5.11. We elaborate on the relationship between the fulfillment of (one of) the three conditions of Theorem 5.2.15

and the structure of a service automaton B in order to derive a Boolean formula for each knowledge set. This formula equivalently expresses the conditions in the sense that one of the conditions are fulfilled by B if and only if the formula is satisfied under the assignment of the currently considered state q_B of B.



(a) K1

$\tau \vee ?\mathsf{b}$

(b) ϕ_{K1}

(c) K2

$(\tau \vee ?\mathsf{a}) \wedge (\tau \vee \mathsf{final})$

(d) ϕ_{K2}

Figure 5.11: (a) and (c) Two knowledge sets K1 and K2 about the service automaton A of Fig. 5.7(a). The set K1 is equal to the knowledge set $k(\mathsf{q1})$ of the overapproximation $\mathcal{F}_{\mathsf{A}}$ of Fig. 5.7(b). The set K2 equals the knowledge set $k(\mathsf{q3})$ of $\mathcal{F}_{\mathsf{A}}$. For convenience, dead situations of A are underlined in Fig. 5.11.
(b) and (d) Two Boolean formulae that equivalently express the fulfillment of the conditions of Theorem 5.2.15 by the respective knowledge sets. This suggests a translation of the conditions of into Boolean annotations.

In Fig. 5.11(a), the situation $[\mathsf{r1}, [\,]]$ of A is transient in A. Hence, condition (i) will always be fulfilled by each B which is checked for $B \in \mathit{Strat}_b(\mathsf{A})$. Hence, $[\mathsf{r1}, [\,]]$ cannot cause B to be no strategy. The same argument holds for the other situations which are not underlined. However, according to Theorem 5.2.15, at least one of the three conditions must hold for *each* knowledge value. Considering the situation $[\mathsf{r4}, [\mathsf{a}, \mathsf{a}, \mathsf{b}]]$, the first condition (i) is not fulfilled as $[\mathsf{r4}, [\mathsf{a}, \mathsf{a}, \mathsf{b}]]$ is stable in A. As $M = [\mathsf{a}, \mathsf{a}, \mathsf{b}]$ is non-empty, also the third condition (iii) cannot be fulfilled completely. However, as there are an a and a b message pending in M, the second condition can be fulfilled by a service automaton B if the considered state q_B of B has an internal or a sending transition, or it has an a-labeled or a b-labeled receiving transition present at q_B. Assuming that a and b are the only interface channels of A, this yields the formula $\tau \vee ?\mathsf{a} \vee ?\mathsf{b}$. However, B must additionally fulfill a condition for the second stable situation $[\mathsf{r4}, [\mathsf{b}]]$ in this knowledge set. There, no a is pending, making an a-labeled transition insufficient for B to be a strategy for A. Hence, the conditions of Theorem 5.2.15 can be translated to the Boolean formula $\tau \vee ?\mathsf{b}$ depicted in Fig. 5.11(b), which we claim to be satisfied by some B if and only if B fulfills at least one of the conditions of Theorem 5.2.15.

Figure 5.11(c) shows another knowledge set about the service automaton A of Fig. 5.7(a). Both situations of A in this set are stable, violating the first condition. The first situation has an a message pending, thus violating also the third condition. Hence, a service automaton B must fulfill condition (ii) for this situation. That is, B must either have an internal transition or an ?a-labeled transition

present at its current state q_B. Furthermore, B must *also* fulfill one condition for the second situation $[\mathsf{r4}, []]$ of A. As M is empty in this situation, B cannot receive any message at this situation. However, as $\mathsf{r4}$ is a final state of A, B can fulfill condition (iii) for $[\mathsf{r4}, []]$. Hence, the corresponding Boolean formula is a conjunction of the formulae $\tau \vee ?a$ (for the first situation) and $\tau \vee \mathsf{final}$ (for the second situation).

These considerations suggest the translation of the three conditions of Theorem 5.2.15 into one Boolean formula per state of $\mathcal{F}^b$. Then, each formula "used" by a state q_B has to be satisfied by B to verify $B \in \mathit{Strat}_b(A)$. That way, the Boolean formulae are sufficient to decide $B \in \mathit{Strat}_b(A)$, and the knowledge of $\mathcal{F}_A$ is not needed anymore.

Definition 5.4.6 (Canonical Boolean annotation, $\psi_{\mathcal{F}^b}$).
Let A be a finite service automaton and let $\mathcal{F}^b$ be its overapproximation of b-strategies.

Then, the *canonical Boolean annotation* of $\mathcal{F}^b$, $\psi_{\mathcal{F}^b}$, is a mapping $\psi_{\mathcal{F}^b} : Q_{\mathcal{F}^b} \to \mathcal{BF}$ where, for each state q_K of $\mathcal{F}^b$:

$$\psi_{\mathcal{F}^b}(q_K) = \bigwedge_{[q_A, M] \in K} \Big(\psi_i(q_A, M) \vee \psi_{ii}(M) \vee \psi_{iii}(q_A, M) \Big)$$

with

- $\psi_i(q_A, M) = \begin{cases} \mathit{true}, & \text{if } [q_A, M] \text{ is transient in } A, \\ \mathit{false}, & \text{otherwise,} \end{cases}$
- $\psi_{ii}(M) = \tau \vee \big(\bigvee_{x \in I_{out\,\mathcal{F}^b}} x \big) \vee \big(\bigvee_{x \in I_{in\,\mathcal{F}^b}, x \in M} x \big),$
- $\psi_{iii}(q_A, M) = \begin{cases} \mathit{final}, & \text{if } q_A \in \Omega_A \text{ and } M = [\,], \\ \mathit{false}, & \text{otherwise.} \end{cases}$

⌟

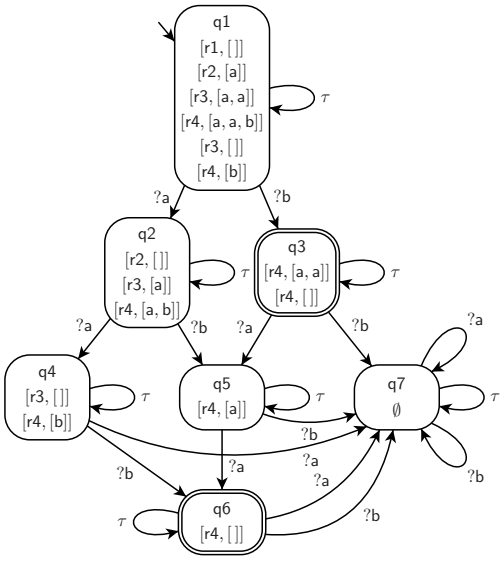
First, we make the observation that the canonical Boolean annotation at a state q_K of $\mathcal{F}^b$ is a negation-free Boolean formula over $\mathcal{MC}^+$ according to Definition 4.1.6 at page 99, as used throughout Chap. 4.

Furthermore, the formula $\psi_{\mathcal{F}^b}(q_K)$ at each state q_K of $\mathcal{F}^b$ is in *conjunctive normal form* (CNF), i.e. $\psi_{\mathcal{F}^b}(q_K)$ is a conjunction of one clause for each situation $[q_A, M] \in K$. Each such clause consists of three disjunctively connected subformulae $\psi_i(q_A, M)$, $\psi_{ii}(M)$, and $\psi_{iii}(q_A, M)$. For satisfying $\psi_{\mathcal{F}^b}(q_K)$, it is therefore sufficient to satisfy one of the subformulae per clause.

As an example for the canonical Boolean annotations, Fig. 5.12(a) depicts the overapproximation $\mathcal{F}^2_{\mathsf{A}}$ of 2-strategies for the service automaton A of Fig. 5.7(a). As motivated above, $\mathcal{F}^2_{\mathsf{A}}$ equals the overapproximation $\mathcal{F}_{\mathsf{A}}$ of arbitrary strategies of Fig. 5.7(b). Figure 5.12(b) depicts the clauses of the canonical annotation of

$\psi_{\mathcal{F}^2_{\mathsf{A}}}$ in the order in which the corresponding situations of A occur in the knowledge of the respective state of $\mathcal{F}^2_{\mathsf{A}}$. For instance, the first clause at state q1, $\mathit{true} \vee \tau \vee \mathit{false}$, corresponds to the situation $[\mathsf{r1}, [\,]]$ of A, which is the first situation in the knowledge of $\mathcal{F}^2_{\mathsf{A}}$ at state q1. Because $[\mathsf{r1}, [\,]]$ is transient in A, the first subformula of $\mathit{true} \vee \tau \vee \mathit{false}$ is *true*; because $M = [\,]$ and $I_{out\,\mathcal{F}^2_{\mathsf{A}}} = \emptyset$, the second subformula is just τ; and because r1 is no final state of A, the third subformula equals *false*. As another example, the fourth clause at state q1, $\mathit{false} \vee \tau \vee ?\mathsf{a} \vee ?\mathsf{b} \vee \mathit{false}$, corresponds to the fourth situation of q1, $[\mathsf{r4}, [\mathsf{a}, \mathsf{a}, \mathsf{b}]]$. Because this situation is stable, the first subformula is *false*; because there are an a and a b message pending, the second subformula is $\tau \vee ?\mathsf{a} \vee ?\mathsf{b}$; and because $M \neq [\,]$, the third subformula is *false*.

The canonical annotation $\psi_{\mathcal{F}^2_{\mathsf{A}}}$ of a state of $\mathcal{F}^2_{\mathsf{A}}$ is the conjunction of all clauses listed for this state in Fig. 5.12(b).



(a) $\mathcal{F}^2_{\mathsf{A}}$ for A

q1 :
- $\mathit{true} \vee \tau \vee \mathit{false}$
- $\mathit{true} \vee \tau \vee ?\mathsf{a} \vee \mathit{false}$
- $\mathit{true} \vee \tau \vee ?\mathsf{a} \vee \mathit{false}$
- $\mathit{false} \vee \tau \vee ?\mathsf{a} \vee ?\mathsf{b} \vee \mathit{false}$
- $\mathit{true} \vee \tau \vee \mathit{false}$
- $\mathit{false} \vee \tau \vee ?\mathsf{b} \vee \mathit{false}$

q2 :
- $\mathit{true} \vee \tau \vee \mathit{false}$
- $\mathit{true} \vee \tau \vee ?\mathsf{a} \vee \mathit{false}$
- $\mathit{false} \vee \tau \vee ?\mathsf{a} \vee ?\mathsf{b} \vee \mathit{false}$

q3 :
- $\mathit{false} \vee \tau \vee ?\mathsf{a} \vee \mathit{false}$
- $\mathit{false} \vee \tau \vee \mathit{final}$

q4 :
- $\mathit{true} \vee \tau \vee \mathit{false}$
- $\mathit{false} \vee \tau \vee ?\mathsf{b} \vee \mathit{false}$

q5 :
- $\mathit{false} \vee \tau \vee ?\mathsf{a} \vee \mathit{false}$

q6 :
- $\mathit{false} \vee \tau \vee \mathit{final}$

(b) $\psi_{\mathcal{F}^2_{\mathsf{A}}}$

Figure 5.12: (a) The overapproximation $\mathcal{F}^2_{\mathsf{A}}$ of 2-strategies for the service automaton A of Fig. 5.7(a), together with the knowledge k of $\mathcal{F}^2_{\mathsf{A}}$ about A. (b) The clauses of the canonical annotation of $\mathcal{F}^2_{\mathsf{A}}$ in the order in which the situations occur in $\mathcal{F}^2_{\mathsf{A}}$. The annotation $\psi_{\mathcal{F}^2_{\mathsf{A}}}$ of the state q1 of $\mathcal{F}^2_{\mathsf{A}}$ (i.e. the conjunction of all depicted clauses) is equivalent to the formula $\phi_{\mathsf{K1}} = \tau \vee ?\mathsf{b}$ of Fig. 5.11(b). The annotation of the state q3 is equivalent to the formula $\phi_{\mathsf{K2}} = (\tau \vee ?\mathsf{a}) \wedge (\tau \vee \mathsf{final})$ of Fig. 5.11(d).

It is easy to see in Fig. 5.12 that the knowledge $k(\mathsf{q1})$ is equal to the knowledge set $\mathsf{K1}$ of Fig. 5.11(a). Consequently, the canonical annotation $\psi_{\mathcal{F}^2_{\hat{\mathsf{A}}}}(\mathsf{q1})$ of the state $\mathsf{q1}$ (cp. Fig. 5.12(b)) is equivalent to the Boolean formula $\phi_{\mathsf{K1}} = \tau \vee ?\mathsf{b}$ of Fig. 5.11(b). Analogously, the knowledge set $k(\mathsf{q3})$ equals the knowledge set $\mathsf{K2}$ of Fig. 5.11(c); and the canonical annotation $\psi_{\mathcal{F}^2}(\mathsf{q3})$ is equivalent to the Boolean formula $\phi_{\mathsf{K2}} = (\tau \vee ?\mathsf{a}) \wedge (\tau \vee \mathsf{final})$ of Fig. 5.11(d).

Intuitively, the subformula $\psi_i(q_A, M)$ of the formula $\psi_{\mathcal{F}^b}(q_K)$ corresponds to condition (i) of Theorem 5.2.15 (as recalled at the beginning of this section). In case the situation $[q_A, M]$ is transient in A, the first condition of the theorem is fulfilled, and the other conditions are irrelevant. Accordingly, $\psi_i(q_A, M)$ is a tautology in this case (expressing that there are no further requirements for B). In case $[q_A, M]$ is stable in A, the first condition is violated and (one of) the other conditions must be fulfilled by B. Correspondingly, the Boolean formula $\psi_i(q_A, M)$ is unsatisfiable, i.e. a contradiction, in this case.

The second subformula $\psi_{ii}(M)$ corresponds to condition (ii) of Theorem 5.2.15. It expresses the constellations making the situation $[q_B, M]$ of B transient in B. Either, B has a present internal transition or sending transition at q_B, or B has a present x-labeled receiving transition at q_B with $x \in M$. One such transition is sufficient to fulfill condition (ii).

The third subformula $\psi_{iii}(q_A, M)$ corresponds to condition (iii). Only if A is in a final state q_A and the multiset of pending messages is empty, then B can fulfill condition (iii) if q_B is also a final state of B. Hence, we set $\psi_{iii}(q_A, M)$ to *final* only if $q_A \in \Omega_A$ and $M = [\,]$. Otherwise, condition (iii) is violated anyway.

In case of the empty state $q_\emptyset$ of an overapproximation $\mathcal{F}^b$, there is no situation $[q_A, M] \in K = \emptyset$ at this state. Hence, the canonical annotation of $\mathcal{F}^b$ at state $q_\emptyset$, $\psi_{\mathcal{F}^b}(q_\emptyset)$, is always equal to the *empty conjunction*. Recall that the empty conjunction is a tautology by definition. Hence, the canonical annotation of the empty state $q_\emptyset$ is the formula *true*, for all service automata A.

It is easy to see in Fig. 5.12(b) that there is no clause depicted for the state $\mathsf{q7}$ of $\mathcal{F}^2_{\hat{\mathsf{A}}}$, because there is no situation in the knowledge at $\mathsf{q7}$ — this state is the empty state of $\mathcal{F}^2_{\hat{\mathsf{A}}}$. Hence its annotation $\psi_{\mathcal{F}^2_{\hat{\mathsf{A}}}}(\mathsf{q7})$ is *true*.

Remark 5.4.7 (Non-empty, *true*-annotated state of $\mathcal{F}^b$).
It is easy to construct a service automaton A where $\mathcal{F}^b$ has a state q_K such that K consists of a single, transient situation $[q, [\,]]$ of A. In this case, the canonical annotation $\psi_{\mathcal{F}^b}(q_K)$ at the state q_K is equivalent to *true* — just as the canonical formula of the empty state. However, q_K is obviously not the empty state (due to the situation $[q, [\,]]$ in K). Hence, not every *true*-annotated state of $\mathcal{F}^b$ is the empty state. ⌟

As motivated above, the canonical annotation at a state q_K of $\mathcal{F}^b$ corresponds to the conditions of Theorem 5.2.15. The next theorem formalizes this connection.

Basically, it states that the canonical annotations are satisfied if and only if the conditions are fulfilled for each situation in the respective knowledge set K. Hence, satisfaction of the annotations can be used to decide $B \in \mathit{Strat}_b(A)$, even without considering the knowledge of $\mathcal{F}^b$ about A. To this end, we employ the assignment β_B of a service automaton B as defined in Chap. 4 (Definition 4.2.4 on page 103) to evaluate the Boolean formulae of $\mathcal{F}^b$.

Theorem 5.4.8 (Formula characterization of *b*-strategies).
Let A be a service automaton, $\mathcal{F}^b$ be the overapproximation of b-strategies for A, and B be a service automaton which is interface equivalent to $\mathcal{F}^b$. Let β_B be the assignment of B, and $\varrho \subseteq Q_B \times Q_{\mathcal{F}^b}$ be the minimal simulation relation between B and $\mathcal{F}^b$.

Then, B is a b-strategy for A iff for all states q_B of B and all state q_K of $\mathcal{F}^b$ with $(q_B, q_K) \in \varrho$: $\beta_B(q_B)$ satisfies the annotation of q_K, i.e. $\beta_B(q_B) \models \psi_{\mathcal{F}^b}(q_K)$. ⌟

We prove the theorem by showing that there exists a deadlock $[q_A, q_B, M]$ in $A \oplus B$ if and only if $\beta_B(q_B) \not\models \psi_{\mathcal{F}^b}(q_K)$ for some state q_K of $\mathcal{F}^b$ with $(q_B, q_K) \in \varrho$. Therefore, we reduce the violation of the annotation $\psi_{\mathcal{F}^b}(q_K)$ to the three conditions of Lemma 5.1.5.

Proof.
Let, throughout this proof, k denote the knowledge of $\mathcal{F}^b$ about A and let, for each state q_K of $\mathcal{F}^b$, $\psi_{\mathcal{F}^b}(q_K) = \bigwedge_{[q_A,M]\in K} \Big(\psi_i(q_A, M) \vee \psi_{ii}(M) \vee \psi_{iii}(q_A, M)\Big)$ be defined as in Definition 5.4.6.

($\rightarrow$): Let $[q_A, q_B, M]$ be a deadlock of $A \oplus B$. By Definition 5.1.3, we have $[q_A, M] \in k_{(B,A)}(q_B)$. By Lemma 5.2.14, we further know that there is a state q_K of $\mathcal{F}^b$ with $(q_B, q_K) \in \varrho$ and $[q_A, M] \in k(q_K)$. We show that q_B violates the annotation of q_K, i.e. (a) $\beta_B(q_B) \not\models \psi_i(q_A, M)$ and (b) $\beta_B(q_B) \not\models \psi_{ii}(M)$ and (c) $\beta_B(q_B) \not\models \psi_{iii}(q_A, M)$.

By $[q_A, q_B, M]$ being a deadlock and Lemma 5.1.5, we know that (i) $[q_A, M]$ is stable in A, (ii) $[q_B, M]$ is stable in B, and (iii) $q_A \notin \Omega_A$ or $q_B \notin \Omega_B$ or $M \neq [\,]$.

From (i) and the construction of $\psi_i(q_A, M)$, we know $\psi_i(q_A, M) = \mathit{false}$, and hence, we conclude (a).

From (ii) and the definition of $\beta_B(q_B)$, we know $\beta_B(q_B)(x) = \mathit{false}$ for $x = \tau$, for all $x \in I_{out\,B}$, and for all $x \in I_{in\,B}$ with $x \in M$. Hence, every literal of $\psi_{ii}(M)$ is set to *false* by $\beta_B(q_B)$, and hence, we conclude (b).

From (iii) and the construction of $\psi_{iii}(q_A, M)$, we know either $\psi_{iii}(q_A, M) = \mathit{false}$ (in case $q_A \notin \Omega_A$ or $M \neq [\,]$), or $\beta_B(q_B)(\mathit{final}) = \mathit{false}$ (in case $q_B \notin \Omega_B$). In both cases we conclude (c).

($\leftarrow$): Let $\beta_B(q_B) \not\models \psi_{\mathcal{F}^b}(q_K)$ for some state q_K of $\mathcal{F}^b$ with $(q_B, q_K) \in \varrho$. We show that there exists a situation $[q_A, M] \in k_{(B,A)}(q_B)$ with $[q_A, q_B, M]$ is a deadlock of $A \oplus B$. By Lemma 5.2.14, it suffices to show that $[q_A, M] \in k(q_K)$.

By $\beta_B(q_B) \not\models \psi_{\mathcal{F}^b}(q_K)$ and the construction of $\psi_{\mathcal{F}^b}(q_K)$, we know that there must exist a situation $[q_A, M] \in k(q_K)$ such that (a) $\beta_B(q_B) \not\models \psi_i(q_A, M)$ and (b) $\beta_B(q_B) \not\models \psi_{ii}(M)$ and (c) $\beta_B(q_B) \not\models \psi_{iii}(q_A, M)$.

