CORRECTNESS OF SERVICES AND THEIR COMPOSITION

NIELS LOHMANN

A library record is available from the Eindhoven University of Technology Library.

Lohmann, Niels

Correctness of services and their composition / by Niels Lohmann
– Eindhoven: Technische Universiteit Eindhoven, 2010. – Proefschrift. –

ISBN 978-90-386-2318-4
NUR 993

SIKS Dissertation Series No. 2010-37

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

The research reported in this theses has been partially supported by the DFG within grant "Operating Guidelines for Services" (WO 1466/8-1) and by the BMBF, project "Tools4BPEL", project number 01ISE08.

Printed by University Press Facilities, Eindhoven.
Cover Design by Paul Verspaget.

CORRECTNESS OF SERVICES AND THEIR COMPOSITION

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 maandag 27 september 2010 om 14.00 uur

door

Niels Lohmann

geboren te Bonn, Duitsland

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. W.M.P. van der Aalst
en
Prof.Dr. K. Wolf

CORRECTNESS OF SERVICES AND THEIR COMPOSITION

DISSERTATION

zur
Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Faktultät für Informatik und Elektrotechnik
der Universität Rostock

vorgelegt von

Niels Lohmann, geboren am 10. Mai 1981 in Bonn
aus Rostock

Rostock, 4. Mai 2010

Gutachter:

1. Prof. Dr. Karsten Wolf
   Universität Rostock
2. prof.dr.ir. Wil M. P. van der Aalst
   Technische Universiteit Eindhoven
3. Prof. Dr. Mathias Weske
   Hasso-Plattner-Institut an der Universität Potsdam

Datum der Verteidigung: Montag, 27. September 2010

CORRECTNESS OF SERVICES AND THEIR COMPOSITION

ABSTRACT

Service-oriented computing (SOC) is an emerging paradigm of system design and aims at replacing complex monolithic systems by a composition of interacting systems, called *services*. A service encapsulates self-contained functionality and offers it over a well-defined, standardized interface.

This modularization may reduce both complexity and cost. At the same time, new challenges arise with the distributed execution of services in dynamic compositions. In particular, the *correctness* of a service composition depends not only on the local correctness of each participating service, but also on the correct interaction between them. Unlike in a centralized monolithic system, services may change and are not completely controlled by a single party.

We study correctness of services and their composition and investigate how the design of correct service compositions can be systematically supported. We thereby focus on the communication protocol of the service and approach these questions using formal methods and make contributions to three scenarios of SOC.

The correctness of a service composition depends on the correctness of the participating services. To this end, we (1) study correctness criteria which can be expressed and checked with respect to a single service. We validate services against behavioral specifications and verify their satisfaction in any possible service composition. In case a service is incorrect, we provide diagnostic information to locate and fix the error.

In case every participating service of a service composition is correct, their interaction can still introduce problems. We (2) automatically verify correctness of service compositions. We further support the design phase of service compositions and present algorithms to automatically complete partially specified compositions and to fix incorrect compositions.

A service composition can also be derived from a specification, called *choreography*. A choreography globally specifies the observable behavior of a composition. We (3) present an algorithm to deduce local service descriptions from the choreography which — by design — conforms to the specification.

All results have been expressed in terms of a unifying formal model. This not only allows to formally prove correctness, but also makes results independent of the specifics of concrete service description languages. Furthermore, all presented algorithms have been prototypically implemented and validated in experiments based on case studies involving industrial services.

KURZFASSUNG

Service-oriented Computing (SOC) ist ein Paradigma des Systementwurfes mit dem Ziel, komplexe monolithische Systeme durch eine Komposition von interagierenden Systemen zu ersetzen. Diese interagierenden Systeme werden *Services* genannt und kapseln in sich abgeschlossene Funktionen, die sie über eine wohldefinierte und standardisierte Schnittstelle anbieten.

Diese Modularisierung vermag Komplexität und Kosten zu senken. Gleichzeitig führt die verteilte Ausführung von Services in dynamischen Kompositionen zu neuen Herausforderungen. Dabei spielt *Korrektheit* eine zentrale Rolle, da sie nicht nur von der lokalen Korrektheit der teilnehmenden Services, sondern auch von der Interaktion zwischen den Services abhängt. Weiterhin können sich Services im Gegensatz zu monolithischen Systemen verändern und werden nicht von einem einzelnen Teilnehmer kontrolliert.

Wir studieren die Korrektheit von Services und Servicekompositionen und untersuchen, wie der Entwurf von korrekten Servicekompositionen systematisch unterstützt werden kann. Wir legen dabei den Fokus auf das Kommunikationsprotokoll der Services. Mithilfe von formalen Methoden tragen wir zu drei Szenarien von SOC bei.

Die Korrektheit einer Servicekomposition hängt von der Korrektheit der teilnehmenden Services ab. Aus diesem Grund (1) studieren wir Korrektheitseigenschaften, die im Bezug auf einen einzelnen Service ausgedrückt und überprüft werden können. Wir validieren Services gegen Verhaltensspezifikationen und verifizieren ihre Gültigkeit in jeder möglichen Servicekomposition. Falls ein Service inkorrekt ist, erarbeiten wir Diagnoseinformationen mit deren Hilfe Fehler lokalisiert und repariert werden können.

Falls alle teilnehmenden Services einer Servicekomposition korrekt sind, kann ihre Interaktion zu Problemen führen. Wir (2) verifizieren automatisch die Korrektheit von Servicekompositionen. Weiterhin unterstützen wir die Entwurfsphase von Servicekompositionen und stellen Algorithmen vor, mit denen teilweise spezifizierte Kompositionen automatisch vervollständigt und mit denen inkorrekte Kompositionen automatisch korrigiert werden können.

Eine Servicekomposition kann weiterhin von einer Spezifikation (*Choreographie* genannt) abgeleitet werden. Eine Choreographie spezifiziert den Nachrichtenaustausch in einer Servicekomposition. Wir (3) erarbeiten einen Algorithmus, mit dem lokale Servicebeschreibungen aus einer Choreographie abgeleitet werden können, die per Konstruktion der Spezifikation genügen.

Alle Resultate wurden in einem einheitlichen formalen Modell ausgedrückt. Dies ermöglicht nicht nur formale Beweise, sondern macht die Resultate von konkreten Spezifikationssprachen unabhängig. Weiterhin wurden alle vorgestellten Algorithmen prototypisch implementiert und anhand von industriellen Fallstudien validiert.

CONTENTS

1 INTRODUCTION

Software and hardware systems are becoming more and more complex. At the same time, such systems are increasingly used by nonexperts — albeit consciously or unconsciously. With the growing influence of computerized systems on nearly every aspect of today's life, *correctness* is of paramount importance in such ubiquitous environments. Incorrect systems, which expose bugs and undefined or unpredictable behavior, do not just affect technical systems any more, but may threaten life in safety-critical systems or compromise the reputation or economical situation of individuals, companies, or governments. A cost analysis from 2002 [145] estimates software bugs to cost alone the U.S. economy nearly 60 billion U.S. dollars a year. This number is growing as a survey from 2008 [86] already reports annual debugging costs for single North American companies of up to 22 million U.S. dollars.

Although postulated for several decades, especially *software systems are not yet designed and implemented in an engineering fashion.* Hence, complex systems usually contain design flaws. This may be because of faster production cycles, benefit-cost analyses, or the sheer size of systems. To still ensure correctness, different approaches have been proposed in the previous decades. From a conceptual point of view, domain-specific languages have been introduced to ease the complexity of specifying systems. They aim at abstracting from specifics (e. g., assembly language or gate-level descriptions) and allow to specify the desired behavior at a human-understandable level of detail, for instance using high-level programming languages such as Java or VHDL. Such languages allow for an intuitive and brief implementation of a system and can help to prevent design flaws in the first place.

As the choice of language does not guarantee correctness alone, bugs often need to be detected in already running systems. As it is undesirable to discover these bugs when the system is operational, *extensive testing is needed.* Testing is an empirical method observing the system's output on given inputs and comparing this output to expected results. This technique is especially effective for software systems or any other system whose behavior can be simulated. The flexibility and nonmaterial nature of software further allows to fix already running systems once a design flaw is detected. Whereas this approach is reasonably cheap and fairly acceptable in noncritical environments, it does not guarantee correctness but just the absence of concrete design flaws in the test runs of the system — testing inherently can only detect the presence of bugs, but not their absence. Notwithstanding, the integration of testing into the development process (called test-driven development) can detect and fix many design flaws in an early stage which in turn may dramatically reduce overall development costs [140].

The only way to guarantee the correctness of a system is to use *formal methods*. Instead of investigating a given concrete system or its outputs, it is translated into a mathematical model on which correctness can be proven. Of course, the model must cover all important aspects of the concrete system that are relevant for the property that needs to be verified. If the model abstracts from important details, it may be proved correct, whereas the implementation still contains errors. For these reasons, formal methods are, compared with testing, expensive and time-consuming, but the only possibility to verify every aspect of life-critical or economically vital systems. However, a formal correctness proof has the disadvantage to be complex and hard to automate. Even worse, proofs tend not to scale with the size of the system under consideration.

To automate verification without losing rigor, *model checking* [39] has been introduced. Model checking treats the verification problem as a search problem in which undesired states (i. e., bugs) are searched in a graph which models the system's behavior. Even though this state graph usually underlies exponential growth in case the number of components is increased, modern techniques allow for model checking of large industrial systems such as hardware circuits, communication protocols, or software drivers. These techniques include abstraction (i. e., irrelevant properties of the system are not modeled or verified), compact representation (e. g., a symbolic representation of the state graph using binary decision diagrams [30]), and compositionally (i. e., deducing the system's correctness from the correctness of its components). Nevertheless, these techniques do not yet scale to large software programs such as operating systems and enterprise systems.

Nowadays, the usage of model checking tools and the formalization of systems and properties are much easier than conducting formal proofs. Additionally, model checking techniques usually return a counterexample which points out a situation in which the model does not meet a specification. A counterexample can help the modeler understand and locate a flaw in the original system where it can be fixed. By iterating model checking and error removal, correctness can be eventually proved — assuming the system is realizable. The approach to achieve correctness this way is called *correctness by verification*.

A different approach to achieve correctness is *correctness by construction*. In this realm, a system or model is constructed from a specification and the correctness immediately follows from the correctness of the construction algorithm. Such completion, recommendation, or correction algorithms focus on the design phase of a system and aim at avoiding design flaws as early as possible. To this end, correctness by construction combines the rigor of verification and the simplicity of test-driven development. However, such constructed models usually need to be refined manually toward actual implementations.

To conclude, different approaches exist to ensure the correctness of systems. They differ in the degrees of maturity and applicability to real-life systems. A close in-

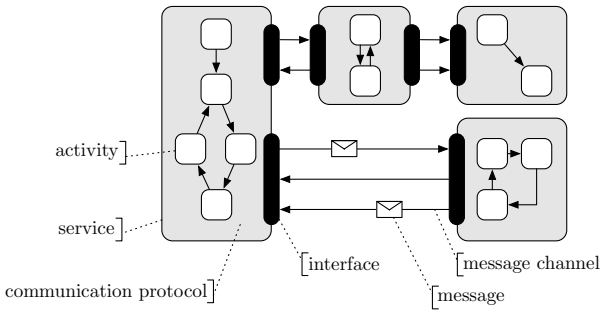


Figure 1.1: A service composition.

tegration of verification techniques into the design process enables the cost-efficient development of correct systems and is a step toward engineering of systems.

Service-oriented computing (SOC) [152] is an emerging paradigm of interorganizational cooperation. It aims at breaking complex monolithic systems into a composition of several simpler and self-contained, yet logically or geographically distributed components, called *services*. A service has an identifier and offers an encapsulated functionality through a well-defined interface; see Fig. 1.1 for an illustration of the concepts. Services are open systems and are designed for being invoked by other services or for invoking other services themselves, and are typically not executed in isolation. Conceptually, SOC revives old ideas from component-based design [132, 175] or from programming-in-the-large [58], for instance.

A simple realization of SOC is the encapsulation of classical computer programs which calculate an output from given inputs as *remote procedure calls* or *stateless services*. Such services only exchange pairs of request/response messages and are capable of implementing simple systems such as stock or weather information systems. This approach is insufficient to implement real-world business scenarios, which do not only calculate an output from given inputs, but in which messages are constantly sent back and forth. Examples for such *stateful* conversations are price negotiations, auctioning, or scenarios in which exception handling is necessary. In this setting, more complex interactions need to be considered and a service needs to implement a *communication protocol* (also called *business protocol* [153]) which specifies the order in which the service's activities are executed and which may distinguish arbitrary states of the interaction with other services. The most prominent class of services are *Web services* [7]. Here, the Internet and several Web-related standards are used to realize SOC. This makes services virtually independent of their geographical location and technological context and allows to entirely focus on the functions a service offers.

The idea of abstracting from underlying technologies and implementations makes it possible to compare services and to replace one service by another service which is, for

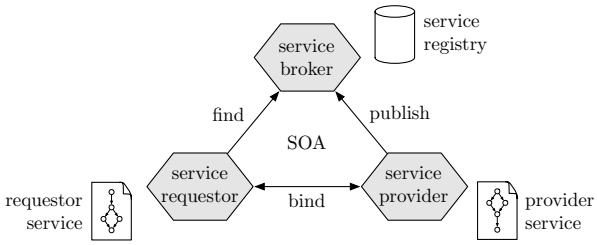
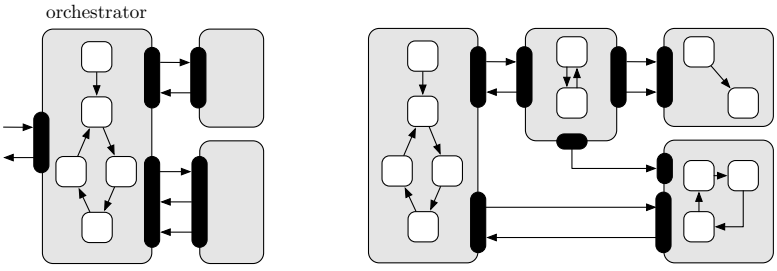


Figure 1.2: The SOA triangle.



(a) service orchestration (b) service choreography

Figure 1.3: Service orchestration versus service choreography.

instance faster, cheaper, compliant with new legal regulations, or more reliable. To this end, SOC allows to effortlessly replace, outsource, and optimize functionalities. This flexible binding is described as a *service-oriented architecture* (SOA) [74]. A SOA provides a general framework for service interaction. This framework — often called the SOA triangle — distinguishes three roles of services (as shown in Fig. 1.2). A *service provider* publishes information about his service to a public registry. A *service broker* manages the registry and allows a *service requester* to find an adequate published service. Then, the provider and the requester may bind their services and start interaction.

In a *service orchestration*, the flexibility to bind formerly unknown providers is employed to offer higher-value added services. It takes the viewpoint of a single participant in the service composition (the *orchestrator*) and abstracts from the internal behavior of other participants. The orchestrator only considers the interfaces of the other participants rather than their concrete behavior or any interaction between third parties (see Fig. 1.3(a)). Service orchestrations are well-suited to describe a business process whose activities are executed by other services.

Services can be also used to specify and implement an entire interorganizational business process. Such a business process is specified by several parties and explicitly

or implicitly describes the behavior of each participant from a global perspective (see Fig. 1.3(b)). From this public description (also called *contract* or *service choreography*), each party derives its share and implements it as a service.

Services received much attention in industry and academia. This is reflected by many standardization efforts for several aspects of services. For instance, there exist various specification and programming languages for services orchestrations (e. g., WS-BPEL [11] or BPMN [150]) and choreographies (e. g., BPEL4Chor [50], WS-CDL [38], iBPMN [48], or BPMN 2.0 choreographies [151]).

1.1 RESEARCH GOAL

Service-orientation allows to construct large distributed systems by composing several heterogenous and decentralized services. This modularization may reduce complexity and cost. At the same time, new challenges arise with the distributed execution of independent services in flexible compositions. In particular, the correctness of a service composition depends on the local correctness of each participating service *and* the correct interaction between them. Unlike in a centralized monolithic system, parts of the system may change and are not completely controlled by a single party. Furthermore, a global state of the system and transitions between states are replaced by local states and local state transitions in addition to message transfer between parties.

Although services have been around for many years and several scientific communities focus on service-related topics, there do not exist widely accepted correctness criteria which are specific to services. From a practical point of view, a system composed of several services can be considered correct if it behaves just as well as a monolithic system. In particular, *the participants should not be aware that the system consists of several decentrally executed components which implement a complex communication protocol and that have been bound without revealing specific implementation details.*

This brings us to the central research question which is investigated in this thesis:

> How can the design of correct services and service compositions be systematically supported?

This question touches upon several challenges:

– *Formalization and verification of correctness.* How to formalize service behavior and correctness notions for services? Can correctness be automatically verified using model checking techniques?

- *Error detection and correction.* In case an error is detected, which participating services are responsible for this error? How can the overall system be fixed toward correct execution?
- *Compositional verification.* Can services be verified in isolation; that is, can local correctness of the participating services be used to derive global correctness of a service composition?
- *Correctness by construction.* Can the design of correct service compositions be supported in a systematic manner? Can errors be avoided in the first place rather than be detected a posteriori? Can service compositions be automatically derived from choreography specifications?
- *Applicability of correctness techniques.* Can the formal methods be applied to industrial services? Do the verification algorithms scale to models of industrial size?

As research goal of this thesis, we want to investigate these challenges on a behavioral level. That said, *we only consider the communication protocol of services and service compositions and abstract from any other aspect which is not immediately related to behavior such as nonfunctional properties, semantics (i. e., ontologies), or instance life cycles.* Our approach complements those aspects: For instance, a proper treatment of semantic discrepancies between services is a prerequisite of our approach, but does not replace the necessity to send and receive messages in a suitable order. Policies and nonfunctional criteria can be integrated into our approach as far as they can be reduced to behavioral constraints. Nonfunctional properties are, however, not the focus of this thesis.

1.2 CONTRIBUTIONS

The contributions of this thesis are all centered around correctness of services and their composition. Parts of the results of this thesis have been published in earlier papers [117, 112, 115, 108, 123]. This thesis summarizes and extends these results. The work presented can be grouped into the following five categories.

CONTRIBUTION 1: FORMAL FOUNDATION

As motivated earlier, formal verification techniques require a formal model of the system under consideration. This thesis investigates correctness in a variety of settings of SOC. As a first contribution, we formalize the aspects sketched in Fig. 1.1 and define with *service automata* a uniform formal model that is able to specify the behavior of single services, service compositions, and service choreographies. Using service automata, we define the correctness notions we shall investigate in the remainder of this thesis:

– *Compatibility.* A service composition is *compatible* iff (1) its execution only terminates in desired final states, (2) message channels are bounded, and (3) nonterminating executions do not exclude a service. We are aware that there exist more sophisticated correctness criteria in literature, for instance, absence of livelocks as an additional requirement. However, our setting is certainly basic enough to be part of any other reasonable concept of correctness of service compositions. Therefore, it can be seen as an intermediate step toward more sophisticated settings.

– *Controllability.* A service is *controllable* [187] iff there exists another service such that their composition is compatible. Controllability is an extension of compatibility to single services and can be seen as a fundamental sanity property for services.

– *Realizability.* A choreography specification is *realizable* [71, 8] iff there exists a compatible service composition which exactly implements the specified interactions. Realizable choreography specifications follow a top-down modeling approach of service compositions which are compatible by design.

The employment of a single formalism throughout this thesis allows us to simplify theory, combine results, and to reuse algorithms and tools. In addition, the results of this thesis can be immediately applied to domain-specific service description languages as soon as a translation into service automata is available. In this thesis, we shall present such translations from WS-BPEL and BPEL4Chor into service automata.

CONTRIBUTION 2: CORRECTNESS OF SERVICES

Compatibility can only be checked for complete service compositions. At design time of single services, such a complete composition is usually not available. To still make a statement on the correctness of a single service, its share of compatibility in *any* possible composition can be analyzed using the notion of controllability. This thesis extends controllability in two aspects:

– *Validation and selection.* In earlier work [117], we focused on the verification of services. We refined the notion of controllability with the help of behavioral constraints. These constraints can be seen as a specification of desired interactions a service can be checked against. If the specification is satisfied by the service, we can synthesize communication partners with the specified communication protocol. Furthermore, we show how a specification can be used to restrict a set of controllable services to only those which additionally satisfy a given specification.

– *Diagnosis.* Not every service is controllable. Unfortunately, the classical analysis algorithm to decide controllability [187] lacks the possibility to provide counterexamples in case a service is uncontrollable. We studied this issue in [112] and presented an algorithm to diagnose uncontrollable services. This diagnosis information (i. e., a

counterexample for controllability) can help to understand the reasons which led to uncontrollability and proposes actions to fix them.

CONTRIBUTION 3: CORRECTNESS OF SERVICE COMPOSITIONS

As described earlier, the composition of logically and geographically distributed services to a compatible overall system can be a challenging task. In this thesis, we propose the following techniques to ease the design of correct service compositions:

- *Verification and completion.* Compatibility of WS-BPEL services and compositions of WS-BPEL services were analyzed in [115]. We provide formal semantics for BPEL4Chor choreographies, which enables the application of existing formal methods to industrial service languages. This includes verification of compositions with respect to compatibility and the completion of partially specified service compositions.
- *Correction.* In case a service composition is not compatible, verification techniques usually provide a counterexample which describes a trace from the initial state to an error state. This trace usually spans over several services of the composition and gives little detail on how to fix the composition. To this end, we defined in [108] an algorithm to suggest changes of a service to achieve overall compatibility.

CONTRIBUTION 4: CORRECTNESS OF SERVICE CHOREOGRAPHIES

A service composition can be built by composing several existing services. A different paradigm follows a top-down approach and globally specifies the interaction protocol, which should be implemented by the service composition. In case this choreography specification is realizable, it can be projected to several services whose composition is compatible and satisfies the specification by design. Our contribution to this topic is as follows.

- *Realization.* In [123], we studied the specification phase of service compositions. We refine existing realizability notions and link the problems related to choreographies to controllability. This allows us to apply all techniques we described so far to the area of choreographies.

CONTRIBUTION 5: TOOL SUPPORT AND EXPERIMENTAL RESULTS

All algorithms presented in this thesis are implemented in several open source free software tools which are available for download at http://service-technology.org/tools. In particular, the following tools were developed in the course of this thesis.

- *Wendy* [121] synthesizes partners for services and implements the decision and diagnosis algorithm for controllability.

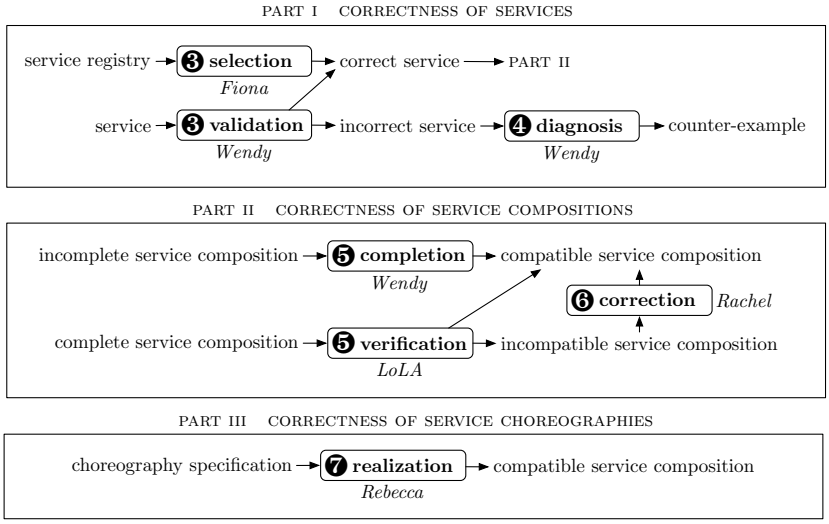


Figure 1.4: Interrelation of the **results**, ❷ chapters, and *tools*.

– *Rachel* [111] provides correction information and recommendations to fix an incompatible service composition toward compatibility.

– *Rebecca* [136] analyzes choreography specifications for realizability and synthesizes realizing services.

We further used the tools *LoLA* [185] and *Fiona* [131] to support additional scenarios investigated in this thesis. In addition, we use the compiler *BPEL2oWFN* [107] to translate industrial services into formal models. Although all tools are proof of concept implementations, experimental results demonstrate the principal feasibility of the approaches. Where possible, we used realistic models translated from WS-BPEL services provided by industrial project partners.

1.3 OUTLINE

The aforementioned list of results sketches an outline for the remainder of this thesis which is illustrated in Fig. 1.4.

The next chapter provides the basic definitions and notions we employ throughout the thesis. It introduces the formal framework we use to model services and service compositions. Furthermore, the correctness criteria introduced informally in this chapter are defined in terms of the formal model. Finally, existing concepts to represent the set of partners of a service are briefly recapitulated.

The remainder of the thesis is divided into three parts, each studying a service-oriented system from a different point of view. We present the software tools used and experimental results obtained within the context of the respective chapters.

– PART I. The first part is dedicated to correctness criteria which can be expressed and checked with respect to a single service. The refinement of the controllability notion to validate services is described in Chap. 3, together with various application scenarios of this notion, for instance the selection of services from a service registry. Chapter 4 focuses on diagnostic information (viz. the construction of counterexamples) in case a service is uncontrollable.

– PART II. In the second part, we go one step further and consider the correctness of service compositions. In Chap. 5, we show how compatibility of industrial service compositions defined in BPEL4Chor can be verified and how a completion algorithm can support the construction of compatible compositions. Chapter 6 shows how correction proposals for incorrect service compositions can be automatically derived.

– PART III. In the last part of the thesis, we study a correctness-by-construction approach for service compositions. Given a choreography specification, we investigate whether this global specification can be realized by several services. Chapter 7 shows how the realizability problem of services can be approached in terms of controllability. This link makes all results of the previous chapters applicable to service choreographies.

Chapter 8 concludes the thesis and summarizes the contributions and the remaining open problems. Furthermore, it sketches directions for future extensions of the presented results.

2 FORMAL MODELS FOR SERVICES

In this chapter, we introduce the basic concepts used in the remainder of this thesis. In particular, we introduce *service automata* as a uniform formalism to define the behavior of a service and a service composition. Based on service automata, we define the correctness notions we investigate in the subsequent chapters. We continue by recalling algorithms to construct and characterize correct service automata. We conclude the chapter with a discussion on the choice of service automata as our formal model.

2.1 PRELIMINARIES

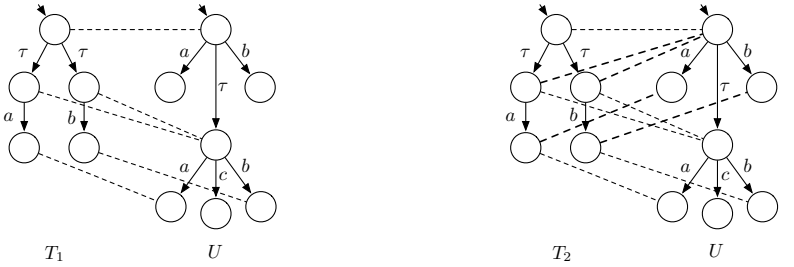
We first recall basic mathematical notions and define several fundamental concepts from computer science.

SETS For a set M, we denote its cardinality with $|M|$ and its powerset with 2^M. We denote the set of natural numbers (including 0) with $\mathbb{N}$ and the set of positive natural numbers (excluding 0) with $\mathbb{N}^+$.

MULTISETS We denote the set of all multisets over a set M with $Bags(M)$. We use the list notation for multisets and, for example, write $[x, y, y]$ for the multiset $\{x \mapsto 1, y \mapsto 2, z \mapsto 0\}$ over $\{x, y, z\}$; $[\,]$ denotes the empty multiset. Addition of multisets $\mathcal{B}_1, \mathcal{B}_2 \in Bags(M)$ is defined pointwise: $(\mathcal{B}_1 + \mathcal{B}_2)(x) := \mathcal{B}_1(x) + \mathcal{B}_2(x)$, for all $x \in M$. For $k \in \mathbb{N}$, we denote with $Bags_k(M)$ the set of multisets such that $\mathcal{B} \in Bags_k(M)$ implies $\mathcal{B}(x) \leq k$, for all $x \in M$.

LABELED TRANSITION SYSTEMS In this thesis, we distinguish visible (i. e., communicating) and invisible (i. e., internal) actions, yielding an extended definition of labeled transition systems: A *labeled transition system* $T = [Q, q_0, \Sigma, \Sigma^\tau, \rightarrow]$ consists of a set of states Q, an initial state $q_0 \in Q$, a set of visible labels Σ, a set of *discriminable invisible* labels Σ^τ with $\Sigma \cap \Sigma^\tau = \emptyset$, and a labeled transition relation $\rightarrow \subseteq Q \times (\Sigma \cup \Sigma^\tau) \times Q$. For $[q, x, q'] \in \rightarrow$, we shall write $q \xrightarrow{x} q'$. If not clear from the context, we add indices to the constituents of T and refer to its states by Q_T, for instance.

A state $q' \in Q$ is *reachable from a state* $q \in Q$, denoted $q \xrightarrow{*} q'$, iff there exists a (possibly empty) sequence of transitions originating in q and ending in q'. A state is *reachable* iff it is reachable from the initial state q_0. If q has no outgoing transitions, we also write $q \not\rightarrow$. We define the set $\tau(q)$ of *internally reachable states* for a state



(a) simulation relation (b) structural matching relation

Figure 2.1: Simulation and structural matching.

$q \in Q$ inductively as follows: (base) $q \in \tau(q)$ and (step) if $q' \in \tau(q)$ and $q' \xrightarrow{x} q''$ with $x \in \Sigma^\tau$, then $q'' \in \tau(q)$.

For a state $q \in Q$, we define $lab(q) := \{x \in \Sigma \mid \exists q' \in Q : q \xrightarrow{x} q'\}$ and $lab^*(q) := \bigcup_{q' \in \tau(q)} lab(q')$. A transition system T is *complete* iff $lab(q) = \Sigma$ for each reachable state $q \in Q$. T is *deterministic* iff $q \xrightarrow{x} q'$ and $q \xrightarrow{x} q''$ implies $q' = q''$ for each reachable state $q \in Q$. T is *τ-free*, if $q \xrightarrow{x} q'$ implies $x \in \Sigma$ for each reachable state $q \in Q$. T is a *finite state* transition system, iff the number of reachable states is finite.

A *strongly connected component* (SCC) of T is a maximal set of states $Q' \subseteq Q$ such that $q, q' \in Q'$ implies $q \xrightarrow{*} q'$. An SCC Q' is a *terminal strongly connected component* iff, for all $q \in Q'$, $q \xrightarrow{*} q'$ implies $q' \in Q'$.

SIMULATION AND STRUCTURAL MATCHING Let T and U be labeled transition systems. A relation $\varrho \subseteq Q_T \times Q_U$ is a *simulation relation*, iff $[q_{0_T}, q_{0_U}] \in \varrho$ and for all states $q_T, q'_T \in Q_T$, all states $q_U \in Q_U$, and for all labels $x \in \Sigma \cup \Sigma^\tau$ holds: if $[q_T, q_U] \in \varrho$ and $q_T \xrightarrow{x}_T q'_T$, then there exists a state $q'_U \in Q_U$ such that $q_U \xrightarrow{x}_U q'_U$ and $[q'_T, q'_U] \in \varrho$.

We use the distinction between visible and invisible labels to define another relation between states of two transition systems: A relation $\varrho \subseteq Q_T \times Q_U$ is a *structural matching relation*, iff $[q_{0_T}, q_{0_U}] \in \varrho$ and for all states $q_T, q'_T \in Q_T$, all states $q_U \in Q_U$, and for all labels $x \in \Sigma \cup \Sigma^\tau$ holds: if $[q_T, q_U] \in \varrho$ and $q_T \xrightarrow{x}_T q'_T$, then (1) there exists a state $q'_U \in Q_U$ with $q_U \xrightarrow{x}_U q'_U$ and $[q'_T, q'_U] \in \varrho$ or (2) $x \in \Sigma^\tau$ and $[q'_T, q_U] \in \varrho$. The first requirement is the same as for a simulation relation. The second requirement allows the transition system T to take an arbitrary number of invisible transitions. This makes a structural matching relation similar to a weak simulation relation or a stuttering simulation relation [29], but stuttering is only allowed in the labeled transition system T.

A transition system U *simulates* (*structurally matches*) a transition system T iff there exists a simulation relation (a structural matching relation) $\varrho \subseteq Q_T \times Q_U$. If the

transition systems T and U are not clear from the context, we shall add indices and write $\varrho_{(T,U)}$. Figure 2.1 illustrates the simulation and structural matching relation.

2.2 MODELING SERVICES AND THEIR COMPOSITION

In this section, we elaborate the core definitions for services and service compositions. We shall introduce the concept of *ports* to model the (purely syntactic) *interface* of a service. To specify the actual behavior of a service (i. e., the order in which messages are sent or received), we employ *service automata*.

Throughout this thesis, we fix a finite set of message channels $\mathbb{M}$ that is partitioned into asynchronous message channels $\mathbb{M}_a$ and synchronous message channels $\mathbb{M}_s$. From $\mathbb{M}$, we derive a set of message events $\mathbb{E}$ that is partitioned into asynchronous send events $!\mathbb{E} := \{!m \mid m \in \mathbb{M}_a\}$, asynchronous receive events $?\mathbb{E} := \{?m \mid m \in \mathbb{M}_a\}$, and synchronous events $!?\mathbb{E} := \{!?m \mid m \in \mathbb{M}_s\}$. Furthermore, we distinguish a noncommunicating event $\tau \notin \mathbb{E}$. For an event $e \in \{!m, ?m, !?m\}$, define its message channel as $\mathbb{M}(e) := m$.

In the following definition, we give message channels a direction and group them into *ports* from which we build *interfaces*. An interface lists the "open" message channels that are exposed to the environment; that is, to other services. Interfaces can be composed by connecting open message channels. This yields "closed" message channels that cannot be used by other services. In this thesis, such closed message channels still belong to the interface. They can be seen as the "composition history" of the respective service automaton that implements the interface. This simplifies subsequent definitions and allows for a unified formal model throughout this thesis.

Definition 2.1 (Port, interface, closed interface).
A pair $P = [I, O]$ is a *port* iff $I \cup O \subseteq \mathbb{M}$ and $I \cap O = \emptyset$. I and O are the *input message channels* and *output message channels* of port P, respectively. For P, define its events as $\mathbb{E}_P := \{?m \mid m \in I \cap \mathbb{M}_a\} \cup \{!m \mid m \in O \cap \mathbb{M}_a\} \cup \{!?m \mid m \in (I \cup O) \cap \mathbb{M}_s\}$.

Let, for $n \in \mathbb{N}$, $\mathcal{P} = \{P_1, \ldots, P_n\}$ be a set of ports with $P_i = [I_i, O_i]$ for $i \in \{1, \ldots, n\}$. $\mathcal{P}$ is an *interface* iff $I_i \cap I_j = \emptyset$ for all $i \neq j$ and $O_i \cap O_j = \emptyset$ for all $i \neq j$. Interface $\mathcal{P}$ is *closed* iff $\bigcup_{i=1}^{n} I_i = \bigcup_{i=1}^{n} O_i$.

From $\mathcal{P}$, derive the set of *closed message channels* $\mathbb{M}_{\mathcal{P}}^{\square} := (\bigcup_{i=1}^{n} I_i) \cap (\bigcup_{i=1}^{n} O_i)$, the set of *open message channels* $\mathbb{M}_{\mathcal{P}}^{\sqcup} := (\bigcup_{i=1}^{n} (I_i \cup O_i)) \setminus \mathbb{M}_{\mathcal{P}}^{\square}$, the set of *internal events* $\mathbb{E}_{\mathcal{P}}^{\tau} := \{e \in \bigcup_{i=1}^{n} \mathbb{E}_{P_i} \mid \mathbb{M}(e) \in \mathbb{M}_{\mathcal{P}}^{\square}\} \cup \{\tau\}$, and the set of *external events* $\mathbb{E}_{\mathcal{P}} := \{e \in \bigcup_{i=1}^{n} \mathbb{E}_{P_i} \mid \mathbb{M}(e) \in \mathbb{M}_{\mathcal{P}}^{\sqcup}\}$.

In a port, each message channel has a direction and is either an input message channel or an output message channel. This is natural for asynchronous communica-

tion where sending and receiving of messages is decoupled. In contrast, synchronous communication is usually undirected. The classification into input and output is of technical nature and can be compared to the complementary labels a and $\overline{a}$ in CCS [138]. Nevertheless, the semantics of a message on a finer level of abstraction may induce a natural initiator. For instance, an asynchronous handshake between two parties (e. g., a pair of a request and an acknowledge message) can be abstracted to an atomic synchronization event.

An interface consists of a set of ports such that communication is bilateral. *A message channel can be used by at most one port as input message channel and by at most one port as output message channel.* If a message channel is used by two ports that way, it is closed and not accessible by other ports any more. This corresponds to *hiding* in process algebra [14]. From the message channels of a port and their direction, potential events can be derived. These events can be partitioned into internal events (including τ) and external events depending on whether the respective message channel is open or closed.

Depending on the context, an interface can be interpreted differently: for a single service, open message channels are exposed to the environment which can invoke the service. An interface with more than one port can be used to model a service orchestrator that interacts with several services simultaneously. In WS-BPEL [11], the term *partner link* has been coined for such a partition of an interface. Finally, a closed interface describes a choreography of services (cf. Chap. 7). In this scenario, no message channel is exposed to the environment: the sender and receiver of each message is specified. This *closed world assumption* is common in choreography description languages such as BPEL4Chor [50] or WS-CDL [90].

Ports and interfaces only describe the syntactic signature of a service consisting of an alphabet of possible events. The behavior itself (i. e., the order in which messages exchange occurs and when a service terminates) is modeled by *service automata*. A service automaton is a state machine whose transitions are labeled with events derived from a given interface.

Definition 2.2 (Service automaton).
A tuple $A = [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ is a *service automaton* iff

- $\mathcal{P}$ is an interface,
- $[Q, q_0, \mathbb{E}_{\mathcal{P}}, \mathbb{E}^{\tau}_{\mathcal{P}}, \rightarrow]$ is a labeled transition system, and
- $\Omega \subseteq Q$ is a set of final states.

A is a *single-port service automaton*, iff $|\mathcal{P}| = 1$; otherwise, A is a *multi-port service automaton*. A is *closed*, iff $\mathcal{P}$ is a closed interface; otherwise, A is *open*.

A service automaton *implements* the ports of its interface by labeling its state transitions with events. If a service automaton implements more than one port,

message channels can be closed and the respective events are internal. Whereas internal events can occur independently of the service's environment, external message events can only be realized together with other services. This interplay with other services is defined in terms of the *composition* of service automata.

Definition 2.3 (Composition of service automata).
Two service automata A and B are *composable* iff $\mathcal{P}_A \cap \mathcal{P}_B = \emptyset$ and $\mathcal{P}_A \cup \mathcal{P}_B$ is an interface.

The *composition* of two composable service automata A and B is the service automaton $A \oplus B = [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ consisting of

- $Q := Q_A \times Q_B \times Bags(\mathbb{M}_a)$,
- $q_0 := [q_{0_A}, q_{0_B}, [\,]]$,
- $\Omega := \Omega_A \times \Omega_B \times \{[\,]\}$,
- $\mathcal{P} := \mathcal{P}_A \cup \mathcal{P}_B$, and
- $\rightarrow$ containing exactly the following elements:
  1. for all $m \in \mathbb{M}^{\sqcup}_{\mathcal{P}_A} \cap \mathbb{M}^{\sqcup}_{\mathcal{P}_B}$ (shared open message channels) and $\mathcal{B} \in Bags(\mathbb{M}_a)$,
     - $[q_A, q_B, \mathcal{B}] \xrightarrow{!m} [q'_A, q_B, \mathcal{B} + [m]]$, iff $q_A \xrightarrow{!m}_A q'_A$,
     - $[q_A, q_B, \mathcal{B}] \xrightarrow{!m} [q_A, q'_B, \mathcal{B} + [m]]$, iff $q_B \xrightarrow{!m}_B q'_B$,
     - $[q_A, q_B, \mathcal{B} + [m]] \xrightarrow{?m} [q'_A, q_B, \mathcal{B}]$, iff $q_A \xrightarrow{?m}_A q'_A$,
     - $[q_A, q_B, \mathcal{B} + [m]] \xrightarrow{?m} [q_A, q'_B, \mathcal{B}]$, iff $q_B \xrightarrow{?m}_B q'_B$,
     - $[q_A, q_B, \mathcal{B}] \xrightarrow{!?m} [q'_A, q'_B, \mathcal{B}]$, iff $q_A \xrightarrow{!?m}_A q'_A$ and $q_B \xrightarrow{!?m}_B q'_B$;
  2. for $e \in \mathbb{E}^{\tau}_{\mathcal{P}_A} \cup \mathbb{E}^{\tau}_{\mathcal{P}_B}$ (internal events) or $e \in \mathbb{E}_{\mathcal{P}}$ (external events) and $\mathcal{B} \in Bags(\mathbb{M}_a)$,
     - $[q_A, q_B, \mathcal{B}] \xrightarrow{e} [q'_A, q_B, \mathcal{B}]$, iff $q_A \xrightarrow{e}_A q'_A$,
     - $[q_A, q_B, \mathcal{B}] \xrightarrow{e} [q_A, q'_B, \mathcal{B}]$, iff $q_B \xrightarrow{e}_B q'_B$.

The composability criteria require that the two services must not share a port (i. e., each port is implemented by exactly one service automaton) and that their union still has the interface property of unidirectional and bilateral communication. Here, keeping closed message channels in the interface is important to keep track of the "composition history" of a service.

The composition of two service automata implements the union of their ports. For the state transitions, we distinguish two cases: (1) communication events between the composed services and (2) other events that are either internal to one of the composed services or external to the composition. Shared message events do not only influence a service's state, but may also add messages to or remove messages from an asynchronous message channel. To this end, each state of the composition contains a multiset of asynchronously sent messages that have not yet been received and that

are pending on the message channel. This represents lossless asynchronous message passing under the assumption that messages can overtake each other. Synchronization between two services does not influence the pending messages. The message buffer is defined to be empty in the initial state and is required to be empty in the final states. The latter requirement rules out interactions that terminate without considering pending messages.

As notational convention, we identify the states of a composition of more than two services by a combination of the participating services' states and a sum of the pending asynchronous messages. For example, we do not identify a state of the composition $(A \oplus B) \oplus C$ with $[[q_A, q_B, \mathcal{B}_{(A\oplus B)}], q_C, \mathcal{B}_{(A\oplus B)\oplus C}]$, but with $[q, \mathcal{B}]$ for $q := [q_A, q_B, q_C]$ and $\mathcal{B} := \mathcal{B}_{A\oplus B} + \mathcal{B}_{(A\oplus B)\oplus C}$. Due to the requirement of bilateral communication and the retainment of closed ports, the composition is — up to isomorphism — commutative and associative. Hence, we may treat composition as a partial operation and write $A \oplus B \oplus C$ instead of $(A \oplus B) \oplus C$.

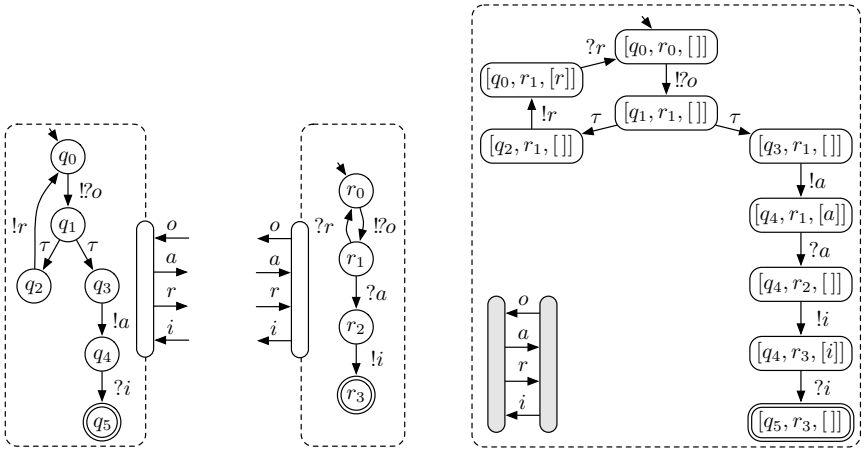
EXAMPLE. As running example for this chapter, consider the service automaton depicted in Fig. 2.2(a). It models a buyer service that receives offers (o) from a client and decides whether to accept (a) or to reject (r) the offer. This decision is modeled by internal τ-steps and is nondeterministic. In case the offer got rejected, the service returns to its initial state (q_0) and waits for another offer. In case the offer got accepted, the service eventually receives an invoice (i) and reaches the final state (q_5). As it can be seen from the graphical representation, the message channels a, r, and i are asynchronous, whereas o is a synchronous channel.

Figure 2.2(b) depicts a composable service automaton modeling a seller service. It sends offers until one gets accepted. The composition of the buyer and the seller service yields the closed service automaton in Fig. 2.2(c). Throughout this thesis, we shall never depict unreachable states.

2.3 CORRECTNESS NOTIONS FOR SERVICES

Services are not executed in isolation, but are designed to communicate with other services. To this end, reasoning about a service's behavior only makes sense if it is part of a closed composition; that is, all message channels are closed and all events are internal. The behavior of a closed composition can then be defined with the concept of *runs*.

Definition 2.4 (Run, terminating run, deadlocking run).
For a closed service automaton $A = [Q, q_0, \to, \Omega, \mathcal{P}]$, a finite or infinite sequence of states $q_0 q_1 \cdots$ is a *run* of A iff there exists an event e_i with $q_i \xrightarrow{e_i} q_{i+1}$ for all $i \geq 0$. A finite run $q_0 \cdots q_n$ is *maximal* iff there exists no state $q_{n+1} \in Q$ with $q_n \xrightarrow{*} q_{n+1}$ and $q_{n+1} \neq q_n$. A maximal run $q_0 \cdots q_n$ *terminates* iff $q_n \in \Omega$ and *deadlocks* iff $q_n \notin \Omega$.



(a) A_{Buy} (b) A_{Sell} (c) $A_{\text{Buy}} \oplus A_{\text{Sell}}$

Figure 2.2: A buyer service A_{Buy} (a) and a seller service A_{Sell} (b) modeled as service automata. These automata are composable and (c) depicts their composition $A_{\text{Buy}} \oplus A_{\text{Sell}}$.

With the set of final states we can distinguish desired terminal states, which model the successful completion of a service composition, on the one hand from design errors or undesired deadlocks, on the other hand. We refer to the absence of deadlocks in a composition as *deadlock freedom*. By definition of the composition of service automata, asynchronous message channels must be empty in a final state.

In this thesis, we do not require every run be extensible to a terminating run (a property that is usually called *livelock freedom* or *weak termination*). We do allow infinite runs even if no final states are reachable as long as no port is excluded from communication. A service with this property is called *responsive*.

Definition 2.5 (Responsiveness).
A service automaton $A = [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ is *responsive* iff for every terminal strongly connected component Q' of Q holds: (1) $Q' \cap \Omega \neq \emptyset$, or (2) for every port $P \in \mathcal{P}$ there exists a state $q \in Q'$ with an outgoing transition that is labeled with an event $e \in \mathbb{E}_P$.

If a closed composition of services is responsive, then every infinite run can reach a final state or contains communication events from every port.

As final requirement, communication must not yield an unbounded number of messages pending on asynchronous message channels. The preceded properties are

combined in the concept of *compatibility*, which is the core correctness criterion we investigate in this thesis.

Definition 2.6 (Message bound, k-compatibility).
Let $A = A_1 \oplus \cdots \oplus A_n$ be a closed service automaton. For a *message bound* $k \in \mathbb{N}^+$, A is *k-compatible* iff

1. every maximal run of A terminates,
2. $\mathcal{B}(m) \leq k$ for every reachable state $[q, \mathcal{B}]$ of A and $m \in \mathbb{M}_a$, and
3. A is responsive.

A closed composition of services is k-compatible iff (1) finite interactions always reach a desired final state in which all message channels are empty, (2) during communication, no asynchronous message channel will ever need to store more than k pending messages, and (3) infinite runs have the possibility to terminate or span all participating ports. A finite and fixed message bound k is motivated by the middleware that realizes the communication of services in reality. The value of k is either known in advance, is derived using capacity considerations or static analysis techniques, or is chosen sufficiently large. In this thesis, we use several models derived from real WS-BPEL processes in which there are hardly any message channels where more than a single pending message made sense. We usually use the term "compatibility" without mentioning a specific message bound if the value itself is not of interest or is clear from the context.

