An Adaptive Service Oriented Architecture

Automatically solving Interoperability Problems

An Adaptive Service Oriented Architecture

Automatically solving Interoperability Problems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit van Tilburg, op gezag van de rector magnificus, prof. dr. Ph. Eijlander, in het openbaar te verdedigen ten overstaan van een door het college voor promoties aangewezen commissie in de aula van de Universiteit op dinsdag 7 september 2010 om 10:15 uur

door

Marcel Hiel

geboren op 5 februari 1978 te Rotterdam.

Promotor: prof. dr. W.J.A.M. van den Heuvel
Copromotor: dr. H. Weigand


The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems (Series No. 2010-32), and CentER, the Graduate School of the Faculty of Economics and Business Administration of Tilburg University.

Voor degene
die er altijd
is geweest.

Acknowledgments

A word of thanks,
A word of gratitude,
A sign of affection,

To those who helped,
Those who assisted,
Those who were there,

Thank you, supervisor,
Thank you, promotor,
Thank you, colleagues,

Despite heavy weather,
Despite a stormy night,
Despite even myself,

There were those who believed,
Who had faith,
Who showed me the road,

Thank you, my friends,
Thank you, my family,
Thank you, my beloved,

It is done,
I did,
You read.

May you all go banana's!

Marcel Hiel
September 2010

Contents

Chapter 1

Introduction

The goal of information systems is to improve the effectiveness and efficiency of an organization (Hevner et al., 2004). In order to achieve this goal, the improvement must be viewed from the perspective of the organization. IT efforts are, or at least should be, driven and measured by an organization's desires and objectives. Currently, two of these desires can be pointed out that drive recent research.

Desire to collaborate: The rise of internet and e-business has put companies into a global competition. In order to differentiate from existing competitors, companies create more complete products and/or services. To do this, companies are integrating their processes (Papazoglou & Ribbers, 2006). Examples are retailers, such as Amazon[1] and eBay[2], that package their service by working together with different shippers, banks, and providers in order to offer a complete service to their customer.

Desire for flexibility: It has often been stated that companies are facing a turbulent, continually changing environment. Changes within the company and in the company's environment cause misfits between the organization and that environment. Flexibility is an essential property for the maintenance of that fit in changing environments (Knoll & Jarvenpaa, 1994). This is reflected in the business process of that organization and hence is called Business Process Flexibility (Regev et al., 2007).

[1] http://www.amazon.com
[2] http://www.ebay.com

An approach aiming to satisfy these desires is Service-Oriented Computing (SOC). SOC is the computing paradigm that utilizes services as fundamental elements for developing applications/solutions (Papazoglou & Georgakapoulos, 2003). It is aimed at designing, building and using distributed software applications in heterogeneous and cross-organizational environments. SOC is gaining a lot of attention, partially because the concept of services is intuitive to business people, which thereby causes a potential convergence of business and IT perspective. Services are the key concept in SOC and are defined as self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications (Papazoglou, 2003). In this thesis, we regard technological issues of services and not the business aspects.

The architectural foundation for SOC is provided by the Service-Oriented Architecture (SOA). SOA is an architectural style that focuses on the definition and interaction of services. It states that applications expose their functionality as services in a uniform and technology independent manner, thereby creating a separation of concerns between implementation and interface. Claimed benefits of SOA are twofold: 1) interoperability and integration of software applications are achieved more easily, and 2) it provides a flexible facilitation of business processes in terms of a composition of loosely coupled services.

However, SOA is not without problems. Across the internet proclamations can be found such as "SOA is dead" (Manes, 2009), and "SOA is a failure" (Kenney, 2008). One of the major reasons for these statements is that SOA is focussed on developing design techniques aimed at guiding developers in how to build services, but does not entail the run-time aspects of the service, i.e. how to manage and maintain services. Without proper management the link with business objectives can not be made, as goals can not be specified and it can not be determined whether targets are reached. Some standards have emerged to cater for management requirements (OASIS, 2004; Bullard & Vambenepe, 2006), however, the standard SOA is not sufficiently equipped to define them in a concise and consistent manner.