From (a) we conclude that $[q_A, M]$ must be stable in A. From (b) we know that all literals of $\psi_{ii}(M)$ must be set to *false* by $\beta_B(q_B)$, and hence we conclude $[q_B, M]$ is stable in B. From (c) and the construction of $\psi_{iii}(q_A, M)$, we know either $\beta_B(q_B)(\mathit{final}) = \mathit{true}$, but $\psi_{iii}(q_A, M) = \mathit{false}$; or $\psi_{iii}(q_A, M) = \mathit{final}$, but $\beta_B(q_B)(\mathit{final}) = \mathit{false}$. In the first case, $q_A \notin \Omega_A$ or $M \neq [\,]$, in the second case we know $q_B \notin \Omega_B$.

Hence, all three conditions of Lemma 5.1.5 are fulfilled and hence, $[q_A, q_B, M]$ is a deadlock of $A \oplus B$. □

With this result, we are finally able to characterize all b-strategies B for a given service automaton A (and a given message bound b) using the $\psi_{\mathcal{F}^b}$-annotated overapproximation $\mathcal{F}^b$ of b-strategies for A.

We will see that $\mathcal{F}^b$ and $\psi_{\mathcal{F}^b}$ constitute a *BSA* as introduced in Chap. 4. Hence, it is a suitable candidate for our main goal, the operating guideline OG_A of A. To emphasize the redundancy of the set K of a state q_K of $\mathcal{F}^b$, however, we will not call this *BSA* operating guideline here, but introduce another formalization in the upcoming Corollary 5.4.10.

Obviously, the annotation at a state q of $\mathcal{F}^b$ can be simplified drastically by applying the syntactical simplifications as described in Remark 4.1.12 (on page 101). There even exist several obvious optimizations already during the construction of $\psi_{\mathcal{F}^b}$. A clause that corresponds to a transient situation of A in K will always be equivalent to *true*, and adds no information to the CNF at the state q_K. Hence, all transient situations could be skipped completely in Definition 5.4.6. If thereby no clause at all is added to a CNF, it is the empty conjunction and is itself equivalent to *true*; as it would be when adding all these *true* clauses. Furthermore, the *false* elements added to a clause due to a stable situation or a non-final state of A can be skipped as well, because they add no information either. Due to the literal τ added by the subformula $\psi_{(ii)}$, a clause can never become empty.

Notation 5.4.9.
For the rest of this thesis, we will simplify the canonical annotation $\psi_{\mathcal{F}^b}(q_K)$ of a state q_K of $\mathcal{F}^b$ if possible. In particular, we omit *true* clauses and *false* elements of a clause. Furthermore, we will not depict the τ-loops (q_K, τ, q_K) at each state q_K of $\mathcal{F}^b$, as well as the τ literals in the annotations, as both can always be derived canonically from the figures. Analogously, we will omit the empty state $q_\emptyset$ if it is not important in the current context. Again, the empty state can canonically be derived from the figures. ⌟

Now we are ready to formalize the notion of the canonical operating guideline OG_A for A, which is derived from $\mathcal{F}^b$ and $\psi_{\mathcal{F}^b}$.

It is easy to see that $\mathcal{F}^b$ is a deterministic service automaton according to Definition 3.3.8. Furthermore, the canonical annotation $\psi_{\mathcal{F}^b}$ of $\mathcal{F}^b$ is obviously a valid Boolean annotation to the states of $\mathcal{F}^b$. Hence, $\mathcal{F}^b$ and $\psi_{\mathcal{F}^b}$ constitute a *BSA* according to Definition 4.2.1. Together with Theorem 5.4.8, we conclude:

Corollary 5.4.10 (Canonical operating guideline, OG_A).
Let $b \in \mathbb{N}$ be some given message bound and let A be a service automaton with $\mathit{closure}(\{[q_{0\,A}, [\,]]\}) \subseteq \mathit{situations}_b(A)$. Let furthermore $\mathcal{F}^b$ be the overapproximation of b-strategies for A and $\psi_{\mathcal{F}^b}$ be the canonical annotation of $\mathcal{F}^b$.

Let B be a service automaton which is isomorphic to $\mathcal{F}^b$ under isomorphism h and let $\phi : Q_B \to \mathcal{BF}$ be a Boolean annotation of B defined as $\phi(q_B) = \psi_{\mathcal{F}^b}(h(q_B))$, for each state q_B of B.

Then, $OG^b_A = B^\phi$ serves as b-operating guideline for A according to Definition 5.3.6, i.e. $\mathit{Match}(OG^b_A) = \mathit{Strat}_b(A)$. Thus, OG^b_A is called the *canonical operating guideline* for A under message bound b.

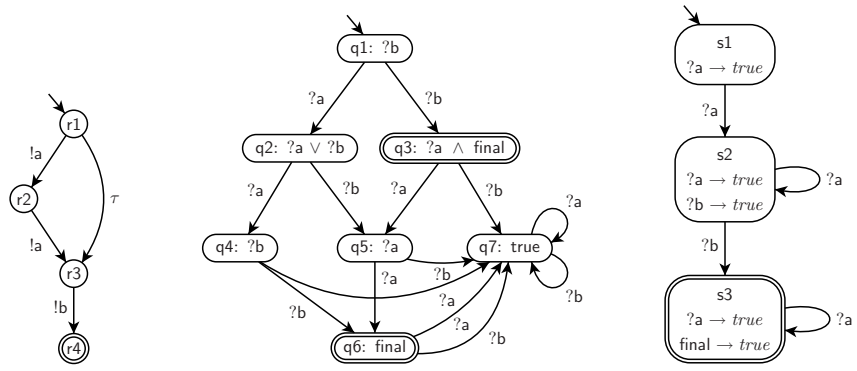
For a service automaton A with $\mathit{closure}(\{[q_{0\,A}, [\,]]\}) \not\subseteq \mathit{situations}_b(A)$, we fix an empty *BSA* as its canonical operating guideline OG^b_A. ⌟

According to Corollary 5.4.10, the semantics of the *BSA* OG^b_A for A is the set of b-strategies for A, i.e. OG^b_A characterizes the set $\mathit{Strat}_b(A)$. Again, we will write OG_A if the message bound b is not important in the current context.

The use of a new service automaton B in the previous corollary emphasizes the redundancy of the knowledge of $\mathcal{F}^b$ about A. Only the annotations are needed to decide $B \in \mathit{Strat}_b(A)$ once they are derived from the knowledge. Consequently, the knowledge can be discarded to hide the internal structure of A. Besides the finiteness of OG_A, this is the second advantage of OG_A over the automaton $\mathcal{F}_A$ of Sect. 5.2.

To exemplify the matching of a service automaton with an operating guideline, we repeat the service automaton A of Fig. 5.7(a) in Fig. 5.13(a). Figure 5.13(b) depicts the canonical operating guideline OG_{A} which is computed from A's overapproximation $\mathcal{F}^2_{\mathsf{A}}$ of Fig. 5.12(a). The Boolean annotations of OG_{A} replace the knowledge of $\mathcal{F}^2_{\mathsf{A}}$. In Fig. 5.13(c), we show the service automaton B of Fig. 5.8(a) for A again. This time, the assignment $\beta_{\mathsf{B}}(q_B)$ of a state q_B of B is depicted inside the state q_B (instead of B's knowledge about A). As done earlier, the elements which are set to *false* are not listed in Fig. 5.13(c). It is easy to see that B does not match with OG_{A} as it already violates the initial state q1's annotation ?b in its state s1. Hence, B is no strategy for A. We could also verify this using the knowledge of B about A, for instance. Checking the conditions according to one of the characterizations that make use of this knowledge, however, would have revealed a concrete deadlock $[\mathsf{r4}, \mathsf{s1}, [\mathsf{b}]]$ in $\mathsf{A} \oplus \mathsf{B}$. Using the annotations instead, only the fact that B must have a present ?b-labeled transition in its state s1 for

being a strategy for A is revealed, which we claim to be a necessary information for deciding deadlock freedom of the composition.



(a) A (b) OG_{A} (c) B with β_{B}

Figure 5.13: (a) The service automaton A of Fig. 5.7(a). (b) Its 1-operating guideline OG_{A}, computed from the overapproximation $\mathcal{F}^2_{\mathsf{A}}$ of Fig. 5.12(a). For better readability, we omit the τ-loops of $\mathcal{F}^2_{\mathsf{A}}$ for OG_{A}. (c) The service automaton B of Fig. 5.8(a) with its assignment $\beta_{\mathsf{B}}(q_B)$ listed inside q_B. Already in the initial state s1 of B we can see that B does not match with OG_A as the annotation ?b of q1 is not satisfied by $\beta_{\mathsf{B}}(\mathsf{s1})$.

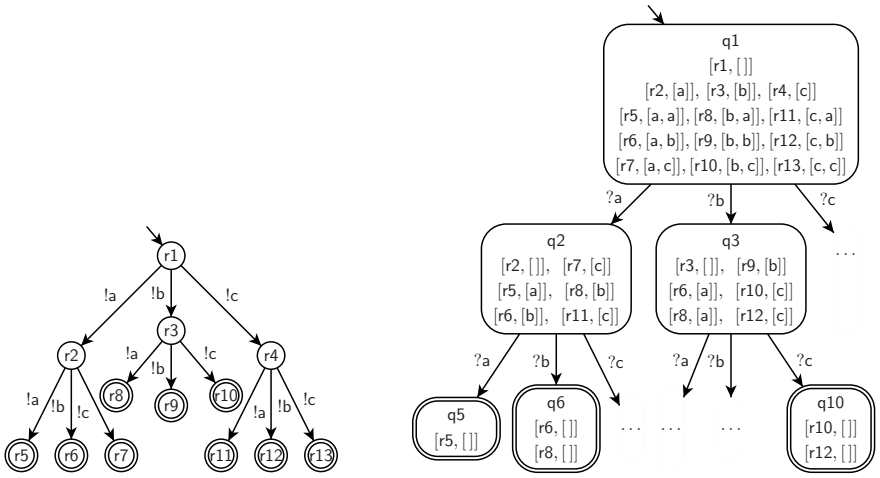
As the operating guideline OG_A for a service automaton A is a *BSA* according to Chapter 4, we can apply all the results presented in that chapter to OG_A as well. That is, we can normalize OG_A and may decide strategy inclusion of two services A and A' by checking the preorder relation between their operating guidelines, i.e. we may check $OG_A \sqsubseteq OG_{A'}$. Furthermore, we can consider the equivalence of operating guidelines or even minimize the canonical operating guideline OG_A. Hence, each other *BSA* which is equivalent to the canonical operating guideline OG_A of A may serve as a valid b-operating guideline according to Definition 5.3.6 as well.

The particular need for a minimization of operating guidelines is illustrated by the following example. It shows that two states q_K and $q_{K'}$ of the intermediate $\mathcal{F}^b$ with different knowledge sets K and K' can result in equivalent annotations, which itself may result in equivalent states q and q' of OG_A. These states can be merged under preservation of the semantics of OG_A (cp. Sect. 4.5.3). As the set $situations_b(A)$ may become quite large, there are potentially many states which can be merged. This may result in quite a lot of reduction by the minimization of operating guidelines constructed from $\mathcal{F}^b$. Furthermore, merging different states of OG_A yields an even greater abstraction of the internal structure of A.

To this end, consider the service automaton A′ of Fig. 5.14(a). Its overapproximation $\mathcal{F}_{\mathsf{A}'}$ of b-strategies for any $b \geq 2$ is depicted in Fig. 5.14(b). Its states $q_K = \mathsf{q2}$

and $q_{K'} = \mathsf{q3}$, for instance, obviously have different knowledge sets K and K', respectively. However, as $\mathcal{F}_{\mathsf{A}'}$ can at most receive a message a, b, or c — and there is at least one situation with an a, b, or c message currently pending in both K and K' — the canonical annotation yields $\psi_{\mathcal{F}_{\mathsf{A}'}}(\mathsf{q2}) = ?\mathsf{a} \vee ?\mathsf{b} \vee ?\mathsf{c} = \psi_{\mathcal{F}_{\mathsf{A}'}}(\mathsf{q3})$. Analogously, the canonical annotation of the states $\mathsf{q5}, \ldots$ are all equal to final. Hence, the operating guideline $OG_{\mathsf{A}'}$ computed from $\mathcal{F}_{\mathsf{A}'}$ looks exactly like the *BSA* presented in Fig. 4.18(a) (on page 134). Hence, the equivalently minimized *OG* is isomorphic to the minimized *BSA* of Fig. 4.18(b), which is significantly smaller than the computed *OG* according to Corollary 5.4.10.

Hence, we suggest to minimize each operating guideline after computation.



(a) A' (b) $\mathcal{F}_{\mathsf{A}'}$

Figure 5.14: Illustration for the need of a minimization of an *OG*. (a) A service automaton A' sending two out of the three messages a, b, and c. (b) The overapproximation $\mathcal{F}_{\mathsf{A}'}$ (for any message bound $b \geq 2$). In $\mathcal{F}_{\mathsf{A}'}$, the states $\mathsf{q2}$, $\mathsf{q3}$, and $\mathsf{q4}$ (not depicted) result in the canonical annotation $?\mathsf{a} \vee ?\mathsf{b} \vee ?\mathsf{c}$, whereas the states $\mathsf{q5}, \ldots, \mathsf{q13}$ result in the annotation final. Hence, the states $\mathsf{q2}$, $\mathsf{q3}$, and $\mathsf{q4}$ are equivalent to each other, as well as the state $\mathsf{q5}, \ldots, \mathsf{q13}$. Hence, the *OG* computed from $\mathcal{F}_{\mathsf{A}'}$ can be reduced drastically.

The construction of the b-operating guideline OG_A of a service A has been prototypically implemented in our service analysis tool Fiona. We refer to Chap. 7 for a presentation of the implementation as well as a case study showing the computability of operating guidelines for real-world services. We will compare the efforts for computing the operating guideline OG_A of A and matching several services B with OG_A on the one hand with the efforts to verify the absence of (reachable) deadlocks in the composition $B \oplus A$ for each service B on the other

hand. This will justify the operating guideline approach to efficiently decide behavioral compatibility of services as sketched in Fig. 5.9.

5.5 Possible Variants of *OG* Definitions

As it was done for our formal representation of services in Chap. 3 and for *BSA*s in Chap. 4, we will motivate some relevant design decisions that we made for introducing operating guidelines in this chapter. We will present and discuss other possibilities and investigate the implications thereof.

5.5.1 Strategy Notion for Non-Closed Compositions

In Sects. 3.2.3 and 3.3.3, respectively, the behavioral correctness notion of well-behavior of a service has been defined for *closed* services only. Hence, the operating guideline OG_A of A characterizes only service automata B where $A \oplus B$ is closed. As we have shown in Sect. 3.5.2, this distinction between open and closed compositions does not restrict generality as every pair of services A and B can be transformed in a way such that their composition is closed.

In this section, however, we discuss the changes needed for directly characterizing strategies B for A where the composition $A \oplus B$ is again an open service automaton, i.e. where A and B have free interface channels (cp. Definition 3.3.16). Basically, to characterize such open strategies B for A, the matching procedure between B and OG_A must be adjusted to tolerate the free channels.

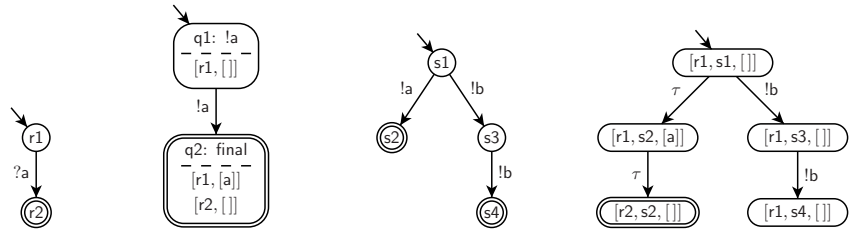
Tolerating Free Channels

If B has a smaller interface than A (in the sense that each free channel belongs to A), then such an open composition does not introduce any trouble, and both the computation of OG_A and the matching procedure can be used without change — only the precondition of interface equivalence in Definition 4.2.5 has to be dropped.

If B has a greater interface than A, however, then OG_A does not simulate B anymore. To overcome this problem, two approaches can be used. On the one hand, one could fix a *finite* name space $\mathcal{MC}$ such that B can at most have additional channels $x \in \mathcal{MC}$. Then, the computation of $\mathcal{F}_A$ and hence OG_A can canonically be extended to include all such channels. This, however, would potentially increase the size of the operating guideline OG_A significantly, which we think is not feasible. On the other hand, one could *tolerate* such additional channels of B during the matching. As the interface of OG_A is fixed, each transition of B not present in OG_A can uniquely be distinguished as a free channel or as a transition which is not allowed at this state. Hence, the simulation relation criterion of

the matching procedure can be relaxed easily. However, such a message is never added to the multiset of pending messages in the composition of A and B and therefore the matching of B with OG_A cannot assure that the message bound is not violated for the additional channels.

As an example, consider the service automaton A and its corresponding canonical operating guideline OG_A under message bound $b = 1$ in Fig. 5.15 (this time depicted *with* the knowledge of $\mathcal{F}_{\mathsf{A}}$). Assume $\mathsf{b} \notin I_{io\mathsf{A}}$. Obviously, A is 1-controllable by a service that sends one a and then rests in its final state, for instance (which can be seen easily by considering OG_A). The service automaton B of Fig. 5.15(c), however, has the possibility to also send two b messages. As b is free between A and B, it occurs as a transition label in their composition $\mathsf{A} \oplus \mathsf{B}$ in Fig. 5.15(d). Obviously, A and B communicate 1-bounded — there is no state of $\mathsf{A} \oplus \mathsf{B}$ with more than one pending message. However, curiously, the composed service automaton $\mathsf{A} \oplus \mathsf{B}$ itself is *not* 1-controllable as it sends two b's (which is now in the interface of $\mathsf{A} \oplus \mathsf{B}$) by itself. This effect should be avoided by introducing a scope of the message bound for the currently considered composition, for instance. That is, the message bound b is considered for the composition $A \oplus B$ only. For a further composition $(A \oplus B) \oplus C$, a new message bound b' (possibly smaller than b) has to be considered.



(a) A (b) OG_{A} with knowledge (c) B (d) $\mathsf{A} \oplus \mathsf{B}$

Figure 5.15: A service automaton A with its operating guideline OG_{A}. Tolerating the free channel !b in the matching of the service automaton B with OG_{A} yields a curious result. B matches with OG_{A} although $\mathsf{A} \oplus \mathsf{B}$ has a deadlock, and the composed service automaton $\mathsf{A} \oplus \mathsf{B}$ is 1-uncontrollable although A is 1-controllable and A and B communicate 1-bounded.

Additionally, $\mathsf{A} \oplus \mathsf{B}$ has an internally reachable deadlock, $[\mathsf{r1}, \mathsf{s4}, []]$. That is, the tolerating matching procedure must nevertheless reject the service automaton B, which cannot be evaluated by the annotations of OG_A so far. Hence, a present sending transition of a service automaton B at a state q_B must be tolerated if and only if q_B matches only with the empty state of OG_A.

Due to the additional need for a distinction between these cases, we decided to use the strict strategy notion in this thesis and demand a closed composition $A \oplus B$.

Due to the availability of the transformation of an open composition into a closed one as described in Sect. 3.5.2 (on page 86), this assumption does not restrict generality of our approach.

5.5.2 Other Correctness Criteria

The correctness criterion considered in this thesis is deadlock freedom of the composition $A \oplus B$ of two service automata A and B. However, there are plenty of other, mostly stricter, correctness criteria, e.g. *livelock freedom* or *weak termination* (i.e. deadlock freedom *and* livelock freedom) that can be considered for correct service interaction.

Note that for services with acyclic behavior, weak termination and deadlock freedom coincide.

In general, the consideration of other correctness criteria yields the notion of an X-strategy, where X denotes the respective correctness notion (as introduced in [Wol09]). For each X, however, a completely different approach may be needed to characterize all X-strategies.

For weak termination, for instance, a *BSA* as introduced in Chap. 4 is insufficient as a characterization of all weak-termination-strategies. One can easily construct two service automata A and A' such that each service automaton B has equal knowledge about A and A', but A is weak-termination-uncontrollable, whereas A' is weak-termination-controllable. Hence, a characterization of such strategies by Boolean formulae as proposed in this chapter is insufficient.

A different approach, called *fragment approach*, tries to solve this issue [SW08]. Basically, a fragment corresponds to a state q_K of our overapproximation $\mathcal{F}_A$. Additionally, q_K stores reachability information between the situations of A inside the knowledge set K, as well as reachability information between the situations of different knowledge sets K and K'. Then, the new matching procedure consists of one step checking for deadlocks (as done in this thesis) and a second step involving *model checking* for livelocks. The major drawback of this approach is the need for storing detailed information about the internals of the service A, as well as the additional model checking step.