Compatibility is a fundamental correctness criterion for closed service compositions. We are aware of more sophisticated criteria, for instance livelock freedom, exclusion of dead activities, or satisfaction of certain temporal logic formulae. Nevertheless, deadlock freedom, bounded communication, and responsiveness would be certainly part of any refined correctness notion. We shall present a refinement of the compatibility notion in Chap. 3.

The notion of compatibility can be extended to an open service, yielding the concept of *controllability*.

Definition 2.7 (k-controllability, k-strategy).
For a message bound $k \in \mathbb{N}^+$, a service automaton A is *k-controllable* iff there exists a service automaton A' such that the composition $A \oplus A'$ is k-compatible. We call A' a *k-strategy* for A and denote the set of all k-strategies of A with $Strat_k(A)$.

The term "strategy" originates from control theory [37, 160]: We may see A' as a controller for A imposing compatibility on $A \oplus A'$.

Controllability allows us to reason about a single service while taking its communicational behavior (i. e., the service's local contribution to overall compatibility) into

account. It is a fundamental correctness criterion for open services, because a service that cannot interact deadlock freely, bounded, and responsively with *any* other service is certainly ill-designed. In Chap. 4, we shall present an algorithm to diagnose the reasons for uncontrollability. From a practical point of view, a k-strategy of a service A does not only prove its k-controllability, but is also a valuable tool to validate, test, or document the service A. Furthermore, a synthesized strategy can be used as a *communication proxy*, which can be implemented (i. e., refined) toward an executable service that is by design compatible to the original service.

2.4 CONSTRUCTION OF STRATEGIES

In this section, we briefly describe an algorithm from Wolf [187] to construct a k-strategy for service automaton if one exists. The approach is limited to finite state service automata. For infinite state services, a related controllability notion is undecidable [130].

To construct a strategy for a finite state service automaton A, we first overapproximate the behavior of *any* service automaton that is composable to A. As the internal state of A is not observable, we can only make assumptions based on the messages sent to and received from A, respectively. These assumptions and the uncertainty about the exact state can be modeled by a *set* of states the service can assume at a certain point of interaction. These sets of states also include pending asynchronous messages.

Definition 2.8 (Closure).
Let $A = [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ be a service automaton. For a set $X \subseteq (Q \times Bags(\mathbb{M}_A))$, we define the set $closure_A(X) \subseteq (Q \times Bags(\mathbb{M}_a))$ to be the smallest set satisfying:

1. $X \subseteq closure_A(X)$.
2. If $[q, \mathcal{B}] \in closure_A(X)$ and $q \xrightarrow{e} q'$ with $e \in \mathbb{E}^{\tau}_{\mathcal{P}}$, then $[q', \mathcal{B}] \in closure_A(X)$.
3. If $[q, \mathcal{B}] \in closure_A(X)$ and $q \xrightarrow{!m} q'$ with $!m \in \mathbb{E}_{\mathcal{P}}$, then $[q', \mathcal{B} + [m]] \in closure_A(X)$.
4. If $[q, \mathcal{B} + [m]] \in closure_A(X)$ and $q \xrightarrow{?m} q'$ with $?m \in \mathbb{E}_{\mathcal{P}}$, then $[q', \mathcal{B}] \in closure_A(X)$.

The closure of a set of states contains all states that can be reached in A without requiring any actions of the environment. That is, it contains those states that are reachable by internal events, by receiving already pending asynchronous messages, and by sending asynchronous messages. Synchronous message events are not considered, because synchronization would involve the environment.

Given an open responsive finite state service automaton A, the following definition constructs a composable service automaton $TS^0(A)$ that overapproximates the behavior of any service automaton that is composable to A. The states of $TS^0(A)$ consist of sets of states of A together with a multiset of pending asynchronous messages. In subsequent steps, those states of $TS^0(A)$ are removed which either violate a given message bound k or deadlock freedom. If the resulting automaton $TS_k(A)$ has a nonempty set of states, A is k-controllable and $TS_k(A)$ is a strategy for A.

Definition 2.9 (Strategy synthesis).
Let $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ be an open responsive finite state service automaton with $\mathcal{P}_A = \{[I_1, O_1], \ldots, [I_n, O_n]\}$. We define the open service automaton $TS^0(A) = [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ with $\mathcal{P} = \{[O, I] \mid [I, O] \in \mathcal{P}_A \cap (\mathbb{M}^{\sqcup}_{\mathcal{P}_A} \times \mathbb{M}^{\sqcup}_{\mathcal{P}_A})\}$ and Q, q_0, $\rightarrow$, and Ω inductively as follows:

- Base: Let $q_0 := closure_A(\{[q_{0_A}, [\,]]\})$. Then $q_0 \in Q$.
- Step: For all $q \in Q$ and $m \in \mathbb{M}$:
  1. If $!m \in \mathbb{E}_{\mathcal{P}}$, let $q' := closure_A(\{[q_A, \mathcal{B} + [m]] \mid [q_A, \mathcal{B}] \in q\})$.
     Then $q' \in Q$ and $q \xrightarrow{!m} q'$.
  2. If $?m \in \mathbb{E}_{\mathcal{P}}$, let $q' := closure_A(\{[q_A, \mathcal{B}] \mid [q_A, \mathcal{B} + [m]] \in q\})$.
     Then $q' \in Q$ and $q \xrightarrow{?m} q'$.
  3. If $!?m \in \mathbb{E}_{\mathcal{P}}$, let $q' := closure_A(\{[q'_A, \mathcal{B}] \mid [q_A, \mathcal{B}] \in q \wedge q_A \xrightarrow{!?m}_A q'_A\})$.
     Then $q' \in Q$ and $q \xrightarrow{!?m} q'$.
- We define $\Omega := \{q \in Q \mid q \cap (\Omega_A \times \{[\,]\}) \neq \emptyset\}$.

For a message bound $k \in \mathbb{N}^+$, let $TS^1_k(A)$ be the service automaton that is obtained from $TS^0(A)$ by removing each state $q \in Q$ that contains a state $[q^*, \mathcal{B}]$ with $\mathcal{B}(m) > k$ for an asynchronous message channel $m \in \mathbb{M}_a$.

Given $TS^i_k(A)$ $(i \geq 1)$, the service automaton $TS^{i+1}_k(A)$ is obtained by removing state $q \in Q_i$ if there exists a $[q_A, \mathcal{B}] \in q$ such that the state $[q, q_A, \mathcal{B}]$ of the composition $TS^i_k(A) \oplus A$ is neither final nor has a successor in $TS^i_k(A) \oplus A$. Thereby, the removal of a state includes the removal of its adjacent arcs and all states that become unreachable from the initial state q_0.

Let $TS_k(A)$ be $TS^j_k(A)$ for the smallest j with $TS^j_k(A) = TS^{j+1}_k(A)$.

The first overapproximaton $TS^0(A)$ is usually an infinite state service automaton that interacts arbitrarily with A. As mentioned earlier, the states $TS^0(A)$ consist of sets of states of the service automaton A. If such a state contains a state in which the message bound k is exceeded, it is removed. This yields the finite state service automaton $TS^1_k(A)$. In subsequent steps, any deadlocking states are removed until

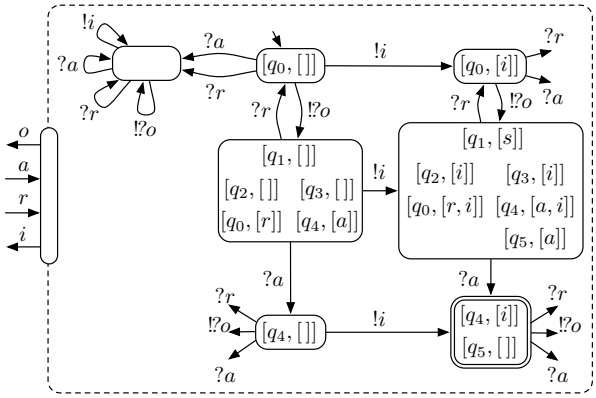


Figure 2.3: The synthesized strategy $TS_1(A_{\text{Buy}})$ of the buyer service A_{Buy} from Fig. 2.2(a) for $k = 1$. Transitions without target states are assumed to have the state $\emptyset \in 2^{Q \times Bags(\mathbb{M}_a)}$ as target.

a greatest fixed point, $TS_k(A)$, is reached. The synthesized service automaton is by design responsive, because it does not contain noncommunicating actions.

This algorithm is correct as it was shown by Wolf [187].

Proposition 2.1 (Synthesis is a strategy [187]).
A is k-controllable iff $Q_{TS_k(A)} \neq \emptyset$.

Wolf [187] uses a slightly different notion of responsiveness, but the difference does not harm. By definition, $TS_k(A)$ closes all open ports of A. Wolf [187] introduced additional controllability notions that take the partition of message channels over ports into account. We shall introduce these notions in Chap. 7 in the context of choreographies.

EXAMPLE. Figure 2.3 depicts the result of Def. 2.9 for the buyer service. Due to asynchronous communication, there are two remarkable details: First, it is also possible to send an invoice message (i) in the initial state. This message keeps pending on the message channel until the selling service is able to receive it. Second, the synthesized strategy also contains an empty state ($q = \emptyset$). This state and its adjacent arcs model behavior that may be present in a strategy, but will be unreachable in the composition with A. For instance, a service that not only places an order (o), but is also ready to receive an acceptance message (a) in the initial state is a valid strategy of the buyer service as long it is also ready to receive a later.

2.5 FINITE CHARACTERIZATION OF STRATEGIES

In this section, we summarize results from Massuthe and Wolf [118, 128] to finitely characterize the possibly infinite set of strategies of a service. As a first observation, strategies can be compared with each other with respect to their behavior. In particular, some strategies permit "more behavior" than others.

Definition 2.10 (k-most-permissive strategy).
A strategy $B^* \in Strat_k(A)$ of a service automaton A is *k-most-permissive* iff B^* structurally matches any other k-strategy of A.

Thereby, the structural matching relation between service automata is defined on their underlying labeled transition systems. A most-permissive strategy can be seen as a top element in a preorder of service behaviors. This preorder [128] is out of scope of this thesis. The strategy synthesized by the algorithm of Def. 2.9 is a most-permissive strategy.

Proposition 2.2 (Synthesis is most-permissive [187]).
Let A be a k-controllable service automaton.
Then $TS_k(A)$ is a k-most-permissive strategy of A.

The proof [118, 187, 128] is based on Prop. 2.1 and exploits that the composition with any service automaton with "more" behavior would not be compatible.

By definition, a k-most-permissive strategy B^* of a k-controllable service A structurally matches any other k-strategy of A. The converse does, however, not hold: there exist service automata $C \notin Strat_k(A)$ which are structurally matched by B^*. Such services can be ruled out by adding Boolean annotations to the states of a B^*.

Definition 2.11 (Annotated automaton).
The tuple $B^\varphi = [B, \varphi]$ is an *annotated automaton* iff $B = [Q, q_0, \rightarrow, \{P\}]$ is a deterministic τ-free single-port service automaton without final states, and φ is an annotation that assigns a Boolean formula to every state $q \in Q$. The formulae are built on $\mathbb{E}_P$, an additional proposition *final*, and the Boolean operators $\wedge$, $\vee$, and $\neg$.

Annotated automata have been introduced by Wombacher et al. [191] to represent sets of automata. In our context, the Boolean formulae are used to refine the structural matching relation by adding constraints on the edges that leave a state of a service automaton that is structurally matched by a most-permissive partner. Annotated automata have no final states; whether a state of a represented automaton needs to

be final is expressed by the proposition *final*. The truth value of an annotated formula is evaluated by an assignment function.

Definition 2.12 (Assignment, model).
Let $A = [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ be a service automaton. We define the *assignment* $\beta : Q \times (\mathbb{E} \cup \{final\}) \rightarrow \{true, false\}$ as follows:

$$\beta(q,p) := \begin{cases} true, & \text{if } p \in lab^*(q), \\ true, & \text{if } p = final \text{ and } \tau(q) \cap \Omega \neq \emptyset, \\ false, & \text{otherwise.} \end{cases}$$

A state $q \in Q$ *models* a formula φ (denoted $q \models \varphi$) iff φ evaluates to *true* under the assignment $\beta(q, \varphi)$. We thereby assume the standard semantics for the Boolean operators $\wedge$, $\vee$, and $\neg$.

An atomic proposition of a formula is true in a state of a service automaton if that state has a respective outgoing edge, possibly reached by a sequence of internal steps. The proposition *final* is evaluated to true exactly in final states. The following definition of *matching* combines structural matching and formulae evaluation.

Definition 2.13 (Matching).
A service automaton A *matches* with an annotated automaton B^φ iff:

1. there exists a structural matching relation $\varrho \subseteq Q_A \times Q_B$ and
2. for all $[q_A, q_B] \in \varrho$: $q_A \models \varphi(q_B)$.

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

The first requirement states that a service automaton matches with an annotated automaton only if there exists a structural matching relation. If such a relation exists, it consists of pairs of states for which the formulae must be satisfied in the second step. With this matching predicate, an annotated automaton implicitly defines a (possibly infinite) set of service automata. In particular, we are interested in annotated automata that exactly characterize the set of k-strategies of a service.

Definition 2.14 (k-operating guideline).
A *k-operating guideline* for a service automaton A is an annotated automaton $OG_A^k = B^\varphi$ such that $Match(OG_A^k) = Strat_k(A)$.

Every k-controllable service has a k-operating guideline; Masuthe et al. [118, 128] provide detailed proofs and a construction algorithm. The core idea is to use a

k-most-permissive strategy and to annotate each state with a formula that is satisfiable iff those events are present that resolve any deadlock within the associated closure of states while still respecting the message bound.

Beside the aforementioned finiteness, Massuthe and Wolf [118, 128] further emphasize the following properties that operating guidelines enjoy. These properties are essential for the results we present in subsequent chapters.

- Matching is only defined in terms of structurally matching and formula evaluation. In particular, it does not take the states of the closure (cf. Def. 2.8) into account. This not only allow for a compact representation (i. e., only the structure of a most-permissive partner needs to be stored), but also avoids an explicit exposure of the service's internal structure which might be subject to trade secrets.

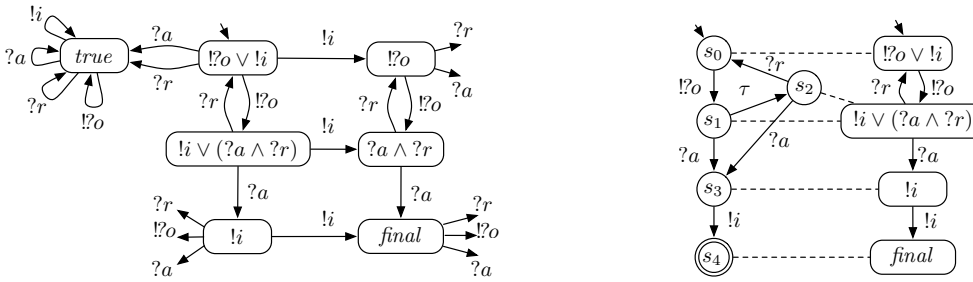
- The formulae of an operating guideline can be transformed into positive formulae (i. e., formulae without negations) [122]. This increases the efficiency of formula evaluation during matching.

- Having a most-permissive strategy as structure, operating guidelines are operational; that is, k-compatible service automata can be easily derived from operating guidelines.

Operating guidelines defined in [118, 128] base on a compatibility notion that does not include responsiveness. To this end, operating guidelines also characterize services that "control" other services by performing an infinite sequence of noncommunicating actions (e. g., τ-loops). Even if the interaction with such unresponsive services is deadlock free and bounded, these services can hardly be used to construct compatible service compositions. Therefore, Defs. 2.9–2.13 have been adjusted to be applicable in the context of our compatibility criterion.

EXAMPLE. Figure 2.4(a) depicts an operating guideline of the seller service. It has the structure of the most-permissive strategy of Fig. 2.3 and each state is annotated with a Boolean formulae. These formulae constrain the behavior of matching services. For instance, formula $?a \wedge ?r$ demands that a matching service must be able to receive an acceptance (a) *and* a rejection (r) message. The state modeling unreachable behavior is annotated with *true*: we pose no constraints on unreachable behavior. Note that the events that lead to this *true*-annotated state (e. g., $?a$ and $?r$ in the initial state) are not mentioned in the formulae.

Figure 2.4(b) depicts an example for a matching service. The dashed lines connect states that are in a structural matching relation between a seller service and a fraction of $OG^1_{A_{\text{Buy}}}$. The states s_1 and s_2 are connected by a τ-annotated edge and match with the same state in the operating guideline. State s_1 satisfies the formula $!i \vee (?a \wedge ?r)$, because the state s_2 has both an edge labeled with $?a$ and $?r$, and this state is internally reachable from s_1.



(a) $OG^1_{A_{\text{Buy}}}$ (b) matching with $OG^1_{A_{\text{Buy}}}$

Figure 2.4: The operating guideline $OG^1_{A_{\text{Buy}}}$ (a) of the buyer service A_{Buy}. Transitions without target states are assumed to have the state with the "*true*" annotation as target. A matching service automaton together with the structural matching relation is depicted in (b).

2.6 EXPERIMENTAL RESULTS

Both the strategy synthesis algorithm [187] (cf. Def. 2.9) and the algorithm to calculate an operating guideline for a service [118, 128] have been implemented in the tool Wendy [121]. These two algorithms were originally implemented in the tool Fiona [131]. The design goal of Fiona was the combination of several analysis and synthesis algorithms for service behavior. This is reflected by a flexible architecture that aims at the reusability of data structures and algorithms. Although this design facilitated the quick integration and validation of new algorithms, the growing complexity made optimizations more and more complicated.

To overcome these efficiency problems, Wendy is a reimplementation of the two synthesis algorithms as compact single-purpose tool. This reimplementation incorporates the experiments made by analyzing performance bottlenecks through improved data structures and memory management, validation of experimental results which gave a deeper understanding of the parameters of the models that affect scalability, and theoretical observations on regularities of synthesized strategies and operating guidelines.

As a proof of concept, we calculated operating guidelines of several WS-BPEL services from a consulting company. Each process consists around 40 WS-BPEL activities and models communication protocols and business processes of different industrial sectors. To apply the algorithms of this chapter, we first translated the WS-BPEL processes into service automata using the compiler BPEL2oWFN [107] implementing the formal semantics we shall discuss in Chap. 5.

Table 2.1 lists details on the processes as well as the experimental results. We see that the service automata derived from the WS-BPEL processes have up to 14,990

Table 2.1: Experimental results for strategy synthesis using Wendy.

	analyzed service automaton			synthesis result		
service	$\|Q\|$	$\|\rightarrow\|$	$\|\mathbb{E}_{\mathcal{P}}\|$	$\|Q_{TS}\|$	$\|\rightarrow_{TS}\|$	time (sec)
Quotation	602	1,141	19	11,264	145,811	0
Deliver goods	4,148	13,832	14	1,376	13,838	2
SMTP protocol	8,345	34,941	12	20,818	144,940	29
Car analysis	11,381	39,865	15	1,448	13,863	49
Identity card	14,569	71,332	11	1,536	15,115	82
Product order	14,990	50,193	16	57,996	691,414	294

states. These large sizes can be explained by the fact that both the positive as well as the negative control flow (i. e., fault and compensation handling) are modeled. The interfaces consist of up to 19 WSDL [38] operations.

The number of states of the operating guidelines (i. e., the most-permissive strategy) are sometimes much larger than the original service. The number of transitions grows even faster. From these transitions, about the moiety have the empty node $q = \emptyset$ as target state. The analysis takes up to 294 seconds on a 3 GHz computer. This is acceptable, because operating guidelines are usually calculated to be used by the service broker many times. Massuthe [128] reports an experiment where the compatibility of two services A and B is verified by model checking the composition $A \oplus B$ on the one hand and calculating the operating guideline OG_A and checking whether $B \in Match(OG_A)$ on the other hand. As result, Massuthe reports that using operating guidelines outperforms model checking in case more than seven checks are made.

In comparison, Fiona could only analyze three of the six services without exceeding 2 GB of memory. For the other models, the analysis was between 5 and 70 times slower than Wendy. To conclude, Wendy allows for the synthesis of strategies and the calculation of operating guidelines of industrial Web services. To give an example of the structure of such strategies, Fig. 2.5 shows an operating guideline of a smaller version of the SMTP protocol.

2.7 DISCUSSION

The original contribution of this chapter is the definition of *service automata as a unified formalism to define and reason about services and service compositions that communicate synchronously or asynchronously.* We conclude this chapter with a discussion and a classification of service automata. A discussion of controllability and operating guidelines is beyond the scope of this theses, and we refer the interested reader to the work of Massuthe et al. [118, 187, 128].

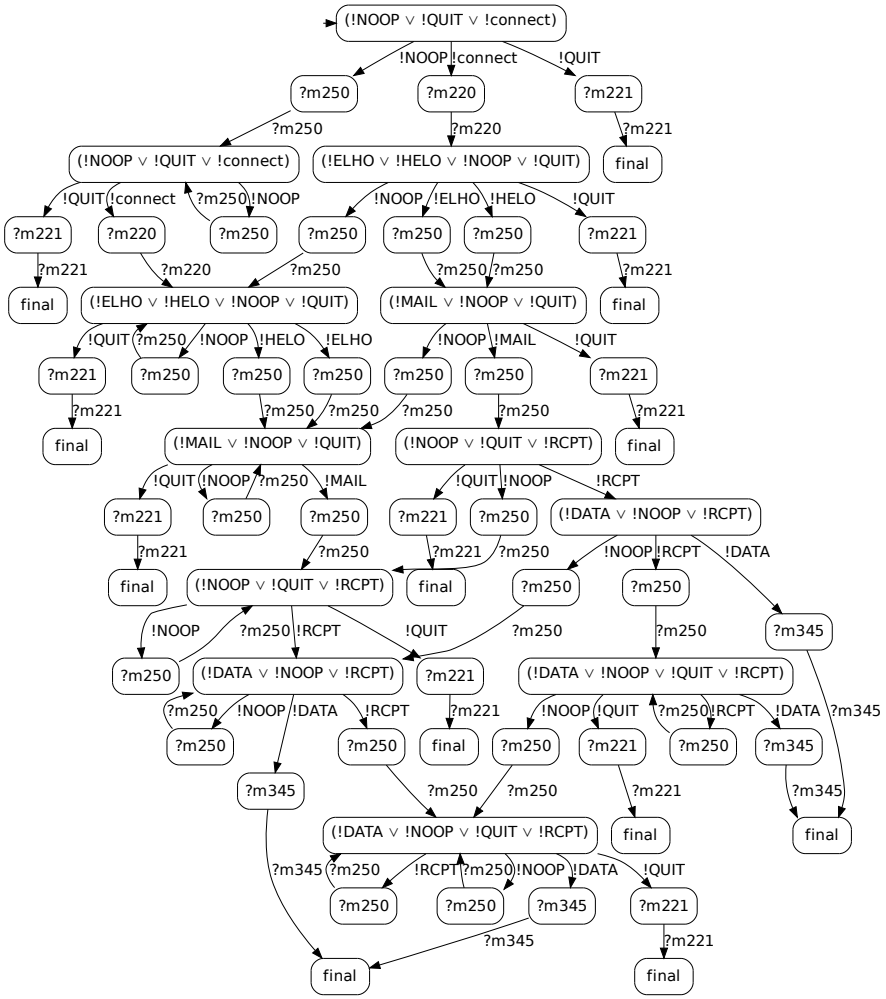


Figure 2.5: Operating guideline for the SMTP protocol (reduced version) as calculated by the tool Wendy.

Communication protocols have been studied and formalized long before the advent of service orientation [137, 25]. Such a formalization must on the one hand specify the protocol or *control flow* itself (i. e., the order in which messages are exchanged) and the underlying *communication model* (i. e., the way messages are transfered) on the other hand. Prominent control flow models are finite automata [85], Petri nets [163],

and process algebras [14]. As communication model, usually a choice is made between either synchronous or asynchronous message transfer.

We first justify our choice for an automaton-based model for the control flow. This choice is motivated by the correctness criteria that are studied in this thesis: compatibility, controllability, and realizability (cf. Chap. 7) are *behavioral* criteria defined in terms of states and runs of services and their composition rather than on their structure. Structural approaches, which avoid a state space exploration, are usually defined for special subclasses (e. g., soundness checks for free-choice Petri nets [180]), or allow only for the definition of either necessary *or* sufficient criteria (e. g., compatibility criteria derived from the state equation [148]). Furthermore, the algorithms to synthesize strategies and operating guidelines are based on states. To this end, we decided to use a formalism with an explicit notion of states rather than models with an implicit notion of states, such as Petri nets or process algebras. This decision also takes into account that none of the algorithms presented in this thesis currently exploits the ability of Petri nets and process algebras to explicitly express concurrency. Nevertheless, Petri nets can be later used to compactly represent service automata and operating guidelines [122].

Service automata are introduced as a uniform instrument to *reason* about correctness of services rather than to *model* services. To create models of services, domain-specific languages, such as BPMN or WS-BPEL, and graphical formalisms, such as Petri nets or MSCs, are far more accessible to domain experts. Such models can, however, be easily translated into service automata: Massuthe [128] presents a bidirectional translation between open nets and service automata, and there exists a variety of translations [27, 120, 119] of service description languages into Petri nets and other formalisms related to automata.

Kazhamiakin et al. [92] compare the expressiveness of different communication models with respect to their ability to detect errors in service compositions. They define a parametrized *state transition system with channels*. Depending on the parameters on numbers, sizes, and ordering abilities of the channels, they constitute a hierarchy of communication models and discuss the tradeoff between expressiveness and analysis performance.

The most restricted communication model is synchronous communication. It allows for simple models and efficient verification, but makes strong assumptions on the underlying infrastructure implementing the message exchange between the services. In particular, the whole message transfer is considered to be instantaneous. Formalisms using synchronous communications include service automata, *I/O automata* [124], *interface automata* [43], the "Roman Model" [21], and *message exchanging finite state automata* [17]. Synchronous communication is also common in interaction models, for instance *interaction Petri nets* [55]. Wolf [189] and Wolf [187] study Petri net models in which multiple synchronous events may occur simultaneously. This extension has an impact on compatibility and controllability, because a set of simultaneously occurring synchronous events can reach different states than an arbitrary interleaving of these

events. Due to increased verification complexity and little practical relevance, we decided not to extend our communication model this way.

A more general communication model decouples the sending and the receiving of a message, but still assumes that the order of sending messages implies an order in which these messages are received; that is, messages are not reordered during communication. This is typically modeled by FIFO queues. Decidability issues in the context of unbounded queues were studied with *communicating finite state machines* [26], and recent work employing FIFO queues usually assume a finite bound [70, 72], sometimes even fixed to the size of one [16, 22].

Finally, the most general communication model assumes unordered message buffers which can be modeled using multisets. Beside service automata, *concurrent automata* [8] and *open nets* [94, 93, 126, 164, 129] follow this approach in which no assumptions are made about the infrastructure other than messages not to get lost.

Bultan et al. [33] stress that the verification of asynchronous communication is more complex than synchronous communication. To this end, Fu et al. [72] examine under which conditions asynchronous communication can be safely abstracted to synchronous communication. They provide sufficient conditions which include the strong requirement that at most one message event is activated in every reachable state of a composition. We investigated the impact of communication models to controllability [113] and showed that small variations in the communication model (e. g., changing the message bound) can make controllable services uncontrollable, and vice versa.

To conclude, service automata support *both* synchronous and (unordered) asynchronous communication and hence cover the entire range of the communication model hierarchy [92]. The ability to mix synchronous and asynchronous communication (similar to [189, 32, 187]) allows us to faithfully represent and reason about service models at different levels of abstraction.

Part I

CORRECTNESS OF SERVICES

3 VALIDATION AND SELECTION

This chapter is based on results published in [117].

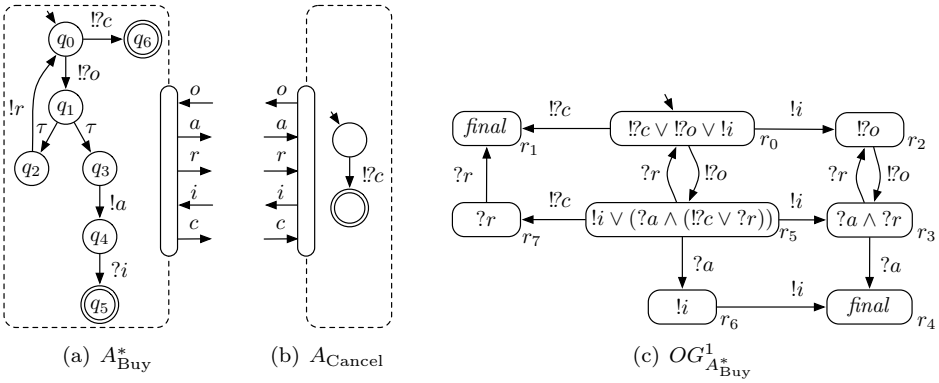
In this chapter, we investigate the set of strategies of a controllable service. Although each strategy models a correct interaction, not every strategy is *intended* in practice. We shall provide means to express intended and unintended behavior as *behavioral constraints*. With such constraints, the set of strategies can be "filtered", and the remaining strategies can be used in several applications from service validation to service discovery. In Sect. 3.2 and Sect. 3.3, we show how constraints can be applied to service automata and operating guidelines, respectively. First experiences with implementations of behavioral constraints are reported in Sect. 3.4. Finally, we discuss related work and give a conclusion.

We motivate, define, and discuss behavioral constraints in the context of service-oriented architectures (SOA). To explain the different scenarios, we distinguish a service provider with a service *Prov*, a service requestor with a service *Req*, and a service broker, which maintains a registry of several provider services (cf. Fig. 1.2). The definitions of this chapter are, however, independent of these roles and are applicable to any setting in which services communicate.

3.1 INTENDED AND UNINTENDED BEHAVIOR

In Chap. 2, we introduced the notion of controllability as a fundamental correctness criterion for services. A controllable provider service *Prov* is correct in the sense that there exists at least one strategy (i. e., a requestor service *Req*) such that their composition is compatible. With operating guidelines, the set of all strategies (i. e., all requestor services) can be characterized. In addition, compatible requestor service automata can be generated from this operating guideline.

In practice, the sole existence of an arbitrary strategy may be a too coarse correctness notion, because there usually exist *intended* and *unintended* strategies. Consider for example the buyer service from the previous chapter. After an update of its functionality, it might introduce the possibility to cancel (c) the negotiation at any time. Figure 3.1(a) depicts this updated buyer service and Fig. 3.1(c) shows the operating guideline of this service. To increase legibility, we refrained from drawing the empty node $q = \emptyset$. The operating guideline now also characterizes sellers that cancel after each step of the negotiation. These interactions with canceling sellers (cf. Fig. 3.1(b)) are still compatible. However, the owner of the buyer service is rather



(a) A^*_{Buy} (b) A_{Cancel} (c) $OG^1_{A^*_{\text{Buy}}}$

Figure 3.1: Adjusted buyer service (a) with operating guideline (c) and compatible seller service that always cancels (b).

interested whether it is still possible to actually buy goods. A filtered operating guideline that only characterizes selling — and hence intended — customers would be helpful in this setting.

Another evaluation of strategies may stem from the owner of a service registry: A service broker might classify provider services as intended or unintended. For example, he may want to ensure certain features for registered services, such as payment only with certain credit cards. Finally, a client requesting the registry might be interested in services implementing a certain protocol. For instance, he could prefer arranged communication such that certain actions occur in a given order (*first* accepting terms of payment and *then* sending a booking confirmation, for instance).

In the remainder of this chapter, we study *behavioral constraints* (constraints for short) that have to be satisfied in addition to compatibility. We provide a formal approach for steering the communication with a service *Prov* into a desired direction and also constrain operating guidelines. A constrained operating guideline of a service *Prov* characterizes all those services *Req* such that $Req \oplus Prov$ is compatible *and* satisfies a given constraint. Technically, a behavioral constraint expresses a criterion that is used to restrict the set $Strat(Prov)$ of strategies of the service *Prov*.

We identify four scenarios involving behavioral constraints.

1. *Validation.* Before deploying a service *Prov* or publishing it to a service registry, the designer wants to check whether an intended feature of that service can be used or whether an unintended communication scenario is excluded.

2. *Selection.* A service requestor queries the broker's registry for a provider service that matches with the requestor service *Req* and satisfies a given constraint.

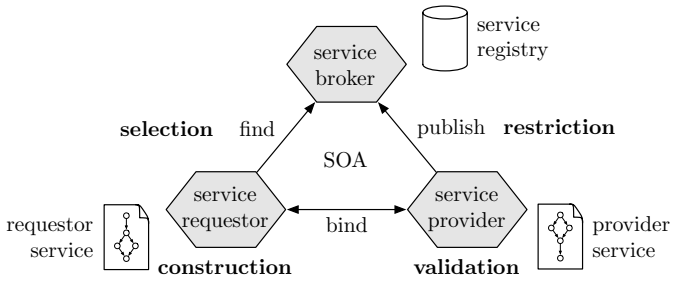


Figure 3.2: Scenarios of behavioral constraints in an SOA.

3. *Restriction.* A specialized registry might require a particular constraint to be satisfied by published services. To add a service *Prov* to this registry, its behavior might have to be restricted to satisfy the constraint.

4. *Construction.* A requester does not have a service yet, but expresses desired features as a constraint. The broker returns all operating guidelines of services providing these features. With this operational description, the requester service can then be constructed.

In the first two scenarios, the operational description — in this thesis given as a service automaton — of the service *Prov* is available. This has the advantage that constraints are not restricted to communication actions, but may involve particular (possibly internal) transitions of the service. That way, a service can, for instance, be customized to legal requirements (publish, for example, an operating guideline where only those strategies are characterized, for which the internal action "add added value tax" has been executed). In contrast, in the previous two scenarios, a constrained operating guideline is computed from a given operating guideline of *Prov*, without having access to an operational description of *Prov* itself. This setting is natural in case of a service registry, which does not store the services itself, but only information the external behavior of the services. As a consequence, only communication events can be constrained.

Figure 3.2 shows how these application scenarios of behavioral constraints can be assigned to the roles and operations of an SOA.

3.2 ADDING CONSTRAINTS TO SERVICE AUTOMATA

The goal of behavioral constraints is to enforce or to exclude certain behavior in the interaction of a service with its environment while maintaining compatiblity. Hence, behavioral constraints are a refinement of compatibility and its derived concept of controllability. One requirement of compatibility is that all maximal runs of a

closed composition terminate in a final state (cf. Def. 2.6). A behavioral constraint restricts these maximal runs by only considering a subset as terminating, whereas other maximal runs are treated as deadlocking. Thereby, a behavioral constraint also restricts the set of strategies of a service. At design time of a service, however, the set of strategies and hence the set of maximal runs of the compositions with the strategies are not known. To this end, we define behavioral constraints in terms of a given service and implicitly change the runs of a composition by explicitly changing transitions of the given service. We model behavioral constraints with *constraint automata*.

Definition 3.1 (Constraint automaton).
Let $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ be a service automaton. The tuple $C = [Q, q_0, \rightarrow, \Omega]$ is a *constraint automaton* for A, iff

1. Q is a finite set of states,
2. $q_0 \in Q$ is an initial state,
3. $\rightarrow \subseteq Q \times (2^{\rightarrow_A} \setminus \{\emptyset\}) \times Q$ is a transition relation, and
4. $\Omega \subseteq Q$ is a set of final states.

A constraint automaton for a service automaton A is a finite state automaton whose transitions are labeled with nonempty sets of transitions of A. Using these labels, a constraint automaton synchronizes with A. As for service automata, final states are used to model desired terminating states. The synchronization is defined as follows.

Definition 3.2 (Product with constraint automaton).
Let $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ be a service automaton and $C = [Q_C, q_{0_C}, \rightarrow_C, \Omega_C]$ a constraint automaton for A. The *product* of A and C is the service automaton $A \otimes C = [Q, q_0, \rightarrow, \Omega, \mathcal{P}_A]$ consisting of

- $Q := Q_A \times Q_C$,
- $q_0 := [q_{0_A}, q_{0_C}]$,
- $\Omega := \Omega_A \times \Omega_C$, and
- $\rightarrow$ containing exactly the following elements: $[q_A, q_C] \xrightarrow{e} [q'_A, q'_C]$ iff
  1. $q_A \xrightarrow{e}_A q'_A$, $q_C \xrightarrow{X}_C q'_C$, and $[q_A, e, q'_A] \in X$ or
  2. $q_A \xrightarrow{e}_A q'_A$, $q_C = q'_C$, and $[q_A, e, q'_A] \notin \bigcup_{q''_C \in Q_C} \{X \mid q_C \xrightarrow{X}_C q''_C\}$.

The product of a service automaton A and a constraint automaton C yields a service automaton with the same interface as A. A state of the product is a pair of a state of A and a state of C, and the product reaches a final state iff both A and C reach a final state. A state transition of A either occurs synchronized with a state transition of C if the former transition is part of the label of the latter transition. In case such synchronization is not possible, A changes its state without synchronization, leaving C in the same state; that is, C *stutters*.

Our product definition is similar to *stuttering synchronization* which is used, for instance, in LTL model checking. Esparza and Heljanko [67] introduced stuttering synchronization to avoid state space explosion by only synchronizing with "relevant" actions of a system. *Our motivation of stuttering is that the constraint automaton must not restrict the behavior of A, but only restricts its set of strategies.* In particular, the product must not disable transitions of A. This requirement was not stated explicitly in the original paper on behavioral constraints [117]. Wolf [187] gave a semantical definition of this *monitor property* in terms of the product of a constraint with a service automaton. In this thesis, we chose a stuttering synchronization to achieve this monitor property, because this type of synchronization changes the shape of A to express a particular constraint and also allows for the efficient analysis of constrained services: Section 3.4 is devoted to implementation details.

As a result, the product of a service automaton with a constraint automaton restricts the set of strategies.

Lemma 3.1 (Product constrains the set of strategies).
Let A be a service automaton and C a constraint automaton for A.
Then $Strat_k(A \otimes C) \subseteq Strat_k(A)$.

Proof. Follows directly from Def. 3.1 and Def. 3.2. □

In a finite-state compatible composition of two services A and B, the set of terminating runs forms a regular language. A constraint automaton C for A specifies a regular language over transitions of A. In the composition $(A \otimes C) \oplus B$, these regular languages are synchronized, yielding a subset of terminating runs. Regular languages allow to express a variety of relevant scenarios, including:

- enforcement of events (e. g., to consider only those strategies in which a delivery notification is sent),
- exclusion of events (e. g., to exclude those strategies in which an error message is received),
- ordering constraints (e. g., to focus on those strategies in which an invoice is never sent *before* a shipping confirmation was received), and
- numbering constraints (e. g., to check whether there exists a strategy that can order an item by sending less than two login messages).

Furthermore, any combinations are possible, allowing to express complex behavioral constraints.

The presented approach is, however, not applicable to nonregular languages. For instance, a constraint requiring that a terminating run must have an equal number

of a and b events or that a and b events must be properly balanced (Dyck languages) cannot be expressed with a finite-state constraint automaton. Hence, $(A \otimes C) \oplus B$ could not be expressed as finite state service automaton. Similarly, constraints that affect infinite runs (e. g., certain LTL formulae [125]) cannot be expressed.

In the remainder of this section, we describe the first two applications of behavioral constraints and how they can support the service provider to validate and restrict his service *Prov*.

FIRST APPLICATION SCENARIO: VALIDATION

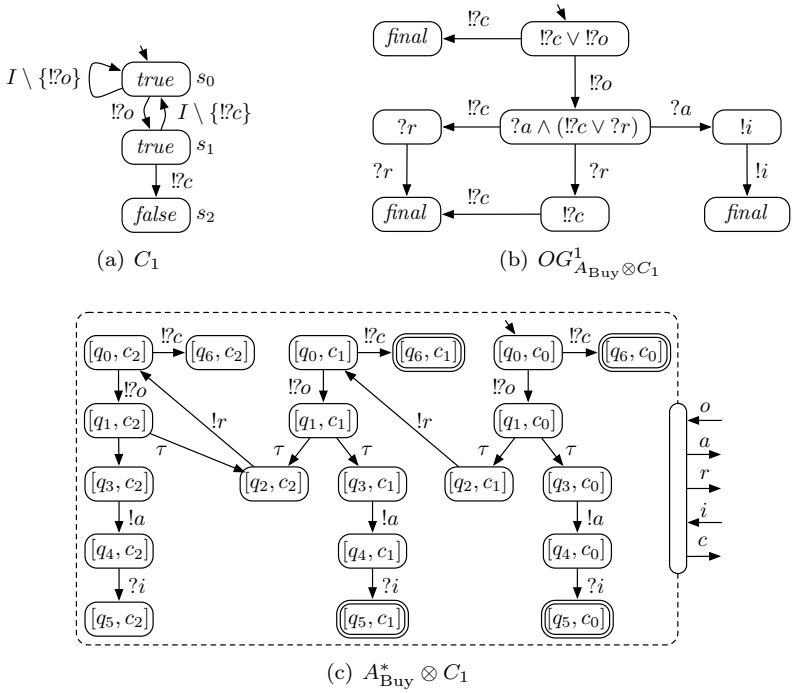
If both services *Req* and *Prov* are given, the satisfaction of a behavioral constraint (i. e., the presence or absence of certain behavior) can be verified on the composition $Req \oplus Prov$ using standard model checking techniques [39]. However — coming back to the scenarios described in the introduction — when a service provider wants to validate his service *Prov* at design time, there is no fixed requestor service *Req*.

In the validation scenario, a service provider wants to make sure that for all strategies *Req* of *Prov* the composition $Req \oplus Prov$ satisfies certain constraints. An example would be that payments will always be made, or that no errors occur. We suggest to describe the constraint as a constraint automaton C. Then, we can analyze the product $Prov \otimes C$ of *Prov* and C. The operating guideline of this product characterizes all strategies *Req* for *Prov* such that $Req \oplus Prov$ satisfies C. The benefit of this approach is that, instead of calculating all strategies *Req* and checking whether $Req \oplus Prov$ satisfies the constraint C, it is possible to characterize *all* C-satisfying strategies *Req*. To this end, we can use the same algorithm to calculate the operating guidelines, because the product is a regular service automaton.

Formally, the validation scenario is as follows: given the provider service *Prov* and a constraint automaton C, check if $Strat(Prov \otimes C) \neq \emptyset$.

SECOND APPLICATION SCENARIO: SELECTION

In the selection scenario, we assume that the service registry already contains several provider services. The requestor queries this service registry to find a provider service *Prov* that matches with his service *Req* and additionally satisfies a given constraint. Similar to the validation scenario, the service requestor is not interested in checking for each matching provider service *Prov* whether $Req \oplus Prov$ satisfies this constraint. We assume that the constraint is given as constraint automaton C. Now, the requestor can calculate the product $Req \otimes C$ and use this product to query the registry for matching services. That way, the consideration of constraints refines the "find" operation of an SOA: Instead of returning *any* provider service *Prov* such that the composition with a requester service *Req* is compatible, only the subset of providers *Prov* for which $Req \oplus Prov$ satisfies the constraint C is returned. Formally, the selection scenario is considering the question whether $(Req \otimes C) \in Strat(Prov)$.



(a) C_1 (b) $OG^1_{A_{\text{Buy}} \otimes C_1}$

(c) $A^*_{\text{Buy}} \otimes C_1$

Figure 3.3: A constraint automaton expressing that at most two offers are rejected (a), the product of this constraint and the modified buyer service (c), and an operating guideline of this product (b).

EXAMPLE. Consider again the updated buyer service in Fig. 3.1. Assume that the provider is only interested in interactions with sellers that reject at most two offers. He can formulate this requirement in a behavioral constraint. Figure 3.3(a) depicts a constraint automaton, which expresses that at most two offers are rejected. Figure 3.3(c) depicts the constrained buyer service. This product also contains two deadlocks, namely $[q_6, c_2]$ and $[q_5, c_2]$.

3.3 ADDING CONSTRAINTS TO OPERATING GUIDELINES

The previous section was devoted to support the service provider to validate his service and the service requestor to query a service registry. The desired behavioral restriction was formulated as constraint automaton. In both scenarios, an operational description of the service (i. e., a service automaton) was available.

In case such an operational description is not accessible, constraint automata cannot be used any more. This excludes the service broker who usually has no access to an operational model, but rather stores service descriptions, such as operating guidelines. However, the service broker plays a central role in the SOA paradigm, comparable to a search engine in the World Wide Web. Thus, the question arises whether it is still possible to satisfy a given constraint *after* having published the service *Prov*; that is, only an operating guideline is accessible. In this section, we extend our operating guideline approach to this regard. We show that it is possible to describe a constraint as an annotated automaton C^φ, called *constraint-annotated automaton*, and apply it by building the product of C^φ and the operating guideline OG_{Prov}. The resulting constrained operating guideline guideline $C^\varphi \otimes OG_{Prov}$ shall describe the set of all requester services *Req* such that $Req \oplus Prov$ satisfies the constraint.

An advantage of this setting is that we do not need the original service automaton model of *Prov*, but can apply constraints directly to the operating guideline OG_{Prov}. This operating guideline contains no trade secrets and is assumed to be public to the service broker. A drawback, however, is that for the same reason we are not able to enforce, exclude, or order concrete transitions of the service automaton any more: C^φ may only constrain send or receive actions as such. For example, if two or more transitions send a message a, then a constraint C^φ excluding a means that all the original transitions are excluded.

A constraint-annotated automaton for a service automaton A is an annotated automaton with the same interface as A.

Definition 3.3 (Constraint-annotated automaton).
Let A be a single-port service automaton. An annotated automaton C^φ is a *constraint-annotated automaton for* A iff A and C^φ have the same interface.

Both the operating guideline to be constrained and the constraint-annotated automaton characterize a set of matching services with the same interface. To apply the constraint to the operating guideline, we again synchronize the automata and construct a product.

Definition 3.4 (Product of annotated automata).
The product of two annotated automata A^φ and B^ψ with the same interface $\mathcal{P}$ is the annotated automaton $A^\varphi \otimes B^\psi = \big[[Q, q_0, \rightarrow, \mathcal{P}], \zeta\big]$ consisting of:

- $Q := Q_A \times Q_B$,
- $q_0 := [q_{0_A}, q_{0_B}]$,
- $[q_A, q_B] \xrightarrow{e} [q'_A, q'_B]$ iff $q_A \xrightarrow{e}_A q'_A$ and $q_B \xrightarrow{e}_B q'_B$, and
- $\zeta([q_A, q_B]) := \varphi(q_A) \wedge \psi(q_B)$.

Structurally, the previous definition is a standard product operation of finite automata which is used to describe the intersection of regular languages [85]. We can observe the following relation between two services and their product.

Corollary 3.1 (Services simulate their product).
Let A^φ and B^ψ be annotated automata and $A^\varphi \otimes B^\psi$ their product.
Then A^φ simulates $A^\varphi \otimes B^\psi$ and B^ψ simulates $A^\varphi \otimes B^\psi$.

Proof. The existence of the simulation relations $\varrho_{(A^\varphi \otimes B^\psi, A^\varphi)}$ and $\varrho_{(A^\varphi \otimes B^\psi, B^\psi)}$ follows directly from Def. 3.4. In particular, for any reachable state $[q_A, q_B]$ of $A^\varphi \otimes B^\psi$ we have $[[q_A, q_B], q_A] \in \varrho_{(A^\varphi \otimes B^\psi, A^\varphi)}$ and $[[q_A, q_B], q_B] \in \varrho_{(A^\varphi \otimes B^\psi, B^\psi)}$. □

In addition, Def. 3.4 also considers the annotated formulae. These formulae are conjuncted, which yields an intersection of the characterized services:

Lemma 3.2 (Product yields intersection).
Let A^φ and B^ψ be annotated automata.
Then $Match(A^\varphi \otimes B^\psi) = Match(A^\varphi) \cap Match(B^\psi)$.

Proof. We prove the lemma by showing that $S \in Match(A^\varphi \otimes B^\psi)$ iff $S \in Match(A^\varphi)$ and $S \in Match(B^\psi)$.

($\Rightarrow$) By assumption $S \in Match(A^\varphi \otimes B^\psi)$, so there exists a structural matching relation $\varrho_{(S, A^\varphi \otimes B^\psi)}$. By Cor. 3.1, there exists a simulation relation $\varrho_{(A^\varphi \otimes B^\psi, A^\varphi)}$. We define the relation $\varrho_{(S,A^\varphi)} \subseteq Q_S \times Q_{A^\varphi}$ as follows: $[q_S, q_A] \in \varrho_{(S,A^\varphi)}$ iff $[q_S, [q_A, q_B]] \in \varrho_{(S, A^\varphi \otimes B^\psi)}$ and $[[q_A, q_B], q_A] \in \varrho_{(A^\varphi \otimes B^\psi, A^\varphi)}$. The relation $\varrho_{(S,A^\varphi)}$ is a structural matching relation between S and A^φ.