In addition to this problem, even if companies were to use SOA as foundation for the IT infrastructure and form inter-organizational business processes, then this would lead to a rapidly growing number of connected systems and corresponding maintenance. As businesses change, so will the services they own (Lehman, 1980). Therefore changes will be introduced in this

inter-organizational business process, producing additional software maintenance. Furthermore, changes can cause other changes in other services, thereby causing a cascade of maintenance through an inter-organizational process. For these reasons, the accompanying complexity as well as the resources needed for maintaining the systems that support such processes will grow rapidly.

One method for dealing with this increase of maintenance is to increase the number of human system administrators. An estimation of the number of maintenance personnel was given in (Jones, 2006). Jones estimates a growth in the US alone, from two million people in the year 2000 to three and a half million people in 2015. However, increasing the number of administrators is not a lasting solution, not only because of the expenses, but also because the number of administrators needed is greater than there are available (Horn, 2001).

An approach that aims to solve the problem of software maintenance and addresses the aspects of run-time management of software, is the vision of Autonomic Computing (AC) (IBM, 2003; Kephart & Chess, 2003; Ganek & Corbi, 2003). Autonomic Computing draws its inspiration from biology, and more specifically from the autonomous nervous system. The human nervous system is the controller in the human body that keeps our vital functions in equilibrium. An example of such a balance is that it keeps our blood-sugar level on a certain point and modifies it when necessary. The idea behind Autonomic Computing is that systems become more self-managing and thereby reduce the amount of time and costs put into maintaining them.

As such, AC is defined as an implementation agnostic computing paradigm, and potentially provides SOA with the concepts and management mechanisms that help to overcome the shortcomings of the existing SOA paradigm. However, AC does not define exactly how business applications in general, and Web services and SOA more in particular, may adapt themselves. Recent work done under the umbrella of AC, for instance (Anthony, 2009; Tosi et al., 2009), maintain the vision of AC but do not realize it.

In this thesis, we address the lack of a conceptual foundation for management and adaptation in SOA by developing an extension of SOA, called the Adaptive Service Oriented Architecture (ASOA). By extending SOA, we reuse the inherent benefits of SOA which are aimed at developing inter-organizational business processes. By incorporating ideas of adaptation and management we aim to make the companies participating in such business

processes flexible. To demonstrate the validity and illustrate the workings of ASOA, we develop and implement a framework for maintaining interoperability in a service orchestration.

This chapter is introductory in nature and provides an overview of the research conducted. In Section 1.1, we present our motivations for this research, followed by the research goal in Section 1.2. The scope of the research is discussed in Section 1.3. The research questions are described in Section 1.4, followed by the adopted research methodology in Section 1.5. The contributions of this thesis are described in Section 1.6. This chapter is concluded in Section 1.7 with an outline of this dissertation.

1.1 Research Motivation

Interoperability is a critical aspect of distributed systems and of SOA in particular (O'Brien et al., 2007). Interoperability is defined by the International Organization for Standardization and International Electrotechnical Comission (2001) as the capability of software to interact with one or more specified systems, meaning that the software is able to coexist and cooperate with other systems. Interoperability is often split into a syntactical, behavioral, and semantical layer. The syntactical layer is concerned with the definition of the data in the messages, for instance specified in XML Schema (Fallside & Walmsley, 2004). The behavioral layer specifies the ordering of messages, also called the business protocol (Benatallah et al., 2004b). The semantical layer is concerned with the meaning of the data and is specified in annotations such as SAWSDL (Farrell & Lausen, 2007).

Problems in interoperability are called incompatibilities. We define an incompatibility as a discrepancy between what is expected to be received and what is actually received. For example, if a service expects to receive a number and instead receives text then this a syntactical incompatibility. For each of the layers of interoperability incompatibilities can occur. To guide these expectations and prevent faults and crashes, services specify what they expect, and also what can be expected by other parties, in an interface. As described in the introduction to this chapter, as businesses change so will their services. These changes will be reflected in the interface of the services, which may result in incompatibilities.