5.6 Related Work

The term strategy, introduced in Chap. 3, already suggests a control-theoretic point of view of the behavioral compatibility relation between two services A and B. That is, we may see B as a controller for A imposing well-behavior of $A \oplus B$.

Accordingly, the constructions presented in this chapter can, in fact, be seen as a variant of *controller synthesis* for discrete event systems (DESs). Controller synthesis approaches consider a *plant* which has to be controlled, a *controller* which may control the plant in different ways, and a *target property* of the composed system of plant and controller. Basically, the controller may hinder the plant to perform specific events if this would result in a violation of the target property. Different settings are considered in which the controller may have only limited access to the plant (i.e. there are controllable and uncontrollable events), limited knowledge (i.e. there are observable and unobservable events), the controller can be most permissive (making as few as possible restrictions to the events of the plant) or not, and plant and controller might either be composed synchronously or asynchronously.

In our terms, the plant is the service automaton A for which the operating guideline has to be computed, the controller is the overapproximation $\mathcal{F}$ of strategies for A, and the target property is deadlock freedom of the composition. The setting of this thesis employs non-deterministic automata with partial observability (as the concrete current state of the controlled service automaton A is hidden for the controller $\mathcal{F}$) and uncontrollable events (as $\mathcal{F}$ may only control A by (not) sending messages to A or may only receive messages from A to observe that *some* transition of A must have sent this message).

Whereas there exist lots of papers on controller synthesis for DESs (see [RW87, CL99], for instance), and even for Petri nets (see [HKG97], for an extensive overview), most of the approaches consider either full observability or full controllability (e.g. [HGZ96]), very restricted classes of controlled systems (as marked graphs [DX03, GRX03] or state machine Petri nets [BBB95]), or very general properties of the target system (like enforcement of arbitrary μ-calculus properties [PR05]).

To the best of our knowledge, there is no controller synthesis approach considering exactly our setting. Furthermore, controller synthesis is at most suited to construct *one* (possibly most permissive) strategy for a service. A *characterization of all strategies* is out of scope of these approaches.

Further related work considers extensions of our original operating guideline approach. In [LW09], the authors present a representation of operating guidelines as a Petri net, i.e. a *Petri net operating guideline*. To this end, they apply the theory of regions concept to compactly represent the structure of the operating guideline OG_N of a service net N as a service net M. Obviously, M is a strategy for N by construction. Furthermore, [LW09] shows how to represent the Boolean annotations of OG_N by three sets of markings of the Petri net operating guideline M. A new matching procedure between another service net M' with the Petri net operating guideline M now requires to construct the state spaces of M' and M (for checking the simulation relation) and to check certain conditions about which marking of M' may occur in combination with a marking of one of the

three sets stored for M' (replacing the annotation evaluation). [LW09] shows a case study illustrating that (classical) operating guidelines can be represented by much smaller Petri net operating guidelines. However, they lack a case study comparing the matching efforts between both approaches. In any case, the approach sets on top of the operating guidelines approach as presented in this thesis as it is only a different *representation* of OG_N and requires the computation of OG_N in a preliminary step.

Another extension considers a stronger behavioral compatibility notion, i.e. weak termination of the composed services [WSOD09]. Therein, the authors show how to construct a set of *fragments* which allows for a characterization of all strategies B for A such that the composition $A \oplus B$ is deadlock-free and livelock-free. That is, from every reachable state of $A \oplus B$, a final state of $A \oplus B$ is reachable. However, the characterization is substantially different from the one presented in this thesis. That is, given a querying service B, the authors propose to construct a state space of the composition $B \oplus A'$ where A' is basically an abstract version of the service A itself, represented by the fragments. Afterwards, the composition has to be checked for the presence of lifelocks using standard model checking techniques. Whilst the result is a strictly stronger compatibility notion than the one presented here, the approach needs to represent the internal structure of the service A (at least in an abstract way) and the matching is by far more difficult and time consuming than the matching procedure in this thesis.

5.7 Concluding Remarks

In this chapter, we have presented the construction of a special Boolean annotated service automaton, called operating guideline OG_A for A, that characterizes the set of b-strategies for a given service A, i.e. the set $Strat_b(A)$. Hence, OG_A can be used to decide whether two services A and B are behaviorally compatible before both services are bound and start to interact.

The computation of OG_A is based on two simple operations on sets K of situations of A, $closure(K)$ and $event(K, x)$, which are used to construct the underlying service automaton $\mathcal{F}$ of OG_A. Furthermore, we were able to translate the conditions of the main Theorem 5.2.15 for the characterization of strategies B for A by $\mathcal{F}$ into a Boolean annotation ψ to the states of $\mathcal{F}$ in case of characterizing only finite strategies.

Hence, checking whether or not $B \in Strat(A)$ reduces to the simple matching problem $B \in Match(OG_A)$, introduced already in Chap. 4.

The major advantage of using the operating guideline OG_A of A to decide $B \in Strat(A)$ is twofold. Firstly, the constant construction effort to compute OG_A easily pays off if a number of B's are checked for their strategy relationship with A, because OG_A has to be constructed only once, and the matching between

B and OG_A is by far more efficient than constructing $A \oplus B$ and checking its state space for deadlocks. Secondly, the operating guideline of A reflects only the requirements that B must meet to be a strategy for A. That way, the reason *why* B is (or is not) a strategy is hidden — only the fact *as such* is preserved.

Finally, operating guidelines are *BSA*s as introduced in Chap. 4. Hence, all results presented there hold for operating guidelines as well. That is, we may normalize operating guidelines, consider the inclusion of the characterized sets of services, or the equivalence of services with respect to their strategies. Furthermore, the operating guideline OG_A of a service automaton A can be minimized, resulting in another, equivalent operating guideline for A, which is possibly much smaller than the original one. Moreover, merging different states of OG_A yields an even greater abstraction of the internal structure of A.

In Chap. 6, we will consider possible applications of operating guidelines in different research areas in the context of services in detail. These areas comprise service discovery, substitutability of service, and the generation of behavioral adapters.

Chapter 7 is then devoted to present the implementation of the construction of operating guidelines (and all other results presented in this Part II of the thesis) in the tool Fiona. Furthermore, we present a case study showing the practical applicability of operating guidelines for the discovery of behavioral compatible services.

Part III

Applications and Implementation

Operating guidelines can be applied in a variety of research areas in the context of behavioral correctness of service interaction. These applications include, but are not limited to, service discovery, substitutability of services, and the generation of behavioral adapters between behaviorally incompatible services. In this part, we briefly introduce these research areas and show how operating guidelines can be helpful to support the solutions in the respective area. Furthermore, we present Fiona, a tool to construct operating guidelines, to match services with operating guidelines, to normalize, to minimize operating guidelines, and many more.

6 Applications of Operating Guidelines

In the previous chapters, we have introduced *BSA*s as general constructs to characterize some set of services and operating guidelines as special *BSA*s to characterize all strategy services R for a given service S.

This chapter is devoted to exemplify the applicability of operating guidelines in the paradigm of service-oriented computing. We will show that operating guidelines constitute a flexible and convenient artifact useful in a variety of analysis questions in this context. To this end, we will apply operating guidelines to approach the open research questions presented in Sect. 1.3.

In particular, we consider how operating guidelines can be employed in the discovery of behaviorally compatible published services by a client in Sect. 6.1, we present decision procedures to decide basic service substitutability notions by comparing the corresponding operating guidelines of the services in Sect. 6.2, and we exemplify the use of operating guidelines for the synthesis of behavioral adapters in Sect. 6.3. Finally, Sect. 6.4 concludes this chapter.

6.1 Service Discovery

As introduced in Sect. 1.3, *service discovery* in an SOA addresses the question whether there exist published services in the service registry that satisfy the specified search criteria of a client. *Behavioral service discovery* refines this question by considering only those published services that are behaviorally compatible to the client service. For being able to *decide* behavioral compatibility, we propose to publish the operating guidelines of the provider services to the registry. The operating guideline OG_S of a service S then serves as a description of the behavior of S that can be used to decide behavioral compatibility of some client service R with S.

A comprehensive approach to service discovery taking into account behavioral, semantical, and non-functional aspects of services is an open research question

crucially important for the success of service-orientation [PTDL08].

The whole process of service brokering under support of operating guidelines comprises the three steps of (1) computing the operating guideline OG_S of S, (2) publishing the computed operating guideline OG_S, and (3) discovering a provider service S which is behaviorally compatible to a querying client's service R by using the published operating guideline OG_S. These three steps are reasonably organized as follows.

6.1.1 Computation of Operating Guidelines

Given a newly created provider service S which shall be made publicly available, the first service brokering step considers the computation of the operating guideline OG_S of S. As described in Chap. 5, the OG computation is directly based on the knowledge values of S and the operations closure and event. That is, the *computation* of operating guidelines requires detailed knowledge of the inner structure of S. As this kind of information is often subject to trade secrets, the responsible SOA role to perform the OG computation step is usually the service provider of the service S itself.

As the knowledge values are only needed during the computation of the operating guideline OG_S of S, they can be discarded once the computation of OG_S is finished. Then, OG_S represents the canonical operating guideline of S according to Corollary 5.4.10 and hides all relevant internals of S. Only the fact *that* a client service satisfies or violates the operating guideline is preserved — the reason *why* is hidden. Accordingly, OG_S can then be published to the registry without interfering with privacy issues.

Another advantage of the computation of operating guidelines by the service provider is that the computational efforts for the OG construction are spent at *design time*. At this stage, investing the time and memory efforts needed to compute operating guidelines is reasonable.

6.1.2 Publishing of Operating Guidelines

The second service brokering step is to publish the computed operating guideline to the service registry. As the operating guideline OG_S of S contains enough information to decide behavioral compatibility between a client service R and the provided service S later on, OG_S thereby serves as a behavioral description of S according to the extended SOA's foundation layer (cp. Fig. 2.2 on page 36).

A normalization and minimization of all published operating guidelines is important in order to reduce the storing capacity requirements for operating guidelines in the registry. Both steps are uncritical for privacy issues and can be performed

by the service provider before publishing OG_S or by the service broker during or after the reception of OG_S.

Additional information describing QoS related properties of the service S, the semantical meanings of the messages and operations of S, or information about the provider itself should accompany the operating guidelines to enable the client to evaluate the additionally specified search criteria in the third step.

6.1.3 Service Discovery with Operating Guidelines

The last step corresponds to the actual SOA operation of service discovery (which is called "find" in the basic SOA of Fig. 2.1). Therein, the client specifies certain search criteria, typically including minimal QoS requirements and business goals the client wants to achieve, and searches the repository for fitting published services. If such services could be found, the client may then select one of these services to collaborate with. This selection can be guided by the expected costs of using the provider service or other preferences of the client.

Operating guidelines as published behavioral descriptions in the registry now enable the client to also take into account behavioral aspects of the selected service. To this end, the client service R should be matched with the operating guidelines of the fitting services in order to detect behavioral errors like deadlocks in the interaction *before* the actual binding has taken place. That way, published services that are not behaviorally compatible to the service R of the client can be filtered and are not considered for the final selection.

To *decide* behavioral compatibility, we have introduced the matching procedure in Sect. 4.2. This procedure amounts to a depth-first traversal of both the service and the operating guideline while evaluating the annotations of the states of the operating guideline. This is a very efficient procedure and therefore well suited for service discovery at *runtime*.

As the matching procedure requires detailed knowledge of the internal structure of the client service R, it should be performed by the client itself.

6.1.4 Conclusion

The operating guidelines approach applied to a service brokering of behaviorally compatible services as described above takes into account all aspects motivated in Sect. 1.3. It respects the privacy issues of both the client and the provider. To this end, the approach separates the computation of the operating guideline OG_S by the provider and the matching of R with OG_S by the client. The only construct that is exchanged between both parties is the operating guideline OG_S, which hides all relevant internals of S. Furthermore, our operating guidelines approach shifts a large part of the computational efforts to the design phase

before publishing a service S. The remaining runtime operation in this setting amounts to a matching procedure of R and OG_S that is very efficient compared to a model checking of the composition of R and S for behavioral errors. That way, we take into account that the publish operation of an SOA typically happens less often than the discovery operation.

Section 7.4 presents a case study comparing the efforts for computing the operating guideline OG_S of a service S and matching several services R with OG_S on the one hand with the efforts to verify behavioral compatibility of R and S by model checking the composition $R \oplus S$ for each service R on the other hand. We will show that even the efforts spent at design time easily pay off after a few matchings. That is, even the *sum* of computing OG_S once and checking n matchings of services R_1 to R_n is smaller than model checking n compositions $R_i \oplus S$ for deadlocks — already for a small number n.

As a prerequisite for being able to apply operating guidelines for service discovery, the OG approach requires the formal modeling of the provider service S (for being able to construct the operating guideline OG_S of S) as well as the client service R (for matching R with OG_S). Without a formal model of services, behavioral errors are hard or even impossible to detect [LMSW06, LMSW08].

6.2 Substitutability of Services

Service substitutability is another important open research challenge raised in the SOC research roadmap of [PTDL08]. *Behavioral* service substitutability considers the following question. Given an existing, well-behaving service interaction (specified either as an orchestration or as a choreography) involving n services S_1 to S_n, under which conditions can one of these services, say S_1, be substituted by another service S_1' such that the overall service interaction of S_1' with the services S_2 to S_n is still well-behaving.

In general, the reasons for a substitution of a service S can be manyfold. For instance, the owner of the service may want to add a new functionality to S or has to restructure S to improve its quality parameters, or S is currently just not accessible and another service providing the functionality of S is needed by the remaining parties (organizations) to achieve the overall business goal.

Another setting in which service substitution plays an essential role is known as the *contract approach* to interorganizational business processes. Therein, the parties involved in a future collaboration jointly specify a global, abstract specification of the overall target process and agree on a distribution of the tasks of the process to the single parties. The overall process specification together with the duties of each party serves as the *service contract*. The contract can be seen as a service composition of n services S_1 to S_n where each service S_i represents the specification for the party i. After the contract has been successfully specified

(and analyzed for its behavioral correctness), each party locally *implements* its specification S_i. That way, the service S_i is substituted by a service S_i', and the actual execution of the contract involves the new services S_1' to S_n'.

In both settings, service substitutability of S by S' is devoted to guarantee that substituting S by S' does not alter well-behavior of any interaction that S is involved in.

One of the main challenges to *decide* service substitutability of S by S' is again found in the restrictions given by privacy issues. The collaborating parties may be competitors in other business areas and usually do not want to reveal any internals of their services to the other parties. Hence, service substitutability must be checked *locally* for each party on the one hand, but nevertheless assure the global behavioral correctness of the whole composition on the other hand. Thus, we are looking for local relations between the services S and S' guaranteeing that no composition of S with other services is "affected" by this substitution.

6.2.1 Behavioral Substitutability of Services

In [SMB09], we have proposed a number of general behavioral substitutability notions. Therein, the most basic service substitutability relation is called *behavioral equivalence* of two services S and S'. It states that behaviorally equivalent services are behaviorally compatible to exactly the same environments. In our terms, S and S' are behaviorally equivalent if they have the same strategies.

Definition 6.2.1 (Equivalent services).
Two services S and S' are *behaviorally equivalent* if $\mathit{Strat}(S) = \mathit{Strat}(S')$. ⌟

If S and S' are behaviorally equivalent, then S can be *equivalently substituted by* S'. That is, each well-behaving service composition that S is involved in is also well-behaving with the new service S' replacing S in this composition (and vice versa).

Corollary 6.2.2 (Equivalent substitution sufficient for well-behavior).
For all services R and all behaviorally equivalent services S and S': $R \oplus S$ is well-behaving iff $R \oplus S'$ is well-behaving. ⌟

As R itself can be a composition of services, this substitutability notion can be applied to arbitrary service compositions S_1 to S_n. Considering the service S_1 for substitution, the services S_2 to S_n correspond to the service R in the previous corollary and the service S_1 can be equivalently substituted by any behaviorally equivalent service S_1'.

If we restrict ourselves to finite service compositions with b-bounded communication (for some message bound b), i.e. to the preservation of the set $\mathit{Strat}_b(S)$ by

S', we derive the notion of (behavioral) *b-equivalence* of the services. It is easy to see that operating guidelines can be directly used to decide b-equivalence of services S and S'.

Corollary 6.2.3 (Decision of equivalent substitution).
Let b be some message bound. Then, two services S and S' are b-equivalent iff $OG^b_S \equiv OG^b_{S'}$. ⌟

That means that we can decide b-equivalence of two services S and S' by computing the operating guidelines OG^b_S and $OG^b_{S'}$ of S and S' and checking the *BSA* equivalence relation $\equiv$ for the operating guidelines (cp. Sect. 4.5.1) which can be decided on the structures of the operating guidelines only.

A drawback of this substitutability notion is that behavioral equivalence is a very strict requirement for services allowing for rather small differences between the services S and S'. For this reason, a more relaxed substitutability relation between S and S' is needed. Consequently, we have introduced the notion of *substitution under accordance* in [SMB09]. Thereby, a service S' accords with a service S if S' is behaviorally compatible to at least all strategies R of S. Formally, we have:

Definition 6.2.4 (Accordance relation).
A service S' *accords* with a service S if $Strat(S') \supseteq Strat(S)$. ⌟

The accordance relation between S and S' allows that the new service S' can have more strategies than S, which is a more relaxed notion than behavioral equivalence of S and S'. According to the following corollary, substituting a service S by a service S' which accords to S does not restrict any current composition of S with another (simple or composite) service R.

Corollary 6.2.5 (Accordant substitution sufficient for well-behavior).
For all services R and S and all services S' that accord with S: $R \oplus S$ is well-behaving implies that $R \oplus S'$ is well-behaving, too. ⌟

Again considering only finite services that communicate b-boundedly, we can decide *b-accordance* using operating guidelines, too.

Corollary 6.2.6 (Decision of accordance).
Let b be some message bound. Then, a service S' b-accords with a service S iff $OG^b_{S'} \sqsupseteq OG^b_S$. ⌟

That is, we can reduce the (b-)accordance relation question of two services S and S' to the problem of deciding the smaller relation $\sqsubseteq$ (as introduced in Sect. 4.4) of their corresponding operating guidelines.

Both substitutability notions, equivalent substitution and substitution under accordance, are defined on the (usually infinite) sets of (b-)strategies of the services

S and S'. Hence, checking structural criteria for deciding the equivalence or preorder relation of the operating guidelines of S and S' is a valuable tool for deciding basic behavioral substitutability notions for services.

6.2.2 Multiparty Service Contracts

A currently quite common approach to realize interorganizational business processes is known as the *contract approach* [AW01, ALM+07, ALM+09]. Therein, a contract serves as a common agreement on the "rules of engagement" between the parties of a future collaboration. The contract can be seen as a composite service consisting of n service specifications S_1 to S_n for the parties (agents) A_1 to A_n. Each service S_i is called the *public view* of the party A_i and describes the duties for this party in the contract.

Definition 6.2.7 (Contract, public view).
Let $\mathcal{A} = \{A_1, \ldots, A_n\}$ be a set representing the involved parties. Then, a *contract* is a service $\mathcal{C}$ such that

- $\mathcal{C}$ is a well-behaving composition of services S_1 to S_n,
- each service S_i belongs to the party A_i and is called *public view* of A_i. ⌟

An important requirement for contracts is that the public views leave most implementation details for the parties open. That way, a party is free to optimize the implementation of its share in the contract. The new, implemented service S_i' of the party A_i then serves as the *private view* of A_i. While the contract itself is agreed upon the public views S_i, the actual execution of the contract is based on the private views S_i' of the parties.

Accordingly, each public view must be (behaviorally) substitutable by the corresponding private view. To this end, different substitutability notions between public and private view can be considered that allow for varying degrees of freedom for filling in implementation details. In [ALM+07, ALM+09], we have shown that accordance is a well-suited substitutability notion in this setting. Accordance is relaxed enough to allow for, e.g., reordering or combining the sending or receiving of messages as well as the introduction of concurrency in the private view. Accordance further allows for introducing alternative branches that will never be executed in the contract. That way, a party may easily reuse an already existing implementation S' with more functionality than is actually needed in the current contract setting [ALM+07, ALM+09]. In sum, accordance is a weaker substitutability notion than behavioral equivalence and even weaker than *projection inheritance* [AB02, BA01], an established refinement notion for workflow nets. Correspondingly, accordance allows for using more services S_i' as behaviorally correct implementations of the contract.

The main result of using accordance in the contract setting is that even if *each* party *independently* implements its public view (and locally analyzes the accordance relation between public and private view), the overall contract implementation is still guaranteed to be well-behaving [ALM+07, ALM+09].

Theorem 6.2.8 (Implementation of a contract).
Let $\mathcal{C}$ be a (well-behaving) contract between parties $\mathcal{A} = \{A_1, \ldots, A_n\}$. If, for all $i \in \{1, \ldots, n\}$, S'_i (the private view of A_i) accords with S_i (the public view of A_i), then the composition $\mathcal{C}'$ of the private views S'_1 to S'_n (i.e. the actual implementation of the contract) is well-behaving, too. ⌟

To *check* the accordance criteria between the public and private views of each party A_i required in this theorem, we can once more apply operating guidelines. Therefore, each party *locally* computes the operating guidelines of both its public and its private view and checks the smaller relation of these OGs.