Let $[q_S, q_A] \in \varrho_{(S,A^\varphi)}$ be arbitrary. By assumption, $q_S \models \varphi(q_A) \wedge \psi(q_B)$. Hence $q_S \models \varphi(q_A)$ and $S \in Match(A^\varphi)$. The arguments for $S \in Match(B^\psi)$ are analogous.

($\Leftarrow$) Let $S \in Match(A^\varphi)$ and $S \in Match(B^\psi)$. Let q_S be an arbitrary state of S, and let q_A and q_B be corresponding states with $[q_S, q_A] \in \varrho_{(S,A)}$ and $[q_S, q_B] \in \varrho_{(S,B)}$, respectively. Then, the state $[q_A, q_B]$ is reachable in $A^\varphi \otimes B^\psi$ and $[q_S, [q_A, q_B]] \in \varrho_{(S, A^\varphi \otimes B^\psi)}$. Hence, S matches with $A^\varphi \otimes B^\psi$. Finally, as the assignment $\beta(q_S)$ satisfies the annotation $\varphi(q_A)$ and the annotation $\psi(q_B)$ of matching states in A or S, $\beta(q_S)$ satisfies their conjunction $\varphi(q_A) \wedge \psi(q_B)$ as well.

□

Lemma 3.2 allows us to restrict the set of strategies of a provider service that do not satisfy a given constraint by calculating a product: The set $Match(OG_{Prov}) \cap$

$Match(C^{\varphi})$ is characterized by $OG_{Prov \otimes C^{\varphi}}$. With this result, we are able to realize the last two scenarios described in the introduction of this section. As already seen in our example, in these scenarios the constraint is modeled as a constraint-annotated automaton C^{φ}. This constraint characterizes the set of accepted behaviors and can be formulated without knowing the structure of the operating guideline needed later on. Only the interface (i. e., the set of input and output message channels of the corresponding service automaton) must be known.

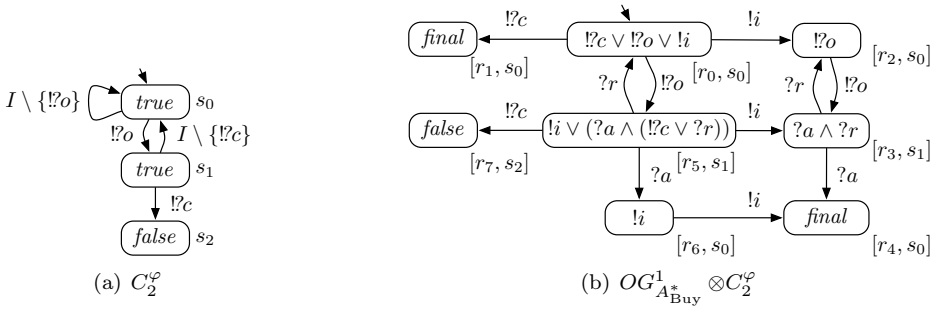
THIRD APPLICATION SCENARIO: RESTRICTION

In this scenario, the service broker wants to ensure that certain constraints are satisfied by the services in his repository. We assume that the service provider formulates his requirements as a constraint-annotated automaton C^{φ}. For each operating guideline stored in the service registry, the service broker can now calculate the product of this operating guideline and the constraint. That is, the restriction scenario can be formalized as considering $Match(OG_{Prov} \otimes C^{\varphi})$. In case the resulting operating guidelines characterizes a nonempty set of strategies, the constraint is satisfiable. Otherwise, the operating guideline can be removed from the registry; Massuthe [128] provides an algorithm to check whether an operating guideline characterizes a nonempty set of strategies. For new provider services to be registered, the service broker has the choice to either calculate the product himself or to publish his constraint. In the latter case, the service provider applies the constraint and publishes $OG_{Prov \otimes C}$ in the service registry.

The service *Prov*, however, can remain unchanged. This is an advantage as — instead of adjusting, reimplementing, and maintaining several versions of *Prov* for each registry and constraint — only a single service *Prov* has to be deployed. From this service the constrained operating guidelines are constructed and published. If, for example, *Prov* supports credit card payment and cash on delivery, then only the strategies using credit card payments would be published to the registry mentioned before. Although there exist strategies *Req* for *Prov* using cash on delivery, those requesters would not match with the published operating guideline.

FOURTH APPLICATION SCENARIO: CONSTRUCTION

In the fourth scenario, the requester service *Req* is yet to be constructed. Therefore, the desired features of *Req* are described as a constraint-annotated automaton. For example, consider a requester who wants to book a flight paying with credit card. If these features are expressed as a constraint automaton C^{φ}, it can be sent to the broker who may return operating guidelines of all provider services *Prov* offering these features (i. e., where the product of OG_{Prov} with C^{φ} is not empty). From



(a) C_2^{φ} (b) $OG^1_{A^*_{\text{Buy}}} \otimes C_2^{\varphi}$

Figure 3.4: A constraint-annotated automaton (a) expressing behavior that excludes offers that are immediately canceled and the product with the operating guideline of the buyer service (b).

such an operational descriptions, the service *Req* can easily be constructed. Formally, $Req \in Match(OG_{Prov} \otimes C^{\varphi})$.

An important aspect of this construction scenario is that the constraint does not need to explicitly specify intermediate steps. This allows the requestor to coarsely describe his desired goals (e. g., receive a plane ticket and pay with credit card) without caring about other protocol steps (e. g., logging in or confirming the terms of payment). These intermediate steps can be specified as "wildcards" in the constraint.

EXAMPLE. In a restriction scenario, a service broker might want to exclude those services that allow to place offers and immediately cancel the negotiation afterward. Figure 3.4 depicts a constraint-annotated automaton characterizing all strategies in which an order (o) is never directly followed by a cancellation (c). In the figure, edges annotated with sets are a shortcut notation for several edges, each labeled with a single element of the set. Such annotations, for instance $I \setminus \{!?c\}$, can be seen as wildcards that match any label but $!?c$.

3.4 IMPLEMENTATION AND EXPERIMENTAL RESULTS

The product operations on service automata and operating guidelines presented in this chapter have been prototypically implemented.

A constraint automaton usually introduces deadlocks, as for instance state $[q_6, c_2]$ in Fig. 3.3(c). Consequently, a maximal terminating run in $A \oplus B$ might reach a deadlock in $(A \otimes C) \oplus B$. The tool Wendy [121], which synthesizes strategies and calculates operating guidelines, also implements *early deadlock detection*. It analyzes the state space of a given open net (which coincides with a service automaton) and

Table 3.1: Experimental results for the validation scenario using Wendy.

	analyzed service automaton			synthesis result		
constraint	$\|Q_{\otimes}\|$	$\|\rightarrow_{\otimes}\|$	deadlocks	$\|Q_{TS}\|$	$\|\rightarrow_{TS}\|$	time (sec)
no constraint	8,345	34,941	0	20,818	144,940	29
numbering constraint	26,667	110,064	102	1,972	11,686	7
enforcement constraint	15,531	66,625	37	23,164	156,796	36
exclusion constraint	20,531	85,053	125	22,880	155,390	36
ordering constraint	9,110	37,616	24	20,786	144,796	29

marks states from which a deadlock will eventually be reached. If such an "inevitable deadlock" is reached during the strategy synthesis, the algorithm does not generate successor states, because the current state will eventually deadlock and hence will not be part of a strategy. This dramatically prunes the state space and still synthesizes most-permissive strategies and operating guidelines. Therefore, an increased size of the product does not necessarily result in longer runtime of subsequent strategy synthesis or the calculation of the operating guidelines.

Table 3.1 lists experimental results for the validation scenario. We applied several behavioral constraints to a service automaton model ("SMTP protocol" in Tab. 2.1) translated from a WS-BPEL process. For the different constraints, the size of the product (columns "$|Q_{\otimes}|$" and "$|\rightarrow_{\otimes}|$") is up to three times larger than the original service. At the same time, the runtime of the synthesis of a most-permissive strategy hardly increases, because of the early detection of the deadlocks that are introduced by the product. We refer the interested reader to [121].

The calculation of the product of two annotated automata has been implemented in the tool Fiona [131, 128]. First, the product of the underlying service automata is built by performing a coordinated depth-first search. This search avoids the calculation of unreachable states. In case one annotated automaton is an operating guideline (as motivated in the third and fourth scenario), this product calculation is very efficient, because operating guidelines are deterministic by construction. During this calculation, also the product's states are annotated with the conjunction of the individual service's formulae. In a final step, each state with an unsatisfiable formulae (e. g., resulting a conjunction with *false*) is deleted together with its adjacent arcs. This is repeated until a fixed point is reached. While this pruning of the constrained operating guideline does not change the characterized set of strategies, it may dramatically reduce the size of the underlying service automaton and thereby speed up subsequent matching.

3.5 DISCUSSION AND RELATED WORK

In the area of model checking, it is a common technique to specify desired or undesired behavior (e. g., traces that satisfy or violate a temporal logic formulae) using automata (e. g., Büchi automata in case of LTL) and to calculate the intersection of the actual and the desired behavior using the product of this automaton and the system to check. Therefore, the presented approach to use behavioral constraints to refine the set of strategies of a service is related to several approaches in the area of computer-aided verification.

SUPERVISORY CONTROL, MODULE CHECKING, ATL In these problem instances, an open system with controllable and uncontrollable actions as well as a formula (LTL or CTL) are given. Supervisory control [160, 161] asks whether an environment *exists* which controls the controllable actions such that the system satisfies the given formula. Module checking [98, 99, 97] checks whether the system satisfies the formula in *all possible* environments. In this setting, deadlock-freedom is a prerequisite for the composition with the environments. That is, supervisory control quantifies the environment existentially and module checking quantifies the environment universally. Alternating-time temporal logic (ATL) [10] allows to selectively quantify the environment. This approach is closest to our approach to use an operating guideline to characterize the set of all environments (i. e., strategies) such that the composition satisfies a given constraint. Admittedly, we do not consider classical temporal logics, but only simple regular constraints. However, with operating guidelines we are able to characterize all constraint-satisfying strategies — a concept that is not yet known in the field of ATL or LTL synthesis [158].

MODEL CHECKING The idea to constrain the behavior of a system by composing it with an automaton is also used in the area of model checking. When a component of a distributed system is analyzed in isolation, it might reach states that are unreachable in the original (composed) system. To avoid these states, Graf and Steffen [75] introduce an *interface specification* to constrain the global communication behavior, which is composed to the considered component and mimics the interface behavior of the original system. Valmari [178] adds *cut states* to the interface specification, which are not allowed to be reached in the composition. These states are similar to deadlocks in a constraint automaton (cf. state c_2 in Fig. 3.3(a)) or states of a constraint-annotated automaton with annotation *false* (cf. Fig. 3.4(a)).

SERVICES There is a lot of research being done to enforce constraints in services. The originality of behavioral constraints as presented in this chapter lies in the application of constraints to the communication between a requester and a provider service (see Fig. 3.2). Furthermore, the presented model of constraints allows us to refine "find" operation in an SOA.

Davulcu et al. [42] describe services with a logic, allowing the enforcement of constraints by logical composition of a service specification with a constraint specification. Similarly, several protocol operators, including an intersection operator are introduced by Benatallah et al. [20]. Although these approaches only consider synchronous communication, they are similar to our product definition (cf. Def. 3.2)

An approach to describe services and desired (functional or nonfunctional) requirements by *symbolic labeled transition systems* is proposed by Pathak et al. [156]. An algorithm then selects services such that their composition satisfies the given requirements. However, the requirements have to be very specific; that is, the behavior of the desired service has to be specified in detail. In our presented approach, the desired behavior can be described by a constraint instead of a specific workflow. However, the discovery of a composition of several services that satisfies a required constraint is subject of future work. Other approaches presented by Berardi et al. [23] and recently by De Giacomo and Patrizi [44] assume a specification of a *target service* which is then realized by composing available services from a registry. Again, this approach is based on synchronous communication. Furthermore, it requires the target service to be completely specified, including all intermediate steps. In contrast, the construction approach of Sect. 3.3 does not require a complete specification, but services can also be discovered using a partial specification.

OPERATING GUIDELINES Both constraint automata and constraint-annotated automata allow to specify the enforcement of desired behavior and the exclusion of undesired behavior. These constraints are implicitly universally quantified. That is, a constraint requires a certain behavior to occur in *all* terminating runs or in *no* terminating run. Such constraints cannot express existential quantification. For instance, a requirement that it should be *possible* to receive a certain message cannot be specified. Stahl and Wolf [172] fill this gap by introducing *cover constraints*. These constraints can only be expressed by *extended operating guidelines*, which require a global formula in addition to the formulae that are annotated to each state.

In this thesis, we already showed how set inclusion (cf. Def. 2.13) and intersection (cf. Lem. 3.2) can be expressed in terms of operating guidelines. To define a union operation or negation, Kaschner and Wolf [89] present another extension of operating guidelines with a global formula, which allows to implement a complete set algebra on operating guidelines. While these extensions increase the complexity of the set operations, especially the possibility to join sets of strategies allows to speed up the "find" operation of an SOA.

Other reasons to discard strategies might stem from the semantics of messages and causalities between messages. These aspects go beyond the protocol level. For instance, a message modeling an acknowledgment might be sent by a participant *before* actually having received a request. While such an interaction might still be compatible, it is not realizable in practice. To this end, Wolf [186, 187] shows how the strategy synthesis can be adjusted to respect semantics or causalities of messages.

3.6 CONCLUSION

In this chapter, we introduced behavioral constraints as means to restrict the set of strategies to enforce or to exclude desired and undesired behavior, respectively. Behavioral constraints can be either applied to service automata or to operating guidelines. This flexibility makes behavioral constraints a valuable tool in different scenarios of an SOA.

These different applications of behavioral constraints contribute to the topic of this thesis — correctness of services and their composition — as follows.

- The validation scenario allows to check a service at design time. The satisfaction of a behavioral constraint can be checked with respect to *any* possible communication partner of the given service. This allows to detect unintended strategies well before implementing, deploying, and publishing the service.
- In the selection and restriction scenarios, the focus lies on correctness by construction. The composition with any service that is returned by the service broker is not only compatible, but also satisfies a given constraint. The construction scenario further supports the design of new services by declaratively querying the service registry for desired behavior.

We deliberately restricted the expressiveness of the behavioral constraints to regular languages. As discussed in the previous section, covering constraints or properties of infinite runs cannot be expressed. First results show that an increased expressiveness of constraints also yields in more complex characterizations of the set of strategies of a service. To this end, we decided to make the application of behavioral constraints transparent to the concept of operating guidelines [172, 89]. As a consequence, existing tools and algorithms remain applicable. With the aforementioned translations [116, 110] from WS-BPEL to service automata, behavioral constraints can be applied to industrial service description languages. First case studies showed that there are hardly any runtime penalties when considering constraints while constructing a service's operating guideline.

We consider an extension of the construction scenario as a promising direction for future work. With the presented techniques, the service registry can be queried for services that satisfy a given constraint. If the constraint models complex behavior (e. g., reserving a hotel and booking a flight), it might not be satisfied by a single service. Instead, several simpler constraints could be formulated, which return several services which need to be orchestrated to achieve the composite behavior. The automatic construction of such an orchestrator could greatly facilitate the construction of new requestor services while improving the reuse of provider services.

4 DIAGNOSIS

This chapter is based on results published in [112].

We introduced controllability as a fundamental correctness criterion for interacting service models. In the previous chapter, we presented behavioral constraints as a means to restrict the set of strategies to refine the analysis of a service. Controllability and the satisfaction of behavioral constraints can be automatically decided. The decision algorithm (cf. Def. 2.9) is constructive: If a strategy for a service exists, it can be synthesized and serves as a witness for controllability. If, however, the service is uncontrollable, no strategy exists and the algorithm neither returns a service nor any diagnosis information. In this chapter, we introduce a *diagnosis framework for uncontrollable services*. In the next section, we present the various reasons which may make a service uncontrollable. In Sect. 4.2 and Sect. 4.3, we informally sketch how counterexamples for controllability (or witnesses for uncontrollability) may be presented to service modelers. Section 4.4 is devoted to a formalization of the problem. The diagnosis algorithm is finally defined in Sect. 4.5 where we also discuss its implementation. Section 4.6 concludes the chapter.

4.1 REASONS FOR UNCONTROLLABILITY

The presence of strategies (i. e., clients, partner services, requestors, customers, etc.) is crucial for a service. To this end, controllability is a fundamental sanity check for services, and any other (behavioral) correctness criterion (e. g., stronger notions which also require the absence of livelocks in the composition) would likely further refine the set of strategies of a service. Controllability is defined as an extension of compatibility to open services, and we shall consider the requirements for compatibility when we reason about uncontrollability. A service is uncontrollable if there does not exist a composable service such that

1. every maximal run of the composition terminates in a final state,
2. the asynchronous message channels are bounded, and
3. the composition is responsive (i. e., no port is excluded from communication on infinite runs).

An uncontrollable service has no strategy. Hence, we cannot analyze a concrete composition for the reasons which led to incompatible behavior. Therefore, we need

to explain the absence of strategies by considering the service itself. In particular, we have to investigate the service's share of the incompatibility of the composition with *any other* service. In the remainder of this section, we give examples how errors and design flaws of a single service can result in uncontrollability. We group these issues according to the three preceding requirements.

4.1.1 DEADLOCKING RUN

The algorithm suggested by Def. 2.9 removes all nodes which contain a deadlocking state; that is, a state which is neither final nor has a successor state in the composition of the service and the strategy overapproximation. Thereby, a state $[q, \mathcal{B}]$ has two components: state q representing the internal state of the service and a multiset $\mathcal{B}$ modeling the pending asynchronous messages. These components help classify deadlocks.

INTERNAL DEADLOCK. First, the state q of the service may be a deadlock itself; that is, q is a nonfinal state without successors. We call such a state an *internal deadlock*, because this deadlock is independent of a communication event. There are different reasons why a service may contain an internal deadlock:

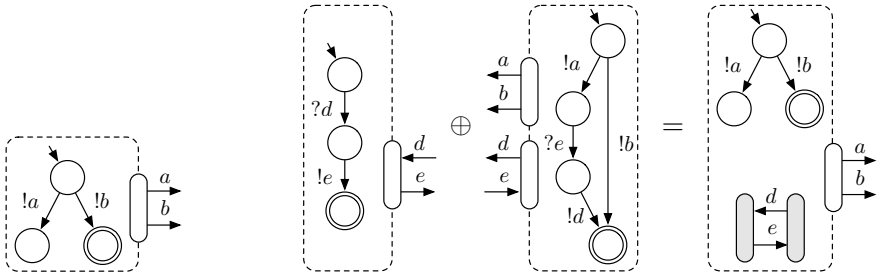
– *Design flaw.* An obvious reason for an internal deadlock is a classical design flaw. Although languages, such as WS-BPEL, have syntactical requirements to avoid modeling potential deadlocks, in graph-based languages, such as BPMN, it is possible to introduce deadlocks, for instance because of mismatching gateways. Such design flaws affect the control flow of a service and can be detected without taking the interaction into account. Classical control flow-oriented correctness notions such as soundness [1] are, however, neither sufficient nor necessary for controllability of a service. We shall discuss this in Sect. 4.2.

  Figure 4.1(a) shows a service automaton, which contains an internal deadlock. The service nondeterministically decides whether to send an a-message or a b-message. The environment can only observe, but not influence this decision. As the deadlock cannot be avoided, the service is uncontrollable.

– *Service choreography.* Not every internal deadlock is the result of a modeling error. Another source of internal deadlocks can be the composition of several services in a service choreography. There, it is possible that the behavior of two participants is mutually exclusive leading to an internal deadlock.

  Figure 4.1(b) depicts two services whose composition has the same behavior as the service automaton in Fig. 4.1(a). The internal deadlock occurs, because the left service waits for a d-message and the right service waits for an e-message.

– *Behavioral constraint.* Another reason for internal deadlocks of a service is the consideration of a behavioral constraint C, cf. Chap. 3. In particular, final states



(a) design flaw (b) deadlocking composition

Figure 4.1: Uncontrollability caused by internal deadlocks.

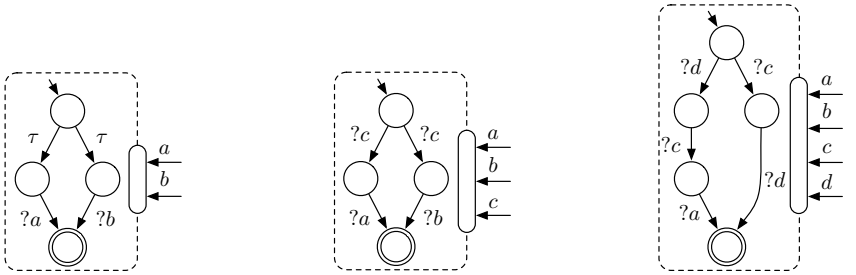
of A may become internal deadlocks in the product $A \otimes C$. These deadlocks are not design flaws, but model undesired situations. This may render a service uncontrollable as it may be impossible to satisfy the constraint.

Figure 3.3(c) depicts a service automaton which contains the deadlocks $[q_5, c_2]$ and $[q_6, c_2]$ which were introduced by the constraint automaton depicted in Fig. 3.3(a). However, this service automaton is still controllable, because the deadlocks can be circumvented by the environment.

COVERED FINAL STATE. A *covered final state* is a situation in which the control flow of A reached a final state without successor state, but an asynchronous message sent to A is still pending on an input channel. This message will never be received from the service. This may be negligible for generic acknowledgment messages, but an unreceived message is typically an undesired situation (e. g., if the message contains private or payment information). In addition, unexpected messages may lead to runtime errors during the execution of a WS-BPEL process. Again, there are many reasons for this problem:

– *Hidden choice.* In case services implement business processes, data-dependent decisions (e. g., WS-BPEL's `<if>` activity or a data-dependent gateway in BPMN) are common. Such a decision may be taken without explicitly informing the communication partner about the outcome. If this *hidden choice* requires different reactions of the partner (i. e., the partner needs to send different messages), it cannot be guaranteed that each of these messages are received.

  Consider for example the service automaton in Fig. 4.2(a), which nondeterministically chooses the left or the right branch. Depending on this internal choice, a partner has to send either an a-message or a b-message. The final marking is only reached, if the partner's "guess" was right. Otherwise, the "wrong" message keeps pending.



(a) hidden choice (b) conflicting receives (c) delayed messages

Figure 4.2: Uncontrollability caused by covered final states.

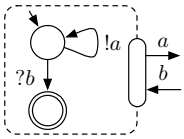
– *Conflicting receives.* If a service can reach a state in which more than one transition can receive the same message from an asynchronous channel or synchronize with the same event, these transitions are *conflicting receives* [11]. The decision which branch to take, can neither be influenced nor observed by a partner yielding a hidden choice situation. Execution languages like WS-BPEL treat conflicting receives as runtime faults, but similar to internal deadlocks, we do not want to forbid such situations in the first place. Instead, we want to investigate whether these problems are the original reason a service is uncontrollable.

  The initial state of the service automaton in Fig. 4.2(b) models a conflicting receive situation. After sending a c-message, a partner has to send either an a-message or a b-message to the service. If the wrong choice is made, the message keeps pending on the input channel. This eventually yields a covered final state.

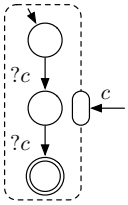
– *Delayed messages.* Service automata support asynchronous message exchange: messages can keep pending on a channel and overtake one other. Therefore, a partner has only limited control over a service, because after sending a message, a partner cannot observe whether this message was already received or whether it is still pending on the channel. Again, this can result in a "hidden choice" situation.

  An example is given in Fig. 4.2(c). The order in which the c-message and the d-message are sent to the service does not determine the order in which these messages are received and, consequently, which branch is taken. However, an a-message is only received if the left branch is taken and remains pending otherwise.

Covered final states can be seen as a "visible symptom" of uncontrollability rather than an original fault. For instance, there can be an arbitrary number of transitions leading from a hidden choice to covered final state. This makes the detection of the reasons which actually led to uncontrollability nontrivial.



(a) unbounded channel



(b) channel bound $k > 1$

Figure 4.3: Uncontrollability caused by message bound excess ($k = 1$).

4.1.2 EXCEEDED MESSAGE BOUND

Beside the requirement that the message channels must be empty in a final state, compatibility demands that the message channels never exceed a given bound k. Consequently, also this message bound k influences in Def. 2.9 the removal of states of $TS^0(A)$ when constructing $TS^1_k(A)$. There are two situations to consider:

UNBOUNDED COMMUNICATION. If a message channel is unbounded (e. g., caused by a loop of the service in which it sends messages without waiting for acknowledgements), then obviously no partner can exist such that the composition is k-bounded. Figure 4.3(a) shows an example where the output channel a is unbounded. Even if the environment sends a b-message to this service, its receipt can be postponed arbitrarily.

INADEQUATE MESSAGE BOUND. If a service is k-controllable for a message bound $k \in \mathbb{N}^+$, it is also l-controllable for any bound $l > k$. The converse does not hold: Figure 4.3(b) shows a service which is 2-controllable, but not 1-controllable, because the receipt of the first c-message cannot be enforced before sending a second c-message. This results in a state where two c-messages are pending and the message bound is violated. Thus, even if a message bound exists for a service, this service may be considered k-uncontrollable if the message bound k chosen for analysis is too small. Again, we do not want to rely on the underlying infrastructure, which may enforce a message bound by discarding messages, but to treat exceeded message bounds as a design flaw we want to diagnose. Note that the message bound can be violated for output message channels (cf. Fig. 4.3(a)) and input message channels (cf. Fig. 4.3(b)).

4.1.3 UNRESPONSIVENESS

Definition 2.9 was only defined for responsive service automata. Similar to internal deadlocks, unresponsive behavior does not necessarily result in uncontrollability. Instead of restricting diagnosis to responsive services, it should be investigated whether it is the original reason of uncontrollability of a service.

(a) internal livelock (b) unresponsive communication

Figure 4.4: Uncontrollability caused by unresponsive behavior.

INTERNAL LIVELOCK. A service can make a service composition unresponsive if it continuously changes its state without interaction with all ports environment and without reaching a final state. Such diverging behavior is an *internal livelock* and can be checked locally similar to internal deadlocks. Figure 4.4(a) shows an example. An internal livelock can also model the closed communication between two implemented ports. Consider the service automaton Fig. 4.4(b). Once this service sends an a-message, the port $[\emptyset, \{a, b\}]$ is excluded from further communication.

The issues presented in this section are the original problems which can make a service uncontrollable. Deadlocks and message bound violations — Def. 2.9 only takes responsive service automata into account — yield to the deletion of such states. This deletion can introduce other deadlocks. These states usually give no further information on the original reasons which make a service uncontrollable. To this end, we focus on the detection of internal deadlocks, covered final states, and message bound violations. In addition, we have to consider unresponsive services; that is, we need to detect internal livelocks as reasons for uncontrollability.

4.2 COUNTEREXAMPLES FOR CONTROLLABILITY

As motivated the synthesis algorithm gives no information on the reasons which make a service uncontrollable. Before we elaborate on how diagnosis information could be presented, we study a related diagnosis approach.

RELATIONSHIP TO SOUNDNESS

Controllability of a service model has a close relationship to soundness in the area of workflow models [1]. However, existing diagnosis techniques for unsound workflow models [180] are not applicable to diagnose uncontrollability, because the service's interaction with the environment has to be taken into account.

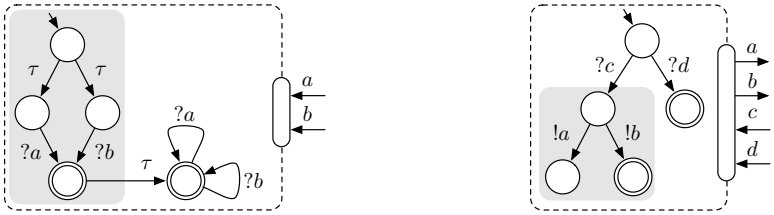
For a controllable service A there exists service B such that $A \oplus B$ are compatible. Compatibility is closely related to *soundness* [1]. In fact, soundness is more strict because it rules out activities which are never executed as well as livelocks. For soundness, an elaborate diagnosis algorithm exists [180], which exploits several properties of the soundness criterion to avoid a complex state space exploration whenever possible. For example, soundness can be expressed in terms of two simpler Petri net properties, namely *liveness* and *boundedness*. An unsound workflow net fails one of these tests. This result can be used to give detailed diagnosis information. In addition, several simple necessary or sufficient criteria for soundness can be checked before liveness and boundedness checks. For example, certain net classes such as *free choice Petri nets* [59] allow for efficient analysis algorithms. However, this diagnosis approach cannot be adapted to diagnose the reasons of why a service automaton is uncontrollable.

First, a sound control flow does not imply controllability, and vice versa. For example, the control flow of the controllable service automaton in Fig. 3.3(c) is not sound (due to internal deadlocks), and the uncontrollable service automata in Fig. 4.2 all have a sound control flow. Similarly, weaker criteria such as *relaxed soundness* or *non-controllable choice robustness* [57] are not applicable. The latter, for example, assumes that the environment can completely observe the service's state, whereas the internal state of a service can only be guessed from observations on the interface (to this end, a state of the synthesized strategy contains of a *set* of states of the service together with its asynchronous interface).

Second, controllability is not a local, but a global criterion: only under restricted preconditions controllability can be decomposed [109]. The previous section shows that there are multiple reasons that can make a service uncontrollable. Unfortunately, these examples cannot serve as antipatterns. Intuitively, every service that contains a bad scenario such as a hidden choice or an internal deadlock, can be extended such that the problem is either resolved or avoided in the first place (cf. Fig. 4.5). To this end, it is impossible to consider only a fraction of the states of a service and make a statement about the correctness of the service. Therefore, only limited necessary or sufficient structural criteria for (un)controllability exist [165, 126]. Finally, structural results like the invariant calculus [103] for Petri nets are not applicable, because these techniques do not take the interface into account.

COUNTEREXAMPLES

In case structural methods are not applicable or can only give partial information on the correctness of a system, the *behavior* of the system (i. e., its state space) needs to be analyzed. The ability to generate *counterexamples* greatly boosted the acceptance of model checking [39] in the field of computer-aided verification. If a model does not meet a given specification, model checking techniques automatically provide such a counterexample. For the modeler, this is a useful artifact (e. g., a deadlock trace) to



(a) resolve problem (b) avoid problem

Figure 4.5: Uncontrollability cannot be checked locally using antipatterns (shaded gray): pending messages can be received later (a) and internal deadlocks can be avoided (b).

understand the reasons *why* the model contains an error, how it is reached, and how to fix the model. Likewise, *witnesses* are useful means to prove that a system satisfies certain properties.

To find a counterexample for controllability is a nontrivial task because of the criterion's nature. Controllability is "proved" by constructing a witness: A is k-controllable iff there *exists* some service B such that the composition $A \oplus B$ is k-compatible. In other words, B can be seen as a counterexample for A's *un*controllability. If A is not controllable, we can only conclude that *no* such service exists, and hence cannot provide a counterexample which can be used to find out, which of the various problems we described in the previous section rendered the service uncontrollable.

The algorithm to decide controllability (cf. Def. 2.9) overapproximates a strategy for A and then iteratively removes states of this overapproximation which will not be part of any strategy of A. If A is uncontrollable, all states will be eventually deleted. In the remainder of this section, we elaborate how a counterexample for A's controllability (or a witness for A's uncontrollability) should be shaped to support to locate and to understand the problems that lead to uncontrollability. In the next two sections, we then define an algorithm to use information why states are deleted from $TS^0(A)$ and $TS^j_k(A)$ to give diagnosis information for an uncontrollable service A.

As a motivation for the desired style of diagnosis information, consider again the service in Fig. 4.2(b). We already described informally why this service is uncontrollable:

> After sending a c-message, a partner has to send either an a-message or a b-message to the service. If the wrong choice is made, the message keeps pending on the input channel. This eventually yields a covered final state.

Let us analyze this informal description of why the service is uncontrollable. It contains:

(I) an indisputable initial part ("after sending a c-message") which describes the communication between the service and a possible interaction partner,

(C) a description of possible continuations ("a partner has to send either an a-message or a b-message") which are derived from the service's control flow, and

(P) the problem which ultimately hinders a partner achieve compatibility of the composition ("If the wrong choice is made, the message keeps pending on the input channel. This eventually yields a covered final marking.").

Before we explain the parts, we need to introduce *waitstates*, which model situations in TS_0 which can only be left with the help of the environment.

Definition 4.1 (Waitstate).
Let A be a service automaton. The pair $[q, \mathcal{B}] \in Q_A \times Bags(\mathbb{M}_a)$ is a *waitstate* if, for all $q \in Q_A$ and $e \in \mathbb{E}$, $q \xrightarrow{e}_A q'$ implies (1) $e \in ?\mathbb{E}$ and $\mathcal{B}(\mathbb{M}(e)) = 0$ or (2) $e \in !?\mathbb{E}$. This waitstate can be *resolved* by (1) sending an asynchronous message e to A or by (2) synchronizing with A via channel e, respectively.

A waitstate is a situation the service automaton A cannot leave without communication with the environment; that is, an asynchronous message needs to be received or a synchronization with the environment is required. The notion of waitstates will be used to define the (I), (C), and (P) parts of the previous description. The initial part (I) consists of communication steps which are necessary to resolve a waitstate and which would also be taken by partners who *know* the outcome of the service's decision in advance. Sending a c-message is not source of the problem, because this message *will* be received by the service. In contrast, after sending an a-message, any continuation (C) *can* lead to a situation where reaching a final marking is not any more guaranteed. Finally, the possible problem which can occur after sending either message is described (P). This subtle distinction between indisputable "safe" interactions and problematic "unsafe" interactions is crucial to construct an artifact that can serve as counterexample.

In the following, we generalize this approach and elaborate the required information to define an algorithm which automatically derives such diagnosis results for an uncontrollable service A consisting of these three parts:

(I) From the strategy overapproximation $TS^0(A)$, we define a maximal subgraph $TS^{0*}_k(A)$ such that the composition $A \oplus TS^{0*}_k(A)$ is free of *bad states*. A state is considered bad if it contains an internal deadlock, a covered final state, an exceeded message bound, or an internal livelock.

(C) The subgraph $TS^{0*}_k(A)$ is not a strategy of A, because its nodes contain waitstates which are not resolved in $TS^{0*}_k(A)$, because the respective edge to a successor

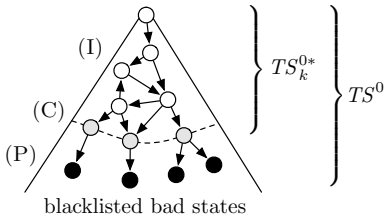


Figure 4.6: A counterexample for controllability consists of an initial part (I), possible continuations (C), and resulting problems (P). The subgraph TS_k^{0*} is defined using blacklists.

is missing. When these waitstates are resolved by sending messages to or by synchronizing with A, the composition may reach a state from which a bad state cannot be avoided any more. Therefore, in the second part of the diagnosis result, each unresolved waitstate is described including a communication trace from the initial state to the state containing this waitstate.

(P) Finally, we give detailed information how the resolution of the waitstate can reach a bad state. For each problem, witness paths to the problematic situation or pointers to the structure of A are given to locate the problem.

Figure 4.6 illustrates the overall shape of a counterexample for controllability. It is a subgraph of TS^0 from which all blacklisted bad states (P) are removed. The actual diagnosis information can then derived from those waitstates from which a transition to blacklisted states is inevitable (C). The initial part (I) may be empty in case a bad situation (i. e., an internal deadlock, etc.) can be reached from the initial state without interaction, cf. Fig. 4.1(a). The final diagnosis algorithm will treat this case separately.

4.3 AN OVERAPPROXIMATION OF A COUNTEREXAMPLE

The counterexample we sketched in the previous section is based on a subgraph of the strategy overapproximation $TS^0(A)$ from Def. 2.9. Before we go into details on how to derive this subgraph, we have to make sure that the counterexample we construct does not contain unnecessary parts.

Definition 2.9 aims at synthesizing a most-permissive strategy for a service A. The algorithm achieves this by first generating the behavior of *any* service communicating with A. This leads to several bad states in the composition $TS^0(A) \oplus A$, which are iteratively removed. Only by starting out with a maximal overapproximation, most-permissiveness is guaranteed.

In case a service is uncontrollable, every state is eventually considered bad. However, not every bad state can be used to derive diagnosis information. To explain the reasons which lead to uncontrollability, the overapproximation should contain as few states as possible. In particular, messages should be only sent if they can resolve a waitstate. Likewise, receive and synchronization events should only occur if they are really possible. If we construct an overapproximation in this fashion (i. e., the construction of every state has a reason), we can derive concrete diagnosis information from bad states.

Although a smaller overapproximation is not suitable to construct a most-permissive strategy, it can dramatically speed up the synthesis of an arbitrary strategy. Weinberg [184] defined several on-the-fly reduction rules to find compact strategies. These strategies can be used in case only the *existence* of a strategy is of interest rather than a complete characterization of all strategies satisfying the constraint.

We use two reduction rules from [184]: The first rule (called "activated events") avoids synthesizing unreachable behavior and only sends messages to resolve waitstates. The second rule (called "receive before send") prioritizes receiving events before sending events. The result is a smaller overapproximation. We adjust Def. 2.9 as follows.

Definition 4.2 (Reduced strategy synthesis).
Let $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ be an open finite state service automaton with $\mathcal{P}_A = \{[I_1, O_1], \ldots, [I_n, O_n]\}$. We define the open service automaton $TS^0_{red}(A) = [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ with $\mathcal{P} = \{[O, I] \mid [I, O] \in \mathcal{P}_A \cap (\mathbb{M}^{\sqcup}_{\mathcal{P}_A} \times \mathbb{M}^{\sqcup}_{\mathcal{P}_A})\}$ and Q, q_0, $\rightarrow$, and Ω inductively as follows:

- Base: Let $q_0 := closure_A(\{[q_{0_A}, [\,]]\})$. Then $q_0 \in Q$.
- Step: For all $q \in Q$ and $m \in \mathbb{M}$:
  1. If $!m \in \mathbb{E}_{\mathcal{P}}$ and $[q_1, \mathcal{B}] \in q$ with (i) $q_1 \xrightarrow{?m}_A q_2$, (ii) $\mathcal{B}(m) = 0$, and (iii) $\mathcal{B}(m') = 0$ for all $m' \in \bigcup_{j=1}^{n} I_j$, let $q' := closure_A(\{[q_A, \mathcal{B}+[m]] \mid [q_A, \mathcal{B}] \in q\})$. Then $q' \in Q$ and $q \xrightarrow{!m} q'$.
  2. If $?m \in \mathbb{E}_{\mathcal{P}}$, let $q' := closure_A(\{[q_A, \mathcal{B}] \mid [q_A, \mathcal{B} + [m]] \in q\})$. If $q' \neq \emptyset$, then $q' \in Q$ and $q \xrightarrow{?m} q'$.
  3. If $!?m \in \mathbb{E}_{\mathcal{P}}$, let $q' := closure_A(\{[q'_A, \mathcal{B}] \mid [q_A, \mathcal{B}] \in q \wedge q_A \xrightarrow{!?m}_A q'_A\})$. If $q' \neq \emptyset$, then $q' \in Q$ and $q \xrightarrow{!?m} q'$.
- We define $\Omega := \{q \in Q \mid q \cap (\Omega_A \times \{[\,]\}) \neq \emptyset\}$.

From $TS^0_{red}(A)$, we proceed as in Def. 2.9 by iteratively removing states where the message bound is violated or which contain deadlocks, yielding $TS_{k_{red}}(A)$. Compared to Def. 2.9, the following adjustments have been made:

– An asynchronous message m is only sent if it resolves a waitstate and if no receiving event is possible. This is expressed by adding to (1.) the requirement that the state q must contain a state $[q_1, \mathcal{B}]$ in which (i) message m can be received by A, (ii) that no message is pending on the input channel m, and that (iii) also all output channels are empty in state $[q_1, \mathcal{B}]$.

– Asynchronous receive events and synchronization events are only added if they are actually possible. In Def. 2.9, state q' and transition $q \xrightarrow{?m} q'$ were added even if there exists no state $[q^*, \mathcal{B}] \in q$ with $\mathcal{B}(m) > 0$. In this case, $q' = \emptyset$ (cf. Fig. 2.3). The same effect can occur if a synchronization event is not possible. Hence, the reduced strategy only contains states q with $q \neq \emptyset$.

– Finally, responsiveness of A is not required. If A is uncontrollable, because it is unresponsive, the diagnosis algorithm should report this.

The first adjustment ensures that every sending event has a "reason", namely the resolution of a waitstate. The second adjustment rules out unreachable behavior in the overapproximation, which does not help to diagnose reasons for uncontrollability. As discussed earlier, the application of the reduction rules do not synthesize a most-permissive strategy and is therefore not applicable during the calculation of an operating guideline. However, Def. 4.2 does synthesize a strategy if and only if the service is controllable [184].

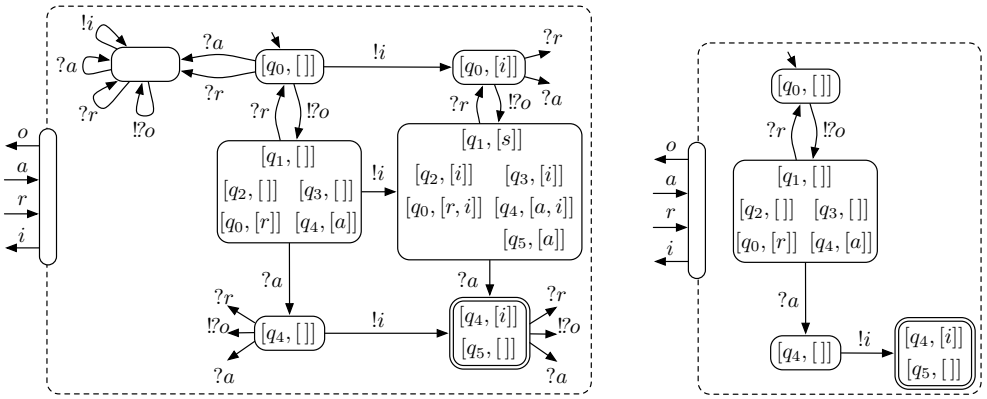
Proposition 4.1 (Reduced strategy proves controllability [184]).
A is k-controllable iff $Q_{TS_{k_{red}}(A)} \neq \emptyset$.

To put it differently: Definition 4.2 preserves the reasons for uncontrollability and $TS^0_{red}(A)$ can be used as an overapproximation for a counterexample rather than $TS^0(A)$.

EXAMPLE. Fig. 4.7 depicts a comparison between a most-permissive strategy and a reduced synthesized strategy. In the reduced strategy, the invoice (i) is only sent to resolve the waitstate $[q_4, [\,]]$.

4.4 BLACKLIST-BASED DIAGNOSIS

To derive diagnosis information — our counterexample demonstrating uncontrollability —, we first need a criterion to decide for each state of $TS^0_{red}(A)$ whether it is a state of the subgraph $TS^{0*}_k(A)$, too. We already motivated that $TS^{0*}_k(A)$ should not contain bad states. Thus, for each problem, we define a *blacklist* which contains such bad states. With these blacklists, we then can define the subgraph $TS^{0*}_k(A)$.



(a) $TS_1(A_{Buy})$ (b) $TS_{1_{red}}(A_{Buy})$

Figure 4.7: A most-permissive strategy (a) and the reduced synthesized strategy (b) for the buyer service from Fig. 2.2(b).

For some bad states, it is also possible to characterize states which eventually *will* be bad. For instance, a state whose successors are all internal deadlocks can likewise be considered bad, because once this state is reached, the service will eventually deadlock. This not only reduces the size of $TS_k^{0*}(A)$, but also the length of the witness paths. Whereas the *early detection* of internal deadlocks is straightforward and already exploited in the setting of behavioral constraints (cf. Sect. 3.4), the early detection of covered final states is more challenging. In particular, a covered final marking does not need to occur immediately after a hidden choice, but can occur many communication steps later.

In addition, we define a witness for each problem. A witness is an artifact which can help to locate the parts of the uncontrollable service that cause the problem (e. g., the transitions modeling a hidden choice or an internal deadlock state).

BLACKLIST FOR DEADLOCKING AND LIVELOCKING CONTROL FLOW

Internal deadlocks and internal livelocks (i. e., unresponsive behavior) are problems that can be detected by analyzing the service in isolation. An internal livelock is a nonempty terminal strongly connected set of states of A, which neither contains a final state nor an open communication event. Because every internal deadlock is a (trivial) internal livelock, we can define a combined blacklist for internal deadlocks and internal livelocks as follows.

Definition 4.3 (Blacklist for internal deadlocks and livelocks).
We define the set of inevitable internal deadlocks of A, $Q_{DL} \subseteq Q_A$, to be the smallest set fulfilling:

– If $q \not\rightarrow_A$ and $q \notin \Omega_A$, then $q \in Q_{DL}$.
– If $q \notin \Omega_A$ and, for all $x \in \mathbb{E}$, $q \xrightarrow{x}_A q'$ implies $q' \in Q_{DL}$, then $q \in Q_{DL}$.

A set of states $Q_{LL} \subseteq Q_A$ is a livelock iff Q_{LL} is a terminal strongly connected component of A and $q \notin \Omega_A$ and $lab(q) = \emptyset$, for all $q \in Q_{LL}$. Let $\mathcal{LL}$ be the set of all internal livelocks of A.

From these sets, define the *blacklist for internal deadlocks and internal livelocks* as $bl_{DLL} := \{q \in Q_{TS^0_{red}(A)} \mid q \cap ((Q_{DL} \cup \bigcup \mathcal{LL}) \times Bags(\mathbb{M})) \neq \emptyset\}$. For each blacklisted state $q \in bl_{DLL}$, define the witness $W_{DLL}(q) := \{q_d \in (Q_{DL} \cup \bigcup \mathcal{LL}) \mid [q_d, \mathcal{B}] \in q\}$.

We not only blacklist states which contain an internal deadlock, but also states which contain a state from which an internal deadlock will be eventually reached. For a blacklisted state q, the witness $W_{DLL}(q)$ is the set of all (inevitable) internal deadlocks and the internal livelocks in q.

BLACKLIST FOR EXCEEDED MESSAGE BOUND

States of the composition which exceed the message bound k can be easily detected by analyzing the states occurring in nodes of TS^0_{red}. The blacklist can be defined straightforwardly:

Definition 4.4 (Blacklist for exceeded message bound).
We define the *blacklist for exceeded message bound* as $bl_{MB} := \{q \in Q_{TS^0_{red}(A)} \mid \exists [q^*, \mathcal{B}] \in q : \exists m \in \mathbb{M}_a : \mathcal{B}(m) > k\}$. For each blacklisted state $q \in bl_{MB}$, define the witness $W_{MB}(q) := \{m \in \mathbb{M}_a \mid \exists [q^*, \mathcal{B}] \in q : \mathcal{B}(m) > k\}$.

Note that the message bound may be exceeded for *both* input and output channels, because the receiving of asynchronous messages may be delayed as Fig. 4.3(b) illustrates.

BLACKLIST FOR COVERED FINAL STATES

In a covered final state q_c reachable in $A \oplus TS^0_{red}(A)$, the control flow of A has reached a final state which cannot be left, but a message is pending on an input channel, which cannot be received from A. By construction of $TS^0_{red}(A)$, this message was originally

sent to A to resolve a waitstate (cf. Def. 4.2). The following observation is needed to justify the later definition of a blacklist for covered final states.

Lemma 4.1 (Covered final states also exist uncovered).
Let A be service automaton with the interface $\mathcal{P}_A = \{[I_1, O_1], \ldots, [I_n, O_n]\}$ and $TS^0_{red}(A)$ as defined in Def. 4.2. Let q_1 be a state of $TS^0_{red}(A)$ and $[q_f, \mathcal{B} + [x]] \in q_1$ a covered final state with $q_f \in \Omega_A$ and $x \in \bigcup_{j=1}^{n} I_j$.
Then exists a state q_2 of $TS^0_{red}(A)$ with $[q_f, \mathcal{B}] \in q_2$.

Lemma 4.1 states that, for each covered final state with a pending x-message occurring in a state of $TS^0_{red}(A)$, there exists a state which contains a covered final state (or a final state if $\mathcal{B} = [\,]$) *without* that pending x-message. Figure 4.8(a) illustrates the lemma and visualizes the interrelations of the states mentions in the following proof.