We distinguish between two methods that can be employed to *prevent* incompatibilities: (1) standardization of communication, and (2) usage of

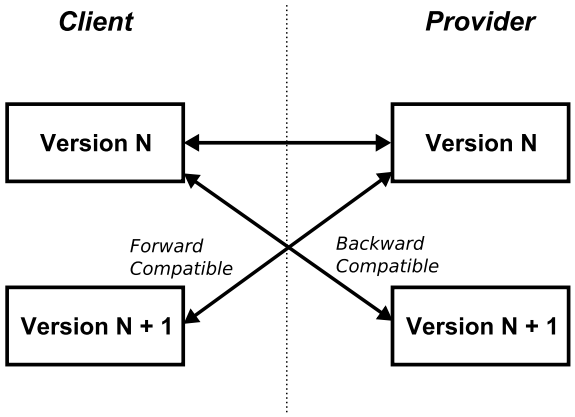


Figure 1.1: Backward and Forward Compatibility

appropriate versioning.

The first method is to standardize communication. SOA as a design philosophy is technology agnostic. Currently, Web services are the technology of choice for realizing SOA. Web services were invented to solve, at least partially, the problem of integrating distributed information systems (Alonso et al., 2004). A key benefit of Web services is that they define a standard for, among others, the data format (XML), the interface definition language (WSDL), and the communication transport mechanism (SOAP). Standardization reduces heterogeneity and makes it easier to develop solutions that integrate systems, other Web services, from other parties. Although standards make interoperability easier, they are not a panacea. They do not prevent all incompatibilities caused by changes to the interface.

The second method is to keep the services backward and/or forward compatible with each release of a new version of the services. Backward compatibility means that a newer version of the service provider can be deployed without breaking interoperability with the client. In other words, a service provider can send a newer version of a message to the client, which understands the new version and is able to processes this message. Similarly, forward compatibility means that a newer version of the client can be deployed in a way without breaking the interoperability with existing providers. Figure 1.1 illustrates backward and forward compatibility.

By guaranteeing backward/forward compatibility, the frequency of change propagation decreases, as the connected services can still use a newer version without problems. However, forward and backward compatibility are only

achievable through extensibility (Orchard, 2006). Furthermore, extensibility can only be exploited a limited number times. Therefore, appropriate versioning can prevent incompatibilities for some time, but can not avoid them all together.

Both methods, standardization and versioning, do not prevent all incompatibilities and if an incompatibility occurs they do no specify how to solve it. In *solving* (potential) incompatibilities, we distinguish in two methods: (1) placement of an adapter, and (2) making the service adaptive.

The first method to solve incompatibilities is to place an adapter between the services. Adapters act as a mediator solving interoperability problems. A research goal in this area is to be able to automatically generate adapters between any two services. However, adapters suffer from a drawback. Adapters are typically placed between existing applications that are regarded as a black box. As adapters can not manipulate the internals of the services they can not exploit any internal flexibility to solve incompatibilities.

The second method is to make the service adaptive to changes concerning interoperability. We define software to be adaptive if it can automatically detect changes (within the software and its environment) in relation to a certain criteria, decide whether and how to react on the change, and execute this decision. In the context of interoperability, this means that a service can automatically detect changes in the interfaces of the services it uses, and that it can adapt itself. Creating an adaptive service has two advantages: 1) adaptive services *can* exploit any internal flexibility of the service, and 2) if the adaptive service is created such that its own interface will not change due to adaptation, then it prevents change propagation and thus lowers the amount of maintenance for other connected services.

1.2 Research Goal

The problems described in the previous section leads us to the research goal, which is formulated as:

> **Design, develop and validate an extension of SOA, which will enable software services to (semi-)automatically adapt to their (evolving) environments.**

The goal of information systems in general is to improve the effectiveness and efficiency of an organization (Hevner et al., 2004). In the context of

our goal description, this means that less resources are spent on software maintenance. Autonomic Computing suggests that software should become self-managing in order to realize this. Self-managing means that software should deal with issues that normally fall under software maintenance, i.e., software is able to perform tasks that reduce the need for human intervention. In order to realize self-managing software, it should be self-adaptive with respect to a certain goal or criteria that is to be managed.