As an advantage over the general substitutability setting, the needed message bound b for the computed operating guideline can be generated automatically from the contract $\mathcal{C}$ by counting the number of simultaneously pending messages between the service specifications S_i in the original contract $\mathcal{C}$.

6.2.3 Conclusion

Many general behavioral substitutability notions are based on a relation between the sets of strategies for a service S and its new version S'. As two examples for such substitutability notions, we have introduced equivalent substitution and substitution under accordance which are based on behavioral equivalence and accordance between the services S and S', respectively. Whereas behavioral equivalence between S and S' requires that S and S' have the same strategies, the accordance relation allows that the new service S' can have more strategies than the substituted service S. Operating guidelines can be used to decide these relations.

In the setting of multiparty service contracts, operating guidelines are well suited to characterize all valid private implementations of a public view of a party's share in a contract. Thereby, all substitutability checks can be performed locally to each party. Thus, operating guidelines help to overcome privacy issues that are encountered when organizations collaborate in interorganizational business processes such as service contracts.

6.3 Service Synthesis for Adapter Generation

Often, services are *not* behaviorally compatible. In this case, they cannot interact with each other without the risk to run into severe behavioral errors such as dead-

locks, for instance. Operating guidelines may help to filter such services. They can be used to discover, select, and bind only those services R and S that are behaviorally compatible to each other. However, in some cases, the incompatibilities between two services R and S are rather small and could be avoided/repaired by *mediating* between the services. Such mediators between services are called *adapters*. A *behavioral adapter* A is devoted to (re-)organize the message flow between the (behaviorally incompatible) services R and S such that the overall composition of R, A, and S is well-behaving. In this sense, the adapter A is a distinguished service which is itself behaviorally compatible to the composition of the services R and S.

A suggesting approach to *automatically generate* such an adapter A is to employ our construction method for a behaviorally compatible strategy service for the services R and S. That is, the computation of the operating guideline $B^{\phi} = OG_{R \oplus S}$ of the composition $R \oplus S$ of the services R and S, as well as methods to translate the underlying service automaton B of $OG_{R \oplus S}$ into a strategy service net for $R \oplus S$ can be used for adapter generation.

However, the requirement that the adapter is behaviorally compatible to the composition $R \oplus S$ is not sufficient. It allows for very trivial and unintended adapters that arbitrarily generate messages which are not yet available. This is obviously not intended for messages that represent real trade items or confidential data like passwords. Hence, the adapter must further respect certain rules for the manipulation of messages. Such rules are often expected as additional input to the adapter generation task.

This approach has also been followed in [GMW08]. Therein, the authors propose to use a *specification of the elementary activities* (SEA) as such adapter rules. The SEA describes which activities the intended adapter is allowed to perform for which messages. It is different for each pair of adapted services R and S and the concrete rules are determined by the semantics of the messages that are exchanged between R and S as well as security and privacy issues. The SEA constitutes the engine part E of the adapter A. Using these rules, a controller part C of A is synthesized controlling the message flow between R and S by only using the allowed adapter rules of the engine part E. If such a controller exists, the adapter is the composition $A = E \oplus C$ and is behaviorally compatible to the services R and S by construction. This concept is illustrated in Fig. 6.1.

In the rest of this chapter, we follow the approach of [GMW08] and assume *service net models* of the adapted services R and S and consider the generation of a service net adapter A. The results, however, are not limited to service nets and can be easily formulated for service automata or other formal methods for which the operating guidelines (or another controller synthesis) approach can be applied. The whole adapter generation approach using operating guidelines is organized as follows.

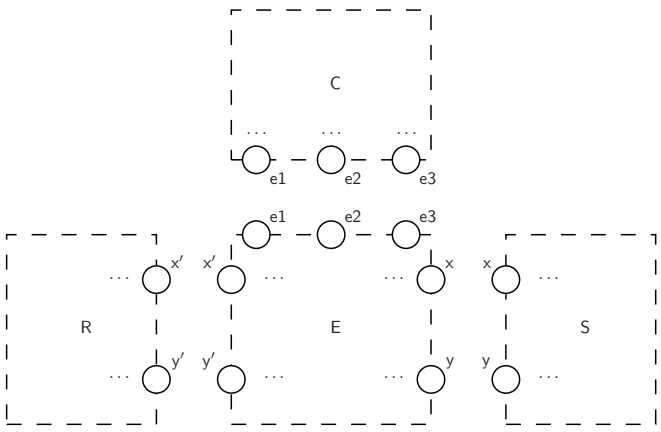


Figure 6.1: Schematic overview of the adapter generation for service nets R and S. The adapter A consists of the engine part E and the controller part C, i.e. $\mathsf{A} = \mathsf{E} \oplus \mathsf{C}$.

6.3.1 Specification of Adapter Rules

In a first step, the possibilities of the adapter with respect to the messages that are exchanged between the services R and S have to be specified as adapter rules, constituting an SEA. The SEA describes whether the intended adapter is allowed to create, copy, delete, transform, split, merge, or reroute a message of a specific type. An SEA can be derived from semantical annotations to the interfaces of R and S, for instance. Alternatively, the rules can be specified manually by a service designer responsible for the adaption of R and S, or they can be inferred from commonly agreed-upon rules in a fixed cooperation environment.

If the rules are specified, they are translated into the engine part E of the adapter. To interconnect E with the original services R and S, these services are completely decoupled by renaming their interface places such that $P_{io\,R} \cap P_{io\,S} = \emptyset$ first. Then, the engine part E may be established as a connection between R and S by naming the interface of E correspondingly (cp. Fig. 6.1).

Additional interface places of E enable the controller part C of the adapter to trigger the individual rules. Therefore, C can control the order of applying the adapter rules, and it can send and receive the messages to/from the services R and S at the right moment.

6.3.2 Synthesizing the Controller Part

The composition $R \oplus E \oplus S$ is an open service net representing the new communication possibilities of R and S with respect to the adapter rules. The next step of the adapter generation is now devoted to find/choose the right order of

performing the rules such that the overall composition is well-behaving. That is, we search for a strategy service net for the composition $R \oplus E \oplus S$. If such a strategy is found, it constitutes our controller part C of the adapter and assures well-behavior by construction. If no strategy exists for $R \oplus E \oplus S$, then R and S are *not adaptable* — at least not with the specified adapter rules.

To *construct* the controller part, we can apply the operating guideline construction for the composition $R \oplus E \oplus S$. In fact, the resulting operating guideline $OG_{R \oplus E \oplus S}$ characterizes *all* such possible controllers C (for a given message bound). More precisely, the operating guideline $OG_{R \oplus E \oplus S}$ of $R \oplus E \oplus S$ characterizes all service automata that represent behaviorally compatible *behaviors* of strategy service nets M for $R \oplus E \oplus S$. That is, each service net M such that the corresponding service automaton $SA(M)$ is characterized by $OG_{R \oplus E \oplus S}$ is a valid candidate for the controller part C of the intended adapter.

The actual selection which M is to take as controller part of the adapter depends on the expected costs for storing, rerouting, or copying the messages exchanged between R and S, for instance. To this end, we suggest to annotate the specified adapter rules with costs for being able to select the smallest M with respect to these costs.

6.3.3 Composing Engine and Controller Part

Having successfully computed (and selected) the controller part C of the adapter, the overall adapter A is the composition of both the engine part E and the controller part C, i.e. $A = E \oplus C$. It is behaviorally compatible to the composition $R \oplus S$ of the original services R and S by construction and, thus, the overall composition $R \oplus A \oplus S$ is well-behaving. Furthermore, the adapter A respects all adapter rules (i.e. the SEA) specified by the service designer of the adapter, or derived from semantical annotations of the interfaces of R and S.

A further structural reduction of the adapter A may help to simplify a further analysis of the composition $R \oplus A \oplus S$ and may reduce the storage requirements for the adapter A as well as enhance the performance of A.

6.3.4 Conclusion

The generation of a behavioral adapter for behaviorally incompatible services R and S consists of two steps. Firstly, adapter rules for the adapter are specified, defining the capabilities of the intended adapter with respect to which action (create, copy, delete, transform, split, merge, or reroute) the adapter is allowed to perform for which message type exchanged between R and S. The adapter rules define the engine part E of the adapter A.

Secondly, a controller is synthesized as a service behavior which is behaviorally compatible to the composition $R \oplus E \oplus S$. To this end, we can apply the operating guideline computation for $R \oplus E \oplus S$. The corresponding operating guideline $OG_{R \oplus E \oplus S}$ characterizes all such behaviorally compatible behaviors. As the operating guideline is an *operational* characterization, one can easily derive one of these characterized behaviors from the operating guideline. The selection may be guided by the expected costs or by other aspects of the target adapter. It may either be the underlying service automaton of the operating guideline $OG_{R \oplus E \oplus S}$ itself or any other matching service automaton. Then, the selected behavior is translated into a service net, representing the controller part C of the adapter. For the translation, we can either apply our construction PN (introduced in Sect. 3.4.3 on page 82) or use the theory of regions approach [BD98]. Alternatively, any service net C where $SA(C)$ matches with the operating guideline $OG_{R \oplus E \oplus S}$ is well suited as controller part.

The overall adapter A is then the composition of E and C and imposes the property of well-behavior for the interaction between R, A, and S.

6.4 Conclusion

Operating guidelines as a characterization of the set of behaviorally compatible services R for a given service S constitute a flexible and convenient concept that can be employed in a variety of analysis questions in the context of service-oriented computing. They can be used to support service discovery in an SOA, to decide a number of service substitutability relations, and to synthesize behavioral adapters for services. These are rather different applications, but are all particularly important and listed as open research challenges for the near future [PTDL08].

Thereby, different properties of operating guidelines contribute to the applicability in these areas. In the application setting of service discovery, operating guidelines can be used as a behavioral description of a published service S. That way, a querying client benefits because the operating guideline of a published service S provides sufficient information to *decide behavioral compatibility* of its service R with S before both services are bound together. The provider, in turn, can publish the operating guideline without risking to reveal crucial trade secrets of its service S.

In a service substitutability setting, the operating guidelines approach enables a service owner to *locally* check the desired behavioral substitutability relation — like behavioral equivalence or accordance — of an existing service S and a new version S' of S while assuring the global property of well-behavior of the overall service composition when S is substituted by S'. Thereby, the comparison of two usually infinite sets of strategies of S and S' is reduced to a simple decision of

structural criteria — like the equivalence relation $\equiv$ or the smaller relation $\sqsubseteq$ — of the corresponding operating guidelines of S and S'.

Finally, operating guidelines are *operational* descriptions of the set of strategies for a service. Hence, they can not only be used as a basis for the decision of behavioral compatibility, but may also be employed for the *synthesis* of behaviorally compatible services for a given service. This property of operating guidelines allows for their use in the adapter setting, where a controller part of the adapter can be synthesized with the help of the underlying service automaton of an operating guideline.

In sum, the operating guideline of a service S is a finite, abstract, and operational characterization of the set of behaviorally compatible services for S. It supports efficient methods for deciding whether a service R is characterized by the operating guideline of S, respects privacy issues and allows for the local check of substitutability notions, and it supports the synthesis of a behaviorally compatible service R for S. That is, the concept of operating guidelines as introduced in the last chapters of this thesis takes into account all requirements introduced for such a characterization in the research goal of Sect. 1.3.

7 Implementation in the Tool Fiona

In Part II of this thesis, we have developed different formal notions and concepts for analyzing the interactional behavior of services. However, the definition of operating guidelines (specifically Definition 5.4.2 of the basis $\mathcal{F}^b$ of an OG), for instance, at most sketches an algorithm to really construct the operating guideline OG_S for a given service S. Furthermore, all definitions are optimized for proving our theoretical results and mostly not suited to be implemented easily.

For being able to show the feasibility of our concepts, all results presented in this thesis have been implemented in Fiona, a tool to check behavioral compatibility of two services and several other behavioral properties of services.

This chapter is devoted to introduce the implementation of the most important results of this thesis in the analysis tool Fiona.

To this end, this chapter is organized as follows. In Sect. 7.1 and Sect. 7.2, we will shortly introduce the tool Fiona and give an overview of its functionality. Section 7.3 then provides implementation details of the computation of operating guidelines and the minimization of *BSA*s. For these two algorithms, the implementation in Fiona significantly deviates from the theory as presented in this thesis. We will point out the differences and explain our design decisions. In Sect. 7.4, we then present experimental results illustrating the sizes of operating guidelines of real-world services and the respective computation time needed to construct the respective OG. Furthermore, we compare the discovery of behaviorally compatible services using operating guidelines with the service discovery approach of verification behavioral compatibility using model checking. Finally, Sect. 7.5 concludes this chapter.

7.1 About Fiona

The development of Fiona [LMSW06, MW08] started in the end of 2005 as a rather small project at the theory of programming group of Wolfgang Reisig

(Humboldt-Universität zu Berlin, Germany) and was partially funded by the Federal Ministry of Education and Research (BMBF) project "Tools4BPEL". Currently, Fiona is developed in close cooperation with the group of Karsten Wolf (University of Rostock, Germany). It involves more than 10 developers and uses a subversion repository to coordinate the distributed development. The author is one of the core developers of Fiona and responsible for the operating guidelines part of Fiona and many related parts.

Fiona is written in C++ and released as free software under the terms of the GNU General Public License. Fiona's distribution is based on the GNU autotools, which provide the possibility to run Fiona on many operating systems. The latest source code, a documentation, and different precompiled binaries are available for download at `http://service-technology.org/fiona`.

Fiona is designed to be used as a background service of existing service modeling tools. Thus, Fiona provides no own graphical user interface (GUI) — the analysis task as well as the input file(s) are given to Fiona via command line options. Fiona then computes and reports the result and, if needed, generates the output file(s).

7.2 Functionality of Fiona

The functionality of Fiona comprises all results presented in Part II of this thesis. Fiona may compute operating guidelines, match a service with an operating guideline, normalize and minimize operating guidelines, and compare two operating guidelines with respect to the sets of characterized services. Furthermore, Fiona can synthesize behavioral adapters and perform several other functionalities.

The implementation, however, in some cases differs significantly from the theory for the respective result. This is mostly caused by the used data structures which are designed to optimize memory usage and to speed up computation. One of the most fundamental differences is that the theory is based on (and was presented for) service automata in this thesis, but Fiona uses Petri net representations of services instead. That is, the main input of Fiona is a service net, and the algorithms of Fiona are directly based on Petri net technologies. Figure 7.1 depicts an overview of the main functionalities of Fiona in terms of a service net.

Thereby, a functionality of Fiona is represented as a transition in Fig. 7.1. The name of a transition is depicted in the upper box of the transition, the (most important) internal data structures that are used by a functionality are placed as boxes at the bottom of the corresponding transition, and major components (algorithms) used are mentioned as boxes in the middle of a transition. For instance, the functionality OG computation uses an internal representation of a service net for which the *OG* shall be computed, a separate data structure to

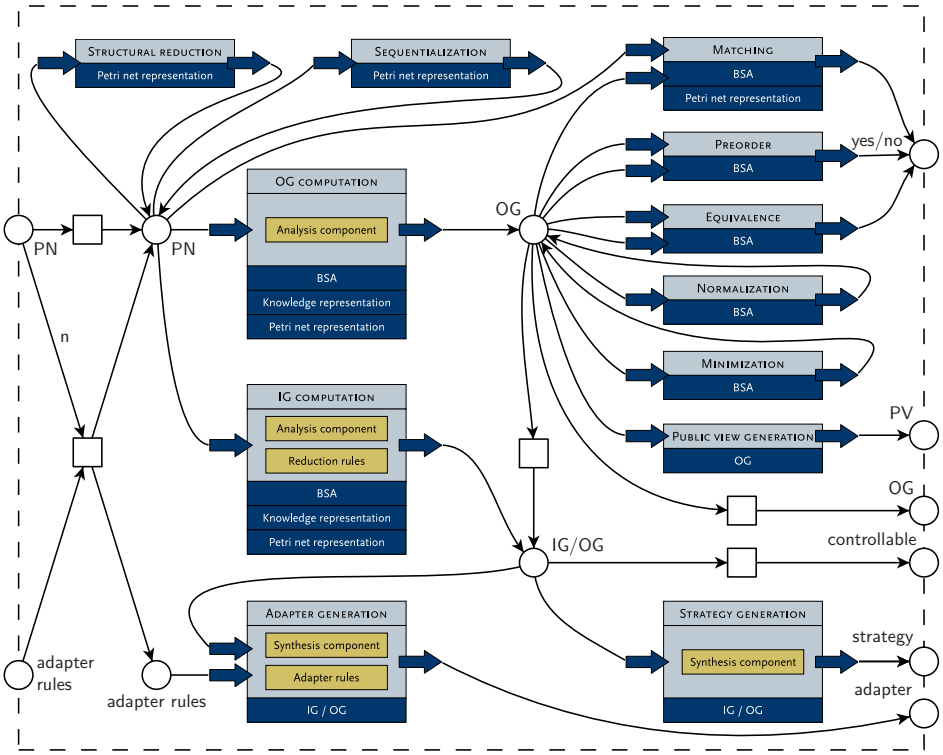


Figure 7.1: The main functionality of FIONA depicted as a service net $\mathsf{N}_{\mathsf{fiona}}$.

store the knowledge values needed to compute the OG, and a data structure for the Boolean annotated service automaton (BSA) that represents the basis of the OG. The computation of OG_N of a service net N uses an analysis component that is devoted to evaluate the knowledge of OG_N about N and to compute the annotation of a state of OG_N.

We consider the following scenarios as the most important analysis features of FIONA.

Operating guidelines construction. In this scenario, we use FIONA to construct the operating guideline OG_N of a service net N. To this end, we expect a service net file as input (represented by the input place labeled PN in $\mathsf{N}_{\mathsf{fiona}}$), which is parsed to an internal representation of N (represented by the PN-labeled internal place of $\mathsf{N}_{\mathsf{fiona}}$). If needed, N may be structurally reduced (transition structural reduction in Fig. 7.1). Then, FIONA computes the operating guideline OG_N of N as a BSA including the knowledge of OG_N about N (transition OG computation). Depending on the parameters, OG_N is normalized and/or min-

imized before it is written to a file representing the output of this scenario (output place OG). Thereby, the knowledge is not written to the file but is discarded.

For constructing the operating guideline of a service net, the corresponding message bound (cp. Sect. 5.3) can be passed to FIONA via a command line option. If not specified, it is set to 1 as the default value.

Decision of controllability. In this scenario, FIONA is used to decide controllability of a service net N. We therefore expect a service net which may again be structurally reduced. Then, we construct a special *BSA* IG_N (from *interaction graph*) for N (transition IG computation). Basically, IG_N is a reduced version of OG_N that does not characterize *each* strategy for N, but *a* strategy (if there is one). To this end, several *reduction rules* are applied that are proven to preserve controllability, but result in a much smaller underlying service automaton of IG_N than the strategy overapproximation $\mathcal{F}$ of OG_N (cp. Sect. 5.4). If IG_N is nonempty, then N is controllable. Otherwise, N is uncontrollable. The respective result is stated by FIONA (represented by the output place controllable in Fig. 7.1).

Note that in principle also operating guidelines can be used to decide controllability which is represented in Fig. 7.1 by the preplace IG/OG of the output transition generating the controllability result in this scenario.

Matching. To match a service net M with the operating guideline OG_N of a service net N (cp. Sect. 4.2), FIONA expects both service nets N and M as input, generates the operating guideline out of N, and then applies the matching algorithm for M and OG_N (transition matching). As the matching algorithm needs to compute a simulation relation between M and OG_N, we have to sequentialize the service net M first (transition sequentialization). As the output in this scenario, FIONA states whether or not M matches with OG_N. In case that M does not match with OG_N, FIONA also reports diagnostic information, i.e. the marking of M and the state of OG_N where either the simulation relation is violated (M has an x-labeled transition not allowed by OG_N) or the assignment of M violates the annotation of OG_N.

Normalization. For a given operating guideline OG_N of a service net N, FIONA can be used to normalize OG_N (transition normalization). Because many functionalities require normal operating guidelines (cp. Sect. 4.3), normalization is default in FIONA and thus applied to all computed operating guidelines. However, it can be switched off by a command line option — mostly used for debugging purposes.

Minimization. Operating guidelines can also be minimized by FIONA (transition minimization) which is, however, not enabled by default. The minimization of an operating guideline requires a normal operating guideline to work correctly (cp. Sect. 4.5). Thus, the minimization scenario always performs a normalization first.