Proof. Let q_1 be as above. Then there exist states q and q_x of $TS^0_{red}(A)$ with $q \xrightarrow{!x} q_x$, and there exists a path σ from q_x to q_1 which does not contain an $!x$-labeled edge. Let $[q_f, \mathcal{B} + [x]] \in q_1$ be as above. The pending x-message was only sent to A to resolve a waitstate (cf. Def. 4.2). Let $[q_w, \mathcal{B}_w] \in q$ be such a waitstate.

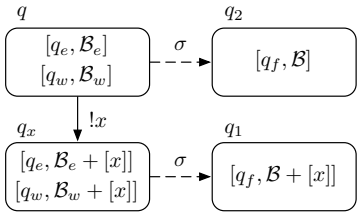
Let $[q_e, \mathcal{B}_e] \in q$ be a state of q. From $q \xrightarrow{!x} q_x$ we can conclude that there exists a state $[q_e, \mathcal{B}_e + [x]] \in q_x$. Let the path σ^* be an extension of the path σ such that $[[q_e, \mathcal{B}_e + [x]], q_x] \xrightarrow{\sigma^*} [[q_f, \mathcal{B} + [x]], q_1]$ in the composition $A \oplus TS^0_{red}(A)$. This path σ^* does not contain a transition labeled with $!x$, because $\sigma^*|_{TS^0_{red}(A)} = \sigma$ does not contain an $!x$-labeled transition. Therefore, σ^* is realizable independently of (i. e., without) the pending x-message. In particular, there exists a state q_2 of $TS^0_{red}(A)$ such that $[[q_e, \mathcal{B}_e], q] \xrightarrow{\sigma^*} [[q_f, \mathcal{B}]], q_2]$. □

After iteratively applying Lem. 4.1, we can conclude that with each covered final state occurring in $TS^0_{red}(A)$, also a respective "uncovered" final marking is present in a state of $TS^0_{red}(A)$.

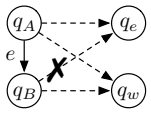
Each application of Lem. 4.1 identifies an $!x$-labeled transition from q to state q_x from which a state q_1 is reached which contains a covered final state with a pending x-message, which is never received. For state q, an alternative continuation to q_2 without an $!x$-transition is possible.

Hence, such a state q_x should be considered critical, which yields the following definition of a blacklist for covered final states.

Definition 4.5 (Blacklist for covered final states).
Let, q, q_x, q_1, q_2, q_e, and q_w as defined to be as in Lem. 4.1 and its proof (cf. Fig. 4.8(a)). We define the *blacklist for covered final states*, bl_{CFS}, to contain exactly those



(a) illustration for Lem. 4.1



(b) hidden choice transition of Def. 4.5

Figure 4.8: Illustrations for Lem. 4.1 and Def. 4.5.

states q_x. Define the witness for a blacklisted state q_x, $W_{CFS}(q_x) := \{q_A \xrightarrow{e}_A q_B \mid q_A \xrightarrow{*} q_w \wedge q_B \xrightarrow{*} q_w \wedge q_A \xrightarrow{*} q_e \wedge q_B \not\xrightarrow{*} q_e\}$, to contain all *hidden choice transitions* of A.

A covered final state is a situation which occurs in case a service automaton A is composed to a partner. With the help of Lem. 4.1, the blacklist for covered final states can be defined only by checking the states of $TS^0_{red}(A)$ and paths in $TS^0_{red}(A)$ and A. This can be realized during the construction $TS^0_{red}(A)$ instead of analyzing paths in $A \oplus TS^0_{red}(A)$. Lemma 4.1 also allows for finding a set of *hidden choice transitions* (see Fig. 4.8(b)), which model a hidden decision as described in Sect. 4.1. These transitions can be the starting point to repair the service to avoid the covered final state.

4.5 DIAGNOSIS ALGORITHM

With the definitions of the blacklists, we are finally able to define the subgraph $TS^{0*}_k(A)$ (i. e., the counterexample for controllability of A) of $TS_{k_{red}}(A)$ which only contains states which are not contained in any of the blacklists. Thereby, we ignore states which have become unreachable from the initial state.

Algorithm 1 combines the defined blacklists together with their witnesses and gives information for each detected problem. After a preprocessing phase (line 1–4) in which TS^0_{red} as well as the blacklists are calculated, the states of TS^0_{red} are analyzed. Thereby, two cases are differentiated: If already the initial state of q_0 is blacklisted, then the service can reach a bad state independently of a partner. Covered final states cannot occur in this setting. As a diagnosis information, the initial state q_0 and the respective problems are printed (line 5–11). The remainder of the algorithm (line 12–27) treats situations in which TS^{0*}_k is nonempty.

Algorithm 1: Blacklist-based diagnosis for uncontrollable services

Input: uncontrollable finite state service automaton A, message bound k
Output: diagnosis information, $TS^{0*}_{k}(A)$

```
1  calculate TS^0_red(A)
2  derive bl_DLL from TS^0_red(A)
3  derive bl_EMB from TS^0_red(A)
4  derive bl_CFS from TS^0_red(A)
5  if q_0 is blacklisted then
6      if q_0 ∈ bl_DLL then
7          foreach witness q* ∈ W_DLL(q_0) do
8              print "internal deadlock/livelock q* reachable without interaction"
9      if q_0 ∈ bl_EMB then
10         foreach witness m* ∈ W_EMB(q_0) do
11             print "message bound of channel m* exceeded without interaction"
12 else
13     foreach nonblacklisted state q reachable from q_0 do
14         foreach waitstate [q', B] ∈ q with q' -e->_A q'' and q_e with q -e-> q_e do
15             if q_e is blacklisted then
16                 print "resolving waitstate [q', B] may reach a bad state"
17                 if q_e ∈ bl_DLL then
18                     foreach witness q* ∈ W_DLL(q_e) do
19                         print "in q_e: internal deadlock/livelock q* reachable"
20                 if q_e ∈ bl_EMB then
21                     foreach witness m* ∈ W_EMB(q_e) do
22                         print "in q_e: message bound of channel m* violated"
23                 if q_e ∈ bl_CFS then
24                     print "in q_e: message e may be left unreceived"
25                     foreach witness [q_1, x, q_2] ∈ W_CFS(q_e) do
26                         print "hidden choice transition: [q_1, x, q_2]"
27     print subgraph TS^0*_k of TS^0_red(A) without blacklisted states
```

The diagnosis messages can be classified into the three categories (initial part (I), possible continuation (C), and occurring problem (P)) as follows:

(I) line 27 prints the nonblacklisted subgraph TS^{0*}_{k},

(C) line 16 prints a nonblacklisted waitstate whose resolution may reach a bad state,

(P) line 8, 11, 19, 22, 24, and 26 print information about the problem which may be unavoidable after resolving the respective waitstate, including witnesses.

The algorithm lists all problems which can occur if TS^{0*}_{k} is "left" by resolving a waitstate. If, for example, sending an x-message can result in a message bound violation *and* yield an internal deadlock, then both problems are reported.

Table 4.1: Experimental results for reduced strategy synthesis using Wendy.

	most-permissive strategy			reduced strategy		
service	$\lvert Q_{TS}\rvert$	$\lvert \rightarrow_{TS}\rvert$	time (sec)	$\lvert Q_{TS_{red}}\rvert$	$\lvert \rightarrow_{TS_{red}}\rvert$	time (sec)
Quotation	11,264	145,811	3	62	77	0
Deliver goods	1,376	13,838	2	53	82	0
SMTP protocol	20,818	144,940	29	62	78	0
Car analysis	1,448	13,863	52	108	183	1
Identity card	1,536	15,115	83	259	1,027	1
Product order	57,996	691,414	303	461	938	0

IMPLEMENTATION AND EXPERIMENTAL RESULTS

The diagnosis has been implemented into the tool Wendy [121]. In a special diagnosis mode, it constructs the reduced strategy from which the blacklists are generated.

The practical applicability of the diagnosis information is hard to measure and needs further investigation. To give an impression on the runtime and the sizes of the counterexamples, Tab. 4.1 lists results on synthesizing reduced strategies for the services we described in Sect. 2.6. The reduced strategies consist only of a fraction of states and all can be calculated in less than a second. The nonblacklisted subgraph which is used as counterexample for controllability is a subgraph of the reduced synthesized strategy, so the numbers of Tab. 4.1 can be seen as an upper bound for the size of the counterexample.

Figure 4.9 depicts an uncontrollable service automaton and the diagnosis output of the tool Wendy. It consists of a graphical representation of the subgraph as well as a textual description of the problems and recommendations how to fix these issues. For instance, if the violation of a given message bound is the only detected problem, then the user is advised to restart the analysis with an increased message bound. The visualization of the counterexample generated by the diagnosis algorithm is in a very early state and needs to be tightly integrated to a service modeling tool. This integration is subject to future work and out of scope of this thesis.

4.6 CONCLUSION

The generation of counterexamples greatly boosted the acceptance of model checking [39] in the field of computer-aided verification. They present the reasons which make a model incorrect and therefore are as important as the verification procedure itself. However, the decision algorithm for controllability (cf. Def. 2.9) does not provide such counterexamples. In this chapter, we investigated uncontrollable service models and presented a variety of reasons why a service does not have any partners

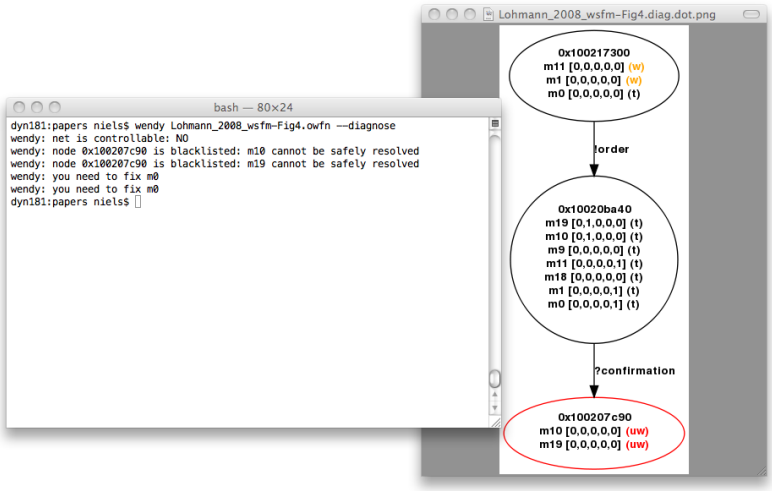


Figure 4.9: Diagnosis output of the tool Wendy.

which interact in a compatible manner. We elaborated how a counterexample for controllability should be shaped to help the modeler understand the reasons which make a service uncontrollable. An algorithm to construct such a counterexample has been defined in terms of blacklists and has been prototypically implemented. The returned diagnosis information can be the starting point for corrections of the service toward controllability. We shall come back to this in Chap. 6. The diagnosis algorithm can be directly used for refinements of controllability, for instance behavioral constraints (cf. Chap. 3), and is likely to be applicable to further extensions.

Several aspects of diagnosing uncontrollable services remain subject of future work. First, service models usually stem from industrial specification languages, such as WS-BPEL. Hence, the retranslation of (automaton-related) diagnosis information back into WS-BPEL is a prerequisite to correlate the problems to the original model. Existing translations between service automata and WS-BPEL [110, 114] could be extended to translate the necessary diagnosis information. In particular, a mapping between diagnosed bad states and activities in the original process could be challenging.

Second, the acceptance of the counterexamples needs to be further investigated. First experiments showed that especially hidden choices are often overlooked even by experienced service modelers. Nonlocality, asynchronous message exchange, and the absence of a concrete interaction partner are only a few of the reasons which make uncontrollable services hard to detect during modeling time.

Finally, further reduction techniques from Weinberg [184] may help to define a more compact counterexample for controllability. Reducing the size of the counterexample not only increases the understandability, but allows for faster calculation. This is crucial to be able to integrate the diagnosis algorithm into modeling tools. A constant analysis of a service model (e. g., each time the model is stored) helps to quickly correlate diagnosed problems to recent changes. For the soundness criterion, it is already possible to integrate verification techniques into industrial modeling tools [68] and verify the model constantly.

Part II

CORRECTNESS OF SERVICE COMPOSITIONS

5 VERIFICATION AND COMPLETION

This chapter is based on results published in [115].

In the previous two chapters, we investigated the correctness of services in isolation; that is, services embedded in arbitrary environments. With the notion of controllability and behavioral constraints, we could reason about the correctness of *one* service with respect to *any* possible service composition. In this chapter, we go one step further and study the correctness of a *concrete* composition of *several* services.

As in the previous chapters, we focus on the behavior of service compositions and employ compatibility as correctness criterion. For this reason, we do not consider other aspects of composing services, such as wiring (i. e., addressing and syntactical issues), instance lifecycles (i. e., how new instances are created, who triggers instantiations, and how many "copies" of each service are needed), or nonfunctional properties (e. g., an agreement on encryption, policies, or quality of service).

In the literature [157, 63], two viewpoints on a service composition are distinguished: *service orchestrations* and *service choreographies*. They are typically considered to be complementary paradigms, whereas other authors (e. g., [154]) criticize a too strict distinction. We shall come back to this discussion in Chap. 7.

A service orchestration (cf. Fig. 5.1) takes the *viewpoint of a single participant*. It focuses on this *orchestrator* and abstracts from the internal behavior of other participants. The service orchestrator only considers the ports to the other participants rather than their concrete behavior or their interaction between third parties. Service orchestrations are well-suited to describe a business process whose activities are executed by other services. For the execution of service orchestrations, the language WS-BPEL [11] emerged as a de-facto standard. A WS-BPEL process specifies how other services are invoked and includes all information that are required to execute it on an engine.

A service choreography (cf. Fig. 5.2) takes the *global viewpoint* on a service composition and does not focus on individual participants. From a modeling perspective, choreographies can be used as a bottom-up approach (called *interconnected models*) or as a top-down approach (called *interaction models*).

In the paradigm of interconnected models (cf. Fig. 5.2(a)), several local service models are merged into a service choreography; that is, services are composed. This can be seen as bottom-up approach, because the global behavior of the choreography is determined by wiring already specified services. It is the classical scenario of SOC (also called *programming in the large* [58]) facilitating the design of large systems

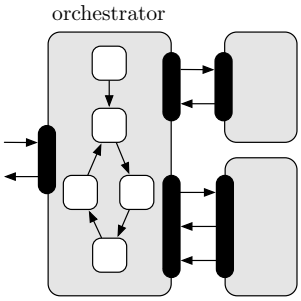


Figure 5.1: Service orchestration.

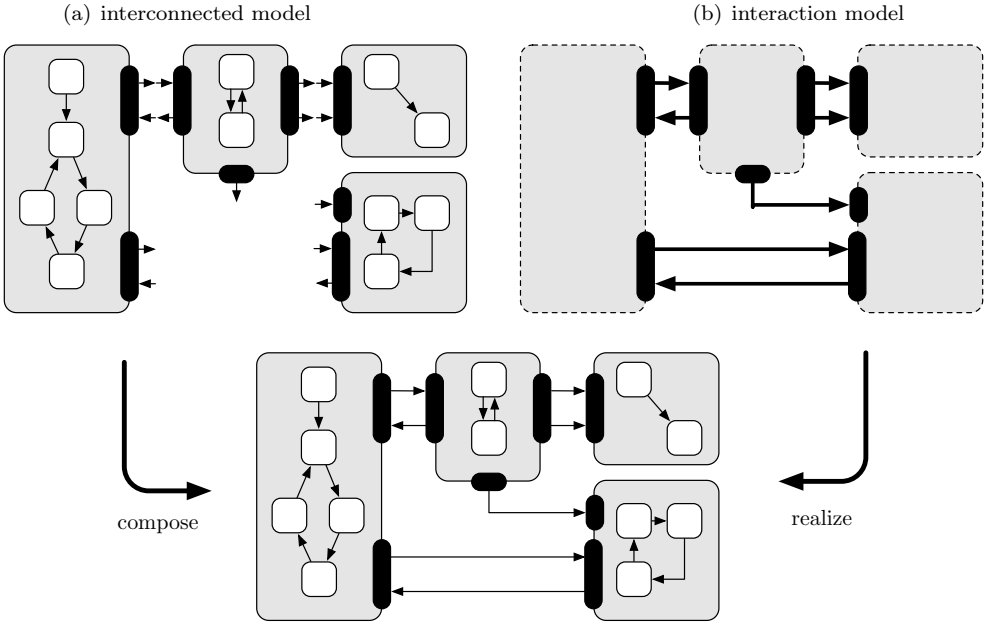


Figure 5.2: Service choreography.

by composing smaller building blocks. The language BPEL4Chor [50] has been introduced to specify global interactions by reusing WS-BPEL processes.

In contrast, the top-down approach, used by the interaction model paradigm (cf. Fig. 5.2(b)), starts with a specification of the desired global behavior of a service composition which is yet to be realized. This interaction model is then projected to the participating services and refined toward execution. Interaction modeling aims at early design stages of service compositions and is typically used to model novel

interorganizational business processes rather than already established compositions. We shall investigate interaction models in Chap. 7.

In this chapter, we investigate the correctness of interconnected models (i. e., service compositions) specified in the language BPEL4Chor. To this end, we continue as follows. The next section briefly introduces the languages WS-BPEL and BPEL4Chor. In Sect. 5.2, we give a formalization of these languages in terms service automata. To facilitate this translation, we employ Petri nets as intermediate formalism, because they offer a compact representation of service automata. Section 5.3 is devoted to the compatibility analysis of BPEL4Chor choreographies. Experimental results show that the verification techniques scale to choreographies with up to a thousand participants. In Sect. 5.4, the completion of partially specified choreographies is studied. By applying results from previous chapters, we can automatically synthesize stub processes for incomplete choreographies. Finally, Sect. 5.5 presents related work and Sect. 5.6 concludes the chapter.

5.1 WS-BPEL AND BPEL4CHOR

WS-BPEL

The *Web Services Business Process Execution Language* (WS-BPEL) [11], is a domain-specific language for describing the behavior of business processes based on Web services. This makes WS-BPEL a language for the *programming in the large* paradigm [58]. Its focus is — unlike modifying variable values in classical programming languages such as C or Java — the message exchange and interaction with other Web services. Advanced concepts such as instantiation, complex exception handling, and compensation of long running transactions are further features which are needed to implement business processes. These features are first-class citizens in WS-BPEL. In this section, we shall only give a brief overview of those concepts of the language which are relevant in this thesis. The interested reader is referred to detailed introductions [18, 183, 7].

For the specification of a business process, WS-BPEL provides *activities* and distinguishes between basic and structured activities. A basic activity can exchange messages with other services (`invoke`, `receive`, `reply`), manipulate and validate data, wait for a period of time or just do nothing (`empty`), signal faults, invoke a compensation handler, or end the entire process instance.

A structured activity defines a causal execution order on basic activities and can be nested in another structured activity itself. The structured activities include sequential execution (`sequence`), parallel execution (`flow`), data-dependent branching (`if`), timeout- or message-dependent branching (`pick`), and repeated execution (`repeatUntil`, `while`, and `forEach`). Within activities executed in parallel, the

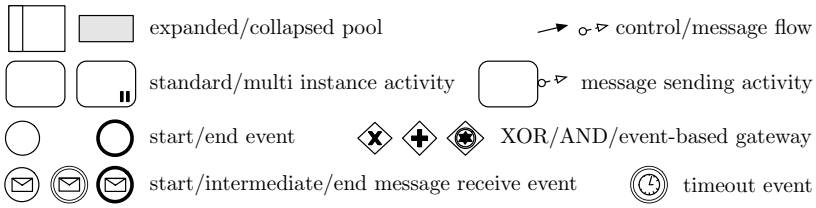


Figure 5.3: BPMN in a nutshell.

execution order can further be controlled by the usage of *control links*. A control link has a source and a target activity. With Boolean conditions, the splitting and joining behavior can be controlled. If a target activity has to be skipped due to negative evaluation of its join condition, all outgoing control links are set to false, which may cause other activities to be skipped, which is called *dead-path elimination* [105].

In addition, the structured activity `scope` links fault, compensation, termination, and event handling to an activity. The `process` is the outmost scope of the described business process. A `faultHandler` provides methods to react to faults, which may occur during execution, whereas a `compensationHandler` can be used to reverse the effects of successfully executed scopes. With the help of an `eventHandler`, external message events and specified timeouts can be handled. The forced termination of running scopes is controlled by a `terminationHandler`.

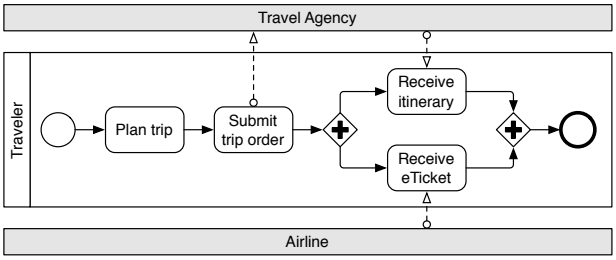
WS-BPEL supports two kind of process specifications. On the one hand, an *executable process* contains all information required to be deployed and executed on a WS-BPEL engine. On the other hand, WS-BPEL further allows to leave parts of the process unspecified. In such *abstract processes*, a placeholder such as an `opaqueActivity` can be used which is later replaced by concrete activities or branching conditions. An abstract process implicitly specifies a set of executable completions.

Although WS-BPEL is intended as exchange and documentation format, it is based on XML and provides no graphical representation. This makes visualization and specification cumbersome. Hence, every vendor of WS-BPEL development tools introduced proprietary graphical notations. In this thesis, we employ BPMN [150] as graphical representation. Figure 5.3 provides an overview of the BPMN constructs used in this thesis. The level of abstraction of BPMN is similar to that of service automata. In particular, the order in which messages are sent and received, the initial state, and final states can be easily derived from a BPMN diagram. In addition, the upcoming BPMN standard [151] provides a basic mapping between BPMN and WS-BPEL.

EXAMPLE. In this chapter, we investigate a choreography modeling a ticket booking scenario taken from [50]. It consists of several participants: a *traveler* who sends

```
<process name="traveler" ... >
  <sequence>
    <opaqueActivity name="PlanTrip" />
    <invoke wsu:id="SubmitTripOrder" />
    <flow>
      <receive wsu:id="ReceiveItinerary" />
      <receive wsu:id="ReceiveETicket" />
    </flow>
  </sequence>
</process>
```

(a) abstract WS-BPEL process



(b) BPMN visualization

Figure 5.4: Traveler service.

a trip order (i. e., a request to book a particular trip) to a *travel agency*, which in turn queries several *airline services* for prices and chooses the cheapest offer. Finally, the traveler receives an itinerary from the travel agency and an e-ticket from the chosen airline. Figure 5.4(a) depicts the traveler's perspective modeled as an abstract WS-BPEL process. In this service orchestration, only the behavior of the traveler is explicitly specified inside an expanded pool, whereas the behavior of the travel agency and the airline services is left unspecified. In BPMN notation, this is modeled by collapsed pools (depicted gray in Fig. 5.4(b)).

BPEL4CHOR

Similar to the traveler service, the other participants' behavior can be specified using WS-BPEL. To describe the interaction of several WS-BPEL processes from a global perspective, BPEL4Chor [50] has been introduced as a choreography description language based on WS-BPEL. BPEL4Chor is not an execution language, but a means to specify all aspects which are required to execute several WS-BPEL processes as a choreography. This approach aims at reducing complexity by reusing services and execution infrastructure. A choreography described by BPEL4Chor consists of (1) the *participant topology*, (2) the *participant behavior descriptions* (PBDs), and (3) the

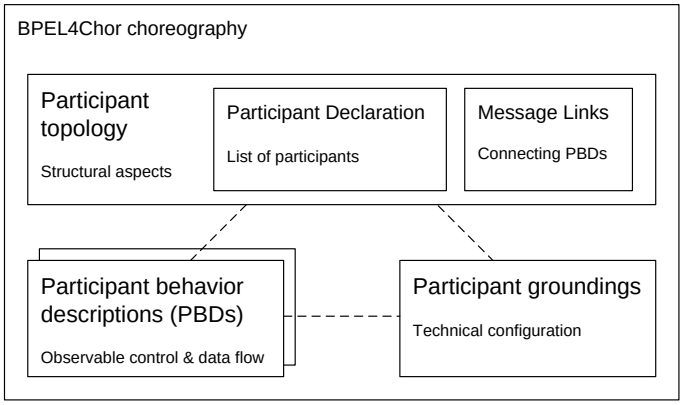


Figure 5.5: Artifacts of a BPEL4Chor choreography [50, 51].

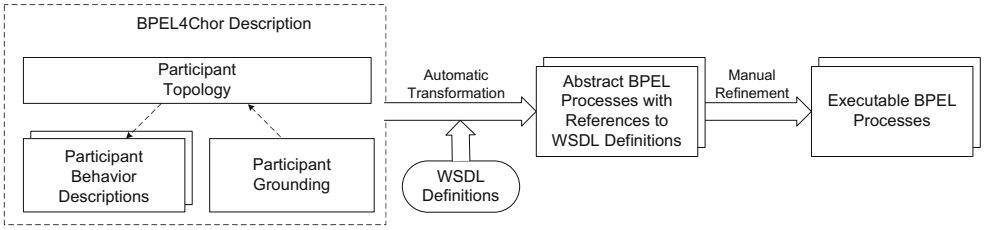


Figure 5.6: Workflow from a BPEL4Chor choreography description to executable WS-BPEL processes [51].

participant groundings (cf. Fig. 5.5 and [50, 51]). The participant topology lists all participants taking part in the choreography and all message links connecting activities of different participants. BPEL4Chor allows for the specification of *participant sets* to group several instances of a participant type. These sets can be (sequentially or parallely) traversed using WS-BPEL's `forEach` activity. A *message link* states that a message is sent from the source of the message link to its target. A BPEL4Chor choreography always describes the behavior of all participants. Thus, a *closed world* is assumed.

Every participant has a certain type. For each participant type, a participant behavior description defined in WS-BPEL is given. In this description, port types and operations are omitted and thus the dependency on interface specifications such as WSDL [38] is removed; that is, the PBDs are abstract WS-BPEL processes. To execute the choreography, every target of a message link has to be grounded to a WSDL operation so that the other participants can use the offered operation. This grounding is done after the choreography design itself, which enables choreography

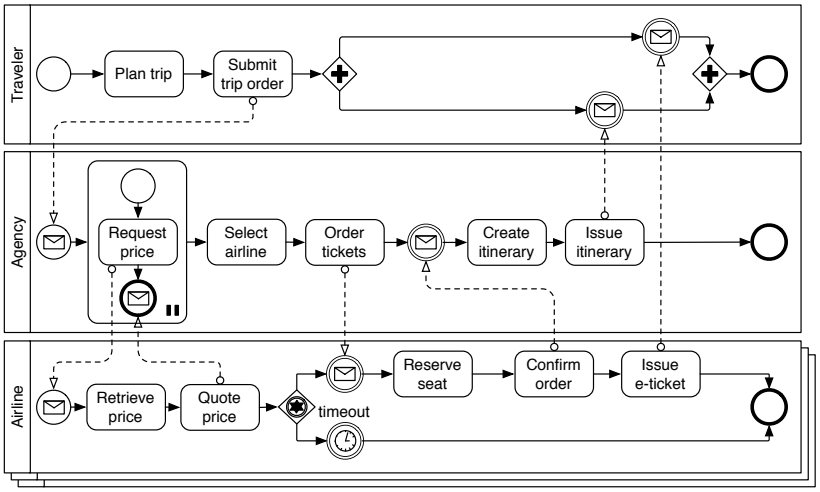


Figure 5.7: Choreography of a ticket booking scenario, taken from [50].

specification reuse. As WS-BPEL is used to specify the behavior of every participant, the development of executable WS-BPEL processes implementing this behavior can be done by using the PBD of a participant as a basis and adding missing details. Reimann et al. [162] elaborate this process. Figure 5.6 diagrams the overall workflow from a BPEL4Chor choreography to executable WS-BPEL processes. Even though other languages can be used to provide implementations of local behavior, using WS-BPEL is a seamless choice using BPEL4Chor.

EXAMPLE. Figure 5.7 shows the complete ticket booking scenario from [50]. It specifies the behavior of all participants; that is, all pools are expanded and there is no message exchange with any undefined participants. The multiple airline instances are modeled as follows. The airline pools are stacked and the "request price" activity is executed in a multiple instances activity modeling a parallel `forEach` activity. After receiving a quote from each of the airline instances, a choice is made. Activity "order tickets" sends a confirmation message to the selected instance. Finally, a timer event is used to terminate unchosen airline instances.

5.2 FORMALIZING WS-BPEL AND BPEL4CHOR

The WS-BPEL language specification [11] describes the operational semantics of WS-BPEL in natural language. This might be sufficient to understand WS-BPEL, but leaves room for ambiguities, contradictions, or unspecified behavior. A formalization [84] of a predecessor specification [12] revealed such unspecified situations which were resolved in the current specification. To *formally* reason about WS-BPEL

processes (i. e., to proof or to verify properties), *formal semantics* are needed. Therefore, a lot of work has been conducted to give formal semantics for the behavior of WS-BPEL processes. The approaches cover many formalisms such as Petri nets, automata, abstract state machines, process algebras, and so on [27, 120, 119]. A few approaches are feature-complete and try to formalize every aspect of WS-BPEL. These approaches usually aim at a deeper understanding the language. Usually, however, only a subset of a language is formalized to investigate a certain aspect.

In the setting of this thesis, we focus on the *behavior* of a WS-BPEL process. We abstract from other aspects such as time, instantiation, or data. In [110, 107], we presented a translation from WS-BPEL to a class of Petri nets. Petri net-based formalisms have the advantage that they are closely related to automata, but can natively express concurrency which facilitates the specification of distributed systems. This make Petri nets an ideal intermediate formalism between WS-BPEL in which concurrency is very common and service automata, the basic formalism of this thesis.

PETRI NETS

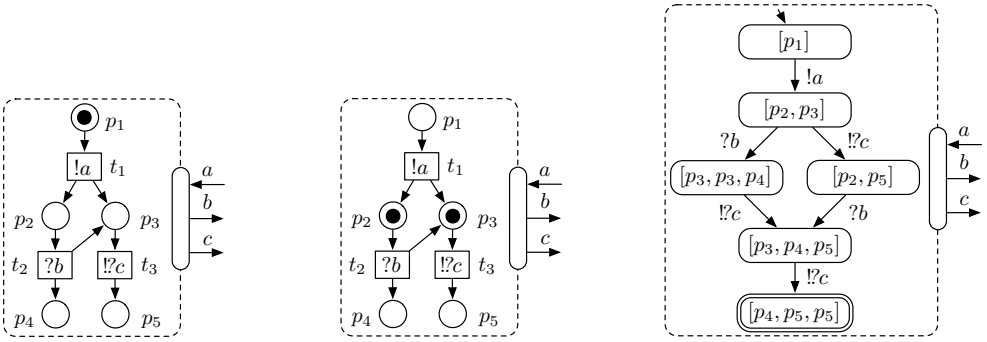
Petri nets [163, 142] are a formalism which was introduced to model and reason about distributed systems. Locality of the cause and effect of actions are realized consequently. This is reflected by the absence of a global notion of a state in favor of a distribution of resources throughout the system. In addition, Petri nets have a natural graphical representation, which was used as inspiration for later graphical notations such as UML activity diagrams or BPMN.

As already discussed in Sect. 2.7, the algorithm for controllability relies on global states and does not exploit concurrency. To this end, we use service automata as formal model in this thesis. However, Petri nets can be used to compactly represent larger service automata. At the same time, the ability to model distributed systems makes Petri nets a convenient intermediate formalism to translate industrial languages such as WS-BPEL into service automata.

As Petri nets are not the main topic of this thesis, but just an intermediate formalism, we do not further discuss the specifics of the model, but continue with defining *service nets*, a class of Petri nets, which is tailored to the needs of this chapter. The interested reader is referred to detailed introductions by Reisig [163], Murata [142], and Desel and Reisig [60].

Definition 5.1 (Service net).
A *service net* is a tuple $N = [P, T, F, m_0, \Omega, \mathcal{P}, \ell]$ such that

- P is a finite set of places,
- T is a finite set of transitions ($P \cap T = \emptyset$),
- $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation,



(a) N_1 (b) N_1 after firing t_1 (c) A_{N_1}

Figure 5.8: A service net with $\Omega = \{[p_4, p_5, p_5]\}$ (a) whose initial marking $m_0 = [p_1]$ enables transition t_1. Firing t_1 yields the marking $[p_2, p_3]$ (b). The net can be translated into a service automaton (c).

- $m_0 \in Bags(P)$ is an initial marking,
- $\Omega \subseteq Bags(P)$ is a set of final markings,
- $\mathcal{P}$ is an interface, and
- $\ell : T \to (\mathbb{E}_{\mathcal{P}} \cup \{\tau\})$ a labeling function.

A service net consists of a classical place/transition net $[P, T, F, m_0]$, a set of final markings, which model desired final states, an interface, and a labeling function that labels each transition with τ or an event that is derived from the interface.

We use the standard graphical notation for Petri nets and depict places by circles, transitions by rectangles, and the flow relation by directed arcs. A marking m is represented by a distribution of $m(p)$ black dots (called "tokens") to each place p. Transition labels are written inside the transitions. Final markings have no graphical representation and are annotated to the net. We depict ports in the same way as for service automata. Figure 5.8(a) shows an example.

Definition 5.1 already suggests a close syntactical relationship to service automata. To give a mapping from a service net to a service automaton, we need to define the operational semantics of a service net; that is, we define a concept of states and state transitions. We do this by applying definitions known from place/transition nets.

Definition 5.2 (Firing rule).
Let $N = [P, T, F, m_0, \Omega, \mathcal{P}, \ell]$ be a service net. For a node $x \in P \cup T$, define the preset of x as ${}^\bullet x := \{y \mid [y, x] \in F\}$ and the postset of x as $x^\bullet := \{y \mid [x, y] \in F\}$.

A transition t is *enabled* at marking $m \in Bags(P)$ iff $m(p) > 0$ holds for all places $p \in {}^\bullet x$. An enabled transition t can *fire* in m, denoted $m\ [t\rangle_N\ m'$, yielding the successor marking m' with

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

In the net of Fig. 5.8(a), transition t_1 is enabled in the initial marking. Firing t_1 yields a successor marking depicted in Fig. 5.8(b). Now we can use markings of a service net as states of a service automaton. Likewise, the labeling of the service net's transitions can be used to derive a labeled transition relation.

Definition 5.3 (Service net translation into service automaton).
Let $N = [P, T, F, m_0, \Omega, \mathcal{P}, \ell]$ be a service net. Define the service automaton for N as $A_N := [Q, q_0, \rightarrow, \Omega, \mathcal{P}]$ with

- $Q := Bags(P)$,
- $q_0 := m_0$, and
- $\rightarrow := \{[m, \ell(t), m'] \mid m\ [t\rangle_N\ m'\}$.

As always, we only consider reachable states of A_N. Figure 5.8(c) depicts the service automaton for the service net of Fig. 5.8(a). Even though the transitions t_2 and t_3 can fire concurrently in N_1, they are explicitly ordered in A_{N_1} which results in several intermediate states. This potential exponential growth of intermediate states in the size of the net is referred to as the *state explosion problem* [177].

FORMAL SEMANTICS FOR WS-BPEL

In the following, we use service nets to define formal semantics for WS-BPEL. The translation of a WS-BPEL process into a service net model is guided by the syntax of WS-BPEL. In WS-BPEL, a process is built by plugging instances of language constructs together. Accordingly, each construct of the language is translated separately into a service net. Such a net forms a *pattern* of the respective WS-BPEL construct. Each pattern has an interface for joining it with other patterns as is done with WS-BPEL constructs. Patterns capturing WS-BPEL's structured activities may

carry any number of inner patterns as its equivalent in WS-BPEL can do. The collection of patterns forms the *service net semantics* for WS-BPEL.

Whereas the original semantics [110, 107] captures the standard as well as the exceptional behavior of a WS-BPEL process, we only consider the standard behavior in this thesis to ease the presentation. We also do not present the formalization of control links and dead-path elimination. Figure 5.9 gives an overview of the used patterns. These patterns can, however, be canonically enhanced to model fault, compensation, and exception handling of the participating WS-BPEL processes. The translation is guided by the structure of WS-BPEL and first translates the basic activities into the respective Petri net patterns. These nets are then embedded into those patterns structured activities. The interested reader is referred to a report [107] which discusses the complete semantics as it is implemented in the tool BPEL2oWFN.

EXAMPLE. Figure 5.10(a) depicts the traveler service from Fig. 5.4 translated into a service net. From this net, a service automaton (cf. Fig. 5.10(b)) can be canonically derived.

PETRI NET SEMANTICS FOR BPEL4CHOR

To translate a BPEL4Chor choreography, two steps are involved: (1) translate each participant's WS-BPEL process into a service automaton and (2) compose the resulting models. All required information can be derived from the BPEL4Chor choreography, cf. Fig. 5.5. In the ticket booking scenario we use as running example, however, there are several *instances* of the airline service involved. To this end, the translation described in [110, 107] needs to be extended to support multiple *instantiation* of participants.

With "instantiation" we do not refer to the lifecycle of a service instance. This lifecycle includes the analysis of incoming messages to decide whether a new instance needs to be created or messages need to be forwarded to existing instances (called *correlation* in WS-BPEL) and the removal of terminated instances from the execution engine. These steps are realized transparently by execution languages such as WS-BPEL and should not influence the behavior of a service composition. In the translation, "instantiation" means creating several copies of a participants and adjusting the wiring of interfaces.

We realized the instantiation of a choreography participant by providing several identical copies of the service net of the participant description. By choosing unique place and transition names (e. g., using prefixes) the behavior of each instance is distinctively modeled. To be able to later compose these models, also the message channel names need to be adjusted to ensure bilateral and unidirectional communication. Therefore, we need to adjust the participant's behavior according to the following scenarios:

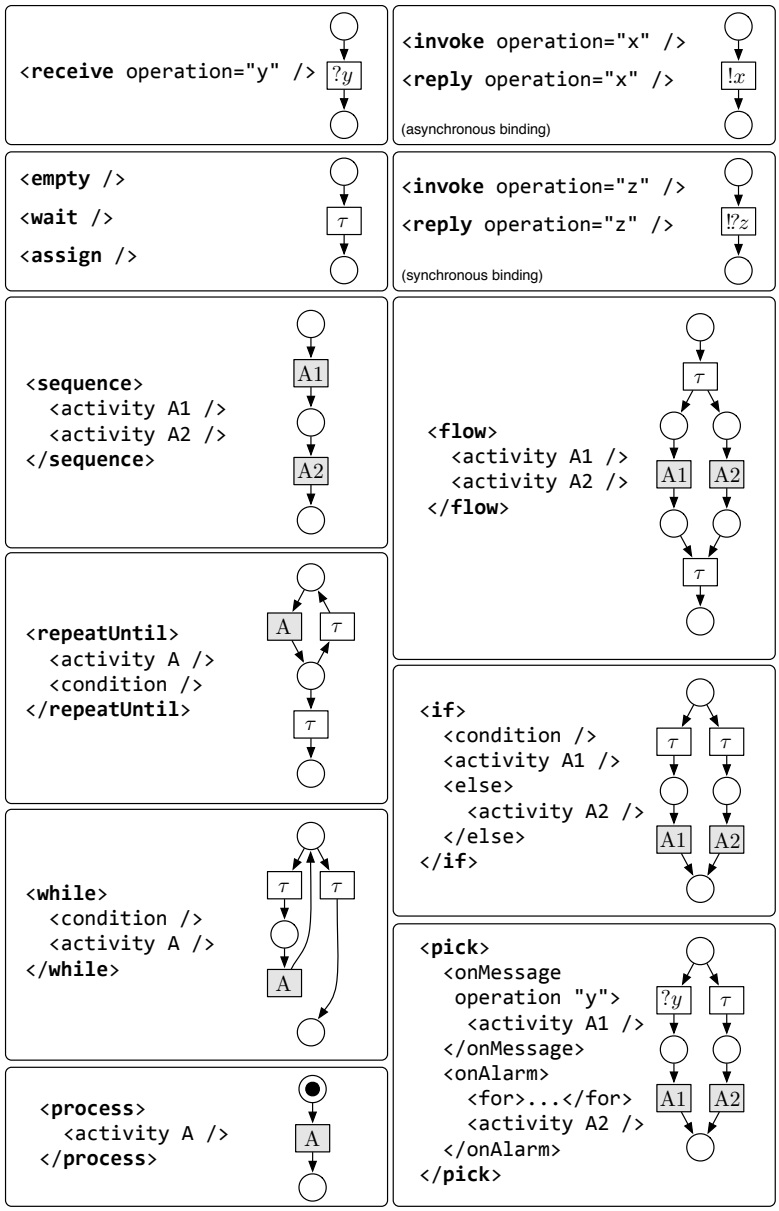
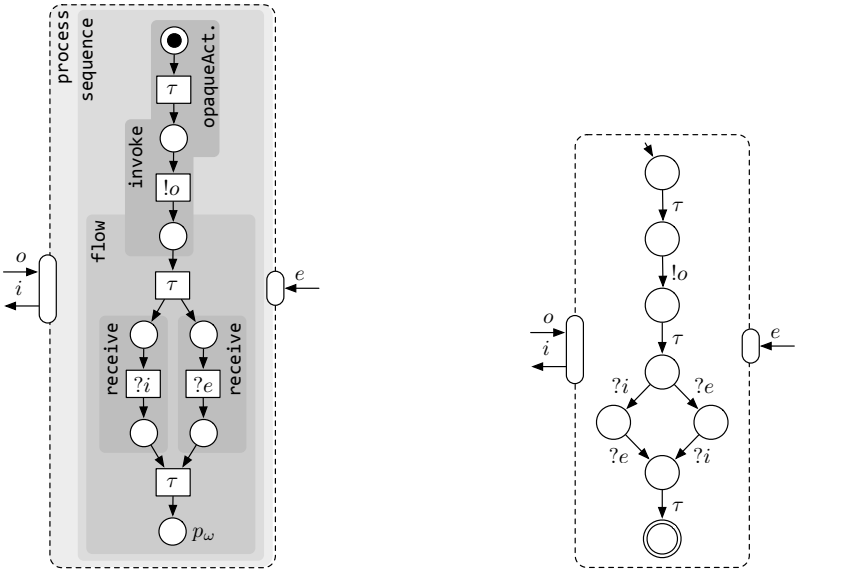


Figure 5.9: Petri net patterns to formalize WS-BPEL.



(a) service net with $\Omega = \{[p_\omega]\}$ (b) corresponding service automaton

Figure 5.10: Translation of the WS-BPEL traveler service.

1. A message is exchanged between two uninstantiated participants (e. g., the trip order sent by the traveler to the agency): no adjustment is needed.
2. A message is exchanged between an uninstantiated participant and one particular instantiated participant (e. g., the price request sent by the agency to each airline instance): The behavior of the uninstantiated participant needs to be duplicated and executed for each instance, either concurrently or sequentially depending on the `forEach` activity used to traverse the instances. In addition, the message channel names need to be adjusted.
3. A message is exchanged between an uninstantiated participant and an arbitrary chosen instantiated participant (e. g., the e-ticket sent by the selected airline to the traveler): The behavior of the uninstantiated participant needs to be duplicated for each instance and executed mutually exclusively. In addition, the message channel names need to be adjusted.
4. A message is exchanged between two instantiated participants (not present in our example choreography): Similar to the second scenario.

As stated earlier, the participant topology holds the necessary information about which process and which message channel has to be instantiated. Admittedly, the topology does not provide the number of instances of each participant. *We therefore*

demand an upper bound of instances to be specified for each participant set. Whereas this upper bound may not be necessary if BPEL4Chor is just a means to *describe* choreographies, its definition is reasonable if such a choreography should be *analyzed* or *executed*.

EXAMPLE. For an example of these scenarios, consider the WS-BPEL code snippet of the agency process depicted in Fig. 5.11(a). For two airline instances, Figure 5.11(b) depicts the resulting subnet. The trip order message (o) sent by the traveler to the agency is an example of the first scenario, as both services (traveler and agency) are uninstantiated. Therefore, the receipt of the trip order message is modeled by a single transition. The price request (p_1 and p_2) sent to and the corresponding price quotes (q_1 and q_2) received from the airline instances are examples for the second scenario. Therefore, the communicating transitions are instantiated, resulting in renamed message channel names. As specified by the parallel `forEach` activity, the agency communicates concurrently with each airline. The ticket order sent to only one airline instance (t_1 and t_2) is an example for the third scenario.

TRANSLATING THE EXAMPLE CHOREOGRAPHY

The presented translation approach is implemented in our compiler BPEL2oWFN [107]. BPEL2oWFN enables us to automatically translate WS-BPEL choreographies into service net models. We translated the example choreography with 10 airline instances into a Petri net, cf. Fig. 5.12. The resulting net has 188 places and 151 transitions. Standard structural reduction techniques [142] simplified the net to 113 places and 76 transitions while preserving compatibility.

In practice, we do not translate the intermediate WS-BPEL services into service automata, but compose the intermediate service nets. The definition of this service net composition operator is a straightforward adaption of Def. 2.3; Wolf [189] provides a formal definition.

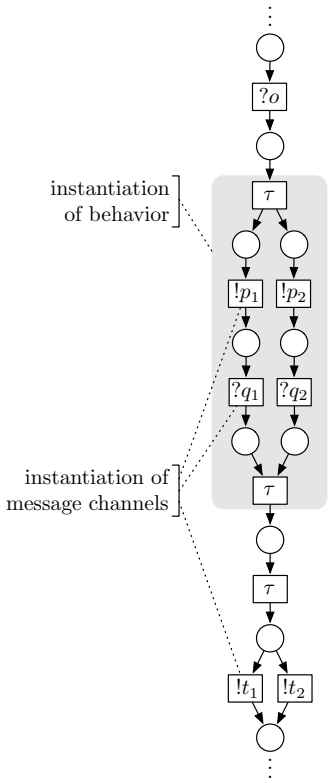
5.3 ANALYZING CLOSED CHOREOGRAPHIES

A BPEL4Chor choreography description specifies not only the behavior of each participant, but also their interaction. In addition, the closed-world assumption ensures that there is no further entity influencing the behavior of the participants. As a result, a complete BPEL4Chor choreography description can be translated into a closed service automaton (i. e., a service automaton with closed interface). Such a closed system can be analyzed without the necessity of taking an environment into account.

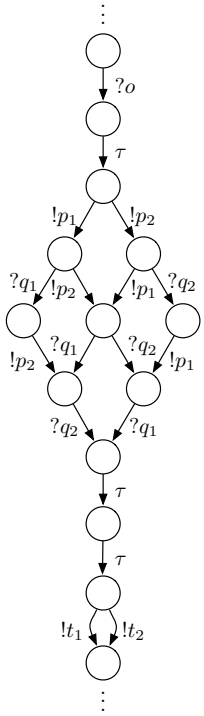
The correctness of a BPEL4Chor choreography is crucial, because it is the basis for technical groundings as well as manual refinement (cf. Fig. 5.6). By checking this

```
...
<receive wsu:id="ReceiveTripOrder" />
<forEach wsu:id="fe_RequestPrice" parallel="yes">
  <scope>
    <sequence>
      <invoke wsu:id="RequestPrice" />
      <receive wsu:id="ReceiveQuote" />
    </sequence>
  </scope>
</forEach>
<opaqueActivity name="SelectAirline" />
<invoke wsu:id="OrderTickets" />
...
```

(a) code snippet of the agency process



(b) resulting subnet of the agency



(c) resulting part as service automaton

Figure 5.11: Example for the instantiation for two airline instances. The WS-BPEL process of the agency (a) translated into a Petri net (b) and a service automaton (c).

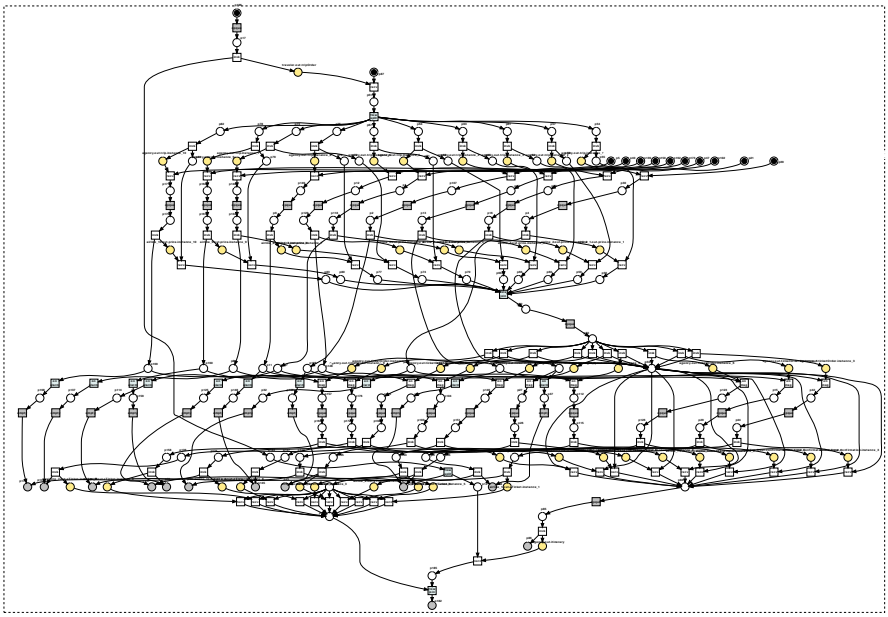


Figure 5.12: Service composition with 10 airline instances translated into a service net using the compiler BPEL2oWFN.