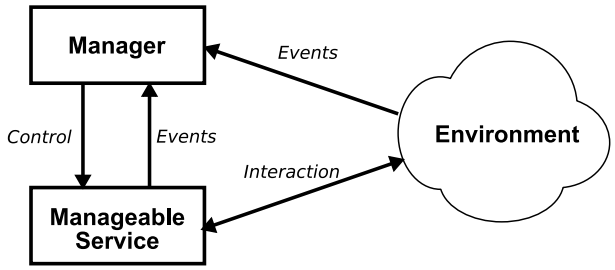


Figure 1.2: Manager and Manageable Service

In order to create adaptive software, there must be some adaptation logic in the software. Following the vision of Autonomic Computing, we aim to realize adaptive software by putting a *manager* on top of a *manageable service*, illustrated in Figure 1.2. The manager controls the manageable service and receives information (events) from the manageable service and the environment. With the introduction of the manager, we make a distinction between managerial interaction and operational interaction. Managerial interaction concerns all interaction about management and is indicated in Figure 1.2 as "control" and "events", located between the manager and the manageable service. Operational interaction concerns all other communication. The line between the manageable service and the environment indicates the operational interaction between the service and other, possibly external, services.

1.3 Scope of the Research

We make a distinction between (1) how to deal with changes on a conceptual level and (2) how to deal with changes concerning interoperability.

Conceptual: At the conceptual level, a generic framework is created by studying concepts of change and adaptation and how adaptation can

be realized in SOA. This includes an analysis of what types of changes can be handled in traditional SOA, state of the art of research work done in SOA and in an adaptive SOA. We provide a framework capable of handling any kind of change in a service's environment.

Applied to Interoperability: The framework created at the conceptual level is implemented for dealing with changes that affect interoperability. At this level, we create a framework for dealing with this type of change in a service orchestration. We create an adaptation strategy that incorporates all aspects of adaptation, i.e., detecting changes, deciding on an adaptation plan, and executing this plan.

Concerning interoperability, our goal is to automatically adapt to changes that cause incompatibilities. We focus on the syntactical and behavioral layer of interoperability. The semantic layer is not considered because (1) syntactic and behavioral already provide enough complexity and (2) the approach we describe in this thesis can easily be extended to cover also semantics, as described in Chapter 4.

In an inter-organizational business process, each of the involved business parties is responsible for managing the evolution of their own services. Although some changes will be handled collaboratively with other business parties (cf. (Wassermann et al., 2009)), the majority of changes will be handled locally. Therefore we take the local perspective of one business party with regard to interoperability. This perspective is called a service orchestration. In a service orchestration a central mediator is assumed to be present, called the service orchestrator. This means that all messages in a service orchestration are either sent or received by the service orchestrator. In our research we are interested in the effect of evolving service provider(s) on the business process of a service orchestrator and how to adapt the service orchestrator in order to maintain interoperability. In particular, we aim to adapt the service orchestrator while keeping its interface to its clients the same, thus preventing change propagation.

1.4 Research Questions

Similar to the scope of the research, the research questions are grouped in two: conceptual and applied to interoperability.

Conceptual

1. *What types of changes occur in a Service-Oriented Architecture?*
2. *How can a (composite) service be made (self-)adaptive?*
   - 2.1 *How can a service be made manageable?*
   - 2.2 *How to design a manager?*
3. *How can a Service-Oriented Architecture be extended to handle (self-) adaptive services?*

Applied to Interoperability

In our research, we deal especially with changes to the interfaces of service providers and set as a goal to be able to maintain interoperability. In relation to this goal, we pose the following questions.

4. *How can we model changes to a service interface?*
5. *What changes result in incompatibilities in a service orchestration?*
6. *How to automatically adapt the service orchestrator in order to maintain interoperability without changing its interface to its clients?*

1.5 Research Methodology

The research design we apply is created to meet the following general three conditions (Stewart, 1995). First, a scientific problem needs to be stated clearly so that it's proposed solutions can be examined properly. Second, experiments should be repeatable. And third, a scientific theory is required to be falsifiable.

The work in this thesis follows a design approach. It is aimed to give software developers new insights into how software, and services in particular, can be built and maintained. We advocate for one approach and explain our motives behind that approach. However, because along the way many choices must be made in realizing this approach, we will describe these choices as well and provide the rationale behind our decisions.