Preorder and equivalence. Given two service nets N and M, FIONA can compare the corresponding sets of strategies $Strat(N)$ and $Strat(M)$ using the operating guidelines OG_N and OG_M. $Strat(N)$ and $Strat(M)$ can be checked for an inclusion relation (transition preorder) of for equality (transition equivalence). In case a check fails, the reason for failure (i.e. the point where the simulation relation or the annotation implication fails) is reported. As in the minimization scenario, both checks require normal operating guidelines (cp. Sect. 4.4 and 4.5.1) and thus perform a normalization first.

The preorder and equivalence scenarios can be used to decide substitutability of the services N and M as described in Sect. 6.2.

Strategy generation. The underlying service automaton of both IG_N and OG_N represents a behaviorally compatible strategy service *automaton* for the service net N. Thus, we may translate this service automaton into a service net in order to construct a strategy service *net* M for N (if N is controllable). However, we do not apply the translation $PN(A)$ as introduced in Sect. 3.4, but use the theory of regions approach [BD98, CKLY98] instead. This results in smaller and more readable service nets. To this end, FIONA calls the tool Petrify [CKK+02] (available at `http://www.lsi.upc.es/~jordicf/petrify/distrib/home.html`).

Adapter generation. As described in Sect. 6.3, FIONA can be used to synthesize a behavioral adapter for n incompatible services N_1 to N_n. To this end, all n service nets as well as the adapter rules (representing the engine part E of the adapter) are given to FIONA as input. Then, in a first step, the nets and the adapter rules are composed to a single, overall service net N^*. For N^*, we then construct either IG_{N^*} or OG_{N^*}, depending on the parameters that are given to FIONA. Again, the underlying service automaton of both IG_{N^*} and OG_{N^*} represents a behaviorally compatible strategy service automaton for N^*. Hence, we translate this service automaton into a service net M (again using Petrify). M represents the controller part of the adapter. The overall adapter consists of both engine and controller part, i.e. it is the service net $E \oplus M$. This net is written to a file representing the output of this scenario (output place adapter of $\mathsf{N}_{\mathsf{fiona}}$).

Public view generation. Based on the operating guidelines concept, Karsten Wolf has formulated a formal notion of a public view N' of a service N and a construction method for N' using the operating guideline OG_N of N. Therein, OG_N is computed in an intermediate construction step in order to completely abstract from the internal structure of N and to preserve only the behaviorally compatible interaction possibilities of strategies for N. Then, the public view N' of N is constructed from OG_N by "reversing" the OG construction. That is, from the abstract representation OG_N, a service net N' is reconstructed that has OG_N as its operating guideline. Obviously, N and N' are equivalent with respect to their strategies by construction. Due to the abstraction achieved by OG_N, N' is a suitable candidate for a public view of N. Although the results are not yet

published, they are already implemented in another application scenario of Fiona (transition public view generation).

Please notice that Fig. 7.1 is a rather abstract representation of the functionalities of Fiona. Some functionalities are not depicted in the figure. As an example, Fiona may also directly read operating guidelines from a file. That way, it supports the matching of a service net M with the operating guideline OG_N without having to (re-)compute OG_N first, for instance. Furthermore, some functionalities need additional internal steps that are not represented in Fig. 7.1 or have preconditions (as a prior normalization before a minimization, for instance) which are not shown in Fig. 7.1.

Another important feature of Fiona is that we do not differentiate between operating guidelines and arbitrary *BSA*s in Fiona. This means that we can read, normalize, minimize, etc. any *BSA* that is given to Fiona as a file.

7.3 Implementation of the Results

In this section, we will introduce the implementation of the results of this thesis in Fiona and explain the differences of the implementation compared to the theoretical algorithm suggested by the corresponding theoretical result.

Because the implementation of the normalization of a *BSA* (Sect. 4.3), as well as the decision of the preorder between two *BSA*s (Sect. 4.4) are implemented rather straightforward, we will not elaborate on the implemented algorithms for these scenarios.

The implementation of the matching procedure of a service net M and a *BSA* B^ϕ bears also no big surprise. First, M is sequentialized. Then, the reachability graph of the inner of M is constructed which basically equals the translation $SA(M)$ as introduced in Sect. 3.4.2. Already during this construction, the currently reached marking of M is matched with the current state of B^ϕ. It is worth to mention that we use a *weak* simulation matching approach in Fiona. That is, we only check markings m of M that enable an interface transition of M for the annotation satisfaction of B^ϕ. This implements the considerations already introduced in Sect. 4.6.2.

The only important difference in the implementation of the equivalence of two *BSA*s B_1^ϕ and B_2^ψ (Sect. 4.5.1) is that the implementation does not check two independent preorder relations $B_1^\phi \sqsubseteq B_2^\psi$ and $B_2^\psi \sqsubseteq B_1^\phi$ (as suggested by Corollary 4.5.2), but constructs the minimal simulation relation ϱ between B_1 and B_2 only once. As both B_1 and B_2 are deterministic, we can rely on Proposition 4.1.5 and check for each pair $(q_1, q_2) \in \varrho$ whether or not (1) q_1 simulates q_2, and (2) q_2 simulates q_1, and (3) $\phi(q_1)$ implies $\psi(q_2)$, and (4) $\psi(q_2)$ implies $\phi(q_1)$. Only if all four conditions are satisfied by each state pair, then B_1 and B_2 are equivalent.

As the scenarios of deciding controllability, constructing public views, and synthesizing strategies or behavioral adapters are not the focus of this thesis, we will omit an introduction of the implementation thereof and concentrate on the implementation of the minimization of a *BSA* (Sects. 4.5.2 and 4.5.3) and the construction of operating guidelines (Chap. 5) in the following.

To this end, we will first describe the basic data structures used to represent service nets, to store knowledge sets about a service net, and to represent *BSA*s (and thus operating guidelines). Afterwards, we will introduce the implemented construction of the operating guideline OG_N of a service net N. Finally, we will show how a *BSA* is minimized in Fiona. In either case, the implementation is compared to the theoretical algorithm suggested by the corresponding theoretical result.

7.3.1 Basic Data Structures

As mentioned earlier in this chapter, Fiona is based on Petri net (more precisely service net) representations of services rather than service automata. The reasons for this fundamental design decision are threefold. Firstly, Petri nets are, from our perspective, much better suited to manually design services than automata dialects — Petri nets have a nice, compact graphical representation and the designer does not have to specify each interleaving separately in case of concurrency. This significantly eases the modeling of test examples and allows for the design of larger case studies. Secondly, the Petri net basis enables us to directly use the output of the compiler BPEL2oWFN [Loh07], which translates services specified in BPEL or BPEL4Chor into Petri net models of the services. Both reasons suggest to use Petri nets at least as a possible input of Fiona. Last but not least, we have had the support of Karsten Wolf to reuse large parts of the code of the explicit Petri net model checking tool LoLA [Sch00] (available at `http://service-technology.org/lola`). The implementation of LoLA is designed to support huge state spaces by very efficient data structures to store Petri nets and their state spaces as well as several state-of-the-art state space reduction techniques. Thus, we decided to develop Fiona based on Petri net data structures and methods that originate from LoLA. Furthermore, the input formats of LoLA and Fiona are intendedly very similar.

Petri Net Representation

The Petri net representation of Fiona is very similar to the one of LoLA. To represent the additional interface, we extended the data structure by representations of the input and output places of a service net and by methods to manipulate the corresponding members.

It is relevant to notice that FIONA (like LoLA) distinguishes one special marking, called current marking, representing the marking that a net N is currently in. Initially, the current marking equals the initial marking of N. This marking is then changed by firing transitions of N. As the transition firing is the core activity of explicit state space generation, a Petri net transition t in FIONA "knows" which other transitions might get enabled or disabled by the firing of t. Thus, only these transitions have to be checked for enabledness after having fired t. That way, time needed for firing transitions is minimized. Thus, even backtracking is organized as a backfiring of afore fired transitions.

Knowledge Representation

In our theory, the knowledge of a *BSA* about a service automaton A at a state q of the *BSA* is a set of situations $[q_A, M]$ of A (cp. Sect. 5.1). As the service is represented by a service net N in FIONA, a situation of the service corresponds exactly to a marking of N. Thereby, the "state part" q_A of the situation corresponds to a marking of the inner of N, and the "message part" M of the situation corresponds to a marking of the interface places of N. Hence, the knowledge of a *BSA* at a state q of the *BSA* is nothing more than a set of reachable markings of N in FIONA.

Consequently, the representation of a knowledge set in FIONA has been adapted from the representation of a set of markings in LoLA. Therein, a marking $m(p)$ of a place p of N is represented by a bit vector. For storing a set $\mathcal{M}$ of markings $m \in \mathcal{M}$ of N, the first marking m of N is simply a concatenated bit vector of all markings $m(p)$ of the places of N (assuming a fixed order of the places). The more states are added to the data structure, the more it converges to a binary decision tree [Sch00]. Thereby, each decision point in the tree represents the first difference of the bit vectors of two markings with a common prefix. Using this data structure, the containment of a marking in the structure can be decided in linear time. Furthermore, although the overhead for storing a small number of states is rather high (due to the additional needs for organizing the binary decision tree), adding more markings to the data structure only requires a logarithmic amount of additional space.

The reachability information (i.e. the step from the set of reachable markings to the reachability graph) is stored separately from the storage structure.

BSA Representation

FIONA uses a class Graph to represent arbitrary service automata with states (which can be marked as initial or final), transitions, and transition labels. This class provides simple methods to traverse the graph and to add states and transitions, for instance. A child class of the class Graph is the class AnnotatedGraph,

representing *BSA*s. In an annotated graph, each state has an additional data member representing the Boolean formula that is annotated to the state and methods to manipulate the annotation. The AnnotatedGraph class has itself a child class, called CommunicationGraph. A communication graph object represents a *BSA* with knowledge. That is, a state of a communication graph has an additional data member representing the set of markings which constitute the knowledge set at this state. The markings itself are stored in the data structure as described above. The communication graph class provides general methods to perform the closure and event operations in order to construct a *BSA* with knowledge, and it enables Fiona to compute an annotation out of a knowledge set. Operating guidelines and interaction graphs are represented by classes OG and IG, respectively. Both classes inherit from the CommunicationGraph class. They provide methods designated to operating guidelines (like the exhaustive iteration over all possible events) and interaction graphs (like the reduction rules to construct small interaction graphs), respectively.

Because especially operating guidelines can have huge graph structures, all these classes are designed in order to be "slim". That is, whenever reasonable, we prefer to save memory by spending computation time. For instance, a graph state only knows its successors, but not its predecessors. This is sufficient for a depth-first traversal of the graph needed during the IG/OG computation but often requires additional steps in other scenarios, like the normalization or minimization of *BSA*s. As the decision of controllability of a service net N and the computation of the operating guideline of N are the core functionalities of Fiona, we have chosen in favor of this design decision.

7.3.2 Construction of Operating Guidelines

Overview

The construction of operating guidelines in Fiona is implemented quite differently from the theory presented in Chap. 5. The main differences are as follows:

1. The operating guidelines computed by Fiona characterize only *responsive* strategies. Thereby, a service net M is responsive with respect to a service net N if and only if there is no livelock in $N \oplus M$ where only transitions of M are enabled. It is easy to see that the set of responsive strategies is a subset of the set of all strategies for N.
2. The construction follows an iterative, depth-first computation approach. That is, we start with the initial state of OG_N and apply the operations closure and event to compute its successor states. From these states, the operations closure and event are applied again to compute more OG states. If thereby the message bound is violated by a knowledge value of a newly

computed OG state, we stop the successor computation and apply backtracking to return to the last computed state. That way, only OG states are computed in Fiona which are δ-reachable from the initial state of OG_N.

The design decision to characterize only responsive strategies results in significantly smaller operating guidelines than operating guidelines characterizing all deadlock-free interacting services M for a service net N. However, non-responsive service nets may perform quite artificial behavior (like sending a lot of messages and then performing an internal τ-loop forever). For this reason, operating guidelines for non-responsive strategies have to characterize also these behaviors (because the composition of such a service net has no deadlock with N). This is not feasible in practice. Furthermore, operating guidelines for responsive strategies can be used as a basis to decide whether or not M is a deadlock-free strategy for N as well. Assume that a service M is not characterized by the responsive operating guideline. Then there are only two possibilities — either M is not responsive with respect to N, or N and M have a reachable deadlock in their composition. Thereby, both cases can usually be easily distinguished from each other by looking at M only. Thus, responsive operating guidelines can be used to characterize all strategies with an additional check after the matching. This check, however, is not yet implemented in Fiona.

In the remainder of this chapter, we will say operating guidelines for short meaning operating guidelines for responsive strategies.

The iterative computation approach to construct an operating guideline has the obvious advantage of only computing those states of operating guidelines that can be used during the matching. The *depth*-first computation is justified by the observation that it is often possible to decide that a state q of the OG is non-normal without knowing all of q's successor states. In this case, we can immediately stop computing the subgraph starting at q. This potentially saves a huge amount of time and memory.

Algorithmic Implementation Details

The states of operating guidelines in Fiona are marked with colors. A blue color of a state q denotes that q is part of the OG later on, a red color marks a non-normal state. Red states will not be part of the OG later on. As introduced in Sect. 4.3.4, a currently normal (i.e. blue) state q can become non-normal (i.e. red) by removing a successor state q' of q (i.e. setting q' to color red). Red states stay red forever. Initially, a state is blue.

The initial state q_0 of the OG of a given service net N is computed as follows. First, we compute the knowledge set K_0 by simply computing those markings of N that are reachable from N's initial marking. This construction step is based on the data structures and methods that origin from LoLA. During the computation

of K_0, we check each reached marking for a message bound violation. If the given message bound is violated, N is not controllable under this message bound and FIONA exits with a respective error message. Having successfully computed K_0, we create a new state object representing q_0 and assign the knowledge K_0 to q_0. Then, we immediately compute the annotation $\phi(q_0)$ from the knowledge K_0 in conjunctive normal form. By setting the color of q_0 to blue, the computation of the initial state q_0 of OG_N is finished.

Given a state q of OG_N, a successor state of q is computed as follows. First, we select an event x from the (yet unprocessed) interface channels of OG_N. Then, the knowledge set K of q is manipulated according to the definition $event(K, x)$ to yield a new knowledge set K'. Then, from each marking of K' we compute the reachable markings and add each such marking to K' (to perform the closure operation). Again, we already check during the firing whether the message bound is violated. If violated, the computation of K' is aborted and FIONA returns to q to choose the next event to perform. Otherwise, K' is eventually computed. Having K', we search for an already computed state of OG_N with K' as knowledge. If such a state is found, it becomes the x-successor of q and FIONA returns to q to choose the next event. If no such state is found, we generate a new, initially blue colored state q' as the x-successor of q and assign K' to q'. Then, we generate the annotation $\phi(q')$ of q' from K' and assign it to q' as well. This completes the computation of q', and FIONA continues computing the successors of q'.

Whenever FIONA returns to a state q (either due to a message bound violation during the computation of a successor or when all events of the interface of OG_N have been performed for a successor of q), the annotation $\phi(q)$ of q is first normalized and then evaluated whether it is still satisfiable. Thereby, transitions that lead to red OG states count as not present and transitions corresponding to yet unprocessed events count as present. If $\phi(q)$ is unsatisfiable, the state q must be non-normal and is thus set to red. In this case, we immediately return to the latest predecessor of q and do not consider the remaining events at q. By the help of this early evaluation of OG states, the computation of possibly large subgraphs of states that will have to be deleted afterwards can be avoided.

For being able to apply the early evaluation as often as possible, we apply two heuristics for "steering" the choice which events are performed to calculate the successors of a state q. To this end, FIONA sorts the clauses of the annotation $\phi(q)$ by their lengths and counts the number of occurrences of each literal in $\phi(q)$. Then, from one of the shortest clauses, one of the events with highest number of occurrences is considered first. In general, such literals have the most impact on the value of $\phi(q)$. If the corresponding successor is set to red, the whole annotation may become unsatisfiable and further analysis of q can be skipped.

Furthermore, FIONA can make use of special annotations to events in the input file of a service net N. These annotations give a maximal number stating how often the respective event can be performed in a successful interaction with N. The

annotations can be gained by static analysis of N or can be specified manually. In fact, all service nets that origin from the compiler BPEL2oWFN carry such annotations. They are derived by static program analysis of the BPEL code before it is translated into service nets. To use this information, FIONA annotates each state q of OG_N with a table listing which event has been performed how often at a path from the initial state q_0 to q and compares these numbers with the specified event annotations. If the specified number is reached for an event, it is not considered at this state, but is suppressed instead.

If the computation of OG_N is finished, the color of the initial state of OG_N determines whether or not N is controllable. In case of a red initial state, OG_N is an empty *BSA* characterizing no service. Hence, N is uncontrollable. Otherwise, the connected blue subgraph containing the initial state q_0 represents the (already normalized) operating guideline OG_N of N. The graph *with* the red states represents a non-normal, equivalent version of OG_N. Depending on the parameters, FIONA now removes red states and/or the knowledge sets (preservation of the knowledge can be used for a diagnosis *why* a service net is uncontrollable, for instance). Finally, FIONA generates a graphical as well as a textual representation of OG_N and stores both in separate files. For the graphical layout, FIONA invokes GraphViz Dot (available at `http://www.graphviz.org`).

Example

In this section, we will give an example for the computation of OG_{N} for the service net N of Fig. 7.2. N corresponds to the service automaton A of Fig. 5.10(a). We illustrate the computation of the operating guideline OG_{A} of A exactly as it is performed by FIONA. Assume therefore a message bound 1, and let the final markings of N be the three markings [p3], [p4], and [p5]. The overapproximation $\mathcal{F}^1_{\mathsf{A}}$ of 1-strategies for A was already shown in Fig. 5.10(b).

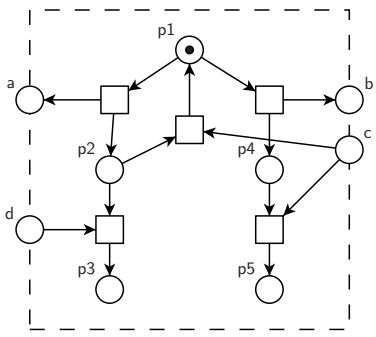


Figure 7.2: Example service net N with three final markings [p3], [p4], and [p5] to illustrate the OG computation. N corresponds to the service automaton A of Fig. 5.10(a).

In the first step, the initial state q0 of OG_{N} is generated. To this end, the markings reachable from the initial marking [p1] of N are computed (Fig. 7.3(a)). They constitute the knowledge set K0 of q0 and are depicted inside the state q0 in Fig. 7.3(a). Then, q0 is set to blue and the annotation of q0 is computed from the knowledge. Therefore, FIONA considers dead markings only. The resulting annotation of q0 is $(!c \vee !d \vee ?a) \wedge (!c \vee !d \vee ?b)$. Therein, !c and !d occur the most often and thus have highest priority.

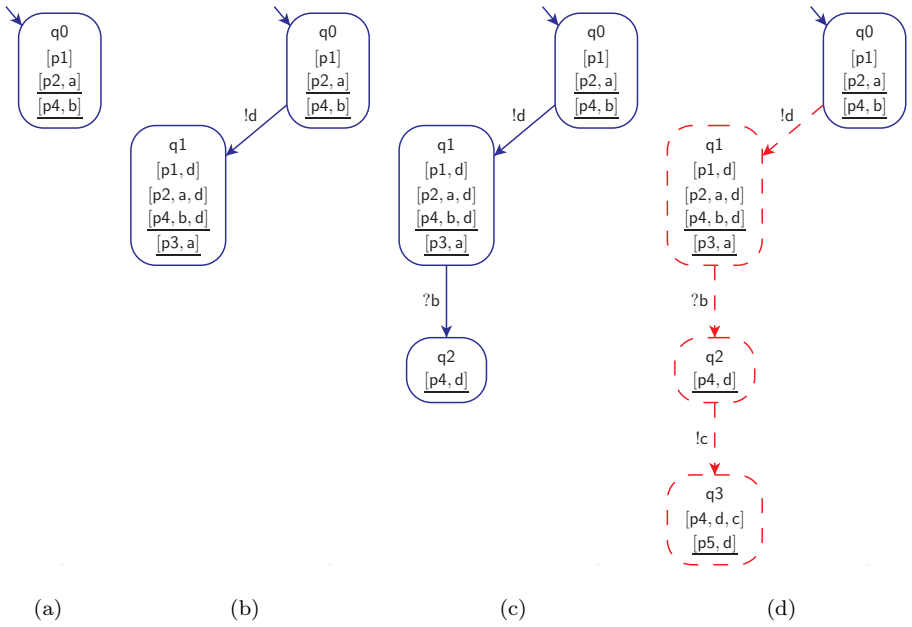


Figure 7.3: Example showing the intermediate steps of the computation of OG_{N} of the service net N of Fig. 7.2. The blue (solid) states of OG_{N} are (currently) normal states; the red (dashed) states represent non-normal states of OG_{N}. Underlined markings are dead markings of N.