BPEL4Chor choreography, we can rule out errors well before refinement, implementation, and deployment.

As motivated in Chap. 2, we employ compatibility as central correctness criterion for closed service compositions. It allows us to derive more elaborate concepts such as controllability, behavioral constraints, or operating guidelines. Beside compatibility, temporal logics allow to express several other properties of closed systems which can be investigated using standard model checking tools [39, 15]. Interesting questions include:

– Will a certain activity of a participant be executed?

– Does there exist a state in which more than one message is pending on a communication channel?

– What is the minimal/maximal number of messages to be sent to reach a final state of the choreography?

– Will a participant always receive an answer? Can a participant enforce the receipt of a certain message?

These properties focus on closed systems and are not applicable to open systems in which an environment has to be taken into account. In this situation, the application of behavioral constraints (cf. Chap. 3) may help to investigate and validate the participants' behavior, but the results cannot be straightforwardly mapped back to the original WS-BPEL or BPEL4Chor model.

ANALYZING THE EXAMPLE CHOREOGRAPHY

We analyzed the Petri net model of Sect. 5.2 with the Petri net verification tool LoLA [168, 185], a state-of-the-art model checker which implements several state space reduction techniques. The unreduced state space consists of 9,806,583 states. Using LoLA, we detected a deadlock in the model. We could map this deadlocking state of the model back to the participating services with the help of a *witness path*. The deadlock occurs, if the agency's choice for an airline takes too much time or if the message sent to the chosen airline is delayed. In this case, the timeout (i. e., the `onAlarm` branch) of *all* participating airlines ends their instances and the agency deadlocks waiting for a confirmation message from the chosen airline.

Even though the presented example does not have the complexity of industrial service choreographies, the design flaw is very subtle and was not detected by the authors of the paper [50], where the example was taken from. Admittedly, our formalization abstracted from time and models timer-based decisions by nondeterminism. Nevertheless, the detected deadlock models a situation in which all airline instances time out, which may happen independently of a concretely chosen timeout interval. In addition, the latency of messages is hard to predict when asynchronous message transfer over the Internet is used to interact.

CORRECTING THE EXAMPLE CHOREOGRAPHY

There are many ways to correct the deadlocking choreography. A straightforward attempt would be to replace the airline service's timeout by a message sent by the agency, which explicitly informs all but one airline that their price quote was not chosen. This would, however, add an unrealistic dependency between the agency and all running airline instances. To this end, we decided to keep the timeout, but at the same time ensure a response of the airline service even if a ticket order is received after the timeout.

Hence, we changed the choreography as follows (cf. gray shapes in Fig. 5.13). The airline's behavior does not change if the agency's ticket order is received before the timeout occurred and if the timeout occurs, the airline service's instance still terminates. However, a new branch was added to the airline: *this branch models the situation in which the agency's ticket order is received after the timeout. In this case, the airline service is restarted and the ticket order is rejected.* In addition, the services

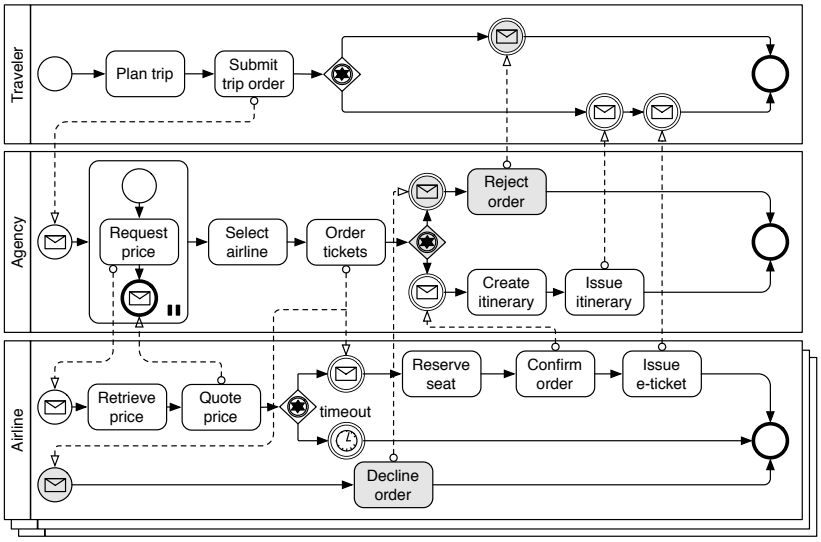


Figure 5.13: Fixed choreography of the ticket booking scenario. The two start events at the airline process denote a WS-BPEL `pick` activity.

of the agency and the traveler are adjusted to handle the case in which all airlines time out and no ticket could be booked.

Note that BPMN can only specify message flow between exactly two activities and does not support the concept of message channels. In the fixed choreography however, the ticket order sent by the agency can be received by two message events of the airline. We denote this in Fig. 5.13 by a branching message flow originating in the "order tickets" activity of the travel agency.

ANALYZING THE FIXED EXAMPLE CHOREOGRAPHY

We translated the fixed choreography with five airline instances into a Petri net model. Because of the newly introduced activities, its structure and its state space have grown. The resulting (structurally reduced) net has 113 places and 97 transitions. The model has 9,805,560 states and is now compatible.

The correction is nontrivial and possibly error-prone. To ensure compatibility, an additional check is needed. In the next chapter, we shall provide an algorithm to automatically suggest corrections for incorrect choreographies.

Table 5.1: Experimental results for compatibility check using LoLA.

first example, cf. Fig. 5.7

airline instances	1	5	10	100	1,000
net places	20	63	113	1,013	10,013
net transitions	10	41	76	706	7.006
states (unreduced)	14	3,483	9,806,583	—	—
states (symmetry)	14	561	378,096	—	—
states (POR)	11	86	261	18,061	1,752,867
states (POR + symmetry)	11	30	50	410	4,010

second example, cf. Fig. 5.13

airline instances	1	5	10	100	1,000
net places	19	63	113	1,013	10,113
net transitions	12	52	97	907	9,007
states (unreduced)	13	3,812	9,805,560	—	—
states (symmetry)	13	704	329,996	—	—
states (POR)	12	88	228	8,361	734,049
states (POR + symmetry)	12	28	43	314	3,014

EXPERIMENTAL RESULTS

In the previous sections, we analyzed the first and the second choreography (cf. Fig. 5.7 and Fig. 5.13, resp.) with five airline instances. For these five airlines, the resulting models already had more than 3,000 states. The states space grows dramatically when the number of airlines is further increased (cf. Tab. 5.1). For ten airlines, the model has over nine million states, and for larger numbers, the full state space could not be constructed due to memory overflow (denoted by "—" in Tab. 5.1). The experiments conducted using a computer with 2 gigabytes of memory.

However, several state space reduction techniques can be applied to reduce the size of the state space while still being able to analyze desired properties such as deadlock-freedom. In our particular example, we applied *symmetry reduction* and *partial order reduction*, both implemented in LoLA (Wolf [185] provides further references). The symmetry reduction exploits that all airline instances have the same structure. This regular structure induces symmetries on the net structure itself, but also on the state space of the choreography. Intuitively, the instances of the airline service act "similar" or "symmetric". During the state space construction, symmetric states are merged. The partial order reduction follows a different approach: As all instances run concurrently, any order of transitions of the airline instances are represented in the state space. These transition sequences introduce an exponential number of intermediate states, resulting in state space explosion. However, the actual order

of independent actions is not relevant to detect deadlocks, for instance. To this end, partial order reduction tries to only construct a single transition sequence of transitions of different airline instances to ease the state space explosion.

In case each of the reduction technique is applied in isolation, the number of states grows more slowly, yet still exponentially in the number of airline instances. The combination of both techniques, however, yields a linear increase of states (cf. Tab. 5.1). Hence, we are able to verify properties of WS-BPEL choreographies with thousands of participating services. This shows that the presented approach should be likewise suitable to analyze real-life examples. The numbers show that the correction of the model only yields few additional states.

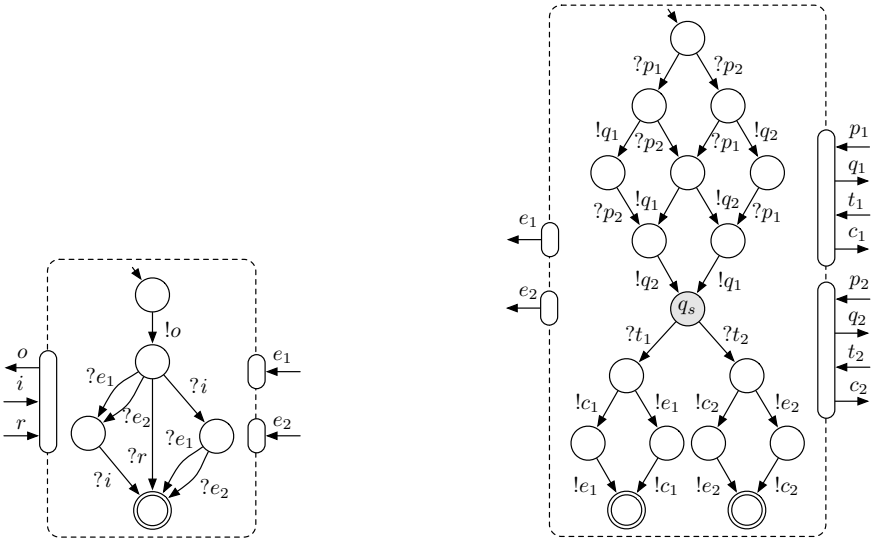
5.4 COMPLETING CHOREOGRAPHIES

While the analysis of closed choreographies may help to find errors such as deadlocks in the interaction between the participating services, service automata may also support the *design* of choreographies. A choreography in which one participating service is missing can, for instance, be *completed* by automatically synthesizing the missing participant service. This synthesized service is then guaranteed to communicate compatibly with the other participants. To this end, controllability is an important property. In Chap. 2, we presented an algorithm to constructively decide controllability of an open service automaton. This algorithm is implemented in the tool Wendy [121]. If a partner exists such that the composition is compatible, it is automatically generated.

SYNTHESIZING A TRAVELER PARTICIPANT

Consider again the fixed choreography of Fig. 5.13. If, for example, only the services of the agency and the airlines were specified, the blueprint of a traveler participant could be synthesized. If such a service exists (i. e., the composition of the existing services is controllable), it completes the choreography which is then compatible by construction. To this end, the incomplete choreography is translated into a service net using BPEL2oWFN. This service automaton is then analyzed by Wendy. If the service automaton is controllable, a service automaton modeling the behavior of a partner service is synthesized.

EXAMPLE. Figure 5.14(a) depicts the synthesized service automaton of a traveler participant which completes the choreography. This traveler participant slightly differs from the traveler participant in the repaired choreography (cf. Fig. 5.13). First, there exists no transition modeling the planning of the trip, because such a transition is internal (i. e., not communicating), but the participant was synthesized based on the external behavior; that is, only the interaction of the service was taken into account. Second, the itinerary and the e-ticket can be received in any order. For example, it



(a) synthesized traveler

(b) two synthesized airline instances

Figure 5.14: Participant services synthesized to complete the example choreography.

would be possible to swap the two receive events in Fig. 5.13. This is because of the asynchronous communication model: messages can keep pending on the interface, so there is no order in which they have to be received. From this service automaton, an abstract WS-BPEL process can be derived using existing approaches [101, 3, 114]. As this translation is out of scope of this thesis, we do not present it here.

EXPERIMENTAL RESULTS

To investigate the applicability of the synthesis algorithm in this scenario, we conducted the following experiment. We fixed the travel agency and synthesized the traveler service for different numbers of airline instances.

Table 5.2 summarizes the results. The first two lines list the size of the structurally reduced service net modeling the composition of the airline instances and the travel agency. The next line lists the number of states of this open composition; that is, the size of the service automaton that is checked for controllability. The next two lines give information on the size of the synthesized traveler service. Finally, the time consumption of the synthesis is listed. With our test setup of 2 gigabytes of memory and a 3 GHz processor, we were able to synthesize a traveler service for up to 14 airline instances. For this number, the service automaton modeling the composition

Table 5.2: Experimental results for participant synthesis using Wendy.

airline instances	4	6	8	10	12	14
net places	35	49	63	77	91	105
net transitions	18	26	34	42	50	58
states	176	1,240	9,120	71,366	588,784	5,045,112
synthesized states	11	15	19	23	27	31
synthesized transitions	56	106	172	254	352	466
synthesis time [s]	0	0	0	3	36	378

of the travel agency and the airline instances has already more than five million states and the synthesis took more than six minutes.

Compared with the compatibility analysis (cf. Tab. 5.1), the synthesis problem does not scale well with respect to the number of airline instances that can be processed (14 vs. 1,000). This is because we currently do not apply state space reduction techniques. Hence, the size of the service automaton to be considered suffers from state explosion and grows exponentially in the number of the airline instances. However, without these techniques, also the compatibility analysis becomes unfeasible if the size of the state space exceeds about 10 million states.

The experiment also shows that just a small number of asynchronously communicating participants are enough to result in an open system which has much more states than industrial Web services (cf. Tab. 2.1). For the ticket booking scenario, the composition of a travel agency and ten airline instances has already four times more states than the largest service automaton we considered in Tab. 2.1.

LIMITS OF THE PARTICIPANT SYNTHESIS

The approach presented allows us to synthesize a participant that interacts in a compatible manner with the other participating services of the choreography. This is, of course, only possible if the open choreography is controllable and thus such a service exists. Currently, it is, however, not possible to synthesize a *set* of services which complete a choreography: Def. 2.9 synthesizes a single strategy. First ideas toward and extension of the synthesis algorithm to multiple services are described by Wolf [187]. We shall come back to this in Chap. 7.

As an example, consider again the first (deadlocking) choreography in Fig. 5.7. The choreography deadlocks because of the airline service's timeout mechanism. If we synthesize a strategy for the composition of the traveler and the travel agency, the result will be a *single* service automaton modeling the behavior of *all* airline service's instances. Figure 5.14(b) depicts this service automaton modeling two airline instances. It receives two price requests from the agency addressed to the different instances (p_1 and p_2) which reply with two price quotes (q_1 and q_2). Then, it waits

to receive a ticket order (either o_1 or o_2) and answers it accordingly (either c_1 or c_2). The resulting choreography would be compatible. However, the airline's instances are not independent of each other. They are implicitly synchronized in state q_s (depicted gray in Fig. 5.14(b)): after this state, only one of the airlines continues the interaction. If this service had to be split into two services (one for each instance), this synchronization would have to be made explicit by adding coordination messages to maintain compatibility. Still, the synthesized airline model can be seen as a starting point for further refinement. In the next chapter, we shall use synthesized strategies as a starting point to automatically propose correction for incompatible compositions. We shall again consider the resolution of dependencies between different choreography participants in Chap. 7 when we study interaction models.

Another aspect of the participant synthesis is the *causality* between messages. As sketched in the description of the generated traveler participant (cf. Fig. 5.14(a)), a generated participant may send and receive messages in different — typically less constrained — orders. This may yield synthesized services which send acknowledgment messages before actually receiving the corresponding request. In such cases, the causality between the request and the acknowledgment is ignored. Such causal effects have been studied by Wolf [187] who further considers semantics of messages [186]. In Chap. 3, we introduced behavioral constraints to rule out such implausible behavior.

5.5 RELATED WORK

As described in the introduction, choreography models can be grouped into interconnected models and interaction models. In this chapter, we only considered the former. We shall consider interaction models in Chap. 7.

WS-BPEL AND BPEL4CHOR. There exists a large number of formalizations of WS-BPEL (see [27, 120, 119] for surveys) which can all be similarly adjusted to model BPEL4Chor as long as they formalize the exchange of messages. Decker et al. [51] give a detailed evaluation of existing choreography languages and an assessment of the features of BPEL4Chor. Whereas we use BPMN for visualization purposes only, Decker et al. [49] provide an extension to BPMN to specify complete BPEL4Chor choreography using BPMN.

ANALYSIS. Compatibility of service compositions and service choreographies has already received much attention in the early days of Web services [193, 144, 81, 33, 69, 127, 159, 54]. Compatibility of a closed composition is closely related to — and motivated by — the *soundness* property of workflows [1, 180]. For soundness, a case study [68] shows that industrial process models can already be checked in few milliseconds using the tool LoLA. Beside verification, Mendling [135] applied empirical studies to *predict* errors in process models from the structure and the used language constructs.

Moser et al. [141] show how to synthesize a WS-BPEL process which properly interacts with a given WS-BPEL process. Decker et al. [52] present a formalization of BPEL4Chor, focusing on *service referrals* (also called link passing). They provide a mapping to the π-calculus, but give no details on possible verification.

REFINEMENT. The refinement from a grounded BPEL4Chor choreography to executable WS-BPEL processes has two aspects. On the one hand, technical details such as WSDL port types or data types have to be added to the participant descriptions. This process is described in Reimann et al. [162] and Decker et al. [51]. On the other hand, a refinement of a participant description should additionally allow a reorganization of the WS-BPEL process as long as compatibility of the overall choreography is preserved. This refinement of *public views* to *private views* is an important aspect in the design of interorganizational business processes and has been studied in Aalst et al. [4, 5]. König et al. [95] define compatibility-preserving transformation rules in terms of WS-BPEL.

5.6 CONCLUSION

In this chapter, we focused on the correctness of service compositions specified in BPEL4Chor. To formally reason about the correctness, we translated a BPEL4Chor choreography into service automata using Petri nets as an intermediate formalism. We thereby applied an existing formalization of WS-BPEL in terms of Petri nets and extended it to model BPEL4Chor choreographies.

A small example choreography demonstrated how subtle errors of choreographies can be. It motivated that the design and verification of compatible choreographies with a larger number of participants or more complex participant services are even more challenging if not impossible to do manually. The example further showed that controllability of each participant does not guarantee compatibility of the composition. As a result, the correctness of a composition of services which are correct (i. e., controllable) by itself needs to verified. In case a participant is uncontrollable, we can diagnose the reasons using the approach described in Chap. 4.

This chapter presented two contributions to the overall goal of this thesis: On the one hand, we illustrated how *correctness by verification* (i. e., a compatibility check) can be realized for choreographies specified in an industrial service language. On the other hand, we showed how choreographies can be completed using strategy synthesis to achieve *correctness by design*.

The experiments on the compatibility check show that a combination of state space reduction techniques known from Petri net theory can effectively tackle the problem of the state space explosion. In the concrete example, the technique scaled to up to a thousand participants. The experimental results might not be directly applicable to real-world choreographies which usually consist of much less participants which in

turn have a more complex behavior. Nevertheless, it gives an idea on the suitability of the tool LoLA as compatibility checker.

By using strategy synthesis to complete choreography models, we use a verification technique to support the modeling of choreographies. Even though the synthesized participant for an incomplete choreography is only a "stub" or "communication skeleton", it is correct by design and avoids an error-prone manual specification. The synthesis technique does not scale as good as the compatibility check, because currently no state space reduction techniques are used. Notwithstanding, we see high potential in this technique to support the design of correct choreographies in early stages.

Finally, the analysis and synthesis approach presented in this chapter are independent of WS-BPEL as input language as the approaches are based on the formal model of Petri nets and service automata. Therefore, the presented techniques can be easily adapted to future service description languages.

A long-term goal is to tightly integrate verification into a modeling tool such that the model can be constantly checked in the background to provide feedback as early as possible. This helps the modeler to relate errors to recent edit actions and to quickly correct these errors. For the soundness criterion, this goal is more or less achieved [68]. For the compatibility check, the experiments of Sect. 5.3 provided promising results, which need to be validated using a case study with industrial service compositions.

Beside scalability and runtime, the presentation of detected errors is an important topic for compatibility checks. This is in particular challenging, as a WS-BPEL has no concept of states or state transitions. To this end, a counterexample needs to be mapped on the participating WS-BPEL processes or their BPMN visualizations to help the modeler locate the error, for instance by coloring executed or blocked activities.

The completion of choreographies requires a retranslation of synthesized service automata into the original input language, for instance WS-BPEL. First approaches in this area [101, 3, 114] focus on the translation of a single Petri net model into an abstract WS-BPEL process. This translation can be improved by incorporating information about the participant topology into the translation process to refine the resulting WS-BPEL process.

Finally, our experiments showed that the synthesis algorithm is — compared with the compatibility check — still in its infancy. To be able to process larger models, it is crucial to integrate state space reduction techniques into the synthesis tool Wendy. These techniques are orthogonal to the reduction techniques presented by Weinberg [184] (cf. Def. 4.2 and Tab. 4.1), which aim at reducing the size the synthesized strategy. These techniques do not avoid the exploration of the full state space of a given service net or service automaton.

6 CORRECTION

This chapter is based on results published in [108].

In the previous chapter, we focused on the verification of service compositions. Experimental results showed that it is possible to detect errors in service compositions with millions of states. In case an error was found, a counterexample is returned which shows how compatibility is violated.

Whereas errors can be *detected* automatically (i. e., with tool support), the *correction* of defective services is usually done manually. Correction steps include investigating the counterexample, determining which participant contains a design flaw, locating the error in the participant model, and finally fixing it. In addition, a subsequent verification is required to prove that the modification really corrected the service composition.

These correction steps are tedious, error-prone, and expensive, because they involve manual interference with the service composition. Hence, it would be desirable to automate the correction of incorrect service compositions to some extent. This is especially crucial, because fixing incorrect services is usually cheaper and takes less time than redesigning and implementing a correct service from scratch. In addition, information on how to adjust an existing service can help the designer *understand* the error more easily compared to confronting him with an entirely newly synthesized service. We shall introduce a graph-based approach to calculate the minimal edit distance between a given defective service and synthesized correct services. This edit distance may help to automatically fix found errors while keeping as much of the service as possible untouched.

In this chapter, we formalize, systematize, and to some extent automate the correction of service compositions. We thereby combine existing work on operating guidelines to characterize all strategies of a service (cf. Chap. 2) with similarity measures and edit distances known in the field of graph correction. We give a motivating example in Sect. 6.1 and briefly sketch the correction approach in Sect. 6.2, before graph similarities are reviewed in Sect. 6.3. In Sect. 6.4, we define an edit distance which aims at finding the *most similar* service from the set of all fitting services. To support the modeler, we further derive the required edit actions needed to correct the originally incorrect service. In Sect. 6.5, we present experimental results conducted with an implementation of the approach that serves as a proof of concept. Section 6.6 discusses related work. Finally, Sect. 6.7 is dedicated to a conclusion and gives directions for future research.

6.1 MOTIVATING EXAMPLE

As the running example for this chapter, consider an example choreography in Fig. 6.1, which is similar to the example of the previous chapter and again visualized in BPMN [150]. It describes the interplay between a travel agency, a customer service, and an airline reservation system. The travel agency sends an offer to the client which either rejects it or books a trip. In the latter case, the travel agency orders a ticket at the airline service which either sends a confirmation or a decline message to the customer. The choreography contains a design flaw as the customer service does not receive the decline message. This leads to a deadlock in case the airline declines the ticket order, because the customer is not able to receive a decline message from the airline, but waits for a confirmation instead.

This incompatibility can be detected using state-of-the-art model checking tools which provide a trace to the deadlocking state, cf. Chap. 5. A concrete counterexample depends on the name of the states and transitions of the service automata modeling the choreography. At the level of detail of the depicted BPMN model, it could be

> 1. send offer, 2. receive offer, 3. send booking, 4. send payment, 5. receive booking, 6. receive payment, 7. send ticket order, 8. receive ticket order, 9. send decline.

This trace, however, gives no insight *which service* has to be changed in *which manner* to avoid the deadlock. Thus, an iteration of manual corrections followed by further checks is necessary to finally remove the deadlock. Even though it is obvious how to correct the flawed example, the manual correction of choreographies of a larger number of more complex services is complex and error-prone, if not impossible.

Moreover, even for this simple choreography there exists a variety of possibilities to correct the customer's service. Figure 6.2 depicts two possible corrections to achieve compatibility. Although either service would guarantee compatibility, the service in Fig. 6.2(a) is to be preferred over the one in Fig. 6.2(b) as it is "more similar" to the original service. Albeit this preference is psychological and is unlikely to be rigorously formalizable, the usage of similarities is accepted in the area of error explanation [77]. The tool chain presented in the previous chapter synthesizes a participant service independently of an existing incorrect service which is either most-permissive (cf. Def. 2.9) or reduced (cf. Def. 4.2 and [184]). Whereas the former most-permissive strategy is usually much larger than a manually specified service, the latter result may be a correct, yet unintuitive result such as the service in Fig. 6.2(b). Hence, the remainder of this chapter is dedicated to the synthesis of a service which not only ensures compatibility of the overall composition, but also is as close to the (incorrect) original service as possible.

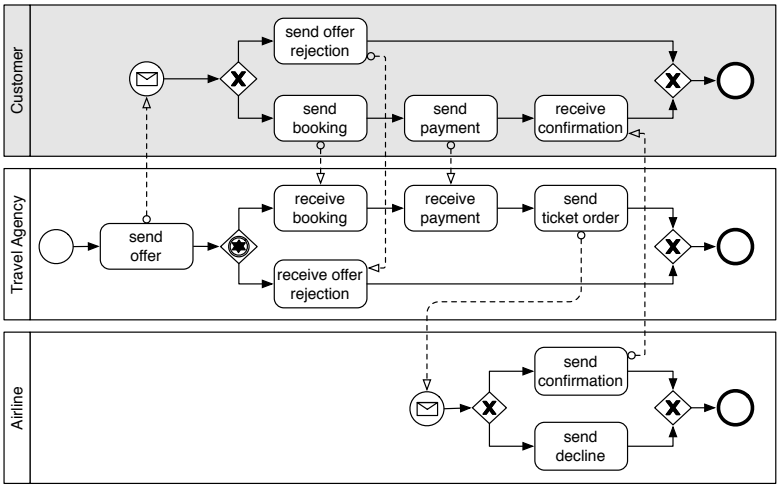
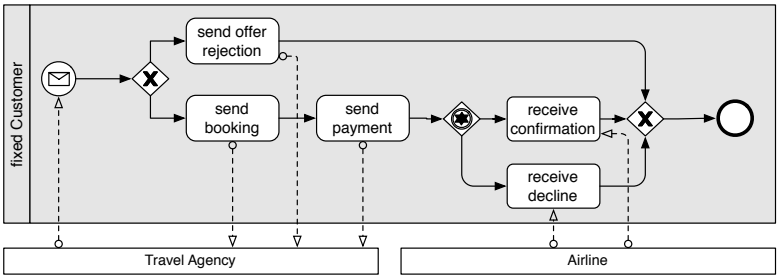
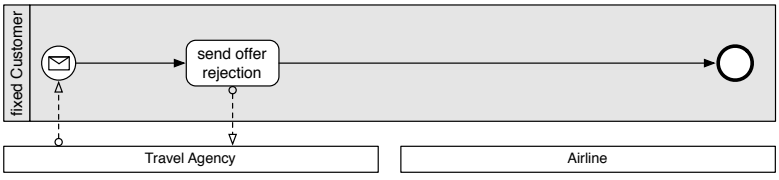


Figure 6.1: Incompatible choreography.



(a) add branch to receive the decline message



(b) delete the booking branch

Figure 6.2: Possible corrections of the customer service to achieve compatibility.

6.2 CORRECTING INCOMPATIBLE CHOREOGRAPHIES

In the remainder of this chapter, we show how the correction procedure of an incompatible service choreography can be supported by automatically providing recommendations for the modeler. This procedure includes the calculation of the candidates for the correction on the one hand and the choice which candidate to take can be automated on the other hand. To provide some intuition, we show how the choreography *completion* described in Chap. 5 can also used to *correct* choreographies.

Consider an incompatible choreography of n participants, $A_1 \oplus \cdots \oplus A_n$. As mentioned before, a counterexample (e. g., a deadlock trace) usually does not give enough information *how* to fix *which* service to achieve compatibility. To find a candidate service which can be changed such that the entire choreography is compatible, we propose the following steps:

1. We check for each service the necessary correctness criterion: If a service taken for itself is not controllable, then there exists no environment in which this service runs correctly — in particular not the choreography under consideration. In that case, that service has to be radically overworked toward controllability using the diagnosis algorithm of Chap. 4.
2. We remove one participant, say A_i. The resulting choreography $Chor_i := A_1 \oplus \cdots \oplus A_{i-1} \oplus A_{i+1} \oplus \cdots \oplus A_n$ can be considered as one large service with an interface to A_i. If this large service is controllable, then there exists a service A'_i which interacts in a compatible manner with the other participants of the choreography; that is, $Chor_i \oplus A'_i$ is compatible. In Chap. 5, we presented a complete tool chain for this participant synthesis for WS-BPEL-based choreographies. We shall discuss the case in which $Chor_i$ is uncontrollable later.

As motivated in the introduction, the mere replacement of A_i by A'_i is not desirable, because A'_i is synthesized independently of the faulty service A_i and totally ignores its structure. Hence, it may be very different to the original, yet incorrect service A_i. Instead of synthesizing *any* fitting service (such as the service in Fig. 6.2(b)), we are interested in a corrected service which is most similar to A_i. To this end, we can use the operating guideline of $Chor_i$, because it characterizes the set of *all* fitting partners. Figure 6.3 illustrates this.

It is important to stress that any automated method can only provide *suggestions* to change a model, and these suggestions always need to be evaluated manually. To this end, the suggestions should be as local as possible.

The problem statement of this chapter is as follows: Given an incompatible service composition $A_1 \oplus \cdots \oplus A_n$ (e. g., the choreography in Fig. 6.1) and a fault service A_i (i. e., a "scapegoat" such as the customer service in Fig. 6.1) such that $Chor_i$ is controllable, what are minimal edit actions to change A_i to A^*_i such that $A_1 \oplus \cdots \oplus A_{i-1} \oplus A^*_i \oplus A_{i+1} \oplus \cdots \oplus A_n$ is compatible?

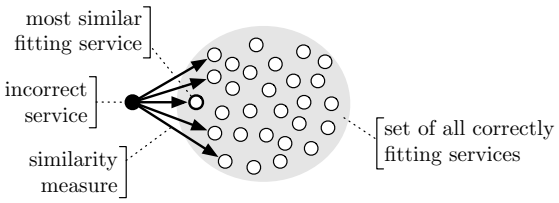


Figure 6.3: The operating guideline as characterization of all correct services can be used to find the most similar correct service.

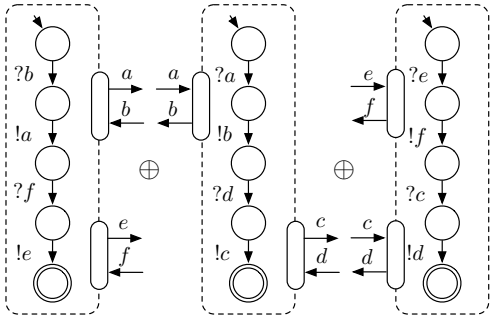
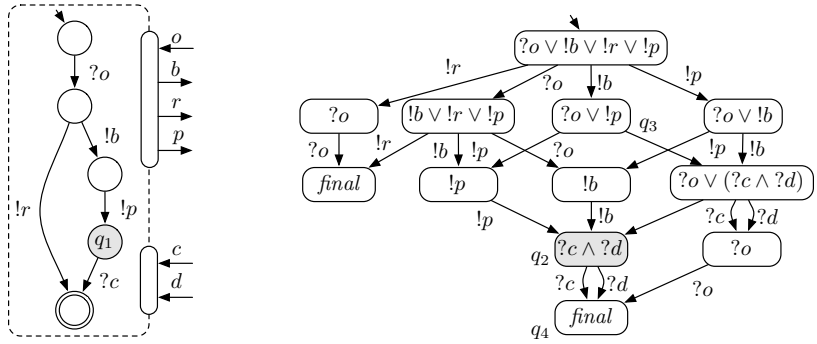


Figure 6.4: An incompatible composition of controllable services.

Unfortunately, controllability of each participating services $A_1, \ldots, A_n$ does not guarantee controllability of $Chor_i$. Figure 6.4 shows an incompatible composition of three controllable services in which the removal of any single service yields an uncontrollable service. This is because of a cyclic dependency between any pair of participants. As any pair of remaining services is uncontrollable, no suggestion can be derived from the composition which service needs to be repaired. In such a situation, we need to diagnose the reasons which lead to uncontrollability of $Chor_i$, choose a different service to repair, or remove a second service. In the remainder of this chapter, *we assume that we can identify a single service for correction.*

EXAMPLE. Figure 6.5(b) depicts an operating guideline of the composition of the travel agency and the airline. The service automaton of Fig. 6.5(a) is structurally matched by the operating guideline and satisfies all but one formula: It does not satisfy the formula $\varphi(q_2) = ?c \wedge ?d$ of the operating guidelines's state q_2, because the service automaton does not receive a decline message (d) in the matched state q_1.

Beside the two corrected services in Fig. 6.2, the operating guideline characterizes 2,302 additional (acyclic, deterministic, and τ-free) strategies (up to isomorphism). This number can be derived from the connected subgraphs of the operating guideline and the labels which satisfy the annotated formulae. We use this number as an



(a) A_{cust} (b) $OG^1_{A_{\text{agency}} \oplus A_{\text{airline}}}$

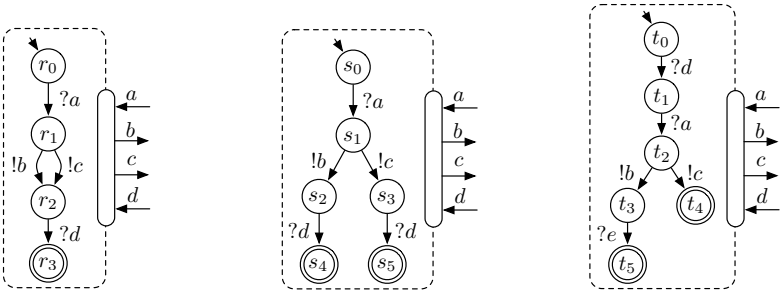
Figure 6.5: The service automaton (a) modeling the customer from Fig. 6.1 and an operating guideline (b) of the composition of the travel agency and the airline service from Fig. 6.1.

approximation, because the set of cyclic or nondeterministic partner services is usually infinite. Alhough each of these services is correct, we are interested in the service which is most similar to the incorrect customer service; that is, instead of iteratively checking an unreasonably high number of candidates, we shall define a similarity measure which exploits the operating guideline's compact representation to efficiently find the desired service of Fig. 6.2(a).

6.3 GRAPH SIMILARITIES

Graph similarities are widely used in many fields of computer science, for example for pattern recognition [167], semantic Web, document retrieval, or in bio informatics. Graph similarities are *quantitative* measures and express the similarity of two graphs in a single value. To gain more insight in the reasons of (un)similarity, cost-based distance measures adapt the *edit distance* known from string comparison [104, 181] to compare labeled graphs [176, 35, 34]. They aim at finding the minimal number of modifications (i. e., adding, deleting, and modifying nodes or edges) needed to achieve a graph isomorphism.

Distance measures aiming at graph isomorphism have the drawback that they are solely rely on the *structure* of the graphs. That is, they focus on the syntax of the graphs rather than their semantics. In case a graph (e. g., a service automaton) models the *behavior* of a system, similarity of graphs should focus on similar behavior rather than on similar structure. Figure 6.6 illustrates that structural and behavioral similarity are not related.



(a) A_r (b) A_s (c) A_t

Figure 6.6: Service automata A_r and A_s simulate each other, but have an unsimilar structure. Service automata A_s and A_t have a similar structure, but very different behaviors.

Sokolsky et al. [171] address this problem (a similar approach is presented by Nejati et al. [146]), motivated by finding computer viruses in a program. The idea is to compare the control flow graph of the program with a library of control flow graphs of known computer viruses and to warn if a certain threshold is exceeded. In that setting, a classical simulation relation as comparison between behavior is too strict, because two systems which are equal in all but one edge label *behave* very similarly, but there exists no simulation relation between them. To this end, Sokolsky et al. introduce a *weighted quantitative simulation* function to compare states of two graphs. Whenever the two graphs cannot perform a transition with the same label, one graph performs a special stuttering step ε, which is similar to τ-steps in stuttering bisimulation [143]. To "penalize" stuttering, a label similarity function assigns low similarity between ε and any other label.

Definition 6.1 (Similarity function, discount factor).
For a set of message events $\mathbb{E}$ and a *stuttering event* $\varepsilon \notin \mathbb{E}$, a *similarity function* is a function $L : (\mathbb{E} \cup \{\tau, \varepsilon\}) \times (\mathbb{E} \cup \{\tau, \varepsilon\}) \to [0, 1]$. A *discount factor* is a value $p \in [0, 1]$.

A label similarity function assigns a value that expresses the similarity between the labels of the service automata under consideration. For example, $L(?a, ?b)$ describes the similarity of an $?a$-labeled transition of service automaton A_1 and a $?b$-labeled transition of service automaton A_2. Furthermore, a discount factor $p \in [0, 1]$ describes the local importance of similarity compared with the similarity of successor states. This discount "smoothens" the simulation results by not only considering local similarity (e. g., by comparing the similarity of labels of outgoing edges), but also the future similarity (i. e., the similarity of successor states). Both L and p will influence

the upcoming definitions to calculate similarities and edit distances. Their values can be chosen freely to adjust the result of the similarity algorithm. The concrete choice of the parameters needs further empirical investigation and is therefore not considered here.

The following definition determines the similarity of two states of A_1 and A_2 by choosing which labels should synchronize. This evaluation is influenced by the label similarity function L and the recursive similarity of the successor states. The label ε further allows one service automaton to stutter rather than to synchronize.

Definition 6.2 (Weighted quantitative simulation, [171]).
For $i \in \{1,2\}$, let $A_i = [Q_i, q_{0_i}, \to_i, \Omega_i, \mathcal{P}_i]$ be service automata. A *weighted quantitative simulation* is a function $S : Q_1 \times Q_2 \to [0,1]$, such that:

$$S(q_1,q_2) := \begin{cases} 1, & \text{if } q_1 \not\to_1 , \\ (1-p) + p \cdot \max\Big(W_1(q_1,q_2), \dfrac{1}{n} \cdot W_2(q_1,q_2)\Big), & \text{otherwise,} \end{cases}$$

$$W_1(q_1,q_2) := \max_{q_2 \xrightarrow{b}_2 q_2'} \Big(L(\varepsilon,b) \cdot S(q_1,q_2')\Big),$$

$$W_2(q_1,q_2) := \sum_{q_1 \xrightarrow{a}_1 q_1'} \max\left(L(a,\varepsilon) \cdot S(q_1',q_2), \max_{q_2 \xrightarrow{b}_2 q_2'} \Big(L(a,b) \cdot S(q_1',q_2')\Big)\right),$$

and n is the number of edges leaving q_1. The weighted quantitative simulation between A_1 and A_2 is defined as $S(q_{0_1}, q_{0_2})$.

The weighted quantitative simulation function S recursively compares the states from the two service automata and finds the maximal similar edges. Thereby, W_1 describes the similarity gain by stuttering of service automaton A_1 on the one hand, and W_2 the tradeoff between simultaneous transitions of A_1 and A_2 and stuttering of service automaton A_2 on the other hand. A sink state of A_1 (i. e., a state without successors) has a maximal similarity with any state of A_2, because there are no obligations for A_2 to simulate this state.

Sokolsky et al. [171] proved that a unique fixed point for S exists and that S generalizes classical simulation: If A_1 is simulated by A_2, then $S(q_{0_1}, q_{0_2}) = 1$, and if A_1 is not simulated by A_2, then $S(q_{0_1}, q_{0_2}) < 1$. In addition the authors provided a linear programming algorithm to calculate the weighted quantitative simulation for arbitrary finite state automata. We adjusted the definitions of [171] to service automata. The original definitions are based on labeled directed graphs and additionally take node labels and similarities between node labels into account. This is not required in the context of this chapter, but may be exploited in the future to further refine the results.

EXAMPLE. Consider the service automata in Fig. 6.6 and assume a discount factor $p = 0.7$ and a label similarity function L, which assigns 1.0 to equal labels and 0.5 to any other label pair. Then $S(r_0, s_0) = 1.0$ (the weighted quantitative simulation is a generalization of the classical simulation) and $S(s_0, t_0) = 0.589975$ which indicates the differences in the behaviors of A_s and A_t. This latter result can be calculated as follows (only the local maxima are shown):

$$
\begin{aligned}
&S(s_0, t_0) = (1-p) + p \cdot L(\varepsilon, ?d) \cdot S(s_0, t_1) = 0.589975\\
&S(s_0, t_1) = (1-p) + p \cdot L(?a, ?a) \cdot S(s_1, t_1) = 0.8285\\
&S(s_1, t_1) = (1-p) + \tfrac{p}{2} \cdot \big(L(!b, !b) \cdot S(s_2, t_4) + L(!c, !c) \cdot S(s_3, t_4)\big) = 0.755\\
&S(s_2, t_3) = (1-p) + p \cdot L(?d, !?e) \cdot S(s_4, t_5) = 0.65\\
&S(s_3, t_4) = (1-p) + p \cdot L(?d, \varepsilon) \cdot S(s_5, t_4) = 0.65\\
&S(s_4, t_5) = 1\\
&S(s_5, t_4) = 1
\end{aligned}
$$

Intuitively, Def. 6.2 can be seen as a system of equations, having the values of S as variables. Some variables (e. g., $S(s_0, t_0)$) depend on other variables, whereas other variables (e. g., $S(s_5, t_4)$) do not. As illustration, consider the definition of $S(s_0, t_0)$:

$$
\begin{aligned}
\boxed{S(s_0,t_0)} = (1-p) + p \cdot \max\big(&\overbrace{L(\varepsilon, ?d) \cdot \boxed{S(s_0,t_1)}}^{W_1(s_0,t_0)},\\
&\underbrace{\frac{1}{n} \cdot \big(\max(L(?a,\varepsilon) \cdot \boxed{S(s_1,t_0)},\ L(?a,?d) \cdot \boxed{S(s_1,t_1)}\big)}_{W_2(s_0,t_0)}\big)
\end{aligned}
$$

The framed values represent variables of the equation system: the value of $S(s_0, t_0)$ depends on the values of $S(s_0, t_1)$, $S(s_1, t_0)$, and $S(s_1, t_1)$.

6.4 A MATCHING-BASED EDIT DISTANCE

The weighted quantitative simulation of Def. 6.2 can be used as a similarity measure for service automata or operating guidelines, but has two drawbacks: First, it is not an edit distance. It calculates a single value which expresses the similarity between the service automata, but gives no information about the modification actions needed to *achieve* simulation. Second, it does not take formulae of the operating guideline into account. Therefore, even a perfect similarity (which is closely related to structural matching) between a service automaton and an operating guideline would not guarantee compatibility as the example of Fig. 6.5 demonstrates: The service automaton of the customer is structurally matched by the operating guideline but the overall choreography deadlocks.

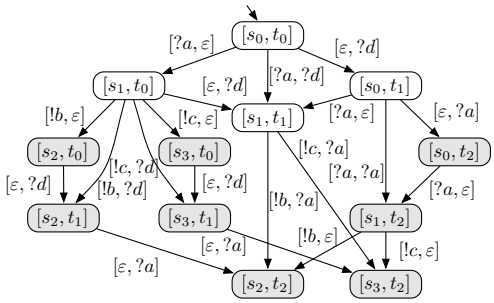


Figure 6.7: Part of the synchronization graph $A_s \odot A_t$.

SIMULATION-BASED EDIT DISTANCE

Before we consider the operating guideline's formulae, we show how the similarity metric of Def. 6.2 (i. e., the result of the algorithm of [171]) can be transformed into an edit distance.

Given two states q_1 and q_2 of two service automata A_1 and A_2, Def. 6.2 determines the similarity $S(q_1, q_2)$ by choosing pairs of labels of transitions leaving q_1 and q_2, respectively. Each pair of labels $[a, b]$ with $q_1 \xrightarrow{a}_1 q_1'$ and $q_2 \xrightarrow{b}_2 q_2'$ determines successor states whose similarity is then recursively determined by $S(q_1', q_2')$. The similarity of q_1 and q_2 is then calculated by locally maximizing the successor state's similarity. From Def. 6.2, we can derive a graph consisting of the state pairs as nodes and the chosen label pairs as transitions:

Definition 6.3 (Synchronization graph).
Let $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ and $B = [Q_B, q_{0_B}, \rightarrow_B, \Omega_B, \mathcal{P}_B]$ be service automata. The *synchronization graph* of A and B is the tuple $A \odot B = [Q, q_0, \rightarrow]$ consisting of

- $Q := Q_A \times Q_B$,
- $q_0 := [q_{0_A}, q_{0_B}]$,
- $\rightarrow$, containing exactly the following elements:
  - $[q_A, q_B] \xrightarrow{[x,y]} [q_A', q_B']$ iff $q_A \xrightarrow{x}_A q_A'$ and $q_B \xrightarrow{y}_B q_B'$,
  - $[q_A, q_B] \xrightarrow{[x,\varepsilon]} [q_A', q_B]$ iff $q_A \xrightarrow{x}_A q_A'$, and
  - $[q_A, q_B] \xrightarrow{[\varepsilon,y]} [q_A, q_B']$ iff $q_B \xrightarrow{y}_B q_B'$.

Intuitively, this graph has the variables of the previously motivated equation systems as nodes. A transition $[q_A, q_B] \xrightarrow{[x,y]} [q_A', q_B']$ states that the value of $S(q_A, q_B)$ depends on the value of $L(x, y) \cdot S(q_A', q_B')$.

EXAMPLE. Figure 6.7 depicts the synchronization graph of the service automata A_s and A_t (cf. Fig. 6.6) to the depth of 2. That is, we do not depict all successor states of the shaded states.

The synchronization graph can be seen as the search space for the optimal result calculated by Def. 6.2. As stated before, we are not just interested in a single value expressing the similarity of two service automata, but in instructions how to change a service automaton to achieve a structural matching. As intermediate result, we restrict the labels of the synchronization graph as follows: from the determined label pairs $[x, y]$ only the second part, y, is kept. Thereby, ε-labels are replaced by τ.

Definition 6.4 (Synchronization graph restriction).
Let $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ and $B = [Q_B, q_{0_B}, \rightarrow_B, \Omega_B, \mathcal{P}_B]$ be service automata and $A \odot B = [Q_{AB}, q_{0_{AB}}, \rightarrow_{AB}]$ their synchronization graph. We define the *restriction* of $A \odot B$ to B as the service automaton $(A \odot B)_{|B} := [Q_{AB}, q_{0_{AB}}, \rightarrow, \Omega, \mathcal{P}_B]$ with:

- $\Omega := \Omega_A \times Q_B$ and
- $\rightarrow$ containing exactly the following elements:
  - $[q, y, q'] \in \rightarrow$ iff $[q, [x, y], q'] \in \rightarrow_{AB}$ and $y \neq \varepsilon$ and
  - $[q, \tau, q'] \in \rightarrow$ iff $[q, [x, y], q'] \in \rightarrow_{AB}$ and $y = \varepsilon$.

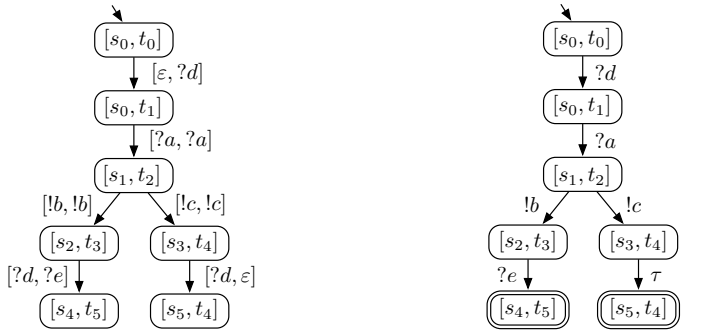
The restriction of the synchronization graph $A \odot B$ to the labels of the interface of service automaton B yields a service automaton $(A \odot B)_{|B}$, which structurally matches B. Additionally, $[q_A, q_B]$ is a final state of $(A \odot B)_{|B}$ iff q_A is a final state of A. Although final states are not considered by structural matching, they are later required to evaluate the operating guideline's formulae.