Our research methodology is illustrated in Figure 1.3. The rounded rectangles in the figure depict the fields of knowledge we deem of interest for

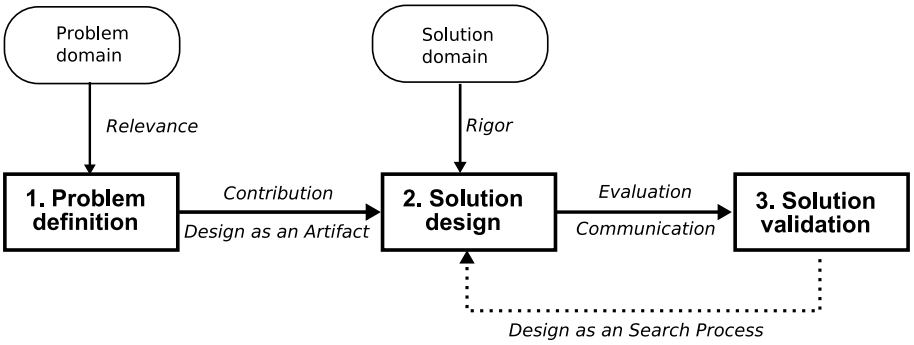


Figure 1.3: Research Methodology

this project. They are the result of a literature review. The arrows in the figure represent steps or derivations which lead from one phase to another. Although Figure 1.3 does not contain iterations (arrows going from the later stages back to the problem fields), we consider that research fields are revisited for refining or creating additional problem scenarios or solution techniques.

The research methodology consists of three main phases, namely problem analysis, solution design and solution validation. In addition to these three phases, we use the seven guidelines of Hevner et al. (2004) to further describe the research process, namely *design as an artifact*, *problem relevance*, *design evaluation*, *research contributions*, *research rigor*, *design as a search process*, and *communication of research*.

We describe Figure 1.3 in more detail for all three phases. For each phase we describe first the general idea after which we describe how we applied it in our research.

1. **Problem definition:** The fields of knowledge of our problem domain provide the essentials to which a solution must adhere to, i.e., they provide the minimum requirements for a solution. Based on a literature study, these fields furthermore contribute to a problem definition. This problem definition is a *world problem* (Wieringa & Heerkens, 2006). A world problem represents a discrepancy between the way the world is and the way we think the world should be. The world problem(s) selected for research should be important, unsolved (business) problems, i.e., they should be of scientific *relevance*.

   In our research, we adopt Service Oriented Architecture, Web services

and Software Evolution as our problem domains. The defined problem is, that due to changes and the lack of adaptive behavior in current software, manual adaptation is needed. The relevance of this problem has been discussed in the introduction to this chapter.

2. **Solution design:** The goal of this phase is to create a solution to the problem identified in the problem definition. Using knowledge from a solution domain provides a direction where the solution is to be found. Finding the right approach for solving a world problem is a *knowledge problem* (Wieringa & Heerkens, 2006). Knowledge problems represent a lack of knowledge about the world. The right approach for solving a research problem should be *rigorous*. A trade-off between rigor and applicability is usually necessary (Hevner et al., 2004). From the problem definition, an IT artifact is designed which should solve the problem (*design as an artifact*). This artifact is usually the goal of the research and functions as the main *contribution* of the research.

   In our research, we use knowledge from the fields of Autonomic Computing (IBM, 2003; Kephart & Chess, 2003; Ganek & Corbi, 2003) and Model Management (Bernstein et al., 2000; Bernstein, 2003). The solution that we design is an extension of SOA, introduced in Section 2.6. We rely on well known mathematical formalisms to ensure the quality of the research, and in addition to this, we choose a formalization of the industry standard for business processes (BPEL) for guaranteeing applicability. The contributions of our research are discussed in Section 1.6.

3. **Solution validation:** The third and last phase focuses on the validation and/or *evaluation* of the research. Part of this validation is performed during the project through *communication*, i.e., dissemination of partial results in academic venues such as conferences and publications in journals. The creation of a solution follows an engineering approach. As is typical for an engineering approach, this phase of the project may require iteration, as the design is flawed or could otherwise be improved. The cycle containing the design and validation of a solution is called generate/test cycle (Simon, 1996) or *design as a search process*.