Arbitrarily, !d is selected to be performed at q0 next (Fig. 7.3(b)). Thus, we add a d to each marking in K0. Thereby, the dead marking $[\mathsf{p2}, \mathsf{a}]$ in K0 results in the transient marking $[\mathsf{p2}, \mathsf{a}, \mathsf{d}]$ in K1, for instance. Then, we compute all reachable markings from the markings of K1 and add them to K1, eventually yielding the final knowledge set K1. As this set has not been computed before, a new, initially blue state q1 is generated with predecessor q0 and knowledge K1. Finally, the annotation of q1 is computed as $(!c \vee !d \vee ?a) \wedge (!c \vee !d \vee ?b)$. Again, !c and !d have highest priority.

Arbitrarily, !d is selected to be performed at q1 next (not depicted). However, dur-

ing the calculation of the new knowledge set, we encounter the marking $[\mathsf{p2}, \mathsf{a}, \mathsf{d}, \mathsf{d}]$ (because a d is added to the marking $[\mathsf{p2}, \mathsf{a}, \mathsf{d}]$ of $\mathsf{K1}$), which violates the message bound. Thus, the calculation is stopped and no new state is added to the graph. Returning to $\mathsf{q1}$, the annotation of $\mathsf{q1}$ is immediately normalized to $(!\mathsf{c} \vee ?\mathsf{a}) \wedge (!\mathsf{c} \vee ?\mathsf{b})$ with $!\mathsf{c}$ having the highest priority now. Thus, it is selected. However, we violate the message bound again, because a c is added to the marking $[\mathsf{p2}, \mathsf{a}, \mathsf{d}]$ of $\mathsf{K1}$, yielding the marking $[\mathsf{p2}, \mathsf{a}, \mathsf{d}, \mathsf{c}]$ which enables the $?\mathsf{c}$-labeled transition in N and subsequently the $!\mathsf{a}$-labeled transition. That way, the marking $[\mathsf{p2}, \mathsf{a}, \mathsf{a}, \mathsf{d}]$ is reachable, which violates the message bound. Hence, we return to $\mathsf{q1}$ once more and normalize its annotation to $?\mathsf{a} \wedge ?\mathsf{b}$.

As both $?\mathsf{a}$ and $?\mathsf{b}$ have the same priority, we arbitrarily select $?\mathsf{b}$ at $\mathsf{q1}$ now (Fig. 7.3(c)). The computed knowledge set is new and we add a blue state $\mathsf{q2}$ with annotation $!\mathsf{c} \vee !\mathsf{d}$. It is easy to see that sending another d violates the message bound. Hence, the remaining (normalized) annotation is $!\mathsf{c}$.

Selecting $!\mathsf{c}$ for $\mathsf{q2}$, we get a new, initially blue state $\mathsf{q3}$ with annotation $!\mathsf{c} \vee !\mathsf{d}$ (Fig. 7.3(d)). However, due to the marking $[\mathsf{p4}, \mathsf{d}, \mathsf{c}]$ in $\mathsf{K3}$, it is easy to see that both events will result in a message bound violation. Thus, the annotation is eventually normalized to *false*. Hence, it is unsatisfiable and $\mathsf{q3}$ is set to red. By backtracking, we return to $\mathsf{q2}$. Because the $!\mathsf{c}$-successor is now red, its annotation $!\mathsf{c}$ is as well normalized to *false* and $\mathsf{q2}$ is as well set to red. Returning to $\mathsf{q1}$, the latest annotation of $\mathsf{q1}$ was $?\mathsf{a} \wedge ?\mathsf{b}$, but the $?\mathsf{b}$-successor $\mathsf{q2}$ is red now. Thus, the red color propagates to $\mathsf{q1}$ as well, and we return to the initial state $\mathsf{q0}$ with normalized annotation $(!\mathsf{c} \vee ?\mathsf{a}) \wedge (!\mathsf{c} \vee ?\mathsf{b})$ (Fig. 7.3(d)). Because $!\mathsf{c}$ occurs more often than all other events, it is performed next at $\mathsf{q0}$ (not depicted). However, again the message bound is violated (the additional c allows N to return to $\mathsf{p1}$ to produce a second a). Thus, the new annotation of $\mathsf{q0}$ is $?\mathsf{a} \wedge ?\mathsf{b}$.

Selecting the event $?\mathsf{b}$ at $\mathsf{q0}$ to be performed next, we get the graph depicted in Fig. 7.4(a) (with the old red states omitted for reasons of space). The new successor state is $\mathsf{q4}$. Because the marking $[\mathsf{p4}]$ is a final marking of N of Fig. 7.2, its annotation is $\mathsf{final} \vee !\mathsf{c} \vee !\mathsf{d}$.

Performing $!\mathsf{d}$ first again at $\mathsf{q4}$, we recompute a knowledge set that is already assigned to state $\mathsf{q2}$ (cp. Fig. 7.3(d)). Thus, $\mathsf{q2}$ becomes the $!\mathsf{d}$-successor of $\mathsf{q4}$. Because $\mathsf{q2}$ is colored red, the annotation of $\mathsf{q4}$ is now normalized to $\mathsf{final} \vee !\mathsf{c}$.

Considering now the event $!\mathsf{c}$ at $\mathsf{q4}$, we get a new state $\mathsf{q5}$ (Fig. 7.4(b)) with annotation $\mathsf{final} \vee !\mathsf{c} \vee !\mathsf{d}$ (because the only dead marking in $\mathsf{K5}$, $[\mathsf{p5}]$, is a final marking of N).

Performing $!\mathsf{d}$ next at $\mathsf{q5}$ recomputes the knowledge of the red state $\mathsf{q3}$, and performing $!\mathsf{c}$ at $\mathsf{q5}$ violates the message bound. Hence, we eventually return to $\mathsf{q5}$ with normalized annotation final. Although there are no further events in the annotation, the remaining events have nevertheless to be performed in order to characterize *all* strategies. These events are $?\mathsf{a}$ and $?\mathsf{b}$. Both events obviously

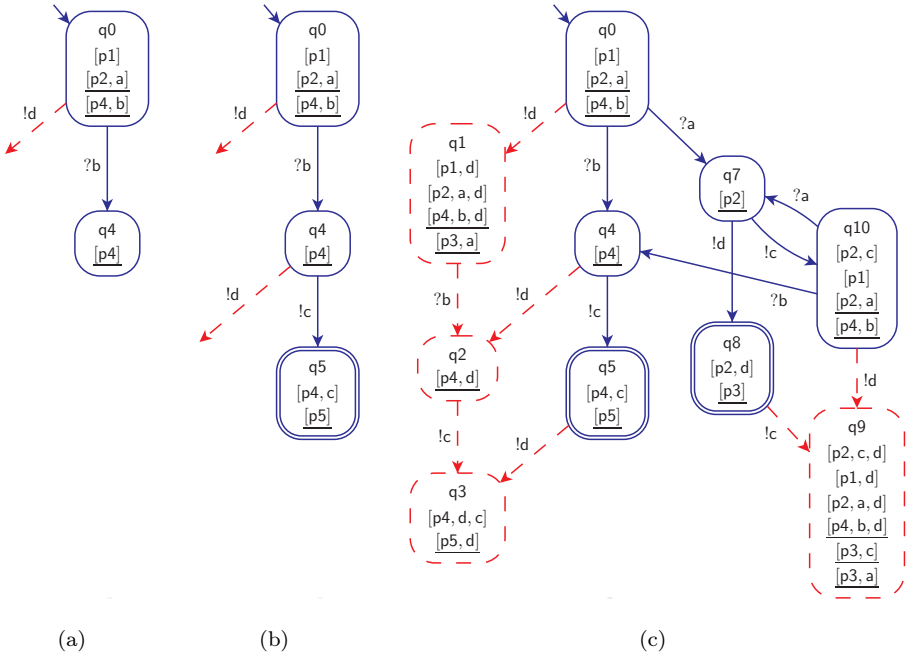


Figure 7.4: Example showing further steps of the computation of OG_{N} of the service net N depicted in Fig. 7.2.

lead to the empty state of OG_{N}. We do, however, not depict this state in the graphics. This is also the default behavior of FIONA. The empty state can be shown by calling FIONA with a respective parameter.

After having considered ?a and ?b at state q5, we return to state q4. As the !c-successor of q4 is blue, the annotation final $\vee$!c of q4 is not changed. Then, the remaining events, again ?a and ?b, are considered, both leading to the empty state. Hence, we return to the initial state q0 with its current annotation ?a $\wedge$?b. Only ?a has not yet been considered at q0. Hence, we consider this event ?a at q0 next. Eventually, the final basis of our OG of the service net N of Fig. 7.2 is computed, as depicted in Fig. 7.4(c).

After normalizing this basic OG (i.e. deleting all red states and transitions) and replacing all knowledge values by the computed annotations, we get the final 1-operating guideline OG_{N} for N. It is depicted in Fig. 7.5.

As mentioned above, the service net N of Fig. 7.2 corresponds to the service automaton A of Fig. 5.10(a). Correspondingly, the automaton with knowledge values depicted in Fig. 7.4(c) corresponds to the overapproximation $\mathcal{F}^1_{\mathsf{A}}$ of Fig. 5.10(b) for A. However, there are some differences between both figures. For instance, $\mathcal{F}^1_{\mathsf{A}}$

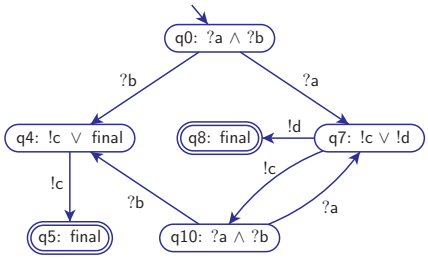


Figure 7.5: The final operating guideline OG_{N} of the service net N of Fig. 7.2.

has a ?a-labeled transition between its states q1 and q8, whereas there is no such transition in Fig. 7.4(c). This is caused by the optimizations of the implementation. At the moment when FIONA would consider this ?a-event, the red color of q1 is already fixed. Hence, any further computation at q1 is suppressed. In the specific example, however, the optimizations only suppress some transitions, and do not help to suppress the computation of a state of $\mathcal{F}^1_{\mathsf{A}}$ of Fig. 5.10(b). This is mostly based on the small message bound of $b = 1$ in the example. For a greater message bound, whole states would have been skipped.

Evaluation of the Implementation

Several of our design decisions have a substantial impact on the implemented *OG* computation's efficiency — both for time and memory needs. We will shortly summarize these decisions in the following and evaluate the respective effects.

First of all, we have decided for using Petri net representations of services. This decision allowed us to reuse very efficient data structures and methods from the Petri net model checking tool LoLA [Sch00]. A re-implementation of these structures and methods would have increased the implementation efforts tremendously without providing any advantage. However, this design decision also bears two disadvantages. Firstly, the firing of transitions in LoLA is based on one distinguished marking, the current marking, which is optimal if each marking is derived by firing or backfiring transitions. This is the case in LoLA. However, the construction of operating guidelines requires FIONA to perform the event operation for computing the knowledge of a successor state q' of a state q. This operation "jumps" directly to a marking without firing. Thus, this marking has to be decoded from the storage data structure and *all* transitions have to be checked for their enabledness. This is a rather time-consuming operation, and it has to be performed at least once per considered event x per state q in FIONA. The second disadvantage is that the additional memory need for organizing the storage data structure only pays off after it stores a certain number of states. Often, this number is not reached in the examples.

However, for a behavioral analysis like the computation of operating guidelines, usually the memory capacity (and not time) is the limiting factor. Thus, data structures and methods optimized for large state spaces are recommended, and we decided for using the data structures and methods of LoLA in Fiona as well.

Two further important design decisions for constructing operating guidelines are the computation in depth-first order with early checks for the red color of OG states and the "steering" of the events which are performed next. These decisions are mainly devoted to speed up OG computation. Although we have to spend time for the additional normalization, the satisfiability analysis of the annotations, and the computation of the event priorities, these efforts easily pay off by avoiding the computation of a possibly large number of markings in the knowledge sets of these states. Furthermore, successful early checks also result in less memory requirements — both for storing the additional markings and the corresponding OG states. The heuristics for choosing the next event for computing the successors of a state q result in a prioritization of sending events (as sending events occur in every clause) and cause red states to be computed at the earliest. This potentially enables us to decide the unsatisfiability of a state's annotation much earlier.

As an important design decision in favor of speeding up computation by *increasing* memory consumption, Fiona does not remove an OG state q from memory when its color changes from blue to red. Instead, red states are stored until the OG is fully computed. Only after the OG computation is finished, red states are removed. Thereby, the memory needs for storing the red states are significant. In most cases, the number of red states is a multiple of the number of blue states for computed OGs. However, removing a state whenever it changes its color to red would result in a multiple re-computation of large (non-normal) parts of the OG. Because the red part is much larger than the blue part of the OG, the loss of performance is enormous. Hence, we opted for storing red states until the OG is fully computed. However, we only compute non-normal OG states that are needed to decide the color of some other state. That is, each computed red state was computed for a certain purpose and OGs are not inflated unnecessarily.

In sum, the performance of the OG computation heavily depends on the power of the optimizations implemented in Fiona. How powerful these optimizations for a concrete service net N are is hard to predict. To further optimize the OG computation, we search for further heuristics that allow us to suppress one or more events at an OG state. For instance, we want to implement static analysis features in Fiona that allow us to derive event annotations like the annotations provided by BPEL2oWFN for service nets that stem from BPEL.

Section 7.4 presents a small case study. It shows the computation time needed to construct the operating guidelines of real-world services and compares the operating guideline approach to the discovery of behaviorally compatible services with the approach of model checking the composition of services to decide their behavioral compatibility.

7.3.3 Minimization of a *BSA*

Overview

The minimization of a *BSA* B^ϕ bases on a decision procedure whether or not two states of B^ϕ are equivalent. The corresponding equivalence check, i.e. the functionality equivalence of Fig. 7.1, had already been implemented before the minimization procedure was implemented. To reduce the implementation efforts, the minimization procedure of a *BSA* B^ϕ in FIONA thus calls the equivalence functionality for pairs of states of B^ϕ. If two states are equivalent, they will be merged to one state representing both of them later on.

The implementation is based on the following considerations. Therefore, assume a fixed order $q_1, \ldots, q_n$ of the states of a *BSA* and call a state q_i smaller than a state q_j if $i < j$. Correspondingly, q_i is greater than q_j if $i > j$.

1. Whenever a state q_i is equivalent to a smaller state q_j, then q_j is also equivalent to q_i (due to symmetry of equivalence). Hence, it is sufficient to consider only the equivalence of q_i with *greater* states q_j.
2. Whenever a state q_i is equivalent to two greater states q_j and q_k, then q_j and q_k are equivalent as well (due to symmetry and transitivity of equivalence). Hence, it is sufficient to consider only the equivalence of q_i with the *next* greater state q_j.
3. If two normal states q_i and q_j are equivalent, then they have equivalent annotations (cp. Lemma 4.5.9), they have the same present transitions (cp. Lemma 4.5.10), and the respective successors of q_i and q_j are equivalent as well (cp. Lemma 4.5.19). Hence, merging q_i and q_j reduces to a redirection of the incoming transitions of q_i to q_j. The leaving transitions do not have to be redirected as the successors will themselves be merged later on, and the annotations need not to be merged.

Algorithmic Implementation Details

The current implementation of the minimization of a *BSA* consists of three independent steps and works as follows.

In the first step, the *BSA* is normalized for being able to rely on all assumptions made above and to reduce the number of state pairs that have to be checked for equivalence. Because we want to be able to minimize also *BSA*s that are specified manually, we do not rely on the correctness of the blue color of a *BSA* state and perform a full normalization (instead of simply removing all red states).

In the second step, we iterate over the states q_1 to q_n. Recall that the *BSA* data structure (i.e. the class AnnotatedGraph) is limited such that a *BSA* state does not know its predecessor states. This saves memory when constructing operating

guidelines. For the minimization, however, we have to know the predecessors of a state and thus remember the current state q_i in this iteration as a predecessor of all of its successors. Furthermore, for each q_i with $i < n$, we iterate over the states q_{i+1} to q_n and check for each pair (q_i, q_j) whether or not they are equivalent. If two equivalent states are found, we store the index pair (i, j), immediately stop the inner loop, and continue with the outer iteration. That way, we eventually have computed the complete predecessor relation, and know for each state either no equivalent state (then it is a singleton equivalence class) or exactly one equivalent greater state.

The third step is then devoted to merge pairs of equivalent states. To this end, we iterate again over the states q_1 to q_n. For each equivalent state pair (q_i, q_j) (with $j > i$), we redirect the predecessors of q_i to the state q_j. Furthermore, if the state q_i has been the initial state of the *BSA*, then q_j is set as the new initial state.

Finally, the computed *BSA* is the minimized version of the original one. Because we have normalized the *BSA* first, the minimized *BSA* is also normal by construction.

Evaluation of the Implementation

The implementation was guided by the reuse of existing algorithms (i.e. the normalization procedure and the equivalence check) on the one hand, and optimized implementation of the new algorithms on the other hand. The result is a prototypic implementation of the minimization of a *BSA* feasible for *BSA*s of moderate sizes.

We currently see several possible optimizations for the minimization implementation. First of all, we claim that the second and third iteration can be united into one iteration. The main problem is that we then have to merge a state q_i with an equivalent state q_j although we do not know all predecessors of q_i. However, it should be possible to redirect all current predecessors of q_i in the moment that we know that q_i and q_j are equivalent, and to redirect a newly found predecessor of q_i later on.

7.4 Case Study

In this section, we present experimental results illustrating the sizes of operating guidelines for real-world services and the corresponding computation time of the operating guidelines. The results are summarized in the table of Fig. 7.6.

Furthermore, we compare the operating guidelines approach to service discovery as described in Sect. 6.1 with the model checking alternative introduced in the problem description of Sect. 1.3. The corresponding results are shown in Fig. 7.7.

7.4.1 Computation of Operating Guidelines

The first case study is devoted to show the computability of operating guidelines of services of realistic sizes. To this end, we have computed the operating guidelines of several processes that stem from two small German consulting companies, as well as other services. The results are listed in Fig. 7.6.

The table shows for each service net N the name of N, the size of N (numbers of all places P, of input places P_{in}, of output places P_{out}, and of transitions T), the size of the respective full operating guideline of N (number of all, i.e. normal (blue) and non-normal (red) states Q and transitions δ), and the size of the normalized version of the operating guideline (number of normal, i.e. blue, states Q and transitions δ only).

In order to compute the normalized operating guideline, the full operating guideline has to be computed first by applying the operations closure and event. As these operations are based on the knowledge sets of the OG states, we also list the number of different knowledge values (column labeled by k), i.e. the number of different markings of N that have been computed and stored in the data structure in order to compute the final operating guideline. Please note that each marking can occur in more than one OG state, and thus the number of references to markings is sometimes a multiple of this number k. The last column, labeled by $t\,(s)$, represents the time in seconds it took to calculate the full operating guideline including the normalization. All experimental results have been computed running FIONA, version 3.1, on a Windows machine (under Cygwin) with 2 GB RAM and an Intel Pentium M 1.73 GHz processor.

The examples stem from various sources and are grouped in Fig. 7.6 according to their origin. Most of these examples were given to us as BPEL processes. In this case, the respective process was translated into a service net using the service net semantics of [Loh08] and the compiler BPEL2oWFN. Afterwards, each service net was structurally reduced by applying standard Petri net reduction rules (i.e. [Mur89]) under special consideration of the interface places. These reduction rules are also implemented in BPEL2oWFN.

The first two services, i.e. the "Loan Approval" and "Purchase Order" process, are realistic web services from the BPEL specification [Alv07]. The corresponding (structurally reduced) service nets are quite simply structured, have 6 (respectively 10) interface places, and result in rather small operating guidelines with 20 to 168 states of the final, normalized operating guidelines. Both operating guidelines could be computed in less than one second.