From the definition of structural matching (cf. Sect. 2.1), Def. 6.3, and Def. 6.4, we can derive the following result:

Corollary 6.1 (Restriction structurally matches.).
Let $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ and $B = [Q_B, q_{0_B}, \rightarrow_B, \Omega_B, \mathcal{P}_B]$ be service automata. Then $(A \odot B)_{|B}$ structurally matches B.

Definition 6.2 choses for each pair of states $[q_a, q_b]$ — or, for each state of the synchronization graph — those successors states where the local similarity is maximal with respect to the label similarity function L. This choice implicitly defines a subgraph of the synchronization graph, because not every state and label pair is part of an optimal solution.

EXAMPLE. The application of Def. 6.2 to calculate $S(s_0, t_0)$, — the weighted quantitative simulation between the service automata A_s and A_t of Fig. 6.6 — implicitly defines



(a) subgraph of $A_s \odot A_t$ (b) restriction $(A_s \odot A_t)_{|A_t}$

Figure 6.8: Subgraph of the synchronization graph $A_s \odot A_t$ (a) and its restriction to $(A_s \odot A_t)_{|A_t}$ (b).

Table 6.1: Deriving edit actions from transition pairs of Def. 6.2.

transition of A_1	transition of A_2	resulting edit action	similarity
a	a	keep label a	$L(a, a)$
a	b	modify label from a to b	$L(a, b)$
a	ε (stutter)	change transition label a to τ	$L(a, \varepsilon)$
ε (stutter)	a	insert transition with label a	$L(\varepsilon, a)$

a subgraph of the synchronization graph $A_s \odot A_t$, which is depicted in Fig. 6.8(a). Applying Def. 6.4, this graph can be restricted to $(A_s \odot A_t)_{|A_t}$ (cf. Fig. 6.8(b)), which by Cor. 6.1 is structurally matched by A_t.

The subgraph of $A \odot B$ is implicitly defined by Def. 6.2 and can be used as an edit distance as follows: Each transition is labeled by a pair of a label of A and a label B. In addition, ε can occur to model stuttering. A pair $[a, b]$ can then be interpreted as an *edit action*; that is, as instructions to change the label a to b. Additionally, a pair $[\varepsilon, b]$ demands adding a b-labeled transition, whereas $[a, \varepsilon]$ demands the removal of the a-label and replacing it by τ. The latter would correspond to the deletion of a letter in the setting of string manipulation. Table 6.1 lists these edit actions.

These edit actions define *basic edit actions* whose similarity is determined by the edge similarity function L. To simplify the representation of a large number of edit actions, the basic edit actions may be grouped to macros to express more complex operations such as swapping or moving of edges and nodes, duplicating of subgraphs, or partial unfolding of loops.

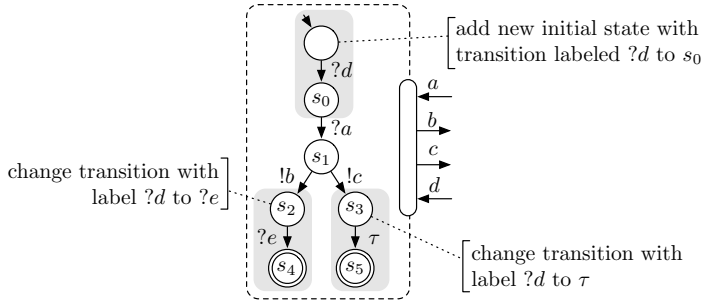


Figure 6.9: Simulation-based edit distance between A_s and A_t.

EXAMPLE. With Tab. 6.1, we can derive from Fig. 6.8(a) edit actions how to change A_s to $(A_s \odot A_t)_{|A_t}$. Figure 6.9 depicts these edit actions. They show how the service automaton A_s needs to be changed to be structurally matched by A_t. We thereby do not explicitly annotate the keeping of edge labels.

COMBINING FORMULA SATISFACTION AND GRAPH SIMILARITY

So far, we defined a simulation-based edit distance which, given two service automata A_1 and A_2, provides minimal editing steps to change A_1 such that it is simulated by A_2. Coming back to the correction scenario motivated in Sect. 6.1, A_1 has the role of an incorrect service and A_2 the role of a correct service. However, we are interested in the similarity to *all* possible correct services characterized by an operating guideline B^φ. In the remainder of this section, we shall extend the simulation-based edit distance accordingly.

The simulation-based edit distance does not respect the formulae of operating guidelines. One possibility to achieve a matching would be to first calculate the most similar simulating service using the edit distance for Def. 6.2 and then to add and remove all nodes and edges necessary in a second step. However, the insertion of nodes would not determine the most similar partner service, because this may result in suboptimal solutions as Fig. 6.10 illustrates. The service automaton (a) is structurally matched by the operating guideline (b), but the formula $?c \wedge ?d \wedge ?e$ is not satisfied. Adding two states and transitions to (a) fixes this (c). However, changing the edge label of (a) from $!a$ to $!b$ also achieves matching, but only requires a single edit action (d).

Because of the suboptimal results achieved through a-posteriori formula satisfaction by node insertion, we need to modify the algorithm of [171] to check any formula-fulfilling *subset* of outgoing transitions. For the remainder of this chapter, we pose the following restrictions on the service automaton and the operating guideline

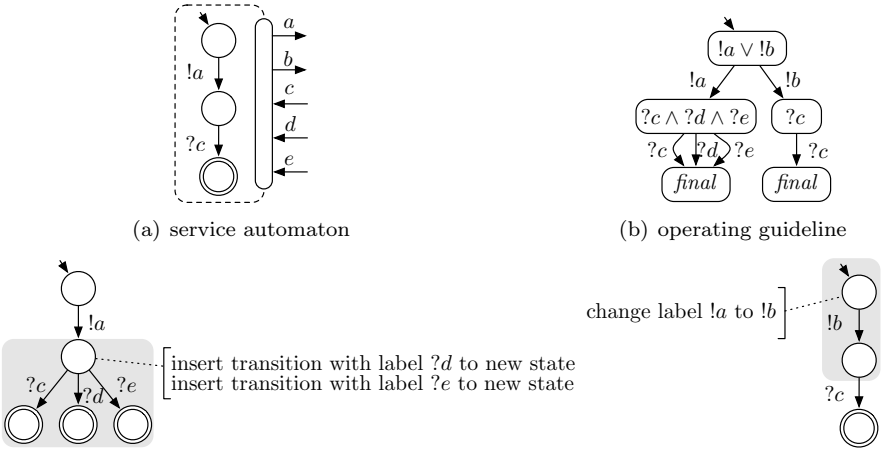


Figure 6.10: Adding states to a simulating service automaton may yield suboptimal results.

under consideration and assume $A = [Q_A, q_{0_A}, \rightarrow_A, \Omega_A, \mathcal{P}_A]$ is a service automaton and $B^\varphi = [Q_B, q_{0_B}, \rightarrow_B, \mathcal{P}_B, \varphi]$ is an operating guideline following these restrictions that we shall discuss in Sect. 6.7.

1. The service automaton A is *deterministic*. Hence, we can treat the transition relation $\rightarrow_A$ as a function and can write $\rightarrow_A(q_A, x) = q'_A$ instead of $q_A \xrightarrow{x} q'_A$.
2. Both the service automaton A and the operating guideline B^φ are *acyclic*. For the operating guideline, we do not consider the empty node $q = \emptyset$ as this node models unreachable behavior which should not be taken into account when correcting a service with respect to a concrete service composition.
3. The final states of the service automaton A must be *sink states*; that is, $q \xrightarrow{x}_A q'$ implies $q \notin \Omega_A$ for all $q \in Q_A$. Furthermore, *final* is assumed to only occur in sink states of the operating guideline.

We need to define additional concepts to include formula satisfaction and to cover the implicit characterization of multiple services by a single operating guideline. We first define label permutations as means to enumerate all the possibilities of changes of the service automaton's labels that are required to satisfy an operating guideline's annotated formula.

Definition 6.5 (Satisfying label set).
Let B^φ be as above and let $q_B \in Q_B$. We define the assignment $\beta' : Q_B \times (\mathbb{E} \cup \{\mathit{final}\}) \to \{\mathit{true}, \mathit{false}\}$ as follows:

$$\beta'(q_B, p) := \begin{cases} \mathit{true}, & \text{if } p \in \mathit{lab}(q_B), \\ \mathit{true}, & \text{if } p = \mathit{final}, \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

A state $q_B \in Q_B$ *models* a formula φ (denoted $q_B \models' \varphi$) iff φ evaluates to *true* under the assignment $\beta'(q_B, \varphi)$. We thereby assume the standard semantics for the Boolean operators $\wedge$, $\vee$, and $\neg$.

Let $\mathit{Sat}(\varphi(q_B)) := \{l \in \mathit{lab}(q_B) \mid l \models' \varphi(q_B)\}$ be the set of all sets of labels of transitions leaving q_B that satisfy formula φ of state q_B.

Compared with Def. 2.12, we evaluate the operating guideline's formulae independent of a service automaton. Thereby, we only take the operating guideline's structure into account. The intuition is that the operating guideline characterizes by itself all correct candidates for the correction (cf. Fig. 6.3). Final states and the *final* predicate do not need to be considered, because we assumed final states to be sink states. The set *Sat* consists of all sets of labels which satisfy a state's formula. To match with this state, a service automaton's state must have outgoing edges with exactly these labels.

EXAMPLE. Consider the operating guideline in Fig. 6.5(b): It holds: $\mathit{Sat}(\varphi(q_2)) = \{\{?c, ?d\}\}$, because the formula $?c \wedge ?d$ has only this satisfying assignment; $\mathit{Sat}(\varphi(q_3)) = \{\{?o\}, \{!p\}, \{?o, !p\}\}$, because the formula $?o \vee !p$ has these three satisfying assignments; and $\mathit{Sat}(\varphi(q_4)) = \emptyset$, because q_4 is a sink state (i. e., has no outgoing transitions) and is annotated with *final*.

The calculation of the weighted quantitative simulation (cf. Def. 6.2) between two service automata A_1 and A_2 is based on pairs of transition labels (one for each service automaton). These pairs of labels are then used to derive the simulation-based edit distance (cf. Tab. 6.1). The pairs were determined by the transition relation of the service automata. To determine the similarity between a service automaton A and an operating guideline B^φ, this is not sufficient. Instead, we need to consider the transition relation of A and the satisfying label sets of B^φ.

Definition 6.6 (Label permutation).
Let A and B^{φ} be as above, and let $q_A \in Q_A$ and $q_B \in Q_B$. For $\beta \in Sat(\varphi(q_B))$, define $perm(q_A, q_B, \beta) \subset \big((\mathbb{E} \cup \{\varepsilon\}) \times (\mathbb{E} \cup \{\varepsilon\})\big)$ to be a *label permutation* of q_A, q_B and β such that:

1. if $q_A \xrightarrow{a}_A q'_A$, then $(a, c) \in perm(q_A, q_B, \beta)$ for some label $c \in \beta \cup \{\varepsilon\}$,
2. if $q_B \xrightarrow{b}_B q'_B$ and $b \in \beta$, then $(d, b) \in perm(q_A, q_B, \beta)$ for some label $d \in \mathbb{E} \cup \{\varepsilon\}$,
3. $(\varepsilon, \varepsilon) \notin perm(q_A, q_B, \beta)$, and
4. if $(a, b) \in perm(q_A, q_B, \beta)$, then (a, c), $(d, b) \notin perm(q_A, q_B, \beta)$ for all labels $c \neq b$ and $d \neq a$.

Define $Perms(q_A, q_B, \beta) \subseteq 2^{(\mathbb{E} \cup \{\varepsilon\}) \times (\mathbb{E} \cup \{\varepsilon\})}$ to be the set of all label permutations of q_A, q_B, and β.

Intuitively, each outgoing edge of q_A is mapped onto at most one outgoing edge of q_B such that we can derive edit actions from this mapping to achieve a structural matching between q_A and q_B. Specifically, the set $Perms$ consists of all permutations of outgoing edges of two states. In a permutation, each outgoing edge of a state of the service automaton has to be present as first element of a pair (1), each outgoing edge of a state of the operating guideline that is part of the label set β has to be present as second element of a pair (2). As the number of outgoing edges of the states may differ, ε-labels can occur in the pairs, but no pair $(\varepsilon, \varepsilon)$ is allowed (3), because we want to exclude simultaneous stuttering. Finally, each edge is only allowed to occur once in a pair (4). This definition exploits the assumption that A is deterministic.

EXAMPLE. For state q_1 of the service automaton in Fig. 6.5(a), state q_2 of the operating guideline in Fig. 6.5(b), and $\beta = \{?c, ?d\}$, one of the permutations in $Perms(q_1, q_2, \beta)$ is $\{(?c, ?c), (\varepsilon, ?d)\}$. Two other permutation are $\{(?c, ?d), (\varepsilon, ?c)\}$ and $\{(?c, \varepsilon), (\varepsilon, ?c), (\varepsilon, ?d)\}$. The permutations can be interpreted just like the label pairs of the simulation edit distance: $(?c, ?c)$ describes keeping the label $?c$, $(?c, ?d)$ describes changing label $?c$ to $?d$, and $(\varepsilon, ?d)$ the insertion of a $?d$-labeled transition.

The insertion and deletion has to be adapted to avoid incorrect or suboptimal results (cf. Fig. 6.10). This is achieved by taking the structure as well as the formulae into account. The following definition relies on the fact that both A and B^{φ} are acyclic and deterministic, and that their final states are sink states.

Definition 6.7 (Subgraph insertion, subgraph deletion).
Let A and B^φ be as above, $q_A \in Q_A$, and $q_B \in Q_B$. We define

$$ins(q_B) = \begin{cases} 1, & \text{if } q_B \not\rightarrow_B, \\ (1-p) + \max\limits_{\beta \in Sat(\varphi(q_B))} \frac{p}{|\beta|} \cdot \sum\limits_{b \in \beta} L(\varepsilon, b) \cdot ins(\rightarrow_B(q_B, b)), & \text{otherwise,} \end{cases}$$

$$del(q_A) = \begin{cases} 1, & \text{if } q_A \in \Omega_A, \\ (1-p) + \frac{p}{n} \cdot \sum\limits_{q_A \xrightarrow{a}_A q'_A} L(a, \varepsilon) \cdot del(q'_A), & \text{otherwise,} \end{cases}$$

where n is the number of outgoing edges of q_A.

Function $ins(q_B)$ calculates the insertion cost of the optimal (acyclic) subgraph of the operating guideline B^φ starting at q_B which fulfills the formulae. Likewise, $del(q_A)$ calculates the cost of deleting of the entire (acyclic) subgraph of the service automaton A from state q_A. Both functions only depend on one of the graphs; that is, *ins* and *del* can be calculated independently from the service automaton and the operating guideline, respectively. Definition 6.7 does not insert or delete nodes, but only calculates the similarity value of the resulting subgraphs. Only this similarity is needed to find the most similar partner service and the actual edit actions can be easily derived from the state from which nodes are inserted or deleted (cf. Tab. 6.1).

With Def. 6.5 and Def. 6.6 describing the means to respect the operating guideline's formulae and Def. 6.7 coping with insertion and deletion, we can finally define the weighted quantitative matching function:

Definition 6.8 (Weighted quantitative matching).
Let A and B^φ be as above. A *weighted quantitative matching* is a function $M : Q_A \times Q_B \to [0,1]$, such that:

$$M(q_A, q_B) = \begin{cases} 1, & \text{if } (q_A \in \Omega_A \wedge q_B \not\rightarrow_B), \\ (1-p) + W_1(q_A, q_B), & \text{otherwise,} \end{cases}$$

$$W_1(q_A, q_B) = \max_{\beta \in Sat(\varphi(q_B))} \max_{P \in Perms(q_A, q_B, \beta)} \frac{p}{|P|} \cdot \sum_{(a,b) \in P} W_2(q_A, q_B, a, b),$$

$$W_2(q_A, q_B, a, b) = \begin{cases} M(\rightarrow_A(q_A, \tau), q_B), & \text{if } a = \tau, \\ L(a,b) \cdot M(\rightarrow_A(q_A, a), \rightarrow_B(q_B, b)), & \text{if } (a \neq \tau \wedge a \neq \varepsilon \wedge b \neq \varepsilon), \\ L(\varepsilon, b) \cdot ins(\rightarrow_B(q_B, b)), & \text{if } (a \neq \tau \wedge a = \varepsilon), \\ L(a, \varepsilon) \cdot del(\rightarrow_A(q_A, a)), & \text{otherwise.} \end{cases}$$

The weighted quantitative matching function is similar to the weighted quantitative simulation function (Def. 6.2). It recursively compares the states of the service automaton and the operating guideline, but instead of statically taking the operating guideline's edges into consideration, it uses the formulae and checks all satisfying subsets (W_1). Additionally, W_2 organizes the successor states determined by the labels a and b, or the insertion or deletion.

MATCHING-BASED EDIT DISTANCE

Again, we can extend the weighted quantitative matching function toward an edit distance, because the permutations give information how to modify the graph. We are, however, not able to define a synchronization graph as in Sect. 6.4, because we compare a service automaton with *all possible* service automata characterized by the operating guidelines. Keeping and modifying of transitions is handled as in Tab. 6.1, whereas adding and deletion of nodes can be derived from Def. 6.7. In fact, the weighted quantitative matching function is not a classical distance. It expresses the similarity between a service automaton and an operating guideline (i. e., a characterization of many service automata) and is hence not symmetric. We still use the term "edit distance" to express the concept of a similarity measure from which edit actions can be derived.

EXAMPLE. Consider once more the example from Fig. 6.5. During the calculation of $M(q_1, q_2)$, the permutation $\{(?c, ?c), (\varepsilon, ?d)\}$ is considered. The first label pair denotes that the $?c$ transition is kept unmodified. The second label pair denotes an insertion of a $?d$-labeled transition. The value of this insertion is defined by

$$L(\varepsilon, ?d) \cdot ins(\rightarrow_{OG_{agency \oplus airline}}(q_2, ?d)) = L(\varepsilon, ?d) \cdot ins(q_4) = L(\varepsilon, ?d)$$

and depends only on the similarity function L.

Figure 6.11 shows the result of the application of the matching-based edit distance to the service automaton of Fig. 6.5(a) assuming a discount factor $p = 0.7$ and a label similarity function L, which assigns 1.0 to equal labels and 0.5 to any other label pair. The states are annotated with edit actions. Interestingly, the values of p and L had little impact on the correction of the example. Only for extreme values (e. g., $p = 1.0$ or $L(x, x) = 1.0$ for all labels x), the result changed. The traveler service automaton was automatically generated from a WS-BPEL process and the state in which a modification has to be made can be mapped back to the original WS-BPEL activity. In the example, a `receive` activity has to be replaced by a `pick` activity with an additional `onMessage` branch to receive the decline message. The result would then coincide with the desired result of Fig. 6.2(a).

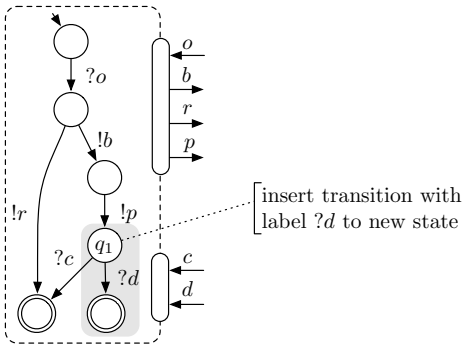


Figure 6.11: Matching-based edit distance applied to the customer's service.

6.5 EXPERIMENTAL RESULTS

The original simulation algorithm of [171] to calculate a weighted quantitative simulation between two service automata A_1 and A_2 (cf. Def. 6.2) needs to check at most $|Q_{A_1}| \cdot |Q_{A_2}|$ state pairs. The extension to calculate the matching between a service automaton A and an operating guideline B^φ (cf. Def. 6.8) takes the operating guideline's formulae and the resulting label permutations into consideration. The length of the operating guideline's formulae is limited by the maximal degree of the nodes which again is limited by the number of open message channels, because operating guidelines are by definition τ-free and deterministic. Let $\mathbb{M}^\sqcup$ be the set of open message channels of B^φ. Then, for each state pair, at most $2^{|\mathbb{M}^\sqcup|}$ satisfying assignments have to be considered. The number of permutations is again limited by the maximal node degree such that at most $|\mathbb{M}^\sqcup|!$ permutations have to be considered for each state pair and assignment. This results in at most $|Q_A| \cdot |Q_B| \cdot 2^{|\mathbb{M}^\sqcup|} \cdot |\mathbb{M}^\sqcup|!$ comparisons.

Although the extension toward a formula-checking edit distance has a discouraging worst-case complexity, operating guidelines of real-life services tend to have simple formulae, a relatively small interface compared to the number of states, and a low node degree. As a proof of concept, we implemented the edit distance in a software prototype Rachel [111]. It takes an acyclic deterministic service automaton and an acyclic operating guideline as input and calculates the edit actions necessary to achieve a matching with the operating guideline. We evaluated the prototype again using service automata generated for WS-BPEL services. In this experiment, we restricted ourselves to acyclic services and translated only the positive control flow (i. e., no exceptional behavior). The edit distance usually could be calculated within few seconds. The prototype exploits that a lot of subproblems overlap and uses dynamic programming techniques [19] to cache and reuse intermediate results which significantly accelerates the runtime. For the examples of the experiment, we observed

Table 6.2: Experimental results on service correction using Rachel.

service	$\lvert\mathbb{E}_{\mathcal{P}}\rvert$	$\lvert Q_A\rvert$ (SA)	$\lvert Q_B\rvert$ (OG)	search space	time (sec)
Online Shop	16	222	153	10^{2033}	2
Supply Order	7	7	96	10^{733}	1
Customer Service	9	104	59	10^{108}	2
Internal Order	9	14	512	$> 10^{4932}$	100
Credit Preparation	5	63	32	10^{36}	1
Register Request	6	19	24	10^{25}	0
Car Rental	7	50	50	10^{144}	3
Order Process	8	27	44	10^{222}	0
Auction Service	6	13	395	10^{12}	0
Loan Approval	6	15	20	10^{17}	0
Purchase Order	10	137	168	$> 10^{4932}$	193

a cache hit ratio of about 97 %. However, no example used more than 5 MB of memory. The experiments were conducted using a computer with a 3 GHz processor. Table 6.2 summarizes the results.

As we had no access to real-life service compositions, we could only evaluate our approach in the case of service compositions consisting of two services. The first seven services of Tab. 6.2 are WS-BPEL processes of a consulting company; the last four services were taken from the WS-BPEL specification [11]. The services were first translated into service automata using the compiler BPEL2oWFN [107]. For these service automata, the operating guidelines were calculated using the tool Wendy [121]. Note that the complexity of the calculation of the edit distance is independent of the fact whether the service automaton matches the operating guideline or not. In case no partner service was available, we synthesized a strategy with Wendy and manually added a few mistakes that lead to incompatibility. As we can see from Tab. 6.2, the services' interfaces are rather small compared to their number of states.

Column "search space" of Tab. 6.2 lists the number of acyclic deterministic services characterized by the operating guideline. This number is calculated by Rachel and is bounded by 10^{4932} due to technical reasons — this is the maximal number that can be represented by an 80 bit floating point data type. All these services are correct partner services and have to be considered during the search for the most similar service. The presented algorithm exploits the compact representation of the operating guideline and allows to efficiently find the most similar service from more than 10^{2000} candidates.

For most services, the calculation only takes a few seconds. The "Internal Order" and "Purchase Order" services are exceptions. The operating guidelines of these services have long formulae with a large number of satisfying assignments (about ten times larger than those of the other services) yielding a significantly larger search space. Notwithstanding the larger calculation time, the service fixed by the calculated

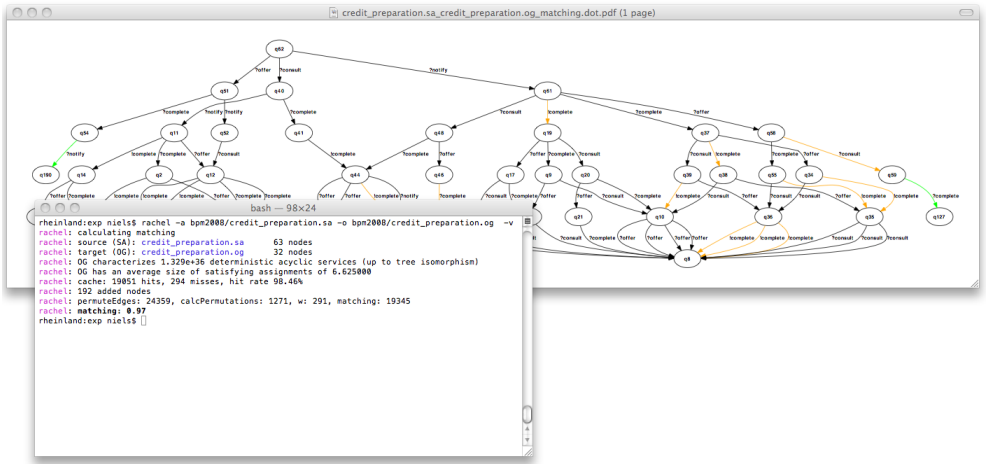


Figure 6.12: Correction output of the tool Rachel for the credit preparation example.

edit actions is correct by design, and the calculation time is surely an improvement compared to iterative manual correction.

Figure 6.12 depicts the output of the tool Rachel. It consists of a graphical representation of the incorrect service automaton whose edges are annotated with edit actions. Different colors (green for insertion, yellow for modification, and red for removal) symbolize the respective edit actions. The proposed edit actions are additionally given as textual output.

6.6 RELATED WORK

The presented matching edit distance is related to several aspects of current research in many areas of computer science:

AUTOMATED DEBUGGING. In the field of model checking, the explanation of errors by using distance metrics [77] has received much attention. Compared to the approach presented in this paper, these works focus on the explanation and location of single errors in classical C (i. e., low-level) programs. The derived information is used to support the debugging of an erroneous program.

SERVICE MATCHING. Many approaches exist to discover a similar partner service. Corrales et al. [40] and Grigori et al. [76] investigate the matching of WS-BPEL processes and Web service conversations, respectively. Both approaches rely on graph isomorphisms and do not consider service behavior. Dijkman et al. [64] compare

several heuristic approaches to speed up matching between business processes and report promising runtimes even for larger sets of processes. Compared to our approach, they focus only on the structure of a process rather than on its behavior. Other approaches [192, 24] use ontologies and take the semantics of activities into account, but do not focus much on the behavior or message exchange. Günay and Yolum [79] represent the behavior of a service as a language of traces and apply string edit distances to compare services. This approach, however, cannot be used in the setting of asynchronously communicating services where the moment of branching is crucial to avoid deadlocks.

SERVICE SIMILARITY AND VERSIONING. The change management of business processes and services is subject of many recent works. An overview of what can differ between otherwise similar services is given by Dijkman [61, 62]. The reported differences go beyond the behavioral level and take authorization aspects under consideration. Weber et al. [182] give an overview of frequent change patterns occurring in the evolution of a business process model. Beside the already mentioned basic operations (adding, changing, and removing of edges or nodes), complex operations such as extracting subprocesses are presented. With a *version preserving graph*, a technique to represent different versions of a process model is introduced by Zhao and Liu [196]. This technique was made independent of a change log by Küster et al. [100]. Again, versioning relies on the structure of the model rather than on its behavior. Aït-Bachir et al. [6] investigate behavioral differences in terms of simulation. They compare behavioral incompatibilities between two services and elaborate an edit distance to overcome these incompatibilities. Their result is much related to our simulation-based edit distance, but only consider synchronous communication.

Li et al. [106] apply an edit distance to find, given several process variants, a reference process model which is most similar to all variants with respect to structural change operations. This approach could be applied to our correction setting by treating each strategy that is characterized by an operating guideline as variant model and the most similar correct participant as a reference model. However, experimental results are only reported for up to 100 processes.

SERVICE MEDIATION. An alternative to changing a service to achieve compatibility in a choreography offer *service mediators* (sometimes called adapters) [28, 66, 73]. Service mediation is rather suited to fit existing services, whereas our approach aims at supporting the design and modeling phase of a service choreography. Still, a mediator between the customer service on the one hand and the travel agency and the airline service on the other hand (cf. Fig. 6.1) would have to receive the airline's decline message and create a confirmation message for the customer which is surely unintended. Furthermore, several service mediation approaches assume total perception of the participants' internal states during runtime [147].

The difference between all mentioned related approaches and the setting of this paper is that these approaches either focus on low-level programs or mainly aim at finding *structural* (and certainly not simulation-based) differences between *two* given services and are therefore not applicable to find the most similar service from a large set (cf. Tab. 6.2) of candidates.

6.7 CONCLUSION AND FUTURE WORK

We presented an edit distance to compute the edit actions necessary to correct a faulty service to interact in a choreography in a compatible manner. We defined this edit distance in two steps. First, we defined an edit distance based on existing work on a quantitative similarity measure, which was originally defined to compare behavior with respect to a simulation relation. Second, we adjusted this approach to suite our setting of comparing a service automaton with a set of service automata, implicitly characterized as operating guideline. The edit distance (i. e., the actions needed to correct the service) can be automatically calculated using a prototypic implementation. Together with translations from [110] and to [114] WS-BPEL processes and the calculation of the characterization of all correct partner services (the operating guideline) [116, 118], an integrated tool chain to analyze and correct WS-BPEL-based choreographies is available. As the edit distance itself is based on service automata, it can be easily adapted to other modeling languages such as UML activity diagrams [149] or BPMN [150] using Petri net or automaton-based formalizations.

The edit distance is an important tool to support the development of correct service compositions. With its help, we are not just able to detect and diagnose incompatibilities, but also to *propose* corrections. The corrected services try to reuse as much behavior of the original faulty service, yet is still guaranteed to be *correct by design*. Although the approach presented in this chapter is still in its infancy and is subject to many restrictions (the structure must be acyclic and deterministic, and final states must be sink states), it is likely to be applicable in several application scenarios. For instance, Parnjai et al. [155] use the presented edit distance to correct a service A such that it can substitute another service B; that is, the corrected service must not exclude strategies of B. Another application could be the organization of services in a service registry (cf. Fig. 1.2): This registry could be partitioned using the similarity of services with respect to certain typical "key services". This partition then could narrow down the search space for a provider service by only searching in these partitions that are sufficiently similar to the query service. Likewise, the edit distance can be used to rank results as reported by Dijkman et al. [64].

However, several questions still remain open. First, the choice *which* service causes the incompatibility and hence needs to be fixed is not always obvious and needs

further investigation. For instance, the choreography in Fig. 6.1 could also have been corrected by adjusting the airline service. Even worse, the composition depicted in Fig. 6.4 shows that for some compositions, the behavior of more than one service needs to be corrected.

A further aspect to be considered in future research is the choice of the *cost function* used in the algorithm, because it is possible to set different values for any transition pairs. This could be, for instance, achieved with questionnaires such as reported by Wombacher [190]. Semantic information on message contents (e. g., derived from an ontology) and relationships between messages can be incorporated to refine the correction. For example, the insertion of the receipt of a confirmation message can be penalized less than the insertion of sending an additional payment message.

The similarity function could also be adjusted by taking the *execution frequencies* of activities into account: Medeiros et al. [134] present a novel process mining algorithm that not only considers execution traces, but also distinguished frequently executed activities ("highways") from rarely used activities ("dirt roads"). They report how this information can be used to improve the quality of mined process models.

Another important field of research is to further increase the performance of the implementation by an early omission of suboptimal edit actions. For instance, heuristic guidance metrics such as used in the A* algorithm [82] may greatly improve runtime performance.

Finally, a translation of the matching edit distance of Def. 6.8 into a linear optimization problem [170] may also help to cope with cyclic and nondeterministic services.

Part III

CORRECTNESS OF SERVICE CHOREOGRAPHIES

7 REALIZABILITY

This chapter is based on results published in [123].

CHOREOGRAPHIES which were modeled in a bottom-up approach by composing existing services (also called interconnected models) were investigated in Part II. In chapters 5 and 6, we demonstrated how the composition of locally correct (i. e., controllable) services can introduce subtle errors such as deadlocks. Using model checking and strategy synthesis techniques, these errors can be automatically detected and — to some extent — corrected.

The converse of the bottom-up approach is the top-down approach which tries to avoid these problems in the first place by creating a service composition out of a choreography specification, called interaction model. This service composition is then *compatible by design*. A choreography specification describes the interaction from a global point of view without specifying unnecessary details about the internal control flow of the involved parties. In any case, there is no distinguished coordinator as it is the case in an orchestration setting.

Several languages have been proposed for specifying choreographies (Su et al. [173] provide a survey). They all have in common that they permit to specify unreasonable interactions. An example for a potentially unreasonable interaction is to require that a message from participant A to participant B must be exchanged before another message from C to D. As long as no other messages are passed between A and C or B and C this requirement cannot be satisfied. For distinguishing between reasonable and unreasonable interaction, the concept of *realizability* [71, 8] was introduced. Intuitively, realizability describes the dilemma of balancing compliance with the global specification on the one hand and flexibility and autonomy of the participating services on the other hand.

In this chapter, we address the following *issues* in existing approaches to choreographies and realizability notions. *First*, several approaches only focus on synchronous interaction: asynchronous interaction is either not considered at all, or is brought into the approach as a derivative of the synchronous approach. For instance, several approaches specify only the order in which messages are sent, but leave the order in which they should be received open. Consequently, we employ service automata for modeling choreographies where *synchronous and asynchronous communications are both first-class citizens*. In our setting, causality between the receipt of a message and sending another one can be specified.

Second, there appear to be several proposals for defining realizability. Consequently, we propose a *hierarchy* of realizability notions which includes and extends existing concepts.

Controllability asks whether a given service has compatible partners. Existing techniques for answering the controllability problem are capable of synthesizing a compatible partner if it exists. Hence, *third*, we suggest techniques to synthesize internals of realizing partners. By relating the realizability problem to controllability, we, *fourth*, get the opportunity to study specifications which involve both a choreography and the specification of the internal behavior of some of the participants. This way, we marry the choreography approach with the orchestration approach as well as interaction models with interconnected models. Both approaches have so far been conceived as complementary paradigms for building up complex processes from services [157, 63].

The remainder of this chapter is organized as follows. In the next section, we show how service automata can be used to model choreographies. In Sect. 7.2, we recall different realizability notions and introduce the novel concept of *distributed realizability*, which is more liberal than complete realizability, yet stricter than partial realizability and hence complements the existing notions. The main contribution is presented in Sect. 7.3: the realizability problem can be approached with algorithms from controllability. Section 7.4 is dedicated to issues arising in case asynchronous communication is considered. In Sect. 7.5, we show how the relationship between controllability and realizability can be used to combine aspects from interaction modeling and interconnected models. Section 7.6 discusses related work, and Sect. 7.7 concludes and gives directions for future research.

7.1 MODELING CHOREOGRAPHIES

A choreography specification usually consists of two parts: First, a description of the participants and the message channels between them (i. e., the structure or syntax of the choreography, called *collaboration*); and second, a specification of the desired interactions between these participants (i. e., the semantics of the choreography). The former, structural, aspects of a choreography can be expressed by sets of ports.

Definition 7.1 (Peer, collaboration).
A *peer* is a set $\mathcal{P} = \{[I_1, O_1], \ldots, [I_n, O_n]\}$ of ports such that (1) $I_i \cap O_i = \emptyset$ for all i and (2) $(I_i \cup O_i) \cap (I_j \cup O_j) = \emptyset$ for all $i \neq j$. A *collaboration* is a set $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ of peers such that $\bigcup_{i=1}^{m} \mathcal{P}_i$ is a closed interface.

A *peer* consists of a set of ports whose message channels are pairwise disjoint. These shall be later implemented by a service automaton. We need to employ *sets* of ports to

(a) collaboration (b) collaboration (shorthand notation)

Figure 7.1: Illustration of the role of peers and ports in a collaboration.

express situations in which one participant communicates with *several* other services using several ports. Figure 7.1 illustrates this.

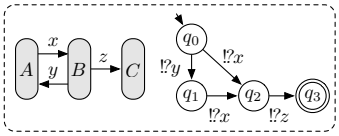
As the union of the peers form a closed interface, a closed multiport service automaton can be used as a formal model for the behavior of a choreography. This ensures a closed world (cf. Def. 2.1) and bilateral communication between the participants (i. e., each message has exactly one sender and one receiver). At the same time, a closed service automaton implicitly defines a set of runs.

Definition 7.2 (Conversation, choreography).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration and $A = [Q, q_0, \rightarrow, \Omega, \bigcup \mathcal{C}]$ be a closed service automaton. A maximal terminating run σ of A is a *conversation* if no τ-transition occurs in σ and, for all $x \in \mathbb{M}_a$, $\#_{!x}(\sigma) = \#_{?x}(\sigma)$ and for every prefix σ' of σ holds: $\#_{!x}(\sigma') \geq \#_{?x}(\sigma')$. Thereby, $\#_x(\sigma)$ denotes the number of occurrences of the message event x in the run σ. A *choreography* is a set of conversations.

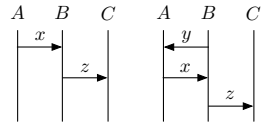
For a run σ, define the *event sequence* of σ as $\sigma|_{\mathbb{E}}$ (i. e., σ without τ-steps). The *language* of A, denoted $\mathcal{L}(A)$, is the union of the event sequences of all runs of A. A is a *choreography automaton*, if A is deterministic and τ-free, and $\mathcal{L}(A)$ is a choreography.

The requirements for a conversation state that asynchronous events are always paired (messages do not get lost), and a send event always occurs before the respective receive event. Synchronous communication is not restricted. Conversations are defined in terms of terminating runs (cf. Def. 2.4) which is similar to *well-behaved* runs defined by Bultan et al. [31].

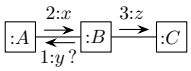
In this thesis, we define choreographies as a set of *message event sequences*, which complies with the common understanding of choreographies [157, 33, 80]. We are aware of proposals to additionally model *local* choices [55] or *internal* behavior in a choreography [96]. This, however, contradicts our understanding that a choreography is a *global* specification of the *interaction behavior* of a service composition. Keeping internals secret may have several reasons. On one hand, trade secrets may be involved



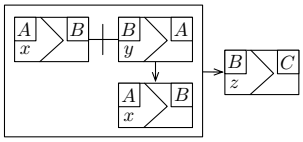
(a) choreography automaton



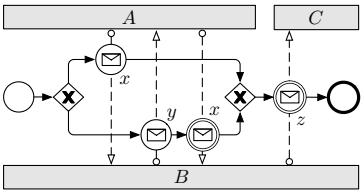
(b) message sequence charts



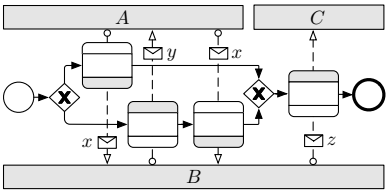
(c) UML collaboration diagram



(d) Let's Dance model



(e) iBPMN model



(f) BPMN 2.0 choreography

Figure 7.2: Different interaction modeling languages specifying the choreography $\{!?x\,!?z, !?y\,!?x\,!?z\}$.

as the parties may be competitors. On the other hand, an internal control flow may not exist in case the choreography is specified in a design-by-contract scenario.

A mapping from existing interaction modeling languages such as interaction Petri nets [55], Let's Dance [194], message sequence charts [8], collaboration diagrams [32], iBPMN [48], or BPMN 2.0 choreographies [151] to choreography automata is straightforward. Whereas these languages differ in syntax and semantics, concepts such as an underlying collaboration (i. e., the set of ports), the choreography (i. e., the intended global behavior) can be easily derived from these languages. Figure 7.2 depicts different models specifying the same choreography.

However, not every τ-free multiport service automaton specifies a choreography. Figure 7.3(a) depicts a service automaton that violates the causal dependency between an asynchronous send and the respective receive event. Another problem arises in settings such as shown in Fig. 7.3(b) in which an arbitrary number of x-messages needs to be buffered. In Sect. 7.4, we show how bounded message buffers can be enforced and all runs that are not conversations can be removed from a service automaton.

(a) dependency violation (b) unbounded message buffer

Figure 7.3: The multiport service automata (a) and (b) do not specify a choreography.

Finally, not every choreography can be expressed by a multiport service automaton, for example the context-free choreography

$$\{(!?d)^i(!?e)^i \mid i \in \mathbb{N}^+\} = \{!?d\,!?e, !?d\,!?d\,!?e\,!?e, \dots\}$$

(an example for a Dyck language) cannot be expressed. In this thesis, we only consider regular choreographies, because language equivalence and language containment is undecidable for context-free languages, and hence realizability is undecidable for context-free choreographies.

7.2 REALIZABILITY NOTIONS

A choreography specifies the desired global behavior of a service composition. This specification can be interpreted as safety properties (no unspecified conversations are allowed) and liveness properties (every interaction sequence can be completed to a conversation). The composition of service automata, each implementing one peer, can be related to a specified choreography, which leads to the concept of *complete realizability* [173, 71, 8, 55, 32, 46, 45] (sometimes called "full realizability" or just "realizability").

Definition 7.3 (Complete realizability).
Let $\mathcal{C} = \{\mathcal{P}_1, \dots, \mathcal{P}_n\}$ be a collaboration and A a choreography automaton implementing the set of ports $\bigcup\mathcal{C}$. The composable service automata $A_1, \dots, A_n$ *completely realize* A if, for all i, A_i implements the set of ports $\mathcal{P}_i$ and $\mathcal{L}(A_1 \oplus \cdots \oplus A_n) = \mathcal{L}(A)$.

Complete realizability is a strong requirement, because it demands that the observable behavior of the participants exactly matches the choreography. That is, all safety and liveness properties must be satisfied. In practice, it is often the case that not all aspects of a choreography can be implemented. To this end, Zaha et al. [195] introduce the notion *local enforceability* (also called *partial realizability* or *weak realizability*), which only demands that a subset of the choreography is realized by the participants:

Definition 7.4 (Partial realizability).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration and A a choreography automaton implementing the set of ports $\bigcup \mathcal{C}$. The composable service automata $A_1, \ldots, A_n$ *partially realize* A if, for all i, A_i implements the set of ports $\mathcal{P}_i$ and $\emptyset \neq \mathcal{L}(A_1 \oplus \cdots \oplus A_n) \subseteq \mathcal{L}(A)$.

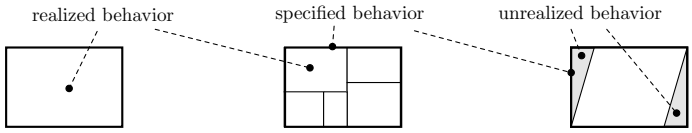
Partial realizability requires all safety properties to hold, but makes no assumption on liveness properties other than demanding the set of realized conversations is not empty. Obviously, complete realizability implies partial realizability. Though this weaker notion ensures that all constraints of the choreography are satisfied, it still only considers a single tuple of service automata. If there does not exist such a tuple of automata that realizes the *complete* choreography, there may still exist a *set* of tuples — each partially realizing the choreography — which *distributedly* realizes the complete choreography:

Definition 7.5 (Distributed realizability).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration and A a choreography automaton implementing the set of ports $\bigcup \mathcal{C}$. The tuples of service automata $[A_1^1, \ldots, A_n^1], \ldots, [A_1^m, \ldots, A_n^m]$ *distributedly realize* A if, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$,

1. A_i^j implements the set of ports $\mathcal{P}_i$,
2. $A_1^j, \ldots, A_n^j$ are composable,
3. $\emptyset \neq \mathcal{L}(A_1^j \oplus \cdots \oplus A_n^j) \subseteq \mathcal{L}(A)$, and
4. $\bigcup_{j=1}^{m} \mathcal{L}(A_1^j \oplus \cdots \oplus A_n^j) = \mathcal{L}(A)$.

Distributed realizability allows for design-time coordination between participants: From a set of different possible implementations, we can choose a specific tuple of implementations which are coordinated in the sense that each participant can rely on the other participant's behavior. In addition, every conversation specified by the choreography can be realized by at least one tuple of implementing service automata; that is, the choreography does not contain "dead code" which would be unusable by any partner set. Although it is a stronger notion than partial realizability (i. e., more of the choreography's behavior is implemented), it is still a weaker notion than complete realizability. Figure 7.4 illustrates the different notions.

EXAMPLE. Consider the choreographies in Fig. 7.5. The choreography in which the participants communicate synchronously (a) is completely realizable by a set of service automata (b) which synchronize at runtime via message x or y. In case the messages are sent asynchronously (c), this is no longer possible. This choreography is not completely realizable, because there does not exist a single pair of service automata which implements the specified behavior. However, the implementations can be coordinated



(a) complete realizability (b) distributed realizability (c) partial realizability

Figure 7.4: Visualization of realizability notions.

at design time: either participant A sends a message and participant B is quiet or the other way around (d). These two pairs distributedly realize the whole choreography. Finally, choreography (e) can only be partially realized, because the conversation $!x!y?x?y$ cannot be implemented by the participants without also producing the unspecified conversations $!y!x?x?y$ or $!y!x?y?x$. However, the conversation $!x?x!y?y$ can be realized (f).

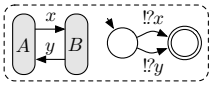
7.3 REALIZING CHOREOGRAPHIES

In this section, we show how the different notions of realizability can be checked and how a realizable choreography can be projected to implementing service automata. A close relationship to a controllability notion allows us to use an existing algorithm to remove unrealizable behavior from choreographies.

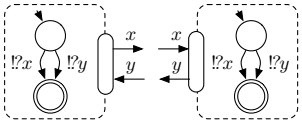
PROJECTION AND INDEPENDENCE

The tuples of service automata which realize a choreography are existentially quantified in the definitions of the different realizability notions. To this end, realizability can be seen as a synthesis problem: given the choreography specification, we are interested in realizing peer implementations. As the choreography describes the global behavior of all peers, it can be used as a starting point to *project* this global behavior to the different peers.

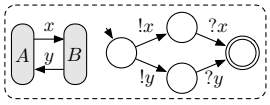
As discussed earlier, the different realizability notions differ in the amount of conversations which must be realized by the peers. They all have in common that no new conversation must be introduced. Hence, the projected peers need to be *coordinated* at design time such that they do not produce unspecified conversations. The example choreographies in Fig. 7.5 showed that this coordination can already be impossible even if two peers share message channels. To characterized possible and impossible coordination, we first introduce *distant* message events. We call two message events distant if there exists no peer which can observe both:



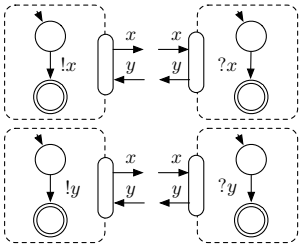
(a) completely realizable choreography



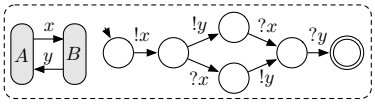
(b) a pair of realizing participants



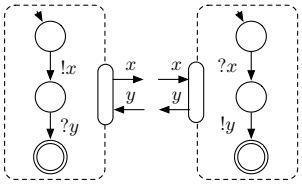
(c) distributedly realizable choreography



(d) two tuples of realizing participants



(e) partially realizable choreography



(f) a pair of realizing participants

Figure 7.5: The choreography (a) is completely realizable, (c) is distributedly realizable, and (e) is partially realizable.

Definition 7.6 (Distant message events).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration. Two message events $a, b \in \mathbb{E}$ are *distant* iff there exist no peer $\mathcal{P} \in \mathcal{C}$ such that $\{a, b\} \subseteq \mathbb{E}_{\bigcup \mathcal{P}}$.

Distant message events cannot be coordinated by any peer. There only exist two scenarios in which distant message events do not jeopardize realizability: either they can be removed from the choreography without resulting in an empty set of conversations, or they do not need to be coordinated in the first place. Whereas the former setting results in implementations which do not completely realize the choreography, message events enjoying the latter property are called *independent*. Informally, two events are independent if they (1) neither activate (2) nor deactivate each other, and (3) if they occur in one specific order reaching a state q, then any

other ordering must be possible, possibly reaching a different state q', which, however, must be equivalent to q.

Definition 7.7 (Independence [187]).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration, $A = [Q, q_0, \to, \Omega, \bigcup \mathcal{C}]$ be a τ-free multiport service automaton, and $a, b \in \mathbb{E}$ be distant message events.