   The work in this thesis has been communicated to the scientific com-

munity and has resulted in publications (Hiel & Weigand, 2006; Rohr et al., 2006; van den Heuvel et al., 2007; Hiel et al., 2008a,b; Hiel & Weigand, 2009). We validate our research on two points:

- We validate the extension of SOA, the ASOA, in two ways. The first way is to show that our conceptual framework does not suffer from the shortcomings of traditional SOA. We realize this in three steps. The first step is defining what constitutes to adaptive behavior based on existing literature. The second step is to demonstrate that services built in SOA *do not* possess adaptive behavior. And the third step is to show that, on the contrary, services in ASOA *do* posses adaptive behavior. The second way of validation is that we demonstrate that (self-)adaptation, as a prototypical implementation of ASOA, serves as a solution for interoperability problems in a service orchestration.
- We validate the framework for interoperability in two ways. The first way is by formally verifying that our adaptation solution is correct with respect to guaranteeing interoperability. The second way is by constructing a prototype which implements this solution.

1.6 Contributions

This dissertation addresses the lack of an architectural style for management of, and design of, (self-)adaptive services, introducing ASOA (Adaptive Service-Oriented Architecture), an extension of SOA, which demonstrates how SOA can incorporate management and adaptive behavior of services.

A conceptual framework for designing services in ASOA is provided using Model Management. This framework specifies what constitutes to a manageable service and provides a measure of, and the means to, achieve a completely adaptable service. Furthermore, the framework specifies how a manager can be created that, by itself, is again a manageable service, thereby creating an manageable manager. We describe how goals are incorporated in the model of a service, such that it can be verified whether a goal is reached or not.

This conceptual framework is implemented with as a goal maintaining the interoperability in a service orchestration. Research on interoperability has

largely focussed on identifying and solving incompatibilities, most of which in the context of adapters. However, how the business process of a service orchestrator can be adapted, not only to solve incompatibilities but also to prevent change propagation, has not been studied before.

Our service orchestrator enacts a hybrid business process, meaning that it consists of a process combined with business rules. We demonstrate that using our model of a business process, we can automatically compose a service from different service interfaces, where the business rules serve as the "glue" that keeps the composition together. Although many approaches exist that study and demonstrate automatic service composition, unlike these approaches we also incorporate business rules in the synthesis process and use policies to choose between alternative business processes.

Based on this hybrid business process, we demonstrate how a manager can automatically adapt the orchestrator based on three adaptation operators, namely *remapping*, *reordering* and *recomposition*. Each of these operators has a different impact on the business process and has a different range of problems that it can solve. *Remapping* adjusts the mapping of the orchestrator but does not change the process. *Reordering* exploits any flexibility in the ordering of messages and attempts to reorder messages to solve incompatibilities. And *recomposition* tries to find an alternative composition that incorporates the changed service. Using these three operators, we not only can decide whether an incompatibility can be solved, we also use them to analyse whether a service can be substituted for another. Not all incompatibilities can be solved, and therefore not all incompatibilities can be solved *automatically*. For this reason, the manager is equipped with an escalation mechanism, meaning that it will call for human intervention, or a higher level manager, when it can not solve a problem.

1.7 Outline of the Thesis

In **Chapter 2** we introduce the Adaptive Service Oriented Architecture. We provide a definition of adaptation and show what concepts need to be added to make SOA adaptive. Using Model Management, we create a parameterizable framework that can tackle every identifiable change. Furthermore, our framework is *model-* and *domain independent*, that is, we make no assumption on the type of models and in which domain they should be used. Next to this, we describe the architecture of a manager that enacts an *adaptable*

adaptation cycle.

Chapter 3 presents our model of an orchestration. We describe operators that capture changes in service specifications as well as operators that can be used to adapt an orchestrator. By defining these operators we create a business process that is *completely adaptable*. Also, we provide the properties that must hold for an orchestrator to be compatible. With compatible we mean that all sent and received messages of the orchestrator are understood by the receiving party.