The second group contains nine processes representing typical services from different practical domains. For instance, the "car breakdown" service models a part of a car rental service in case a car breaks down. Other examples represent banking and online shopping services. All examples have been modeled in the modeling suite of a small German consulting company by customers of the company. Each

Service name	service net				full *OG*		normal *OG*			
	P	P_{in}	P_{out}	T	Q	δ	Q	δ	k	$t\,(s)$
Loan Approval	26	3	3	9	21	84	20	36	76	0
Purchase Order	22	4	6	7	169	1,182	168	548	464	0
Breakdown	24	6	7	16	697	2,400	33	73	2,528	3
Car Breakdown	13	3	5	6	33	155	24	46	56	0
Car Repair	24	5	11	10	1,117	6,494	250	783	2,304	3
Customer Service	31	5	9	22	1,489	7,819	289	1,045	5,248	5
Mortgage Guarantee	17	6	4	6	153	970	152	484	288	0
Order processing	22	6	7	9	465	2,565	208	691	976	1
Rental Info Delivery	9	2	3	4	19	62	15	26	34	0
Reservations	18	2	7	14	259	1,887	249	973	564	0
Ticket Booking	20	3	7	11	105	706	92	292	352	0
Identity Card Issue	36	2	9	9	1,537	15,115	1,536	7,936	13,828	5
Registration Office	81	3	3	26	25	100	24	48	1,148	0
Database Service	12	2	5	5	140	551	37	95	311	0
Help Desk Service	15	4	4	8	43	136	16	30	104	0
Olive Oil Ordering	12	3	3	6	23	77	16	27	51	0
Broker with 5 Airlines	59	1	7	44	15	107	14	18	3,954	2
Broker with 7 Airlines	79	1	9	60	19	173	18	24	82,475	45
Broker with 8 Airlines	89	1	10	68	21	212	20	27	400,350	325
Sequence of length 5	11	5	0	5	32	80	32	80	63	0
Sequence of length 14	29	14	0	14	16,384	114,688	16,384	114,688	32,767	66
5 Dining Philosophers	36	5	5	16	1,434	8,395	1,432	3,435	9,757	8
6 Dining Philosophers	43	6	6	19	6,140	43,134	6,138	17,676	61,113	81

Figure 7.6: Experimental results of computing operating guidelines with Fiona.

service was exported to BPEL and then translated to service nets. Although most of these examples have a rather simple internal structure, the number of reached

markings of the service net varies from some 30 markings to several thousands of markings. The "Customer Service" is the largest one with respect to both the operating guideline and the number of computed markings. It could be computed in approximately 5 seconds.

The next two services, "Identity Card Issue" and "Registration Office", stem from another industrial cooperation partner and model administrative workflows. Again, the services were exported as BPEL processes and then translated into service nets. Although the internal structure of the "Identity Card Issue" service net is rather complex and results in $13,828$ computed markings, the operating guideline could be computed in 5 seconds as well.

The "Database Service" [BCB+06], the "Help Desk Service" (from the Oracle BPEL Process Manager), and the "Olive Oil Ordering" process [AFK05] are other web services that use BPEL features like fault and event handling. They have been directly modeled as BPEL processes and were then translated into service nets. The structural reduction of the translated service nets results in quite small nets despite the involved structure of the BPEL processes.

It is easy to see that all these practical examples so far could be computed in reasonable time. Nevertheless, tool support for computing operating guidelines is necessary because the operating guidelines of these services partly have more than one thousand states.

The last two groups of services represent academic, technical examples. Therein, the "Broker with n Airlines" is an open BPEL4Chor [DKLW07] choreography adapted from [LKLR08]. It models a travel agency (the Broker) who expects a ticket request of a traveler and then asks n different airlines for a ticket offer. The cheapest offer is finally selected and returned to the traveler. For the computation of the operating guideline, we have considered the agency together with 5 (or 7, or 8) airlines as one composite service and translated this composition into a service net using the compiler BPEL2oWFN as well. These examples yield the biggest service nets (with up to 89 places and 68 transitions after structural reduction) as well as the biggest operating guidelines (with more than 5 minutes computation time). Thereby, the size of the OG (i.e. the number of OG states) is not very big. However, more than $400,000$ markings (i.e. different knowledge values) have to be computed which is already a rather complex state space for a service.

The last group of services consists of sequence services and services representing the known dining philosophers problem. The "sequence" services wait for 5, respectively 14, different messages in a sequence. Due to the asynchronous communication of services, the operating guidelines reflect the possibility of strategies to send all 5 (14) messages in arbitrary order. This results in diamond-structured operating guidelines with lots of intermediate states. Raising the number of messages by 1 doubles the number of OG states and more than doubles the computation time. These examples can thus be used to evaluate the computation of

operating guidelines with many states. The "Dining Philosophers" model a version of the philosophers problem, where the strategy has to choose an order in which the philosophers take their forks. It is an even harder challenge example for FIONA as adding one philosopher to the system approximately quadruples the number of OG states. Furthermore, the internal structure of the philosophers is much more complicated than the structure of the sequence services. This results in both large operating guidelines and huge numbers of computed markings. Consequently, the computation of the operating guideline for 7 philosophers already takes a few minutes and several hundred thousands of markings (comparable to the "Broker with 8 Airlines" service).

For computing the operating guidelines, we mostly used a message bound of 1. Only the database service requires a message bound $b = 2$ to be controllable. Raising this number usually raises the computation time significantly without necessarily increasing the (normalized) operating guideline's size. Hence, choosing a realistic message bound for the computation of operating guidelines is important for not wasting computation time unnecessarily.

7.4.2 Service Discovery with Operating Guidelines

As motivated in Sect. 6.1, the operating guideline OG_S of a service S is usually computed before S is made publicly available, i.e. at *design time*. At this stage, spending the computational efforts to construct the operating guideline is reasonable. After S has been published to a registry, when a client searches for a behaviorally compatible published service for its service R, however, the efforts for deciding behavioral compatibility of R and S, i.e. deciding the question $R \in \mathit{Strat}(S)$, are spent at *runtime*. Thus, an efficient decision procedure to check $R \in \mathit{Strat}(S)$ is crucial for the service brokering concept of publishing and discovering services in an SOA. Consequently, an approach that shifts at least a part of the computational efforts to the first phase may help to enhance the performance of service discovery in the second phase substantially.

In the introduction, we have motivated the concept of operating guidelines as an approach to support the discovery of behaviorally services S for a querying client's service R. The corresponding service discovery approach with operating guidelines has been described in Sect. 6.1. It takes into account all relevant aspects motivated in the problem description of Sect. 1.3. Most notably, it shifts the (potentially) costly computation of the operating guideline OG_S of a service S to the design phase and allows for an efficient matching procedure to decide $R \in \mathit{Strat}(S)$ at the discovery phase.

The following small case study shall back this claim with numbers. It recalls the "Broker with n Airlines" example service from the previous section. In the table of Fig. 7.7, we repeat the needed computation time of the respective operating guideline for 5, 7, and 8 airlines. Furthermore, we translated a fitting traveler for this

open choreography (also taken from [LKLR08]) into a service net and considered the matching of the traveler with the computed operating guideline. The time needed for the matching takes approximately 130 milliseconds, which underlines the intended distribution of the efforts between design time and runtime.

Open Choreography	Computation of OG, $t\,(s)$	Matching with Traveler, $t\,(s)$	Model checking of Composition, $t\,(s)$
Broker with 5 Airlines	1.5	0.128	0.350
Broker with 7 Airlines	44.7	0.130	7.511
Broker with 8 Airlines	324.9	0.130	57.512

Figure 7.7: Experimental comparison of Model Checking with LoLA versus Matching with FIONA. In the cases of 5 and 7 Airlines, seven matchings are sufficient to outperform the model checking approach. In the case of 8 Airlines, even six matchings are sufficient to break even with the model checking approach.

As a competing approach, we translated the same choreography, this time with all participants including the traveler, into a closed Petri net, representing all parties in interaction. To this end, we used the compiler BPEL2oWFN once again, which is capable of translating BPEL and BPEL4Chor into the input format of the Petri net model checking tool LoLA [Sch00] as well.[1] Again, we applied the structural reduction techniques implemented in BPEL2oWFN to reduce the structure of the translated Petri net. Furthermore, the final states induced by the BPEL4Chor specification have been translated into a CTL formula expressing that, from each reachable marking of the Petri net, a final marking is reachable. That way, we are easily able to verify the absence of deadlocks (i.e. non-final, dead, and reachable markings) of the net. The corresponding time needed by LoLA is given in the last column of the table of Fig. 7.7. All results were computed on the same machine.

It is easy to see that the runtime efforts between both approaches drastically differ — whereas the matching basically takes constant time of approximately 130 milliseconds, the time for model checking the composite service of agency, airlines, and traveler significantly grows with the number of considered airlines.

Even if we take into account the overall sum of computation time (i.e. when also including the OG computation), then our OG approach still easily pays of after a small number of performed matchings. In case of less than 8 Airlines, only seven matchings are sufficient to outperform the model checking approach. In the case of a price request of 8 Airlines, the break even is already reached after matching the traveler six times. As we assume that the publish operation of the service brokering in an SOA typically happens significantly less often than the discovery operation, we summarize that the operating guideline approach for service discovery is feasible.

[1] Recall that FIONA is based on the data structures and algorithms of LoLA, and that the input formats of LoLA and FIONA are very similar.

7.5 Conclusion

All results presented in Part II of this thesis have been prototypically implemented in our service analysis tool Fiona. That is, Fiona can decide controllability of a service and compute its operating guideline. Fiona can normalize and minimize *BSA*s (and thus operating guidelines), match a service with an operating guideline, and compare the sets of strategies of two services for equivalence or inclusion. Furthermore, the functionality of Fiona comprises additional features that have not been the focus of this thesis. For example, Fiona can be used to synthesize strategy services for a given service, to compute a public view of a service, or to generate a behavioral adapter between behaviorally incompatible services.

Experimental results have shown that the computation of operating guideline is feasible in practice. That is, it is possible to construct the operating guidelines of service of realistic sizes in reasonable time. Furthermore, the operating guideline approach for the discovery of behaviorally compatible services outperforms the competing model checking approach to verify behavioral compatibility after only a few matchings. This justifies the application of operating guidelines in service-oriented architectures as a behavioral description of a provider service S published to the service registry.

The implementation of Fiona was guided by the use of efficient data structures and methods for being able to analyze services of practical sizes. The implementation of additional heuristics can further increase performance.

8 Conclusions and Future Work

8.1 Conclusions

In the service-oriented approach, services are used as building blocks to develop large dynamical interorganizational business processes. A key concept in this approach is the discovery of published services that meet the search criteria of a client. Therein, one major challenge is to ensure behavioral compatibility between the client and the (selected) published service. In case of behavioral incompatibilities, severe global errors in the whole interorganizational business process can be the result.

In this thesis, we have presented a framework to formally analyze the interaction of services for behavioral compatibility. We introduced service nets and service automata as equally well-suited formal methods to model services and their interaction and formalized behavioral correctness in the notions of well-behavior and controllability of a service. Suitability of service nets for service modeling has been proven through a feature-complete formal service net semantics for the industrial Web service description language BPEL. The semantics is implemented in the compiler BPEL2oWFN and therefore allows for the automatic translation of a BPEL process into a service net. That way, our algorithms and techniques can directly be applied to real-world processes. Experimental results have shown that we can detect non-trivial model flaws of interacting services that would have been hard or impossible to find manually.

With the concept of the operating guideline OG_S of a service S, we proposed an artifact that characterizes the set $\mathit{Strat}(S)$ of all behaviorally compatible services R for S, i.e. all strategies for S. The semantics of OG_S is formalized by a matching algorithm in form of the set $\mathit{Match}(OG_S)$. With the help of this algorithm, we can efficiently decide whether or not R is characterized by OG_S, i.e. decide $R \in \mathit{Match}(OG_S)$. Together with the main property of operating guidelines, $\mathit{Strat}(S) = \mathit{Match}(OG_S)$, we are able to decide behavioral compatibility between R and S, i.e. the question $R \in \mathit{Strat}(S)$, by simply deciding $R \in \mathit{Match}(OG_S)$.

OG_S is finite, operational, and reveals only such information about S that are

inevitably needed to decide behavioral compatibility with S. Hence, OG_S is well suited to support the discovery of behaviorally compatible published services. Consequently, we propose to publish the operating guideline of a service as additional available information in a service repository.

We further developed a procedure to decide the inclusion relation between the semantics $Match(OG_S)$ and $Match(OG_{S'})$ of two operating guidelines OG_S and $OG_{S'}$ of two services S and S'. To this end, we introduced a preorder relation $\sqsubseteq$. It is defined on the structures of OG_S and $OG_{S'}$ only and can therefore be efficiently decided. The preorder relation can be used to analyze several notions that are relevant in the context of behavioral substitutability of services and for service contracts.

The construction of operating guidelines is implemented in our analysis tool Fiona and experimental results have shown that the computation of operating guidelines is feasible for services of realistic sizes. The compiler BPEL2oWFN complements our analysis possibilities. Together with BPEL2oWFN, we have a technology chain that starts out with a BPEL process, transforms it into a service net, and computes its operating guideline. Analogously, we may also start with another BPEL process, transform it into a service net, and match this net with the published operating guideline to decide behavioral compatibility of both services.

The presented approach is based on service nets and service automata in this thesis. However, it is not limited to these formalisms and can easily be translated into other frameworks using asynchronous message passing as a communication paradigm. Furthermore, our approach can also be applied to services specified in a service description languages other than BPEL as long as this language can be translated into some formal model and therefore into service nets or service automata. For example, there exists tool support to translate languages like BPEL4Chor [DKLW07], WS-CDL [KBR+05], UML activity diagrams [OMG07, Stö04], and BPMN [OMG06] into a formal model.

Finally, the work presented in this thesis already serves as a foundation for other ongoing doctoral projects in the area of substitutability, adapter generation, or test case generation. All these works heavily base on operating guidelines and partially extend the concept of operating guidelines for their respective needs.

8.2 Future Work

The many the applications of operating guidelines are, the many are the possible directions for future research. From our perspective, the most important future research questions are the following ones.

Distributed operating guidelines. So far, operating guidelines reflect correct interaction of two services only. The interaction of a service S with more than one client $R_1, \ldots, R_n$ is not addressed. This question is closely related to the question of *distributed controllability* as considered in [Wol09]. The main challenge for distributed operating guidelines is the availability of choices. That is, the interaction of R_1 with S has an impact on which interaction of R_2 with S is correct, and which is not. To characterize *all* combinations of correct interactions of R_1 with S and R_2 with S, the corresponding operating guideline of S should have annotations like "If R_1 performs/has performed this (or that) action, then R_2 must satisfy this (or that) Boolean formula."

Compositional computation. The current computation approach for constructing the operating guideline OG_S for a service S requires a complete recalculation of OG_S whenever S is changed. Consequently, S can only be analyzed reasonably after it has been designed completely. An approach that can at least reuse some part of the construction of the old operating guideline could decrease the needed computational efforts significantly.

Boundedness of communication. In Sect. 5.3, we have introduced the restriction of our characterization to strategies R that have b-bounded communication with S. This restriction was introduced to ensure a finite state composition of R and S. However, there are pairs of services where the composition is finite although there is no message bound b such that R and S have b-bounded communication. In other words, b-bounded communication is only sufficient for a finite composition, but not necessary. A criterion that is also necessary is an interesting theoretical future work.

Combination with other properties. As a goal in the broader context of services and SOC, the results of this thesis should be combined with other correctness notions of services. Especially in the area of service discovery, the notion of behavioral compatibility of services should be integrated into results regarding the compatibility of the semantics of services as well as non-functional, quality of service (QoS) aspects. Only a service discovery process that takes care of all aspects of service compatibility will satisfy the needs to realize the vision of high flexibility and loose coupling of service compositions.

Bibliography

[Aal98] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[Aal03] W. M. P. van der Aalst. Inheritance of Interorganizational Workflows: How to agree to disagree without loosing control? *Information Technology and Management Journal*, 4(4):345–389, 2003.

[AB02] W. M. P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.

[ACKM03] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, September 2003.

[AFK05] J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Applying Model Checking to BPEL4WS Business Collaborations. In *The 2005 ACM Symposium on Applied Computing (SAC)*, pages 826–830. ACM, 2005.

[AH01] L. de Alfaro and T. A. Henzinger. Interface Automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001.

[ALM+07] W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf. From Public Views to Private Views – Correctness-by-Design for Services. In *4th International Workshop on Web Services and Formal Methods (WS-FM 2007)*, volume 4937 of *Lecture Notes in Computer Science*, pages 139–153, Brisbane, Australia, 2007. Springer.

[ALM+09] W. M. P. van der Aalst, N. Lohmann, P. Massuthe, C. Stahl, and K. Wolf. Multiparty Contracts: Agreeing and Implementing Interorganizational Processes. *The Computer Journal*, 2009. (to appear).

[Alv07] A. Alves et al. Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Committee specification, Organization for the Advancement of Structured Information Standards (OASIS), January 2007.

[AW01] W. M. P. van der Aalst and M. Weske. The P2P approach to Interorganizational Workflows. In *13th International Conference on Advanced Information Systems Engineering (CAiSE 2001)*, volume 2068 of *Lecture Notes in Computer Science*, pages 140–156, Interlaken, Switzerland, August 2001. Springer.

[BA01] T. Basten and W. M. P. van der Aalst. Inheritance of Behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.

[Baj06] S. Bajaj et al. Web Services Policy 1.2 - Framework (WS-Policy). Technical report, W3C, April 2006.

[BBB95] R. K. Boel, L. Ben-Naoum, and V. van Breusegem. On forbidden state problems for a class of controlled Petri nets. *IEEE Transactions on Automatic Control*, 40(10):1717–1731, 1995.

[BBMP06] M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. A Priori Conformance Verification for Guaranteeing Interoperability in Open Environments. In *4th International Conference on Service-Oriented Computing (ICSOC 2006)*, volume 4294 of *Lecture Notes in Computer Science*, pages 339–351. Springer, 2006.

[BCB+06] J. Bolie, M. Cardella, S. Blanvalet, M. Juric, S. Carey, P. Chandran, Y. Coene, K. Geminiuc, M. Zirn, and H. Gaur. *BPEL Cookbook: Best Practices for SOA-based integration and composite applications development.* Packt Publishing, 2006.

[BCGM05] D. Berardi, D. Calvanese, G. De Giacomo, and M. Mecella. Composition of services with nondeterministic observable behavior. In *3rd International Conference on Service-Oriented Computing (ICSOC 2005)*, volume 3826 of *Lecture Notes in Computer Science*, pages 520–526. Springer, 2005.

[BCPV04] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing Web Service Choreographies. *Electronic Notes in Theoretical Computer Science*, 105:73–94, 2004.

[BD98] E. Badouel and P. Darondeau. Theory of Regions. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 529–586. Springer, 1998.

[BDH05] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Service Interaction Patterns. In *3rd International Conference on Business Process Management (BPM 2005)*, volume 3649 of *Lecture Notes in Computer Science*, pages 302–318, 2005.

[BFHS03] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *12th international conference on World Wide Web (WWW 2003)*, pages 403–410. ACM, 2003.

[BFM02] C. Bussler, D. Fensel, and A. Maedche. A Conceptual Architecture for Semantic Web Enabled Web Services. *ACM SIGMOD Record*, 31(4):24–29, 2002.

[BHL+05] B. Benatallah, M.-S. Hacid, A. Leger, C. Rey, and F. Toumani. On automating Web services discovery. *The International Journal on Very Large Data Bases*, 14(1):84–96, 2005.

[BN08] B. Benatallah and H. R. Motahari Nezhad. Service Oriented Architecture: Overview and Directions. In *Advances in Software Engineering: Lipari Summer School 2007*, volume 5316 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2008.

[BSBM04] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *5th International Workshop on Technologies for E-Services (TES 2004)*, volume 3324 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2004.

[BZ83] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, Los Angeles, California, January 1977. ACM Press, New York, NY, USA.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weeravarana. Web Service Discription Language (WSDL) 1.1. Technical report, Ariba, International Business Machines Corporation, Microsoft, March 2001.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 2000.

[CKK+02] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Springer, 2002.

[CKLY98] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, 1998.

[CL99] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[CTD05] I. Chebbi, S. Tata, and S. Dustdar. Cooperation Policies for Inter-organizational Workflows. In *The 2005 Symposium on Applications and the Internet Workshops (SAINT-W)*, pages 222–225, Washington, DC, USA, January 2005. IEEE Computer Society.

[DBN08] M. Dumas, B. Benatallah, and H. R. Motahari Nezhad. Web Service Protocols: Compatibility and Adaptation. *IEEE Data Engineering Bulletin*, 31(3):40–44, 2008.

[DHM+04] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services. In *30th International Conference on Very Large Data Bases (VLDB 2004)*, pages 372–383. VLDB Endowment, 2004.

[DK76] F. DeRemer and H. H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

[DKLW07] G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 296–303, Salt Lake City, UT, USA, July 2007. IEEE Computer Society Press.

[DS05] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, August 2005.

[DSGF06] V. Deora, J. Shao, W. A. Gray, and N. J. Fiddian. Modelling Quality of Service in Service Oriented Computing. In *2nd IEEE International Symposium on Service-Oriented System Engineering (SOSE 2006)*, pages 95–101, Washington, DC, USA, 2006. IEEE Computer Society.

[DX03] P. Darondeau and X. Xie. Linear control of live marked graphs. *Automatica*, 39(3):429–440, 2003.

[FBS05] X. Fu, T. Bultan, and J. Su. Synchronizability of Conversations among Web Services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.

[Fer04] A. Ferrara. Web services: a process algebra approach. In *2nd International Conference on Service-Oriented Computing (ICSOC 2004)*, pages 242–251, New York, NY, USA, 2004. ACM Press.