- *a activates b* in $q \in Q$, if (1) there exist states $q_a, q_{ab} \in Q$ with $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$, but there exists no state $q_b \in Q$ with $q \xrightarrow{b} q_b$ and (2) $\mathbb{M}(a) = \mathbb{M}(b)$ implies $a \notin !\mathbb{E}$.
- *a disables b* in $q \in Q$, if there exist states $q_a, q_b \in Q$ with $q \xrightarrow{a} q_a$, $q \xrightarrow{b} q_b$, but there exists no state $q_{ab} \in Q$ with $q_a \xrightarrow{b} q_{ab}$.
- Two states $q_1, q_2 \in Q$ are *equivalent* iff $\mathcal{L}([Q, \delta, q_1, F, \mathcal{P}]) = \mathcal{L}([Q, \delta, q_2, F, \mathcal{P}])$.
- a and b are *independent* iff, for all states $q \in Q$ holds: a neither activates nor disables b in q and, if $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$ and $q \xrightarrow{b} q_b \xrightarrow{a} q_{ba}$, then q_{ab} and q_{ba} are equivalent.

These independence requirements introduced by Wolf [187] are weaker than the *lossless-join* property [71] and the *well-informed* property [32] which both aim at complete realizability only. They are, however, similar the *autonomous property* [71].

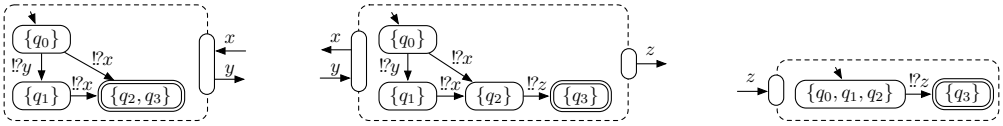
The second requirement of the definition of activation is an extension of the original definition: Wolf [187] investigates independence of the states of a most-permissive strategy. There, each state has all possible asynchronous receiving events — possibly leading to the empty state (see Sect. 2.4). Hence, an asynchronous send event $!x$ would never activate a subsequent asynchronous receive event $?x$. The second requirement rules out such scenarios for choreography automata.

If all distant events of a choreography are independent, the choreography can be safely projected to the peers, resulting in a tuple of service automata which realize the choreography.

Definition 7.8 (Participant projection).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration and $A = [Q, q_0, \to, \Omega, \bigcup \mathcal{C}]$ be a τ-free multiport service automaton. Define the *projection* of A to the peer $\mathcal{P}_i$, denoted $A|_{\mathcal{P}_i}$, as the service automaton $[Q', q_0', \to', \Omega', \mathcal{P}_i]$ with the initial state $q_0' := project_{\mathcal{P}_i}(\{q_0\})$ and Q', $\to'$, and Ω' inductively defined as follows:

- $q_0' \in Q'$.
- If $q \in Q'$ with $q_1 \in q$, $q_1 \xrightarrow{x} q_2$, and $x \in \bigcup \mathcal{P}_i$, then $q' \in Q'$ with $q' := project_{\mathcal{P}_i}(\{q_2\}) \in Q'$ and $[q, x, q'] \in \to'$. $q' \in \Omega'$ iff $q' \cap \Omega \neq \emptyset$.

Thereby, define $project_{\mathcal{P}_i}(S) := \{q' \mid q \in S, q \xrightarrow{x_1} \cdots \xrightarrow{x_n} q', x_i \notin \bigcup \mathcal{P}_i\}$ for a set of states $S \subseteq Q$.



(a) peer A (b) peer B (c) peer C

Figure 7.6: Choreography projection to service automata.

The set $project_{\mathcal{P}_i}(S)$ contains all states reachable with a (possibly empty) sequence from a state of S which does not contain an event from $\bigcup \mathcal{P}_i$. The definition an adaption of the *closure* operation (cf. Def. 2.8) and was first proposed to be used as a projection algorithm by Decker [46, 45].

From Def. 7.7 and Def. 7.8 we can conclude:

Corollary 7.1 (Projection of independent choreographies).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration, $A = [Q, q_0, \rightarrow, \Omega, \bigcup \mathcal{C}]$ be a choreography automaton such that all distant events are independent.
Then $\mathcal{L}(A) = \mathcal{L}(A|_{\mathcal{P}_1} \oplus \cdots \oplus A|_{\mathcal{P}_n})$.

EXAMPLE. The choreography of Fig. 7.2(a) does not contain distant message events and is completely realizable. Figure 7.6 depicts its projection to the peers A, B, and C.

LINK TO CONTROLLABILITY

Definition 7.7 describes independence in a declarative way. It can be used to *check* whether the events of a choreography are independent. If this is the case, the choreography is completely realizable and Def. 7.8 provides realizing service automata. In case there exist distant events which are not independent, we can only conclude that the choreography is not completely realizable, but we can neither make a statement on the weaker notions of realizability nor can we apply Def. 7.8.

However, Wolf [187] not only provides a definition of independence, but also an algorithm to systematically restrict behavior to enforce independence. This algorithm was originally introduced to check a related controllability notion, *decentralized controllability*. Decentralized controllability is an extension of controllability to respect the ports of a service automaton.

Definition 7.9 (Decentralized k-controllability [169, 187]).
Let A be an open multiport service automaton implementing the ports

$\{[I_1, O_1], \ldots, [I_n, O_n]\}$. Then A is *decentralized k-controllable* iff there exists a tuple of single-port service automata $[B_1, \ldots, B_n]$ such that B_i implements the port $\{[O_i, I_i]\}$ and $A \oplus B_1 \oplus \cdots \oplus B_n$ is k-compatible.

In contrast to Def. 2.7, decentralized controllability takes the ports of A into account and requires each port to be implemented by a distinct service automaton B_i. The multiport service automaton A can be seen as an orchestrator for the other service automata. We call $[B_1, \ldots, B_n]$ a *decentralized k-strategy* of A. Thereby, the single-port service automata $B_1, \ldots, B_n$ only communicate with A and do not share message channels. Hence, they cannot communicate directly with each other during runtime. Only during design time of $B_1, \ldots, B_n$ it is possible to coordinate their behavior.

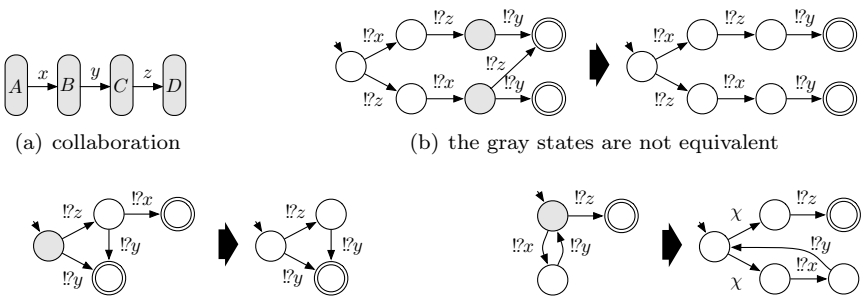
This setting is similar to the realizability scenario and is related to independence as follows. To synthesize a decentralized strategy, first a "centralized" strategy (i. e., a strategy as constructed by Def. 2.9) is synthesized. This strategy is an overapproximation of any decentralized strategy, because it serves all ports at once does not require any coordination. This strategy is then "massaged" to enforce independency.

To apply the algorithm from Wolf [187], the choreography automaton needs to be made *deterministic*. This is a standard operation for regular automata [85] and does not restrict generality. It ensures that in every state q and for each event x there is exactly one x-labeled edge leaving q. In case such a transition was not specified, the new introduced edge leads to a nonfinal sink state.

Independency can be achieved by removing those edges and states from the automaton which are dependent. In the case of disabling of events, this removal contains nondeterminism: If, for instance, an event a disables an event b in a state q, we can decide to either remove the a-successor or the b-successor of q. This nondeterminism seems to be necessary, as nondeterminism (or backtracking) is one of the few tools to break symmetry [169]. This mutually exclusive deletion yields two different tuples of implementing peers. To avoid restricting the set of local service implementations, we introduce a *global decision event* χ in the following definition to express the different outcomes of this nondeterminism. These decision events allow us to postpone the decision which event to remove after the dependency resolution.

Definition 7.10 (Resolution of dependency).
Let $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ be a collaboration, $A = [Q, q_0, \to, \Omega, \bigcup \mathcal{C}]$ be a τ-free multiport service automaton, and $a, b \in \mathbb{E}$ be distant message events.

1. If a disables b in a state $q \in Q$, then introduce two new states q_a and q_b with $q \xrightarrow{\chi} q_a$, $q \xrightarrow{\chi} q_b$ such that q_a has all outgoing edges of q that are not labeled with



(a) collaboration

(b) the gray states are not equivalent

(c) $!?z$ enables $!?x$ in the gray state

(d) $!?x$ disables $!?z$ in the gray state

Figure 7.7: Examples for the resolution of dependencies.

b and q_b has all outgoing edges of q that are not labeled with a. Then remove all outgoing edges of q that are not labeled with χ.

2. If a enables b in a state $q \in Q$, then delete the state q_{ab} with $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$.
3. If the states $q_{ab}, q_{ba} \in Q$ with $q \xrightarrow{a} q_a \xrightarrow{b} q_{ab}$ and $q \xrightarrow{b} q_b \xrightarrow{a} q_{ba}$ are not equivalent, then delete q_{ab}, q_{ba} and unite A with the deterministic τ-free multiport service automaton A' with $\mathcal{L}(A') = \mathcal{L}([Q, q_{ab}, \to, \Omega, \mathcal{P}]) \cap \mathcal{L}([Q, q_{ba}, \to, \Omega, \mathcal{P}])$ and add the edges $q_a \xrightarrow{b} q'_0$ and $q_b \xrightarrow{a} q'_0$.

The first step introduces the global decision events if an event is disabled. The second step removes states to avoid the enabling of an event. In the third step, equivalence of states that are reached by different interleavings of events is enforced by intersecting the runs reachable from these states. As we consider regular languages, the automaton having this intersection as language can be constructed easily [85].

The original definition of Wolf [169, 187] is based on acyclic service models. *The algorithm is sound, but not complete: When applied to cyclic models, correctness is guaranteed, but the dependency resolution might not terminate.* A sound and complete algorithm is still subject to future work. We are, however, currently not aware of an example for that Def. 7.10 does not terminate.

EXAMPLE. Figure 7.7(b)–(d) depict examples for each step. The removal of states and edges can introduce new deadlocks and make other states unreachable from the initial state. Such states need to be removed before projection. A multiport service automaton is — similar to Prop. 2.1 — not decentralized controllable iff all states are removed.

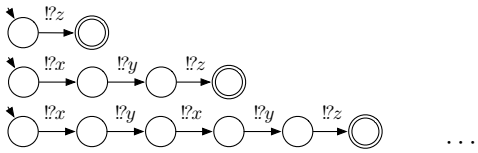


Figure 7.8: Resolutions of the global decisions in the automaton of Fig. 7.7(b).

A multiport service automaton with global decision events such as the example in Fig. 7.7(c) implicitly characterizes a set of multiport service automata in which these decisions have been resolved. Each resolution of these decisions results in a tuple of implementing peers which can be derived using the projection defined in Def. 7.8. For the example of Fig. 7.7(c), the global decision is resolved independently each time the initial state is reached. Figure 7.8 depicts the different resolutions of the global decisions. Each resolution represents a design-time coordination between the peers A and C on how often the $!?x!?y$ loop should be traversed. As the peers A and C cannot communicate with each other, this coordination cannot be done during runtime. The set of all possible implementations distributedly realize the choreography.

The approach aims at finding the strongest applicable realizability notion. If a state needs to be deleted due to dependencies, we can derive diagnosis information:

- If a state is deleted by step (2) or (3) in Def. 7.10, the choreography is neither completely realizable nor distributedly realizable.
- If a global decision (i. e., a χ-event) is introduced by step (1), the choreography is not completely realizable, because the considered events are mutually exclusive.
- If the initial state is removed, the choreography is not partially realizable.

In any case, the respective state and the events that require state deletion can be used to diagnose the choreography and to introduce messages that restore independency.

IMPLEMENTATION

The dependency resolution algorithm and the participant projection has been implemented in a software prototype *Rebecca* [136]. It analyzes a given choreography specification, resolves dependencies, returns the strongest possible realizability notion, and outputs realizing service automata. We checked various choreography models from literature and it turns out that a lot of models that were unrealizable in a classical sense (i. e., not completely realizable) are in fact distributedly realizable.

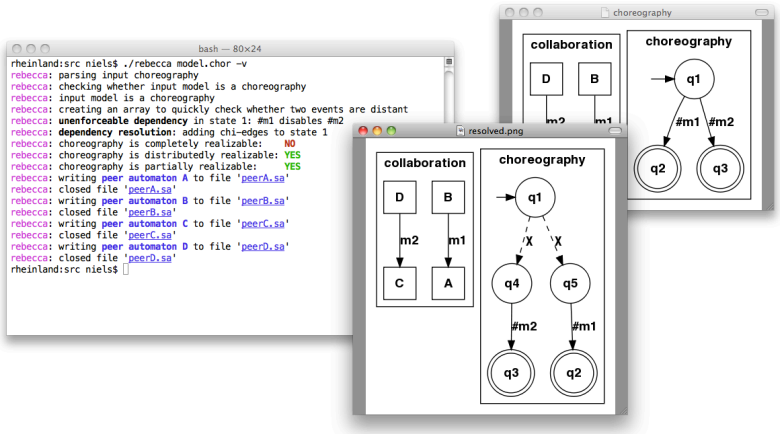


Figure 7.9: Screenshot of the tool Rebecca analyzing a choreography.

7.4 REALIZING ASYNCHRONOUS COMMUNICATION

Many interaction modeling languages (e. g., WS-CDL or interaction Petri nets) assume atomic and hence synchronous message exchange; that is, the sending and receiving of a message is specified to occur at the same time. Participants realizing such a choreography model inherit this synchronous message model. In implementations, however, asynchronous communication is often preferred over synchronous communication as a "fire and forget" send action is more efficient than a blocking handshaking.

To this end, we studied how synchronous peers that realize a choreography can be "desynchronized" [47]; that is, atomic message exchange is decoupled to a pair of asynchronous send and receive actions. This desynchronization in turn might introduce deadlocks, and the correction toward compatibility results in refinements of the choreography which require domain information and can hardly be automatized. Though the correction approach of Chap. 6 remains applicable, this would contradict the correctness by construction idea of interaction modeling. Fu et al. [72] propose a reverse approach and study *synchronizability* of choreographies — a property under which asynchronous communication can be safely abstracted to synchronous communication. Synchronizability can help to detect problems introduced by asynchronous communication, but is only a sufficient criterion and offers only limited support in resolving these issues.

To avoid both restrictions during the design time of a choreography and a later change of the communication model, we used service automata which allow to individually define, for each message, whether it should be transferred in an asynchronous

or synchronous manner. We claim that the nature of the message transfer is usually known in an early design phase and helps to refine the choreography model.

Unlike related work on collaboration diagrams or conversation protocols, we thereby do not just specify the order in which *send* events occur, but also describe the moment of the respective *receive* events. This is crucial to be able to specify dependencies between asynchronous messages. For instance, one is able to express that a customer must not send an order message to a shop before he *received* the terms of payment. If modeled synchronously, the shop would be blocked as long as the customer reads the terms of payment.

In addition, the precise specification of message receipts ensures that the message exchange between the peers can be realized with bounded message buffers. This is motivated by implementation issues. In addition, unbounded queues would result in an infinite state automaton for which controllability and realizability would be undecidable [130].

In Def. 7.2, we restricted choreographies to only consist of conversations. This does not constrain synchronous message events, but only the asynchronous message events. The following definition manipulates an arbitrary closed service automaton such that every terminating run is a conversation; that is, its collaboration language is a choreography. Furthermore, no run will exceed a given message bound k. This is done by explicitly taking count of the asynchronous messages on the message channels, and is very similar to the composition of service automata (see Def. 2.3).

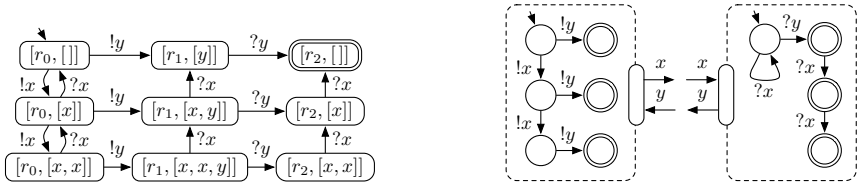
Definition 7.11 (k-bounded service automaton).
Let $A = [Q, q_0, \to, \Omega, \mathcal{P}]$ be a closed τ-free service automaton and $k \in \mathbb{N}$. Define the *k-bounded service automaton* $A_k := [Q', q_0', \to', \Omega', \mathcal{P}]$ with $Q' := Q \times Bags_k(M_A)$, $q_0' := [q_0, [\,]]$, $\Omega' := \Omega \times \{[\,]\}$ and $\to'$ contains exactly the following elements ($\mathcal{B} \in Bags_k(\mathbb{M}_a)$):

- $\big[[q, \mathcal{B}], !?x, [q', \mathcal{B}]\big] \in \to'$ iff $q \xrightarrow{!?x} q'$,
- $\big[[q, \mathcal{B}], !x, [q', \mathcal{B} + [x]]\big] \in \to'$ iff $q \xrightarrow{!x} q'$ and $\mathcal{B}(x) < k$, and
- $\big[[q, \mathcal{B} + [x]], ?x, [q', \mathcal{B}]\big] \in \to'$ iff $q \xrightarrow{?x} q'$.

The bound k can also be used as a parameter for realizability:

Definition 7.12 (k-realizability).
Let C be a choreography automaton and $k \in \mathbb{N}$. C is (completely/distributedly/partially) *k-realizable* iff C_k is (completely/distributedly/partially) realizable.

EXAMPLE. The multiport service automaton of Fig. 7.3(b) did not specify a choreography, because the message buffer for message x is unbounded. Applying Def. 7.11,



(a) 2-bounded choreography

(b) partially 2-realizing peers

Figure 7.10: Enforcing a message bound in the choreography of Fig. 7.3(b) to achieve (partial) 2-realizability.

we can derive the 2-bounded choreography automaton in Fig. 7.10(a). The resulting choreography is partially 2-realizable by the service automata in Fig. 7.10(b).

7.5 COMBINING INTERACTION MODELS AND INTERCONNECTED MODELS

We already described the two approaches to model a choreography: The first approach focuses on the interaction between services and uses message exchange events as basic building blocks. These *interaction models* and the corresponding languages such as Let's Dance, MSCs, iBPMN have already been discussed in Sect. 7.1. Interaction models are a means to quickly specify a choreography by only modeling the desired observable behavior instead of the local control flow of each participant. With the notion of realizability, these missing local behaviors can then be derived from the choreography. To this end, interaction models are best suited if all peer implementations are unknown. Interaction modeling follows a top-down approach from an abstract global model to concrete participant implementations.

In contrast, the second approach is to specify the choreography implicitly by providing a set of peer implementations and information on their interconnection. As these *interconnected models* specify both the local behavior of the participating services and their interaction, they are close to implementation. Examples of specification languages that follow this modeling style are BPMN and BPEL4Chor [50]. Interconnected models aim at reusing existing services in new settings. Though first approaches exist to synthesize individual peers, this modeling style can only be used in a late stage of development, see Chap. 5.

However, a setting in which the local behaviors of some peers are completely specified whereas other peers are not specified at all is not supported by any of the modeling style mentioned. In such a scenario, the completely specified peers can be seen as a constraint of the choreography: The set of all realizing peers is constrained to the set of those peers that not only realize the choreography, but are also compatible to the completely specified peers. By using service automata as uniform formalism

to model both choreographies and peer implementations, we can support this mixed scenario as follows.

The choreography specifies the global interaction of all peers, whereas a completely specified peer only specifies its local communication protocol. The service automaton describing this protocol can be transformed into a constraint automaton (see Def. 3.1) be replacing each x-labeled transition with a transition that is labeled with the set of all x-labeled transition of the choreography automaton. The product of the choreography automaton and this constraint automaton is then a multiport automaton whose terminating runs are conversations of the choreography and than can be projected to terminating runs of the service automaton. This multiport automaton can then be transformed into a choreography automaton by removing all deadlocking states and by collapsing all τ-transitions. The latter operation is standard for finite automata [85] and preserves the language of the automaton. The resulting choreography automaton can then be analyzed as before.

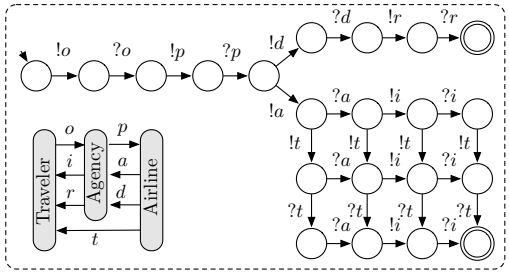
A combination of classical BPMN constructs together with iBPMN extensions [48] (i. e., modeling processes both inside and and outside pools) could be used to present this mixed choreography modeling approach to modelers with a unique graphical representation. The combination of both modeling approaches is also supported the recent standard of BPMN 2.0 [151].

EXAMPLE. Figure 7.11 sketches an example for the proposed combination of choreography models. The observable behavior of a simplified version of the traveler example (cf. Fig. 5.13) is given as choreography automaton (a). It specifies the interplay between a traveler that sends an order (o) to a travel agency which quotes a price (p) from an airline. The airline can either accept (a) or declines (d) the booking. In the former case, the traveler receives an itinerary (i) from the travel agency and a ticket (t) from the airline. In the latter case, only a rejection (r) message is sent. This choreography can be completely realized (b). The realizing service automaton of the traveler can receive the itinerary and the ticket in any order.

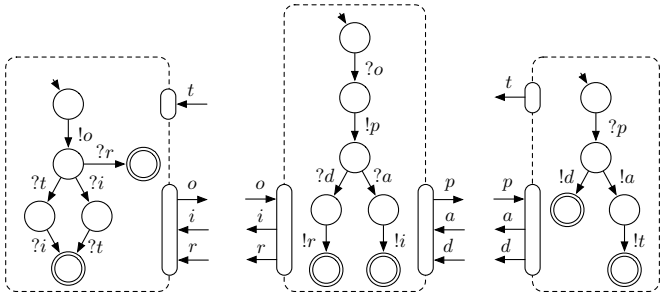
Now assume that instead of the realized service automaton, an already implemented traveler service (c) with more restricted behavior should be used in the service composition under design. This service automaton can be used as a constraint to the choreography specification, yielding a constrained choreography (d) in which the invoice is always received before the ticket.

7.6 RELATED WORK

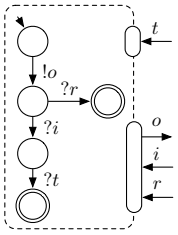
Realizability received much attention in recent literature, and was studied for most of the aforementioned interaction modeling languages, see [173] for a survey. Note that the term realizability as used in this thesis focuses on service choreographies. The classical notion of realizability is tightly linked to program/process synthesis



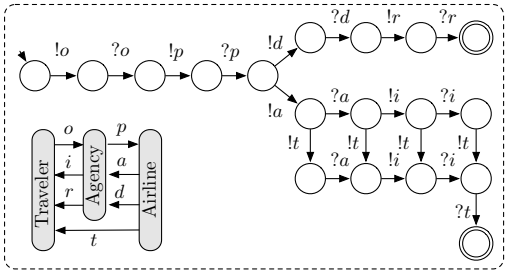
(a) choreography



(b) completely realizing service automata



(c) implemented service



(d) constrained choreography

Figure 7.11: Combination of choreography models.

from logical specification, see, for instance, [179]. Beside the different specification languages, the approaches differ in (1) the expressiveness of the specification language (the main differences concern the support of arbitrary looping) and (2) the nature of the message exchange (synchronous vs. asynchronous) of the realizing peers. In the following, we classify related approaches into these two groups.

STRUCTURAL RESTRICTIONS. Alur et al. [8] present necessary and sufficient criteria to realize a choreography specified by a set of message sequence charts (MSCs) with a set of concurrent automata. Both synchronous and asynchronous forms of communication are supported. Their proposed algorithms are very efficient, but are limited to acyclic choreography specifications, because the MSC model used in the paper does not support arbitrary iteration which excludes models such as Fig. 7.7(c).

Salaün and Bultan [166] investigate complete and partial realizability of choreographies specified by collaboration diagrams. The authors express the realizability problem in terms of LOTOS and present a case study conducted with a LOTOS verification tool. Their approach tackles both synchronous and asynchronous communication (using bounded FIFO queues). Collaboration diagrams, however, provide only limited support for repetitive behavior (only single events can be iterated and cycles such as in Fig. 7.7(c) cannot be expressed) and choices (events can be skipped, but complex decisions cannot be modeled). These restrictions also apply to [32] in which sufficient conditions for complete realizability of collaboration diagrams are elaborated. A tool to check the sufficient criteria of [32, 71, 72] is presented by Bultan et al. [31]. Using this tool, the authors showed that many collaboration diagrams in literature are unrealizable.

COMMUNICATION MODELS. Realizability of conversation protocols by asynchronously communicating Büchi automata is examined by Fu et al. [71]. The authors show decidability of the problem and define a sufficient condition for complete realizability. One of the prerequisites, *synchronous compatibility*, heavily restricts asynchronous communication.

Algorithms to check choreographies for partial realizability are discussed by Zaha et al. [195]. Both the global and local model are specified in Let's Dance and only atomic message exchanges considered. Decker and Weske [55] study realizability of interaction Petri nets. To the best of our knowledge, it is the only approach in which (complete and partial) realizability is not defined in terms of complete trace equivalence (cf. Def. 7.3). Instead, the authors require the participant implementations and the choreography to be branching bisimilar. Message exchange specified by interaction Petri nets is, however, inherently synchronous.

Kazhamiakin and Pistore [91] study a variety of communication models and their impact on realizability. They provide an algorithm that finds the "simplest" communication model under which a given choreography can be completely realized. Their approach is limited to complete realizability and gives no diagnosis information in case the choreography cannot be implemented by participants. Furthermore, they fix the communication model for all messages instead of allowing different communication models for each message.

OTHER ASPECTS. For other issues of choreographies such as instantiation, reference passing, or compliance checking, verification techniques are available for Let's

Dance [56]) and WS-CDL [36, 41]. Alur et al. [9] report decidability results in context of MSC graphs.

McIlvenna et al. [133] describes an approach to derive an orchestrator service from a service choreography given as interconnected model. This orchestrator is motivated by possible added value and flexibility for the participating services rather than by choreography verification.

Declarative modeling of choreographies is studied by Montali et al. [139]. The authors compare declarative models with interconnected models and report several advantages of the former modeling style. In particular, a declarative model does not necessarily be closed, but additionally constraints can be easily added. Furthermore, such a model is guaranteed to specify maximal behavior. However, the question how to realize a declarative choreography model by local service implementation is not addressed yet.

The original contribution is an automaton framework to specify arbitrary regular choreographies, check for various realizability notations, and to synthesize participants services that implement as much behavior as possible. Thereby, it is possible to define the message model individually for each message. Additionally, the defined synthesis algorithm provides diagnosis information that can help to fix choreographies toward complete realizability.

7.7 CONCLUSION

In this chapter, we linked the realizability problem of choreographies to the controllability problem of orchestrations. The close relationship between orchestrations and choreographies on the one hand and controllability and realizability on the other hand has been anticipated before: Papazoglou et al. [154] sketch a research road map for service-oriented computing. On service orchestrations and service choreographies, they comment:

> This sharp distinction between orchestration and choreography is rather artificial, and the consensus is that they should coalesce in a single language and environment.

Dumas et al. [65] review related notions for services such as realizability, substitutability and controllability. Substitutability can be realized with operating guidelines and is hence naturally related to controllability. However:

> The problem of controllability is intuitively related to that of realizability — as that they both result when internal choices are not externalized as messages. However, a formal relation between controllability and realizability is yet to be established.

The close relationship between these problems offers a uniform way to analyze and model arbitrary interacting services. Hence, we were able to reuse techniques that were originally proposed to check for controllability. These techniques resulted in a formal framework that allows to specify and analyze choreographies with both synchronous and asynchronous communication. Realizing choreographies offers a *correctness-by-construction* alternative to the composition of existing services: Compatibility of the realized peers follows from the realizability algorithm rather than from an a posteriori check. In addition, we refined the existing hierarchy of realizability notions by defining the novel notion of distributed realizability. Finally, we proposed to combine interaction models and interconnected models.

By reducing realizability to decentralized controllability, we also inherited the limitations of the synthesis algorithm. That is, we currently cannot guarantee that the algorithm sketched in Def. 7.10 always terminates for cyclic choreography models. This is subject of future work. The introduction of χ-arcs allows us to resolve dependencies without deleting states. The resulting service automaton can be seen as a most-permissive strategy for decentralized controllability. In future work, we need to study the step toward a finite representation of decentralized strategies; that is, a decentralized operating guideline.

Finally, further consequences of the relationship between controllability and realizability need to be examined. For instance, controllability is used in several other applications such as test case generation [88] or service mediation [73]. We expect these techniques to be similarly applicable to choreographies.

8 CONCLUSIONS

The central research topic of this thesis was the correctness of services and their compositions. We investigated several scenarios of service-oriented computing (SOC) and defined formalizations, correctness notions, and verification algorithms. In this chapter, we provide a summary of the contributions and discuss the theoretical and practical limitations of the results presented. We conclude the thesis by sketching directions for future extensions of the contributions of this thesis.

8.1 SUMMARY OF CONTRIBUTIONS

We approached correctness of SOC from three different directions, each investigated in a separate part of this thesis. In the following, we briefly summarize the contributions of this thesis.

CORRECTNESS OF SERVICES

A fundamental correctness criterion for services is controllability. The presence of interaction partners is a necessary requirement for a service to be used in any service composition. In this thesis, we extended controllability in two aspects.

- In Chap. 3, we presented a refinement of controllability. Using behavioral constraints, the set of all potential interaction partners can be restricted to only those partners which additionally satisfy specified properties. We formalized behavioral constraints with constraint automata and product operators and extended the respective tools. As behavioral constraints are very flexible, they can be used in a variety of scenarios, ranging from service validation to service construction.
- Chapter 4 extended the controllability analysis algorithm with the ability to generate counterexamples in case a service is uncontrollable. We first discussed various design flaws which can render a service uncontrollable and showed that it is not possible to derive antipatterns to check and diagnose uncontrollability on the structure of a service. Consequently, we presented several modifications of the controllability algorithm and introduced a diagnosis approach to construct a counterexample.

CORRECTNESS OF SERVICE COMPOSITIONS

A service composition should behave similar to a monolithic system; that is, distribution aspects should not change the functionality in an undesirable manner. To this

end, correctness of service compositions can be expressed with classical requirements such as absence of deadlocks and boundedness of the system. Conceptually, correctness of service compositions can be verified using existing state-of-the-art checking techniques. To support the *design* of correct service compositions, this thesis makes three contributions in this area.

- In Chap. 5, we formalized with WS-BPEL and BPEL4Chor industrial service specification languages to make the verification techniques directly applicable to industrial service models. We showed that compatibility can be verified for large service compositions, because existing state space reduction techniques are effective in the setting of complete service compositions; that is, closed systems.
- We applied verification techniques presented in the first part of this thesis to support the design of service compositions. By treating an incomplete service composition as a single service with an interface to missing participants, we can check controllability of this service to synthesize a "communication skeleton" of the missing participants which is by construction compatible to the other participants.
- In Chap. 6, we presented techniques to automatically derive recommendations on how to correct incompatible service choreographies. Unlike the previous completion approach, the correction algorithm additionally takes the incorrect service model into account to derive minimal invasive edit actions.

CORRECTNESS OF SERVICE CHOREOGRAPHIES

As an alternative to create large systems by composing existing services, service choreographies have been introduced to specify the behavior of a service composition from a global perspective. Realizability of such specifications has already been studied in terms of various formalisms. In this thesis, we presented three contributions to realizability of choreographies.

- We defined a hierarchy of realizability notions. The novel concept of *distributed realizability* complements existing notions, and it turns out that many examples from literature, which are not realizable in the classical sense, are in fact distributedly realizable.
- We linked choreography realization to decentralized controllability [187]. This relationship not only allows us to reuse an existing algorithm to remove unrealizable conversations from a choreography specification, but it also facilitates the application of other techniques of this thesis, for instance behavioral constraints to refine choreography models.
- By defining choreographies with service automata, both synchronous and asynchronous communication are first-class citizens. This allows us to study choreography realization in the context of both communication paradigms.

		TOOL SUPPORT
CORRECTNESS BY CONSTRUCTION		
construction (Chap. 3)	restriction (Chap. 3)	Rachel
selection (Chap. 3)	realization (Chap. 7)	Wendy
correction (Chap. 6)	completion (Chap. 5)	Rebecca
CORRECTNESS BY VERIFICATION		
diagnosis (Chap. 6)		LoLA
verification (Chap. 5)	validation (Chap. 3)	Wendy
FORMALIZATION		
compatibility (Chap. 2)		
service automata (Chap. 2)	realizability (Chap. 7)	BPEL2oWFN
operating guidelines (Chap. 2)	controllability (Chap. 2)	

Figure 8.1: Classification of the thesis' contributions.

8.2 CLASSIFICATION OF CONTRIBUTIONS

The previous section summarized the contributions with respect to the point of view on a service-oriented system. For each scenario, we made contributions toward the ultimate goal of correct systems which are composed of interacting services. In this section, we classify the contributions alongside with the topics which we discussed in the introduction of this thesis, see Fig. 8.1. *This classification is not specific to SOC, but can be seen as a general roadmap to achieve correctness in other domains.*

FORMALIZATION

We formalized the relevant aspects of services with a *single mathematical formalism*: service automata. The formalization is not only a prerequisite to formally reason about services and to apply verification techniques, but also makes the results independent of specific service description languages. Using a simple formalism consisting only of states, transitions, and an interface, upcoming and existing languages are likely to be canonically translated into service automata. We only briefly sketched the translation of WS-BPEL and BPEL4Chor in this thesis, but also more complex features of such languages can be translated in a straightforward manner [119].

The choice of a single formalism can not be stressed enough. We used service automata to model single services, incomplete and complete service compositions, and service choreographies; that is, all aspects of SOC we studied in this thesis. This allowed us to naturally apply advanced techniques without the need to refine all results

for different questions. Additionally, the choice of only once formalism also simplified the development of software tools.

CORRECTNESS BY VERIFICATION

Once a system can be described by a single formal model, verification techniques, such as model checking, are applicable. This became apparent in Chap. 5 in which we could apply a standard model checking tool to verify large service compositions. In this setting, also the generation of counterexamples (i. e., witness paths) was straightforward.

The verification of open systems (i. e., single services or incomplete service compositions) was considerably more complex. On the one hand, parts of the overall system were unknown and needed to be synthesized. Consequently, we needed to include the specification (i. e., a behavioral constraint) into the synthesis algorithm to realize the validation scenario in Chap. 3. On the other hand, the generation of counterexamples for open systems is a challenging task, because we need to explain the absence of communication partners in terms of the given open system. In Chap. 4, we elaborated counterexamples for controllability which do not consist of witness paths to erroneous states, but rather of a summary of reasons which make compatible communication impossible.

CORRECTNESS BY CONSTRUCTION

Decades of research and tool development allow to investigate certain properties with such an efficiency [68] that verification techniques can be integrated into modeling tools where the model can be constantly checked and errors are — similar to a spell-checker in text editors — immediately displayed. Whereas such techniques are certainly effective in *detecting* errors as soon as possible, they do not constructively support the *design* of correct systems. To this end, we studied several correctness-by-construction scenarios for SOC:

- *Partner synthesis.* In Chap. 3, we constructed partners of services such that the composition satisfies a given behavioral constraint or restricted given operating guidelines using a behavioral constraint. Similarly, we completed choreographies in Chap. 5 by synthesizing missing participants. These constructed partners are "communication skeletons" which need to be manually refined. They are, however, correct by construction and may reduce the modeler's effort similar to the integration of frequently used patterns during the modeling of business processes [78].
- *Service selection.* Operating guidelines are a finite characterization of potentially infinite sets of compatible services which can be efficiently queried. We exploit this property in three scenarios: In Chap. 3, we used behavioral constraints (1) to query a service registry for any service which satisfies a given constraint. The returned services

can then be used as building blocks for larger service orchestrations. Furthermore, (2) we used behavioral constraints to refine the "find" operation of SOA for a given requestor service. Any returned service is correct by design in the sense that the composition is compatible and satisfies the given constraint. Finally, (3) we further refined the search of a partner within a set of services by using the similarity measure defined in Chap. 6.

– *Choreography realization.* Another instance of a correctness-by-construction scenario was studied in Chap. 7. Here, the actual construction phase (i. e., the projection of the choreography specification to the participants) is a straightforward operation compared with the prior analysis of the choreography to ensure the preservation of the specified global behavior.

TOOL SUPPORT

All algorithms of this thesis have been prototypically implemented in several open source software free tools which can be downloaded at http://service-technology.org/tools. In this thesis, we used the following tools:

– *Wendy* [121] is a tool to synthesize partners for services (see Chap. 5) and is used to check controllability and the satisfaction of behavioral constraints (see Chap. 3), generate diagnostic information (see Chap. 4) for uncontrollable services, and to calculate operating guidelines for services (see Chap. 3 and Chap. 6).

– *Rachel* [111] implements the correction algorithm described in Chap. 6. It takes a service automaton and an operating guideline as input and calculates minimal edit actions to correct the service automaton such that it matches the operating guideline.

– *Rebecca* [136] analyzes choreography specifications and projects realizable choreographies to a set of realizing service automata as described in Chap. 7.

– *Fiona* [131] implements (among other features) the product operation for operating guidelines as described in Chap. 3.

– *LoLA* [185] is a general-purpose Petri net model checking tool and is used to verify compatibility in Chap. 5.

– *BPEL2oWFN* [107] is a compiler to translate WS-BPEL services and BPEL4Chor choreographies into formal models and is used to derive the service models which are used in the case studies throughout this thesis. The implemented formal semantics are described in Chap. 5.

The first three tools mentioned (Wendy, Rachel, and Rebecca) were originally developed to conduct the experiments presented in this thesis. These experiments

prove basic applicability of the results of this thesis. Whereas some implemented algorithms already scale to industrial service models, other algorithms still need further optimizations.

Unsurprisingly, the compatibility verification of service compositions in Chap. 5 could efficiently verify even large models. As the analysis of closed system is has been studied well before the advent of SOC, we could use existing techniques. Moreover, the model checking tool LoLA [185] implements a variety of reduction techniques. For service-specific properties, such as controllability, currently only the reduction rules of Weinberg [184] are known. These rules are, however, not applicable in case operating guidelines are considered.

Each of the described scenarios was approached on a technical level. The presented tools were optimized with respect to performance and memory consumption — an integration into modeling tools and acceptance tests are out of scope of this thesis.

8.3 LIMITATIONS AND OPEN PROBLEMS

All results of this thesis use compatibility as basic correctness notion for service behavior. Whereas some problems can be straightforwardly be extended to more elaborate correctness notions, other results required several restrictions and are not yet applicable to arbitrary service automata. This section evaluates the core concepts used in this thesis with respect to their limitations and extendability.

COMPATIBILITY. To check a closed service composition for compatibility is a standard model checking problem. In this realm, not only absence of deadlocks or livelocks, but temporal logics such as LTL or CTL can be automatically checked. To this end, the verification results of Chap. 5 should easily be adjusted to any refined compatibility criterion. Furthermore, several tools exist to automatically check these more sophisticated compatibility notions.

CONTROLLABILITY. The partner synthesis scenarios of Chap. 3 and Chap. 5 are based on controllability which has been defined as an extension of compatibility in this thesis. Wolf [187] already presented an algorithm to extend controllability to *weak termination*; that is, also livelocks are excluded. This algorithm is already implemented in the tool Wendy [121] together with an extension of the diagnosis algorithm [112] which was originally defined for weak termination.

OPERATING GUIDELINES. Operating guidelines — as characterization of all compatible partners — are used in the validation scenario of Chap. 3 and the correction algorithm of Chap. 6. With annotated automata, more elaborate compatibility notions such as weak termination cannot be expressed: Wolf et al. [188] employ *state space fragments* to characterize weak terminating strategies. Whereas the application

of behavioral constraints is likely to be adjusted to this new characterization, the correction algorithm of Chap. 6 heavily relies on annotated automata.

Furthermore, the correction algorithm is currently only applicable to acyclic and deterministic services. The former requirement allows to treat the incorrect service and the operating guideline as trees which in turn allows to define local edit actions which do not affect other states. Determinism in turn can be seen as a simplification of the approach rather than a conceptual problem. As both restrictions exclude a large class of practically relevant service models, the extension of the correction approach to arbitrary service automata is subject of future work.

REALIZABILITY. The algorithm to remove unrealizable conversations from choreography specifications inherits the restrictions of the original algorithm from Wolf et al. [188] to synthesize decentralized strategies: the model needs to be acyclic. The current algorithm is sound, but not complete, and the analysis of cyclic choreographies may not terminate. Similar to the correction algorithm, the removal of dependencies must neither exclude choices nor affect other states. Consequently, an extension to cyclic models is subject of future work.

To conclude, we employed our notion of compatibility as a "least common multiple" of all techniques in this thesis and decided not to switch the correctness criteria between the chapters. Moreover, for practical applications, more refined correctness notions may be needed.

8.4 FUTURE WORK

In this thesis, we fixed with compatibility a correctness notion and formalized services, service compositions, and service choreographies with service automata and investigated correctness in a variety of scenarios. We already justified our design decisions and listed the limitations of the contributions. This naturally brings us to several directions of future work.

REFINED VERIFICATION. The extension of compatibility toward weak termination is a canonic next step. Several approaches are immediately applicable to this refined correctness notion, cf. Sect. 8.3. In Chap. 3, we discussed *cover constraints* [172] as potential extensions to behavioral constraints. This extension only requires little changes to our formal model compared to more expressive constraints such as temporal logics.

Another field of research is to include further aspects to the verification. So far, we entirely focused on the communication protocol of a service and abstracted from other aspects such as data, nonfunctional properties, or instantiation. Consequently, these aspects are not considered during the verification and their integration would

broaden applicability of our results. Heinze et al. [83] present such an extension with an algorithm to refine service models to faithfully model data-driven decisions.

RELAXATION OF RESTRICTIONS. The previous section also listed the current restrictions of the correction and the realization algorithms. The former algorithm is based on work of Sokolsky et al. [171], and the authors present a translation into *linear programming*. This allows to analyze cyclic models and may also be applicable to the edit distance used for correction. For the realization algorithm, its relationship to *region theory* [13] needs to be investigated, because both approaches aim at deriving a distributed model from a global specification.

IMPROVED ALGORITHMS. For the verification of compatibility, several effective state space reduction techniques are available. For controllability, we applied reduction rules of Weinberg [184] which aim at reducing the size of synthesized partners. To also fight state space explosion during partner synthesis, reductions such as the *partial order reduction* need to be adjusted to preserve controllability.

LINK TO PETRI NETS. In this thesis, we only employed Petri nets in Chap. 5, because they allowed for the application of effective state space reduction techniques implemented in the tool LoLA. Other approaches use the *state equation* [102] to efficiently decide necessary or sufficient criteria to realize efficient "quick checks" for compatibility [174, 148]. These checks can be used to restrict the search space in the selection and the correction scenario, and may also be integrated in the diagnosis algorithm. Recently, we presented a compositional calculation of operating guidelines for Petri net models with free choice conflict clusters [109]. Regularities of operating guidelines allow for a compact representation as Petri net [122]. This translation is based on region theory and is currently only applicable a posteriori. Therefore, an investigation of how to realize further algorithms based on Petri nets appears to be a promising direction for future work.

INTEGRATION WITH MODELING. This thesis focused on the formal verification of service models. As a next step, the presented verification tools need to be integrated into modeling tools to allow for correctness checks in early design phases of services. Currently, we are working on an integration into the modeling tool *Oryx* [53] and the process mining framework *ProM* [2]. Such an integration also requires to visualize the verification results (e. g., counterexamples or synthesized partners) in the original modeling language. First approaches [101, 3, 114] study the translation of formal models to WS-BPEL.

THESES

1. Correctness plays an important role in distributed systems, such as service-oriented architectures. As service compositions often implement interorganizational business processes, a single flawed service can cause unpredictable problems, which in turn may cause legal and financial consequences.
2. The behavior of a service composition should similar to that of a monolithic system; that is, distribution aspects should not change the functionality in an undesirable manner. To this end, correctness of service compositions can be expressed with classical requirements, such as absence of deadlocks and boundedness of the system.
3. Existing correctness criteria of business processes are not suitable to embrace the communicating nature of services. Hence novel correctness criteria are required to examine services. Controllability is a fundamental correctness notion for services. Although introduced to analyze service orchestrations, it is also suitable for investigating service compositions or even service choreographies.
4. Although services are always executed in a composition, it is possible to analyze and validate a service in isolation and still make statements about the correctness of the interaction with any other service. This local check can detect and avoid errors in the early design stages of service compositions.
5. When checking correctness, simple yes-no answers are insufficient during any development phase of a service composition. Only detailed diagnosis information and counterexamples help to understand, avoid, and fix errors.
6. Errors in interorganizational business processes further raise questions with respect to responsibility. If a participant can be identified as a scapegoat for the error, correction proposals can be automatically calculated.
7. Various artifacts in the area of service-oriented computing (ranging from single services to service compositions and service choreographies) and related correctness notions can be expressed in terms of a single formalism: service automata. This allows (1) to reuse and combine algorithms and techniques to examine the behavior of services, (2) to facilitate the development of software tools, and (3) to elaborate results that are independent of concrete industrial specification languages.
8. Formal methods also offer support for the early design phase of service compositions. Missing participants of a service composition can be synthesized. Service models can also be derived automatically from global behavioral specifications (contracts, choreographies). Such generated models are correct by construction.

BIBLIOGRAPHY

[1] Aalst, W.M.P.v.d.: The application of Petri nets to workflow management. Journal of Circuits, Systems and Computers **8**(1), 21–66 (1998) (Cited on pages 60, 64, 65, and 103.)

[2] Aalst, W.M.P.v.d., Dongen, B.F.v., Günther, C.W., Mans, R.S., de Medeiros, A.K.A., Rozinat, A., Rubin, V., Song, M., Verbeek, H.M.W.E., Weijters, A.J.M.M.: ProM 4.0: Comprehensive support for *real* process analysis. In: ICATPN 2007, LNCS 4546, pp. 484–494. Springer (2007) (Cited on page 162.)

[3] Aalst, W.M.P.v.d., Lassen, K.B.: Translating unstructured workflow processes to readable BPEL: Theory and implementation. Information & Software Technology **50**(3), 131–159 (2008) (Cited on pages 101, 105, and 162.)

[4] Aalst, W.M.P.v.d., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: From public views to private views — correctness-by-design for services. In: WS-FM 2007, LNCS 4937, pp. 139–153. Springer (2008) (Cited on page 104.)

[5] Aalst, W.M.P.v.d., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty contracts: Agreeing and implementing interorganizational processes. Comput. J. **53**(1), 90–106 (2010) (Cited on page 104.)

[6] Aït-Bachir, A., Dumas, M., Fauvet, M.C.: Detecting behavioural incompatibilities between pairs of services. In: ICSOC Workshops 2008, LNCS 5472, pp. 79–90. Springer (2008) (Cited on page 128.)

[7] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer (2003) (Cited on pages 13 and 83.)

[8] Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. IEEE Trans. Software Eng. **29**(7), 623–633 (2003) (Cited on pages 17, 39, 133, 136, 137, and 151.)

[9] Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theor. Comput. Sci. **331**(1), 97–114 (2005) (Cited on page 152.)

[10] Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**, 672–713 (2002) (Cited on page 55.)

[11] Alves, A., et al.: Web Services Business Process Execution Language Version 2.0. OASIS Standard, OASIS (2007). URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf` (Cited on pages 15, 24, 62, 81, 83, 87, 126, and 176.)

[12] Andrews, T., et al.: Business Process Execution Language for Web Services, Version 1.1. Tech. rep., BEA Systems, IBM, Microsoft (2003). URL `http://www.ibm.com/developerworks/library/ws-bpel` (Cited on page 87.)

[13] Badouel, E., Darondeau, P.: Theory of regions. In: Advanced Course on Petri Nets, LNCS 1491, pp. 529–586. Springer (1996) (Cited on page 162.)

[14] Baeten, J.C.M.: A brief history of process algebra. Theor. Comput. Sci. **335**(2-3), 131–146 (2005) (Cited on pages 24 and 38.)

[15] Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008) (Cited on page 96.)

[16] Balbiani, P., Cheikh, F., Feuillade, G.: Composition of interactive Web services based on controller synthesis. In: 2008 IEEE Congress on Services, pp. 521–528. IEEE (2008) (Cited on page 39.)

[17] Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments. In: ICSOC 2006, LNCS 4294, pp. 339–351. Springer (2006) (Cited on page 38.)

[18] Barreto, C., et al.: Web Services Business Process Execution Language Version 2.0 Primer. Primer, OASIS (2007). URL `http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf` (Cited on page 83.)

[19] Bellman, R.: Dynamic Programming. Princeton University Press (1957) (Cited on page 125.)

[20] Benatallah, B., Casati, F., Toumani, F.: Representing, analysing and managing Web service protocols. Data Knowl. Eng. **58**(3), 327–357 (2006) (Cited on page 56.)

[21] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Mecella, M.: Automatic composition of e-services that export their behavior. In: ICSOC 2003, LNCS 2910, pp. 43–58. Springer (2003) (Cited on page 38.)

[22] Berardi, D., Calvanese, D., Giacomo, G.D., Hull, R., Mecella, M.: Automatic composition of transition-based semantic Web services with messaging. In: VLDB 2005, pp. 613–624. ACM (2005) (Cited on page 39.)

[23] Berardi, D., Calvanese, D., Giacomo, G.D., Mecella, M.: Composition of services with nondeterministic observable behavior. In: ICSOC 2005, LNCS 3826, pp. 520–526. Springer (2005) (Cited on page 56.)

[24] Bianchini, D., Antonellis, V.D., Melchiori, M.: Evaluating similarity and difference in service matchmaking. In: EMOI-INTEROP 2006, CEUR Workshop Proceedings Vol. 200. CEUR-WS.org (2006) (Cited on page 128.)

[25] Bochmann, G.V., Sunshine, C.A.: Formal methods in communication protocol design. IEEE Transactions on Communications **28**(4), 624–631 (1980) (Cited on page 37.)

[26] Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983) (Cited on page 39.)

[27] Breugel, F.v., Koshkina, M.: Models and verification of BPEL (2006). URL `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`. Unpublished manuscript (Cited on pages 38, 88, and 103.)

[28] Brogi, A., Popescu, R.: Automated generation of BPEL adapters. In: ICSOC 2006, LNCS 4294, pp. 27–39. Springer (2006) (Cited on page 128.)

[29] Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing finite Kripke structures in propositional temporal logic. Theor. Comput. Sci. **59**(1-2), 115–131 (1988) (Cited on page 22.)

[30] Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Computers **C-35**(8), 677–691 (1986) (Cited on page 12.)

[31] Bultan, T., Ferguson, C., Fu, X.: A tool for choreography analysis using collaboration diagrams. In: ICWS 2009, pp. 856–863. IEEE (2009) (Cited on pages 135 and 151.)

[32] Bultan, T., Fu, X.: Specification of realizable service conversations using collaboration diagrams. SOCA **2**(1), 27–39 (2008) (Cited on pages 39, 136, 137, 141, and 151.)

[33] Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: WWW 2003, pp. 403–410. ACM (2003) (Cited on pages 39, 103, and 135.)

[34] Bunke, H.: On a relation between graph edit distance and maximum common subgraph. Pattern Recogn. Lett. **18**(8), 689–694 (1997) (Cited on page 112.)

[35] Bunke, H., Shearer, K.: A graph distance metric based on the maximal common subgraph. Pattern Recogn. Lett. **19**(3-4), 255–259 (1998) (Cited on page 112.)

[36] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: COORDINATION 2006, LNCS 4038, pp. 63–81. Springer (2006) (Cited on page 152.)

[37] Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Kluwer Academic Publishers (1999) (Cited on page 28.)

[38] Chinnici, R., Gudgin, M., Moreau, J.J., Weerawarana, S.: Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language. World Wide Web Consortium, Working Draft

WD-wsdl12-20030611 (2003) (Cited on pages 15, 36, 86, and 176.)

[39] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999) (Cited on pages 12, 48, 65, 76, and 96.)

[40] Corrales, J.C., Grigori, D., Bouzeghoub, M.: BPEL processes matchmaking for service discovery. In: OTM Conferences 2006 (1), LNCS 4275, pp. 237–254. Springer (2006) (Cited on page 127.)

[41] Corredini, F., De Angelis, F., Polzonetti, A.: Verification of WS-CDL choreographies. In: YR-SOC 2007, pp. 13–18. University of Leicester (2007) (Cited on page 152.)

[42] Davulcu, H., Kifer, M., Ramakrishnan, I.V.: CTR-S: A logic for specifying contracts in semantic web services. In: WWW 2004, pp. 144–153. ACM (2004) (Cited on page 56.)

[43] de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC 2001, pp. 109–120. ACM (2001) (Cited on page 38.)

[44] De Giacomo, G., Patrizi, F.: Automated composition of nondeterministic services. In: WS-FM 2009, LNCS. Springer (2009). (in press) (Cited on page 56.)

[45] Decker, G.: Design and analysis of process choreographies. Ph.D. thesis, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany (2009) (Cited on pages 137 and 142.)

[46] Decker, G.: Realizability of interaction models. In: ZEUS 2009, CEUR Workshop Proceedings Vol. 438, pp. 55–60. CEUR-WS.org (2009) (Cited on pages 137 and 142.)

[47] Decker, G., Barros, A., Kraft, F.M., Lohmann, N.: Non-desynchronizable service choreographies. In: ICSOC 2008, LNCS 5364, pp. 331–346. Springer (2008) (Cited on page 146.)

[48] Decker, G., Barros, A.P.: Interaction modeling using BPMN. In: BPM Workshops 2007, LNCS 4928, pp. 208–219. Springer (2007) (Cited on pages 15, 136, 149, and 175.)

[49] Decker, G., Kopp, O., Leymann, F., Pfitzner, K., Weske, M.: Modeling service choreographies using BPMN and BPEL4Chor. In: CAiSE 2008, LNCS 5074, pp. 79–93. Springer (2008) (Cited on page 103.)

[50] Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for modeling choreographies. In: ICWS 2007, pp. 296–303. IEEE (2007) (Cited on pages 15, 24, 82, 84, 85, 86, 87, 97, 148, and 175.)

[51] Decker, G., Kopp, O., Leymann, F., Weske, M.: Interacting services: From specification to execution. Data Knowl. Eng. **68**, 946–972 (2009) (Cited on pages 86, 103, and 104.)

[52] Decker, G., Kopp, O., Puhlmann, F.: Service referrals in BPEL-based choreographies. In: YR-SOC 2007, pp. 25–30. University of Leicester (2007) (Cited on page 104.)

[53] Decker, G., Overdick, H., Weske, M.: Oryx - an open modeling platform for the bpm community. In: BPM 2008, LNCS 5240, pp. 382–385. Springer (2008) (Cited on page 162.)

[54] Decker, G., Weske, M.: Behavioral consistency for B2B process integration. In: CAiSE 2007, LNCS 4495, pp. 81–95. Springer (2007) (Cited on page 103.)

[55] Decker, G., Weske, M.: Local enforceability in interaction Petri nets. In: BPM 2007, LNCS 4714, pp. 305–319. Springer (2007) (Cited on pages 38, 135, 136, 137, and 151.)

[56] Decker, G., Zaha, J.M., Dumas, M.: Execution semantics for service choreographies. In: WS-FM 2006, LNCS 4184, pp. 163–177. Springer (2006) (Cited on page 152.)

[57] Dehnert, J., Aalst, W.M.P.v.d.: Bridging the gap between business models and workflow specifications. Int. J. Cooperative Inf. Syst. **13**(3), 289–332 (2004) (Cited on page 65.)

[58] DeRemer, F., Kron, H.H.: Programming-in-the-large versus programming-in-the-small. IEEE Trans. Software Eng. **2**(2), 80–86 (1976) (Cited on pages 13, 81, and 83.)

[59] Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press (1995) (Cited on page 65.)

[60] Desel, J., Reisig, W.: Place or transition Petri nets. In: Advanced Course on Petri Nets, LNCS 1491, pp. 122–173. Springer (1998) (Cited on page 88.)

[61] Dijkman, R.M.: A classification of differences between similar BusinessProcesses. In: EDOC 2007, pp. 37–50. IEEE (2007) (Cited on page 128.)

[62] Dijkman, R.M.: Diagnosing differences between business process models. In: BPM 2008, LNCS 5240. Springer (2008) (Cited on page 128.)

[63] Dijkman, R.M., Dumas, M.: Service-oriented design: A multi-viewpoint approach. Int. J. Cooperative Inf. Syst. **13**(4), 337–368 (2004) (Cited on pages 81 and 134.)

[64] Dijkman, R.M., Dumas, M., García-Bañuelos, L.: Graph matching algorithms for business process model similarity search. In: BPM 2009, vol. LNCS 5701, pp. 48–63. Springer (2009) (Cited on pages 127 and 129.)

[65] Dumas, M., Benatallah, B., Nezhad, H.R.M.: Web service protocols: Compatibility and adaptation. IEEE Data Eng. Bull. **31**(3), 40–44 (2008) (Cited on page 152.)

[66] Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In: BPM 2006, LNCS 4102, pp. 65–80. Springer (2006) (Cited on page 128.)

[67] Esparza, J., Heljanko, K.: Unfoldings: A Partial-Order Approach to Model Checking. Springer (2008) (Cited on page 47.)

[68] Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: BPM 2009, LNCS 5701, pp. 278–293. Springer (2009) (Cited on pages 78, 103, 105, and 158.)

[69] Foster, H., Uchitel, S., Magee, J., Kramer, J.: Compatibility verification for Web service choreography. In: ICWS 2004, pp. 738–741. IEEE (2004) (Cited on page 103.)

[70] Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL Web services. In: WWW 2004, pp. 621–630. ACM (2004) (Cited on page 39.)

[71] Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. Theor. Comput. Sci. **328**(1-2), 19–37 (2004) (Cited on pages 17, 133, 137, 141, and 151.)

[72] Fu, X., Bultan, T., Su, J.: Synchronizability of conversations among Web services. IEEE Trans. Software Eng. **31**(12), 1042–1055 (2005) (Cited on pages 39, 146, and 151.)

[73] Gierds, C., Mooij, A.J., Wolf, K.: Reducing adapter synthesis to controller synthesis. IEEE T. Services Computing (2010). (accepted for publication in July 2010) (Cited on pages 128 and 153.)

[74] Gottschalk, K.: Web Services Architecture Overview. IBM whitepaper, IBM developerWorks (2000). URL `http://ibm.com/developerWorks/web/library/w-ovr` (Cited on pages 14 and 176.)

[75] Graf, S., Steffen, B.: Compositional minimization of finite state systems. In: CAV 1990, LNCS 531, pp. 186–196. Springer (1991) (Cited on page 55.)

[76] Grigori, D., Corrales, J.C., Bouzeghoub, M.: Behavioral matchmaking for service retrieval: Application to conversation protocols. Inf. Syst. **33**(7-8), 681–698 (2008) (Cited on page 127.)

[77] Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. STTT **8**(3), 229–247 (2006) (Cited on pages 108 and 127.)

[78] Gschwind, T., Koehler, J., Wong, J.: Applying patterns during business process modeling. In: BPM 2008, LNCS 5240, pp. 4–19. Springer (2008) (Cited on page 158.)

[79] Günay, A., Yolum, P.: Structural and semantic similarity metrics for Web service matchmaking. In: EC-Web 2007, LNCS 4655, pp. 129–138. Springer (2007) (Cited on page 128.)

[80] Haas, H., Brown, A.: Web Services Glossary. W3C Working Group Note 11 February 2004, W3C (2004). URL `http://www.w3.org/TR/ws-gloss` (Cited on page 135.)

[81] Hamadi, R., Benatallah, B.: A Petri net-based model for Web service composition. In: ADC 2003, CRPIT 17, pp. 191–200. Australian Computer Society (2003) (Cited on page 103.)

[82] Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths in graphs. IEEE Trans. Syst. Sci. and Cybernetics **SSC-4**(2), 100–107 (1968) (Cited on page 130.)

[83] Heinze, T.S., Amme, W., Moser, S.: A restructuring method for WS-BPEL business processes based on extended workflow graphs. In: BPM 2009, LNCS 5701, pp. 211–228. Springer (2009) (Cited on page 162.)

[84] Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: BPM 2005, LNCS 3649, pp. 220–235. Springer (2005) (Cited on page 87.)

[85] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979) (Cited on pages 37, 51, 143, 144, and 149.)

[86] IDC: Improving Software Quality to Drive Business Agility (2008). Available at `http://www.coverity.com/library/pdf/IDC_Improving_Software_Quality_June_2008.pdf`. (Cited on page 11.)

[87] ITU-T: Message Sequence Chart (MSC). Recommendation Z.120, International Telecommunication Union (2004) (Cited on page 176.)

[88] Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: ICSOC 2008 Workshops, LNCS 5472, pp. 66–78. Springer (2009) (Cited on page 153.)

[89] Kaschner, K., Wolf, K.: Set algebra for service behavior: Applications and constructions. In: BPM 2009, LNCS 5701, pp. 193–210. Springer (2009) (Cited on pages 56 and 57.)

[90] Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web Services Choreography Description Language Version 1.0. W3C Candidate Recommendation, W3C (2005). URL `http://www.w3.org/TR/ws-cdl-10` (Cited on pages 24 and 176.)

[91] Kazhamiakin, R., Pistore, M.: Analysis of realizability conditions for Web service choreographies. In: FORTE 2006, LNCS 4229, pp. 61–76. Springer (2006) (Cited on page 151.)

[92] Kazhamiakin, R., Pistore, M., Santuari, L.: Analysis of communication models in Web service compositions. In: WWW 2006, pp. 267–276. ACM (2006) (Cited on pages 38 and 39.)

[93] Kindler, E.: A compositional partial order semantics for Petri net components. In: ICATPN 1997, LNCS 1248, pp. 235–252. Springer (1997) (Cited on page 39.)

[94] Kindler, E., Martens, A., Reisig, W.: Inter-operability of workflow applications: Local criteria for global soundness. In: BPM 2000, LNCS 1806, pp. 235–253. Springer (2000) (Cited on page 39.)

[95] König, D., Lohmann, N., Moser, S., Stahl, C., Wolf, K.: Extending the compatibility notion for abstract WS-BPEL processes. In: WWW 2008, pp. 785–794. ACM (2008) (Cited on page 104.)

[96] Kopp, O., Leymann, F.: Do we need internal behavior in choreography models? In: ZEUS 2009, CEUR Workshop Proceedings Vol. 438, pp. 68–73. CEUR-WS.org (2009) (Cited on page 135.)

[97] Kupferman, O., Vardi, M.: Interactive Computation – The New Paradigm, chap. Verification of Open Systems, pp. 97–117. Springer (2006) (Cited on page 55.)

[98] Kupferman, O., Vardi, M.Y.: Module checking. In: CAV 1996, LNCS 1102, pp. 75–86. Springer (1996) (Cited on page 55.)

[99] Kupferman, O., Vardi, M.Y., Wolper, P.: Module checking. Inf. Comput. **164**(2), 322–344 (2001) (Cited on page 55.)

[100] Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and resolving process model differences in the absence of a change log. In: BPM 2008, LNCS 5240. Springer (2008) (Cited on page 128.)

[101] Lassen, K.B., Aalst, W.M.P.v.d.: WorkflowNet2BPEL4WS: A tool for translating unstructured workflow processes to readable BPEL. In: OTM Conferences 2006 (1), LNCS 4275, pp. 127–144. Springer (2006) (Cited on pages 101, 105, and 162.)

[102] Lautenbach, K.: Liveness in Petri nets. Interner Bericht ISF-75-02.1, GMD Bonn, Bonn, Germany (1975) (Cited on page 162.)

[103] Lautenbach, K., Ridder, H.: Liveness in bounded Petri nets which are covered by T-invariants. In: Application and Theory of Petri Nets 1994, LNCS 815, pp. 358–375. Springer (1994) (Cited on page 65.)

[104] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Dokl. **10**(8), 707–710 (1966) (Cited on page 112.)

[105] Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR (1999) (Cited on page 84.)

[106] Li, C., Reichert, M., Wombacher, A.: Discovering reference models by mining process variants using a heuristic approach. In: BPM 2009, LNCS 5701, pp. 344–362. Springer (2009) (Cited on page 128.)

[107] Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. Informatik-Berichte 212, Humboldt-Universität zu Berlin, Berlin, Germany (2007). Tool available at `http://service-technology.org/bpel2owfn`. (Cited on pages 19, 35, 88, 91, 94, 126, and 159.)

[108] Lohmann, N.: Correcting deadlocking service choreographies using a simulation-based graph edit distance. In: BPM 2008, LNCS 5240, pp. 132–147. Springer (2008) (Cited on pages 16, 18, and 107.)

[109] Lohmann, N.: Decompositional calculation of operating guidelines using free choice conflicts. In: AWPN 2008, CEUR Workshop Proceedings Vol. 380, pp. 63–68. CEUR-WS.org (2008) (Cited on pages 65 and 162.)

[110] Lohmann, N.: A feature-complete Petri net semantics for WS-BPEL 2.0. In: WS-FM 2007, LNCS 4937, pp. 77–91. Springer (2008) (Cited on pages 57, 77, 88, 91, and 129.)

[111] Lohmann, N.: Rachel: a tool to correct service choreographies. Tool available at `http://service-technology.org/rachel`. (2008) (Cited on pages 19, 125, and 159.)

[112] Lohmann, N.: Why does my service have no partners? In: WS-FM 2008, LNCS 5387, pp. 191–206. Springer (2009) (Cited on pages 16, 17, 59, and 160.)

[113] Lohmann, N.: Communication models for services. In: ZEUS 2010, CEUR Workshop Proceedings Vol. 563, pp. 9–16. CEUR-WS.org (2010) (Cited on page 39.)

[114] Lohmann, N., Kleine, J.: Fully-automatic translation of open workflow net models into simple abstract BPEL processes. In: Modellierung 2008, *Lecture Notes in Informatics (LNI)*, vol. P-127, pp. 57–72. GI (2008) (Cited on pages 77, 101, 105, 129, and 162.)

[115] Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and participant synthesis. In: WS-FM 2007, LNCS 4937, pp. 46–60. Springer (2008) (Cited on pages 16, 18, and 81.)

[116] Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting BPEL processes. In: BPM 2006, LNCS 4102, pp. 17–32. Springer (2006) (Cited on pages 57 and 129.)

[117] Lohmann, N., Massuthe, P., Wolf, K.: Behavioral constraints for services. In: BPM 2007, LNCS 4714, pp. 271–287. Springer (2007) (Cited on pages 16, 17, 43, and 47.)

[118] Lohmann, N., Massuthe, P., Wolf, K.: Operating guidelines for finite-state services. In: ICATPN 2007, LNCS 4546, pp. 321–341. Springer (2007) (Cited on pages 32, 33, 34, 35, 36, and 129.)

[119] Lohmann, N., Verbeek, H., Dijkman, R.M.: Petri net transformations for business processes – a survey. LNCS ToPNoC **II**(5460), 46–63 (2009). Special Issue on Concurrency in Process-Aware Information Systems (Cited on pages 38, 88, 103, and 157.)

[120] Lohmann, N., Verbeek, H., Ouyang, C., Stahl, C.: Comparing and evaluating Petri net semantics for BPEL. Int. J. Business Process Integration and Management **4**(1), 60–73 (2009) (Cited on pages 38, 88, and 103.)

[121] Lohmann, N., Weinberg, D.: Wendy: A tool to synthesize partners for services. In: PETRI NETS 2010, LNCS 6128, pp. 297–307. Springer (2010). Tool available at `http://service-technology.org/wendy`. (Cited on pages 18, 35, 53, 54, 76, 100, 126, 159, and 160.)

[122] Lohmann, N., Wolf, K.: Petrifying operating guidelines for services. In: ACSD 2009, pp. 80–88. IEEE Computer Society (2009) (Cited on pages 34, 38, and 162.)

[123] Lohmann, N., Wolf, K.: Realizability is controllability. In: WS-FM 2009, LNCS 6194, pp. 110–127. Springer (2010) (Cited on pages 16, 18, and 133.)

[124] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996) (Cited on page 38.)

[125] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer (1992) (Cited on page 48.)

[126] Martens, A.: Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services. Dissertation, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, Berlin, Germany (2003). (in German) (Cited on pages 39 and 65.)

[127] Martens, A.: Analyzing Web service based business processes. In: FASE 2005, LNCS 3442, pp. 19–33. Springer (2005) (Cited on page 103.)

[128] Massuthe, P.: Operating guidelines for services. Dissertation, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, Berlin, Germany; Eindhoven University of Technology, Eindhoven, The Netherlands (2009) (Cited on pages 32, 33, 34, 35, 36, 38, 52, and 54.)

[129] Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. Annals of Mathematics, Computing & Teleinformatics **1**(3), 35–43 (2005) (Cited on page 39.)

[130] Massuthe, P., Serebrenik, A., Sidorova, N., Wolf, K.: Can I find a partner? Undecidablity of partner existence for open nets. Inf. Process. Lett. **108**(6), 374–378 (2008) (Cited on pages 29 and 147.)

[131] Massuthe, P., Weinberg, D.: Fiona: A tool to analyze interacting open nets. In: AWPN 2008, CEUR Workshop Proceedings Vol. 380, pp. 99–104. CEUR-WS.org (2008). Tool available at `http://service-technology.org/fiona`. (Cited on pages 19, 35, 54, and 159.)

[132] Mcilroy, D.: Mass-produced software components. In: Software Engineering Concepts and Techniques, pp. 138–155. NATO Science Committee (1969) (Cited on page 13.)

[133] McIlvenna, S., Dumas, M., Wynn, M.T.: Synthesis of orchestrators from service choreographies. In: APCCM 2009, CRPIT 96, pp. 129–138. Australian Computer Society (2009) (Cited on page 152.)

[134] Medeiros, A.K.A.d., Aalst, W.M.P.v.d., Weijters, A.J.M.M.: Quantifying process equivalence based on observed behavior. Data Knowl. Eng. **64**(1), 55–74 (2008) (Cited on page 130.)

[135] Mendling, J.: Metrics for Process Models: Empirical Foundations of Verification, Error Prediction, and Guidelines for Correctness. LNBIP 6. Springer (2008) (Cited on page 103.)

[136] Mennicke, S., Lohmann, N.: Rebecca: a tool to realize service choreographies. Tool available at `http://service-technology.org/rebecca`. (2009) (Cited on pages 19, 145, and 159.)

[137] Merlin, P.M.: Specification and validation of protocols. IEEE Transactions on Communications **27**(11), 1671–1680 (1979) (Cited on page 37.)

[138] Milner, R.: A Calculus of Communicating Systems. Springer (1980) (Cited on pages 24 and 175.)

[139] Montali, M., Pesic, M., Aalst, W.M.P.v.d., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographies. TWEB **4**(1) (2010) (Cited on page 152.)

[140] Moody, D.L.: Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data Knowl. Eng. **55**(3), 243–276 (2005) (Cited on page 11.)

[141] Moser, S., Martens, A., Häbich, M., Mülle, J.: A hybrid approach for generating compatible WS-BPEL partner processes. In: BPM 2006, LNCS 4102, pp. 458–464. Springer (2006) (Cited on page 104.)

[142] Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989) (Cited on pages 88 and 94.)

[143] Namjoshi, K.S.: A simple characterization of stuttering bisimulation. In: FSTTCS 1997, LNCS 1346, pp. 284–296. Springer (1997) (Cited on page 113.)

[144] Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of Web services. In: WWW 2002, pp. 77–88. ACM (2002) (Cited on page 103.)

[145] National Institute of Standards and Technology: Software Errors Cost U.S. Economy $ 59.5 Billion Annually (2002). Available at `http://www.nist.gov/public_affairs/releases/no2-10.htm`. (Cited on page 11.)

[146] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of statecharts specifications. In: ICSE 2007, pp. 54–64. IEEE (2007) (Cited on page 113.)

[147] Nezhad, H.R.M., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: WWW 2007, pp. 993–1002. ACM (2007) (Cited on page 128.)

[148] Oanea, O., Wolf, K.: An efficient necessary condition for compatibility. In: ZEUS 2009, CEUR Workshop Proceedings Vol. 438, pp. 81–87. CEUR-WS.org (2009) (Cited on pages 38 and 162.)

[149] OMG: Unified Modeling Language (UML), Version 2.1.2. Tech. rep., Object Management Group (2007). URL `http://www.uml.org` (Cited on pages 129 and 176.)

[150] OMG: Business Process Model and Notation, V1.1. OMG Available Specification, Object Management Group (2008). URL `http://www.omg.org/spec/BPMN/1.1` (Cited on pages 15, 84, 108, 129, and 175.)

[151] OMG: Business Process Model and Notation (BPMN). FTF Beta 1 for Version 2.0, Object Management Group (2009). URL `http://www.omg.org/spec/BPMN/2.0` (Cited on pages 15, 84, 136, and 149.)

[152] Papazoglou, M.P.: Agent-oriented technology in support of e-business. Commun. ACM **44**(4), 71–77 (2001) (Cited on pages 13 and 176.)

[153] Papazoglou, M.P.: Web Services: Principles and Technology. Pearson - Prentice Hall (2007) (Cited on page 13.)

[154] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: State of the art and research challenges. IEEE Computer **40**(11), 38–45 (2007) (Cited on pages 81 and 152.)

[155] Parnjai, J., Stahl, C., Wolf, K.: A finite representation of all substitutable services and its applications. In: ZEUS 2009, CEUR Workshop Proceedings Vol. 438, pp. 29–34. CEUR-WS.org (2009) (Cited on page 129.)

[156] Pathak, J., Basu, S., Honavar, V.: Modeling Web services by iterative reformulation of functional and non-functional requirements. In: ICSOC 2006, LNCS 4294, pp. 314–326. Springer (2006) (Cited on page 56.)

[157] Peltz, C.: Web services orchestration and choreography. IEEE Computer **36**(10), 46–52 (2003) (Cited on pages 81, 134, and 135.)

[158] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL '89, pp. 179–190. ACM (1989) (Cited on page 55.)

[159] Puhlmann, F., Weske, M.: Interaction soundness for service orchestrations. In: ICSOC 2006, LNCS 4294, pp. 302–313. Springer (2006) (Cited on page 103.)

[160] Ramadge, P., Wonham, W.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987) (Cited on pages 28 and 55.)

[161] Ramadge, P., Wonham, W.: The control of discrete-event systems. Proceedings of the IEEE **77**(1), 81–98 (1989) (Cited on page 55.)

[162] Reimann, P., Kopp, O., Decker, G., Leymann, F.: Generating WS-BPEL 2.0 processes from a grounded BPEL4Chor choreography. Tech. Rep. 2008/07, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology (2008) (Cited on pages 87 and 104.)

[163] Reisig, W.: Petri Nets. EATCS Monographs on Theoretical Computer Science edn. Springer (1985) (Cited on pages 37 and 88.)

[164] Reisig, W., Schmidt, K., Stahl, C.: Kommunizierende Workflow-Services modellieren und analysieren. Inform., Forsch. Entwickl. **20**(1-2), 90–101 (2005) (Cited on page 39.)

[165] Richter, W.: Syntaktische Erkennung von Modellierungsfehlern in Web Services. Studienarbeit, Humboldt-Universität zu Berlin, Berlin, Germany (2001). (in German) (Cited on page 65.)

[166] Salaün, G., Bultan, T.: Realizability of choreographies using process algebra encodings. In: IFM 2009, LNCS 5432, pp. 167–182. Springer (2009) (Cited on page 151.)

[167] Sanfeliu, A., Fu, K.S.: A distance measure between attributed relational graphs for pattern recognition. IEEE Trans. on SMC **13**(3), 353–362 (1983) (Cited on page 112.)

[168] Schmidt, K.: LoLA: A low level analyser. In: ICATPN 2000, LNCS 1825, pp. 465–474. Springer (2000) (Cited on page 97.)

[169] Schmidt, K.: Controllability of open workflow nets. In: EMISA 2005, *Lecture Notes in Informatics (LNI)*, vol. 75, pp. 236–249. GI (2005) (Cited on pages 142, 143, and 144.)

[170] Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & sons (1998) (Cited on page 130.)

[171] Sokolsky, O., Kannan, S., Lee, I.: Simulation-based graph similarity. In: TACAS 2006, LNCS 3920, pp. 426–440. Springer (2006) (Cited on pages 113, 114, 116, 119, 125, and 162.)

[172] Stahl, C., Wolf, K.: Covering places and transitions in open nets. In: BPM 2008, LNCS 5240, pp. 116–131. Springer (2008) (Cited on pages 56, 57, and 161.)

[173] Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a theory of Web service choreographies. In: WS-FM 2007, LNCS 4937, pp. 1–16. Springer (2008) (Cited on pages 133, 137, and 149.)

[174] Sürmeli, J., Weinberg, D.: Creating a message profile for open nets. In: ZEUS 2009, CEUR Workshop Proceedings Vol. 438, pp. 74–80. CEUR-WS.org (2009) (Cited on page 162.)

[175] Szyperski, C.: Component Software–Beyond Object-Oriented Programming. Addison-Wesley and ACM Press (1998) (Cited on page 13.)

[176] Tsai, W., Fu, K.: Error-correcting isomorphisms of attributed relational graphs for pattern analysis. IEEE Trans. on SMC **9**(12), 757–768 (1979) (Cited on page 112.)

[177] Valmari, A.: The state explosion problem. In: Advanced Course on Petri Nets, LNCS 1491, pp. 429–528. Springer (1996) (Cited on page 90.)

[178] Valmari, A.: Composition and abstraction. In: MOVEP 2000, LNCS 2067, pp. 58–98. Springer (2001) (Cited on page 55.)

[179] Vardi, M.Y.: From Church and Prior to PSL. In: 25 Years of Model Checking, LNCS 5000, pp. 150–171. Springer (2008) (Cited on page 150.)

[180] Verbeek, H.M.W., Basten, T., Aalst, W.M.P.v.d.: Diagnosing workflow processes using Woflan. Comput. J. **44**(4), 246–279 (2001) (Cited on pages 38, 64, 65, and 103.)

[181] Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM **21**(1), 168–173 (1974) (Cited on page 112.)

[182] Weber, B., Rinderle, S., Reichert, M.: Change patterns and change support features in process-aware information systems. In: CAiSE 2007, LNCS 4495, pp. 574–588. Springer (2007) (Cited on page 128.)

[183] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall (2005) (Cited on page 83.)

[184] Weinberg, D.: Efficient controllability analysis of open nets. In: WS-FM 2008, LNCS 5387, pp. 224–239. Springer (2009) (Cited on pages 69, 70, 78, 105, 108, 160, and 162.)

[185] Wolf, K.: Generating Petri net state spaces. In: ICATPN 2007, LNCS 4546, pp. 29–42. Springer (2007) (Cited on pages 19, 97, 99, 159, and 160.)

[186] Wolf, K.: On synthesizing behavior that is aware of semantical constraints. In: AWPN 2008, CEUR Workshop Proceedings Vol. 380, pp. 49–54. CEUR-WS.org (2008) (Cited on pages 56 and 103.)

[187] Wolf, K.: Does my service have partners? LNCS T. Petri Nets and Other Models of Concurrency **5460**(2), 152–171 (2009) (Cited on pages 17, 29, 31, 32, 35, 36, 38, 39, 47, 56, 102, 103, 141, 142, 143, 144, 156, and 160.)

[188] Wolf, K., Stahl, C., Ott, J., Danitz, R.: Verifying livelock freedom in an SOA scenario. In: ACSD 2009, pp. 168–177. IEEE (2009) (Cited on pages 160 and 161.)

[189] Wolf, M.: Synchrone und asynchrone Kommunikation in offenen Workflownetzen. Studienarbeit, Humboldt-Universität zu Berlin, Berlin, Germany (2007). (in German) (Cited on pages 38, 39, and 94.)

[190] Wombacher, A.: Evaluation of technical measures for workflow similarity based on a pilot study. In: OTM Conferences 2006 (1), LNCS 4275, pp. 255–272. Springer (2006) (Cited on page 130.)

[191] Wombacher, A., Fankhauser, P., Mahleko, B., Neuhold, E.J.: Matchmaking for business processes based on choreographies. Int. J. Web Service Res. **1**(4), 14–32 (2004) (Cited on page 32.)

[192] Wu, J., Wu, Z.: Similarity-based Web service matchmaking. In: SCC 2005, pp. 287–294. IEEE (2005) (Cited on page 128.)

[193] Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. **19**(2), 292–333 (1997) (Cited on page 103.)

[194] Zaha, J.M., Barros, A.P., Dumas, M., Hofstede, A.H.M.t.: Let's dance: A language for service behavior modeling. In: OTM Conferences 2006 (1), LNCS 4275, pp. 145–162. Springer (2006) (Cited on page 136.)

[195] Zaha, J.M., Dumas, M., Hofstede, A.H.M.t., Barros, A.P., Decker, G.: Service interaction modeling: Bridging global and local views. In: EDOC 2006, pp. 45–55. IEEE (2006) (Cited on pages 137 and 151.)

[196] Zhao, X., Liu, C.: Version management in the business process change context. In: BPM 2007, LNCS 4714, pp. 198–213. Springer (2007) (Cited on page 128.)

All links were last followed on August 30, 2010.

GLOSSARY

$\#_x(\sigma)$ ..... the number of occurrences x in the run σ
2^M ..... the powerset of M
$[\,]$ ..... the empty multiset
$\neg$ ..... Boolean negation
$\odot$ ..... synchronization of service automata
$\oplus$ ..... composition of service automata
$\otimes$ ..... product with constraint(-annotated) automaton
$\rightarrow$ ..... a transition relation
$\vee$ ..... Boolean disjunction
$\wedge$ ..... Boolean conjunction
A ..... a service automaton
A_N ..... the translation of service net N to a service automaton
$A|_{\mathcal{P}}$ ..... projection of choreography A to port $\mathcal{P}$
β, β' ..... assignment functions
$\mathcal{B}$ ..... a multiset
$Bags(M)$ ..... the set of all multisets over the set M
$Bags_k(M)$ ..... the set of all k-bounded multisets over the set M
BPEL4Chor .... WS-BPEL extension for choreography modeling [50]
$B^\varphi = [B, \varphi]$ .... an annotated automaton
BPMN ..... Business Process Modeling Notation [150]
C ..... a constraint automaton
CCS ..... Calculus of Communicating Systems [138]
χ ..... global decision event
$closure(X)$ ..... the closure of a set X of states
ε ..... stuttering step
$\mathbb{E}$ ..... the set of all message events
$!\mathbb{E}$ ..... the set of all asynchronous send message events
$?\mathbb{E}$ ..... the set of all asynchronous receive message events
$!?\mathbb{E}$ ..... the set of all synchronous message events
$\mathbb{E}_P$ ..... the events of a port P
$\mathbb{E}_{\mathcal{P}}$ ..... the external events of an interface $\mathcal{P}$
$\mathbb{E}^\tau_{\mathcal{P}}$ ..... the internal events of an interface $\mathcal{P}$
final ..... proposition that evaluates to true in a final state
I ..... the input message channel of a port $P = [I, O]$
iBPMN ..... BPMN extension for interaction modeling [48]
k ..... a message bound ($k \in \mathbb{N}$)
$L(a, b)$ ..... label similarity ($a, b \in \mathbb{E} \cup \{\tau, \varepsilon\}$)

$\mathcal{L}(A)$ ..... the language of the closed service automaton A
$\mathbb{M}$ ..... the set of all message channels
$\mathbb{M}_a$ ..... the set of all asynchronous message channels
$\mathbb{M}_s$ ..... the set of all synchronous message channels
$\mathbb{M}(e)$ ..... the message channel of an event $e \in \mathbb{E}$
$\mathbb{M}_{\mathcal{P}}^{\square}$ ..... the closed message channels of an interface $\mathcal{P}$
$\mathbb{M}_{\mathcal{P}}^{\sqcup}$ ..... the open message channels of an interface $\mathcal{P}$
$Match(B^{\varphi})$ .... the set of service automata that match with B^{φ}
MSC ..... Message Sequence Chart [87]
$m\,[t\rangle_N\,m'$ ..... firing transition t in marking m resulting marking m'
N ..... a service net
$\mathbb{N}$ ..... the set of natural numbers (including 0)
$\mathbb{N}^+$ ..... the set of positive natural numbers (excluding 0)
O ..... the output message channel of a port $P = [I, O]$
OG_A^k ..... a k-operating guideline of A
$\mathcal{P}$ ..... an interface
φ ..... a Boolean formula
p ..... discount factor ($p \in [0, 1]$)
$Prov$ ..... a provider service
Q ..... a set of states
q ..... a state ($q \in Q$)
q_0 ..... the initial state ($q_0 \in Q$)
Req ..... a requestor service
ρ ..... a simulation or structural matching relation
σ ..... a run of a closed service automaton
$\sigma_{|\mathbb{E}}$ ..... a run of a closed service automaton without τ-steps
SOA ..... service-oriented architecture [74]
SOC ..... service-oriented computing [152]
$Strat_k(A)$ ..... the set of all k-strategies of a service automaton A
τ ..... a noncommunicating event ($\tau \notin \mathbb{E}$)
UML ..... Unified Modeling Language [149]
Ω ..... a set of final states ($\Omega \subseteq Q$)
WS-BPEL ..... Web Service Business Process Execution Language [11]
WS-CDL ..... Web Service Choreography Description Language [90]
WSDL ..... Web Service Description Language [38]

SELBSTSTÄNDIGKEITSERKLÄRUNG

Ich erkläre, dass ich die eingereichte Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Rostock, den 30. August 2010

Niels Lohmann

TABELLARISCHER LEBENSLAUF

10. Mai 1981	Geburt in Bonn
08/1987 – 06/1991	Melanchthon-Gemeinschaftsgrundschule Brühl
08/1991 – 06/2000	Max-Ernst-Gymnasium der Stadt Brühl; Allgemeine Hochschulreife
07/2000 – 08/2000	Praktikum bei GE CompuNet Information Technology Solutions, Köln
09/2000 – 06/2001	Grundwehrdienst im Fernmeldeaufklärungsregiment 940 in Daun/ Eifel
10/2001 – 09/2005	Studium der Informatik mit Nebenfach Volkswirtschaftslehre an der Humboldt-Universität zu Berlin; Studienschwerpunkt: Spezifikation und Verifikation verteilter Systeme, formale Methoden; Abschluss als Diplom-Informatiker
10/2005 – 10/2007	Wissenschaftlicher Mitarbeiter an der Humboldt-Universität zu Berlin; Leitung des Projektes Tools4BPEL („Korrektheit und Zuverlässigkeit zusammengesetzter Web Services am Beispiel der Geschäftsprozess-Modellierungssprache BPEL“) am Lehrstuhl für Theorie der Programmierung (Prof. Wolfgang Reisig) des Instituts für Informatik
seit 06/2006	Mitglied im B.E.S.T-Projekt (Berlin-Rostock-Eindhoven Service Technology) mit regelmäßigen Forschungsaufenthalten bei der „Architecture of Information Systems Group“ (Prof. Kees van Hee/Prof. Wil van der Aalst) der Technischen Universität Eindhoven
11/2007 – 10/2009	Wissenschaftlicher Mitarbeiter an der Universität Rostock; Leitung des Projektes „Automatischen Generierung von Bedienungsanleitungen für Services“ am Lehrstuhl für Theorie der Programmiersprachen und Programmierung (Prof. Karsten Wolf) des Instituts für Informatik
11/2009 – 03/2010	Forschungsaufenthalt in der „Architecture of Information Systems Group“ (Prof. Kees van Hee/Prof. Wil van der Aalst) an der Technischen Universität Eindhoven (Niederlande).
seit 04/2010	Wissenschaftlicher Mitarbeiter an der Universität Rostock; Landesstelle am Lehrstuhl für Theorie der Programmiersprachen und Programmierung (Prof. Karsten Wolf) des Instituts für Informatik

CURRICULUM VITÆ

May 10, 1981	Born in Bonn, Germany
08/1987 – 06/1991	Primary school *Melanchthon-Gemeinschaftsgrundschule Brühl*, Germany
08/1991 – 06/2000	Grammar school *Max-Ernst-Gymnasium der Stadt Brühl*, Germany; university-entrance exam
07/2000 – 08/2000	Internship at GE CompuNet Information Technology Solutions in Cologne, Germany
09/2000 – 06/2001	Basic military service at the SIGINT regiment 940 in Daun/Eifel, Germany
10/2001 – 09/2005	Studies of informatics with minor subject economics at the Humboldt-Universität zu Berlin, Germany; Specialization in specification and verification of distributed systems, formal methods; degree *Diplom-Informatiker*
10/2005 – 10/2007	Research associate at the Humboldt-Universität zu Berlin; conducting the project Tools4BPEL ("Correctness and reliability of composed Web services modeled in BPEL") at the Theory of Programming group (Prof. Wolfgang Reisig) at the Department of Computer Science
since 06/2006	Member of the B.E.S.T-project (Berlin-Rostock-Eindhoven Service Technology) with regular research visits at the Architecture of Information Systems Group (Prof. Kees van Hee/Prof. Wil van der Aalst) of the Technical University of Eindhoven, The Netherlands
11/2007 – 10/2009	Research associate at the University of Rostock, Germany; conducting the project "Automatic Generation of Operating Guidelines for Services" at the Theory of Programming Languages and Programming (Prof. Karsten Wolf) at the Department of Computer Science
11/2009 – 03/2010	Visiting researcher at the Architecture of Information Systems Group (Prof. Kees van Hee/Prof. Wil van der Aalst) of the Technical University of Eindhoven
since 04/2010	Research associate at the University of Rostock at the Theory of Programming Languages and Programming (Prof. Karsten Wolf) at the Department of Computer Science

ACKNOWLEDGMENTS

My work on correctness of services started 2005 in Berlin. Three years later, I moved to Rostock. Then earlier this year, I finished my thesis in Eindhoven. Needless to say, there are *many* people to thank...

The only person to start thanking is Karsten Wolf. In Berlin, he introduced me to the world of theoretical computer science, formal methods, and model checking. Some years later, he could offer me a job in Rostock, which really boosted my motivation to work in the area of controllability and operating guidelines. Without him, I never would have thought about an academic career. I thank him for his idea to focus on correctness in my thesis and I am looking forward to continue working with him.

Second, it is my pleasure to thank Wil van der Aalst for being a promotor of my thesis. I am grateful for his thorough feedback, his amazing enthusiasm, and his incomparable motivation. The presentation of this thesis is a result of his feedback. He was always a great guide and I thank him for the time I could spend in Eindhoven.

Third, I am grateful to Wolfgang Reisig to introduce me to Petri nets and the scientific world. He offered me a position in Berlin when I never thought about staying at university after my diploma thesis. In the years in Berlin, he made it possible for me to attend many conferences and to build a network of colleagues and friends.

I thank Mathias Weske, Marlon Dumas, Adelinde Uhrmacher, Natalia Sidorova, and Kees van Hee for their service as committee members and their valuable feedback.

Thank you Christian Stahl for always helping out, reading drafts of papers and this thesis, and taking the time to even discussing the most uncooked ideas; Gero Decker for so many fruitful discussions under Queensland's sun on choreographies and service modeling; Oliver Kopp for a constant reminder that there are people out there actually using WS-BPEL; Peter Massuthe for his scientific rigor; Kathrin Kaschner for giving me a warm welcome in Rostock and being such a great colleague through all these years; Christian Gierds for the many fruitful discussions on our tools and the WS-BPEL semantics; Dirk Fahland for the many nice years of joint research; Daniela Weinberg for her help with Wendy; and all my other colleagues.

I express great gratitude to all members of the B.E.S.T program: Jan Martijn van der Werf, Carmen Bratosin, Arjan Mooij, Helen Schonenberg, and Marc Voorhoeve. In particular, I thank Eric Verbeek and Boudewijn van Dongen for their tremendous support and patience during integration of our ~~girls~~ tools into ProM. Despite the weather, it was a great time in Eindhoven!

Many steps toward this thesis were accompanied by coauthors. I really enjoyed the work with you. Thank you Ahmed Awad, Alistair Barros, Jan Bretschneider, Cédric Favre, Remco Dijkman, Jana Koehler, Dieter König, Frank Michael Kraft, Marcello

La Rosa, Frank Leymann, Nannette Liske, Simon Moser, Chun Ouyang, and Hagen Völzer. I learned a lot!

The software tools used in this thesis are the result of solid teamwork. I thank Stephan Mennicke, Dennis Reinert, Georg Straube, Christian Sura, Robert Waltemath, and Martin Znamirowski for their help and timely last-minute support.

In the years of my PhD, there were so many helpful people in the background that made the whole procedure so incredibly smooth that I nearly forgot them. I thank Birgit Heene, Riet van Buul, and Ine van der Ligt.

Thank you!

Niels Lohmann
Rostock, July 2010

SIKS DISSERTATIONS

1998

1998-1 Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects
1998-2 Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information
1998-3 Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspectives
1998-4 Dennis Breuker (UM) Memory versus Search in Games
1998-5 E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products
1999-2 Rob Potharst (EUR) Classification using decision trees and neural nets
1999-3 Don Beal (UM) The Nature of Minimax Search
1999-4 Jacques Penders (UM) The practical Art of Moving Physical Objects
1999-5 Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
1999-6 Niek J.E. Wijngaards (VU) Re-design of compositional systems
1999-7 David Spelt (UT) Verification support for object database design
1999-8 Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1 Frank Niessink (VU) Perspectives on Improving Software Maintenance
2000-2 Koen Holtman (TUE) Prototyping of CMS Storage Management
2000-3 Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.
2000-4 Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design
2000-5 Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval.
2000-6 Rogier van Eijk (UU) Programming Languages for Agent Communication
2000-7 Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management
2000-8 Veerle Coupâ (EUR) Sensitivity Analyis of Decision-Theoretic Networks
2000-9 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

2001-1 Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks
2001-2 Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models
2001-3 Maarten van Someren (UvA) Learning as problem solving
2001-4 Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5 Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style
2001-6 Martijn van Welie (VU) Task-based User Interface Design
2001-7 Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization
2001-8 Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9 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 politiäle 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
2009-12 Peter Massuthe (TUE, Humboldt-Universität zu Berlin) Operating Guidelines for Services
2009-13 Steven de Jong (UM) Fairness in Multi-Agent Systems
2009-14 Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
2009-15 Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense
2009-16 Fritz Reul (UvT) New Architectures in Computer Chess
2009-17 Laurens van der Maaten (UvT) Feature Extraction from Visual Data
2009-18 Fabian Groffen (CWI) Armada, An Evolving Database System
2009-19 Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
2009-20 Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making
2009-21 Stijn Vanderlooy (UM) Ranking and Reliable Classification
2009-22 Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence
2009-23 Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment
2009-24 Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations
2009-25 Alex van Ballegooij (CWI) RAM: Array Database Management through Relational Mapping
2009-26 Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services
2009-27 Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web
2009-28 Sander Evers (UT) Sensor Data Management with Probabilistic Models
2009-29 Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications
2009-30 Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage
2009-31 Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text
2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors
2009-33 Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?
2009-34 Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach
2009-35 Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling
2009-36 Marco Kalz (OUN) Placement Support for Learners in Learning Networks
2009-37 Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks
2009-38 Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context
2009-39 Christian Stahl (TUE, Humboldt-Universität zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets
2009-40 Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language
2009-41 Igor Berezhnyy (UvT) Digital Analysis of Paintings
2009-42 Toine Bogers (UvT) Recommender Systems for Social Bookmarking
2009-43 Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients
2009-44 Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations
2009-45 Jilles Vreeken (UU) Making Pattern Mining Useful
2009-46 Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010

2010-01 Matthijs van Leeuwen (UU) Patterns that Matter
2010-02 Ingo Wassink (UT) Work flows in Life Science
2010-03 Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents
2010-04 Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments
2010-05 Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems
2010-06 Sander Bakkes (UvT) Rapid Adaptation of Video Game AI
2010-07 Wim Fikkert (UT) Gesture interaction at a Distance
2010-08 Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments
2010-09 Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging
2010-10 Rebecca Ong (UL) Mobile Communication and Protection of Children
2010-11 Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning
2010-12 Susan van den Braak (UU) Sensemaking software for crime analysis
2010-13 Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques
2010-14 Sander van Splunter (VU) Automated Web Service Reconfiguration
2010-15 Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models
2010-16 Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice
2010-17 Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications
2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation
2010-19 Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems
2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative
2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation
2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data
2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions
2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies
2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective
2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines
2010-27 Marten Voulon (UL) Automatisch contracteren
2010-28 Arne Koopman (UU) Characteristic Relational Patterns
2010-29 Stratos Idreos(CWI) Database Cracking: Towards Auto-tuning Database Kernels
2010-30 Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval
2010-31 Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web
2010-32 Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
2010-33 Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval
2010-34 Teduh Dirgahayu (UT) Interaction Design in Service Compositions
2010-35 Dolf Trieschnigg (UT) Proof of Concept: Concept-based Biomedical Information Retrieval
2010-36 Jose Janssen (OU) Paving the Way for Lifelong Learning; Facilitating competence development through a learning path specification
2010-37 Niels Lohmann (TUE) Correctness of services and their composition
2010-38 Dirk Fahland (TUE) From Scenarios to components