[Gie08] C. Gierds. Finding Cost-Efficient Adapters. In *15th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2008)*, volume 380 of *CEUR Workshop Proceedings*, pages 37–42. CEUR-WS.org, September 2008.

[GMW08] C. Gierds, A. J. Mooij, and K. Wolf. Specifying and generating behavioral service adapter based on transformation rules. Preprint CS-02-08, Universität Rostock, Rostock, Germany, August 2008.

[Got00] K. Gottschalk. Web Services Architecture Overview. IBM Whitepaper, IBM developerWorks, September 2000. `http://ibm.com/developerWorks/web/library/w-ovr/`.

[GRX03] A. Ghaffari, N. Rezg, and X. Xiaolan. Feedback Control Logic for Forbidden-State Problems of Marked Graphs: Application to a Real Manufacturing System. *IEEE Transactions on Automatic Control*, 48(1):18–29, 2003.

[HBCS03] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: A Look Behind the Curtain. In *Proceedings of the 22nd ACM symposium on Principles of database systems (PODS 2003)*, pages 1–14. ACM Press, 2003.

[HGZ96] L. E. Holloway, X. Guan, and L. Zhang. A Generalization of State Avoidance Policies for Controlled Petri Nets. *IEEE Transactions on Automatic Control*, 41(6):804–816, 1996.

[HKG97] L. E. Holloway, B. H. Krogh, and A. Giua. A Survey of Petri Net Methods for Controlled Discrete Event Systems. *Discrete Event Dynamic Systems: Theory and Applications*, 7(2):151–190, 1997.

[HR00] R. Hauck and H. Reiser. Monitoring Quality of Service across Organizational Boundaries. In *3rd International IFIP/GI Working Conference on Trends in Distributed Systems*, pages 124–137, London, UK, 2000. Springer.

[KBR+05] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language (WS-CDL) Version 1.0. W3C Candidate Recommendation 9 November 2005, W3C, Cambridge, Massachusetts, USA, 2005.

[Kin97] E. Kindler. A compositional partial order semantics for Petri net components. In *18th International Conference on Application and Theory of Petri Nets (ICATPN 1997)*, volume 1248 of *Lecture Notes in Computer Science*, pages 235–252. Springer, June 1997.

[KL09] K. Kaschner and N. Lohmann. Automatic Test Case Generation for Interacting Services. In *6th International Conference on Service-Oriented Computing (ICSOC 2008), Workshops Proceedings*, volume 5472 of *Lecture Notes in Computer Science*, pages 66–78. Springer, December 2009. (in press).

[KP06] R. Kazhamiakin and M. Pistore. Choreography Conformance Analysis: Asynchronous Communications and Information Alignment. In *3rd International Workshop on Web Services and Formal Methods (WS-FM 2006)*, volume 4184 of *Lecture Notes in Computer Science*, pages 227–241. Springer, 2006.

[LDL08] F. Lécué, A. Delteil, and A. Léger. Towards the Composition of Stateful and Independent Semantic Web Services. In *The 2008 ACM symposium on Applied computing (SAC)*, pages 2279–2285, New York, NY, USA, 2008. ACM.

[LK08] N. Lohmann and J. Kleine. Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes. In *Modellierung 2008*, volume P-127 of *Lecture Notes in Informatics (LNI)*, pages 57–72. GI, March 2008.

[LKLR08] N. Lohmann, O. Kopp, F. Leymann, and W. Reisig. Analyzing BPEL4Chor: Verification and Participant Synthesis. In *4th International Workshop on Web Services and Formal Methods (WS-FM 2007)*, volume 4937 of *Lecture Notes in Computer Science*, pages 46–60, Brisbane, Australia, September 2008. Springer.

[LMSW06] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing Interacting BPEL Processes. In *4th International Conference on Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32, Vienna, Austria, September 2006. Springer.

[LMSW08] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation. *Data Knowledge Engineering*, 64(1):38–54, January 2008.

[LMW07a] N. Lohmann, P. Massuthe, and K. Wolf. Behavioral Constraints for Services. In *5th International Conference on Business Process*

Management (BPM 2007), volume 4714 of *Lecture Notes in Computer Science*, pages 271–287, Brisbane, Australia, September 2007. Springer.

[LMW07b] N. Lohmann, P. Massuthe, and K. Wolf. Operating Guidelines for Finite-State Services. In *28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 321–341, Siedlce, Poland, June 2007. Springer.

[Loh07] N. Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0 and its Compiler BPEL2oWFN. Informatik-Berichte 212, Humboldt-Universität zu Berlin, August 2007.

[Loh08] N. Lohmann. A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *4th International Workshop on Web Services and Formal Methods (WS-FM 2007)*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91, Brisbane, Australia, April 2008. Springer.

[LRS02] F. Leymann, D. Roller, and M. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2), 2002.

[LW09] N. Lohmann and K. Wolf. Petrifying Operating Guidelines for Services. In *9th International Conference on Application of Concurrency to System Design (ACSD 2009)*. IEEE Computer Society, July 2009.

[Lyn96] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mar04] A. Martens. *Verteilte Geschäftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services*. PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, 2004.

[MBM+07] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D. L. Mcguinness, E. Sirin, and N. Srinivasan. Bringing Semantics to Web Services with OWL-S. *World Wide Web*, 10(3):243–277, 2007.

[MCHP08] M. Mancioppi, M. Carro, W.-J. van den Heuvel, and M. P. Papazoglou. Sound Multi-party Business Protocols for Service Networks. In *6th International Conference on Service-Oriented Computing (ICSOC 2008)*, volume 5364 of *Lecture Notes in Computer Science*, pages 302–316, 2008.

[MGB+07] M. Mrissa, C. Ghedira, D. Benslimane, Z. Maamar, F. Rosenberg, and S. Dustdar. A Context-Based Mediation Approach to Compose Semantic Web Services. *ACM Transactions on Internet Technology*, 8(1):1–23, 2007.

[Mil71] R. Milner. An Algebraic Definition of Simulation Between Programs. In *2nd International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 481–489, London, UK, 1971.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MRS05] P. Massuthe, W. Reisig, and K. Schmidt. An Operating Guideline Approach to the SOA. *Annals of Mathematics, Computing & Teleinformatics*, 1(3):35–43, 2005.

[MS05] P. Massuthe and K. Schmidt. Operating Guidelines - An Automata-Theoretic Foundation for the Service-Oriented Architecture. In *5th International Conference on Quality Software (QSIC 2005)*, pages 452–457, Melbourne, Australia, September 2005. IEEE Computer Society.

[MSSW08] P. Massuthe, A. Serebrenik, N. Sidorova, and K. Wolf. Can I find a Partner? Undecidablity of Partner Existence for Open Nets. *Information Processing Letters*, 108(6):374–378, Nov 2008.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[MW07] P. Massuthe and K. Wolf. An Algorithm for Matching Nondeterministic Services with Operating Guidelines. *International Journal of Business Process Integration and Management (IJBPIM)*, 2(2):81–90, 2007.

[MW08] P. Massuthe and D. Weinberg. Fiona: A Tool to Analyze Interacting Open Nets. In *15th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2008)*, volume 380 of *CEUR Workshop Proceedings*, pages 99–104. CEUR-WS.org, September 2008.

[MWF05] B. Mahleko, A. Wombacher, and P. Fankhauser. Process-annotated Service Discovery facilitated by an n-gram-based index. In *International Conference on e-Technology, e-Commerce, and e-Services (EEE 2005)*, pages 2–8. IEEE Computer Society, 2005.

[NNH05] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2nd edition, 2005.

[OMG06] Business Process Modeling Notation (BPMN) Specification. Final adopted specification, dtc/06-02-01, Object Management Group, February 2006.

[OMG07] UML superstructure, v2.1.2, formal/07-11-02. Standard, Object Management Group, 2007.

[Pap01] M. P. Papazoglou. Agent-oriented technology in support of e-business. *Communications of the ACM*, 44(4):71–77, 2001.

[Pap03] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *4th International Conference on Web Information Systems Engineering (WISE 2003)*, pages 3–12, Rome, Italy, December 2003. IEEE Computer Society.

[Pap07a] M. P. Papazoglou. *Web Services: Principles and Technology*. Pearson - Prentice Hall, Essex, July 2007.

[Pap07b] M. P. Papazoglou. What's in a Service? In *1st European Conference on Software Architecture (ECSA 2007)*, volume 4758 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2007.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer.

[PH07] M. P. Papazoglou and W.-J. van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The International Journal on Very Large Data Bases*, 16(3):389–415, 2007.

[PR05] S. Pinchinat and S. Riedweg. You Can Always Compute Maximally Permissive Controllers Under Partial Observation When They Exist. In *American Control Conference (ACC 2005)*, volume 4, pages 2287–2292, Portland, Oregon, June 2005.

[PTDL08] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: a Research Roadmap. *International Journal on Cooperative Information Systems*, 17(2):223–255, 2008.

[Rei85] W. Reisig. *Petri Nets*. Springer, Berlin, Heidelberg, New York, Tokyo, EATCS Monographs on Theoretical Computer Science edition, 1985.

[RPD06] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. *IEEE International Conference on Web Services (ICWS 2006)*, 0:205–212, 2006.

[RW87] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal Control and Optimization*, 25(1):206–230, 1987.

[Sch00] K. Schmidt. LoLA: A Low Level Analyser. In *21st International Conference on Application and Theory of Petri Nets (ICATPN 2000)*, volume 1825 of *Lecture Notes in Computer Science*, pages 465–474. Springer, June 2000.

[Sch05] K. Schmidt. Controllability of Open Workflow Nets. In *Enterprise Modelling and Information Systems Architectures (EMISA 2005)*, volume 75 of *Lecture Notes in Informatics (LNI)*, pages 236–249. GI, 2005.

[SMB09] C. Stahl, P. Massuthe, and J. Bretschneider. Deciding Substitutability of Services with Operating Guidelines. *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 5460(II):172–191, March 2009.

[Stö04] H. Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2004)*, pages 235–242. IEEE Computer Society, September 2004.

[SW08] C. Stahl and K. Wolf. An Approach to Tackle Livelock-Freedom in SOA. In *15th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2008)*, volume 380 of *CEUR Workshop Proceedings*, pages 69–74. CEUR-WS.org, September 2008.

[TP04] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *3rd International Semantic Web Conference (ISWC 2004)*, volume 3298 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2004.

[VMK+07] T. Vitvar, A. Mocan, M. Kerrigan, M. Zaremba, M. Zaremba, M. Moran, E. Cimpian, T. Haselwanter, and D. Fensel. Semantically-enabled service oriented architecture : concepts, technology and application. *Service Oriented Computing and Applications*, 1(2):129–154, 2007.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture*. Prentice Hall PTR, March 2005.

[WFMN04] A. Wombacher, P. Fankhauser, B. Mahleko, and E. J. Neuhold. Matchmaking for Business Processes Based on Choreographies. *International Journal of Web Services Research (JWSR)*, 1(4):14–32, 2004.

[Wol07] M. Wolf. Synchrone und asynchrone Kommunikation in offenen Workflownetzen. Studienarbeit, Humboldt-Universität zu Berlin, May 2007. (In German).

[Wol09] K. Wolf. Does my service have partners? *LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)*, 5460(II):152–171, March 2009.

[WSOD09] K. Wolf, C. Stahl, J. Ott, and R. Danitz. Verifying Livelock Freedom in an SOA Scenario. In *9th International Conference on Application of Concurrency to System Design (ACSD 2009)*. IEEE Computer Society, July 2009.

[YS97] D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997.

[ZBDH06] J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *14th International Conference on Cooperative Information Systems (CoopIS 2006)*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162, Montpellier, France, 2006. Springer.

Index

A bold page number refers to the definition or introduction of a terminology or notation. A page number in normal font refers to relevant information on a subject, or the informal explanation of a concept. An italic page number emphasizes important notations or main results.

Symbols

Operating Guidelines for Services – Summary

In the paradigm of service-oriented computing, companies organize their core competencies as services and may request other functionalities from services of other companies. Services provide high flexibility, platform independent loose coupling, and distributed execution. They may thus help to reduce the complexity of dynamically binding and integrating heterogenous processes within and across organizations. The vision of service-oriented architectures is to provide a framework for publishing new services, for on demand searching for and discovery of existing services, and for dynamically binding services to achieve common business goals. That way, each individual organization gains more flexibility to dynamically react on new challenges.

As services may be created or modified, or collaborations may be restructured at any point in time, a new challenge arises in this setting — the challenge for deciding the compatibility of the composed services before their actual binding. Recent literature distinguishes four different aspects of service compatibility: syntactical, behavioral, semantical, and non-functional compatibility. In this thesis, we focus on behavioral compatibility and abstract from the other aspects. Potential behavioral incompatibilities between services include deadlocks (two services wait for a message of each other), livelocks (two services keep exchanging messages without progressing), and pending messages that have been sent but cannot be received anymore.

For stateful services that interact via asynchronous message passing, deciding behavioral compatibility is far from trivial. Local changes to one service may introduce errors in some or even all other services of an interaction. The verification of behavioral compatibility suffers from state explosion problems and is restricted by privacy issues. That is, the parties of an interaction are essentially autonomous and may be competitors in other business fields. Consequently, they do not want to reveal the internals of their processes to the other participants in order to hide trade secrets.

To systematically approach this challenge, we introduce a formal framework based on Petri nets and automata for service modeling and formalize behavioral compatibility as deadlock freedom of the composition of the services. The main contribution of this thesis is to introduce the concept of the operating guideline of a service. Operating guidelines provide a formal characterization of *the set of all behaviorally compatible services* R for a given service S. Usually, this set is infinite. However, the operating guideline OG_S of a service S serves as a finite representation of this infinite set. Furthermore, the operating guideline of S reveals only internals that are inevitably necessary to decide behavioral compatibility with S. We provide a construction method of operating guidelines for finite-state services with bounded communication.

Operating guidelines can be used in many applications in the context of service-oriented computing. The most fundamental application is to support the discovery of behaviorally compatible services. To this end, we develop a matching procedure that efficiently *decides* whether a given service R is characterized by the operating guideline OG_S of a service S. If R matches, then both services R and S are behaviorally compatible and can be bound together to interact with each other. If R does not match with OG_S, then the services are behaviorally incompatible and may run into severe behavioral errors and not reach their common business goal.

Operating guidelines can furthermore be applied in the novel research areas of service substitutability and the generation of adapter services, for instance. To this end, we develop methods to *compare* the sets of services characterized by the operating guidelines OG_S and $OG_{S'}$. If $OG_{S'}$ characterizes more services than OG_S, then the service S can be substituted by the service S' without loosing any behaviorally compatible interaction partner R. Furthermore, we show how to *synthesize* a service R from the operating guideline OG_S such that R is behaviorally compatible to S by construction.

All results presented in this thesis are implemented in our service analysis tool Fiona. Fiona may compute operating guidelines for services modeled as Petri nets. It may match a service with an operating guideline, compare operating guidelines for equivalence or an inclusion relation, and synthesize service adapters for behaviorally incompatible services. Together with the tool BPEL2oWFN — which translates web services specified in BPEL into Petri net models of the services — we can immediately apply our results to services that stem from practice.

Acknowledgements

First of all, I would like to thank Karsten Wolf for all his support during the last years. He always impresses with his theoretical understanding as well as his visions for applications and new research questions. It has been a pleasure to work with you.

Furthermore, I want to thank Wolfgang Reisig and Kees van Hee for supervising my thesis and giving me both organizational support and constructive comments. In particular, I want to acknowledge the fruitful discussions with Wolfgang about how to formally introduce service nets and automata. This has led to big improvements in the quality of the respective chapter of this thesis. I am grateful to Wil van der Aalst, Schahram Dustdar, and Johann-Christoph Freytag to act as reading committee members. Especially, I thank Wil for his detailed comments which led to many improvements of the presentation.

I also thank my colleagues from the theory of programming group for all the support over the last years. I really enjoyed the warm and pleasant working atmosphere and our productive discussions as well as the spirit of critical evaluation of a presentation. Especially, I want to express my great gratitude to Christian Stahl for the time he invested in proof reading the thesis. He always gives very fruitful comments and finds every minor lack in an argumentation. Thank you. I also want to thank Daniela Weinberg, Dirk Fahland, and Christian Gierds for their proof reading. The presentation in this thesis is also a result of their feedback.

Lots of thanks also go to the members of the architecture of information system group who I got to know when we started with our BEST program and where I experienced an open and warm welcome.

Last, but definitely the most, I thank my family for their patience and support during long periods of stressful hours for writing the thesis. Most of all I want to thank Antje for showing me a light in times where I could not see it by myself.

Peter Massuthe
Berlin, 13th March 2009

Erklärung

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift „Operating Guidelines for Services“ selbständig und ohne unerlaubte Hilfe angefertigt sowie nur die angegebene Literatur verwendet habe,
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder ich einen solchen besitze und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin (veröffentlicht im Amtlichen Mitteilungsblatt Nr. 34/2006) bekannt ist.

Peter Massuthe
Berlin, den 13. März 2009

Curriculum Vitae

Peter Massuthe was born on the 23^{rd} of July 1976 in Wriezen, Germany. From 1990 to 1995, he attended the grammar school "Gymnasium B. Brecht" in Bad Freienwalde. In 1996, he started his study in computer science at Humboldt-Universität zu Berlin, with focus on theoretical computer science. Peter graduated at the Theory of Programming group of Prof. Wolfgang Reisig in 2004. His thesis with the topic "Refinement Methods for the Temporal Logic of Distributed Actions (TLDA)" got the best thesis award of the computer science department of Humboldt-Universität zu Berlin.

During his study, he worked from 1999 to 2001 at DaimlerChrysler Berlin, Department of Research and Technology in the working group on System Safety. In 2001, Peter joint the Theory of Programming group of Prof. Reisig as a student assistant. He worked in the field of Temporal Logics and took part in the "Task Force: Business Process Execution Language for Web Services (BPEL4WS)", where he came in contact with the formal analysis of (Web) services.

Since April 2004, Peter has been working as a research assistant at the Theory of Programming group in the area of Petri nets, model checking, and formal methods for the analysis of service interaction. In 2005, he became part of the "Tools4BPEL" project on "Correctness and Reliability of Interacting Web Services on the Example of the Web Services Business Process Execution Language (WS-BPEL)", funded by the Federal Ministry of Education and Research (BMBF).

Peter Massuthe
Berlin, 13th March 2009

SIKS Dissertatiereeks

====
1998
====

1998-01 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-02 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-03 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-04 Dennis Breuker (UM)
Memory versus Search in Games

1998-05 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

====
1999
====

1999-01 Mark Sloof (VU)
Physiology of Quality Change Modelling;
Automated modelling of Quality Change of Agricultural Products

1999-02 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-03 Don Beal (UM)
The Nature of Minimax Search

1999-04 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-05 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-06 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-07 David Spelt (UT)
Verification support for object database design

1999-08 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

====
2000
====

2000-01 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-02 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-03 Carolien M.T. Metselaar (UVA)
Sociaal-organisatorische gevolgen van kennistechnologie;
een procesbenadering en actorperspectief.

2000-04 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-05 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval.

2000-06 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-07 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-08 Veerle Coup (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-09 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations,
Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

====
2001
====

2001-01 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-02 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-03 Maarten van Someren (UvA)
Learning as problem solving

2001-04 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with
Instance-Based Boundary Sets

2001-05 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-06 Martijn van Welie (VU)
Task-based User Interface Design

2001-07 Bastiaan Schonhage (VU)
Diva: Architectural Perspectives on Information Visualization

2001-08 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-09 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models,
Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice
BRAHMS: a multiagent modeling and simulation language
for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management:
The Role of Mental Models in Business Systems Design

====
2002
====

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments
inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology;
Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative
E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and
Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

====
2003
====

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to

Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

====
2004
====

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

====
2005
====

2005-01 Floor Verdenius (UVA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM))
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)

Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by
Exploiting Application Semantics

====
2006
====

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)
Software-aided Service Bundling - Intelligent Methods & Tools
for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic' (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA)
Focused Information Access using XML Element Retrieval

====
2007
====

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramírez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

====
2008
====

2008-01 Katalin Boer-Sorbán (EUR)
Agent-Based Simulation of Financial Markets: A modular,continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA)
Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU)
Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM)

Adaptive Active Vision

2008-19 Henning Rode (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA)
Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 Krisztian Balog (UVA)
People Search in the Enterprise

2008-22 Henk Koning (UU)
Communication of IT-Architecture

2008-23 Stefan Visscher (UU)
Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU)
Using background knowledge in ontology matching

2008-25 Geert Jonker (UU)
Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT)
Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU)
Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN)
On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT)
Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU)
Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM)
Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT)
Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA)
Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU)
Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT)
Dendritic morphologies: function shapes structure

====
2009
====

2009-01 Rasa Jurgelenaite (RUN)
Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU)
Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT)
A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU)
Understanding Classification

2009-07 Ronald Poppe (UT)
Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU)
Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN)
Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA)
Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA)
Legal Theory, Sources of Law & the Semantic Web