Chapter 4 presents our approach to automatic service composition. Whereas traditional automatic composition only consists of other business processes, we use additional business requirements, such as *business rules* and *policies*, to specify the desired orchestrator. We synthesize an orchestrator based on the business protocols published in the interfaces of service providers. Our synthesis satisfies additional constraints such that our orchestrator can be employed using *both synchronous and asynchronous communication semantics.*

Chapter 5 describes our approach for realizing an adaptive orchestrator. We show how changes can be captured, how to analyze whether changes are applicable, and how an orchestrator can be adapted. For each of these phases we define an operator. Together with the composition operator of the previous chapter, these operators describe a *complete adaptation circle* for handling changes in the interfaces of a service provider. In other words, Chapter 4 and 5 together provide a complete description of a manager dealing with changes that affect interoperability.

In **Chapter 6** we evaluate our approach through the work of others. We discuss other approaches focussing on interoperability such as adapters and software versioning, as well as different management approaches for handling service evolution.

The last chapter of this dissertation, **Chapter 7**, summarizes our work and main results, points out some open problems and presents research challenges for future work.

Chapter 2

Adaptive Service Oriented Architecture

2.1 Introduction

Currently, Service Oriented Architecture (SOA) is heralded as the de-facto distributed computing technology for developing and managing a new breed of highly-adaptive business applications. This far, much progress is booked in development of methods and techniques but management of services has been largely neglected. Some standards have emerged to cater for management requirements, however SOA is not sufficiently equipped to define them in a concise and consistent manner.

At the same time, Autonomic Computing (AC)(IBM, 2003; Kephart & Chess, 2003; Ganek & Corbi, 2003) was presented in response to the rising complexity of business applications as well as to the critical need for tools and techniques to facilitate their evolution. AC is touted by industry leaders, mainly IBM, as the comprehensive solution to leverage the maintenance of business applications by making them self-managing. As such, AC is defined as an implementation agnostic computing paradigm, and potentially provides SOA with the concepts and management mechanisms that help to overcome the shortcomings of the existing SOA paradigm. However, AC does not define exactly how business applications in general, and Web services and SOA in particular, may adapt themselves.

One of the motivating questions of this chapter is: can services be designed in a generic way to adapt to changes in the service's environment?

In this chapter we introduce an architecture that guides the design of such adaptive systems in a structured manner. The architecture we introduce - Adaptive Service Oriented Architecture, ASOA for short - is built on top of the SOA architecture and recent developments in service management and Model Management (Bernstein et al., 2000; Bernstein, 2003).

This chapter is structured as follows: the next section introduces our motivating and running example that is used throughout this thesis. It exemplifies the concepts and the need for adaptivity in a realistic and concise scenario. Section 2.3 defines adaptation and the concepts related to it. In Section 2.4 we describe SOA, the concepts that constitute to a service, and the types of changes that occur in SOA. Section 2.5 presents the adaptiveness of SOA and our motives for extending SOA. Section 2.6 introduces the ASOA. Using ASOA as foundation, Section 2.7 discusses a conceptual framework that is applied to our running example. Section 2.8 concludes this chapter with a discussion.
This chapter is largely based on publications (Rohr et al., 2006; Hiel et al., 2008a,b).

2.2 Motivating Example

Consider the order management process depicted in Figure 2.1 (Adapted from (Papazoglou & van den Heuvel, 2007)). Our example involves five parties: a customer, a retailer, a shipper, an inventory, and a bank. The process is enacted by a composite web service and is structured as follows: after receiving a purchase order from a customer, the retailer executes three tasks. First, the retailer ascertains that sufficient parts are in stock. Second, the retailer checks the creditworthiness of the customer. For this purpose, he invokes an external Web service offered by a trusted third party, the bank. Third, the retailer inquires a shipper whether it can deliver the parts to the customer before the requested date. Once these tasks are completed, the order management process is concluded by sending an invoice to the customer, indicating the expected shipping date. This process was executed many times, until a change is introduced. Assume that the bank unilaterally upgrades the interface of its service, and sends a notification about this to its clients. In particular, the functionality of the service is extended allowing its clients not only to check the creditworthiness of a specific bank account, but

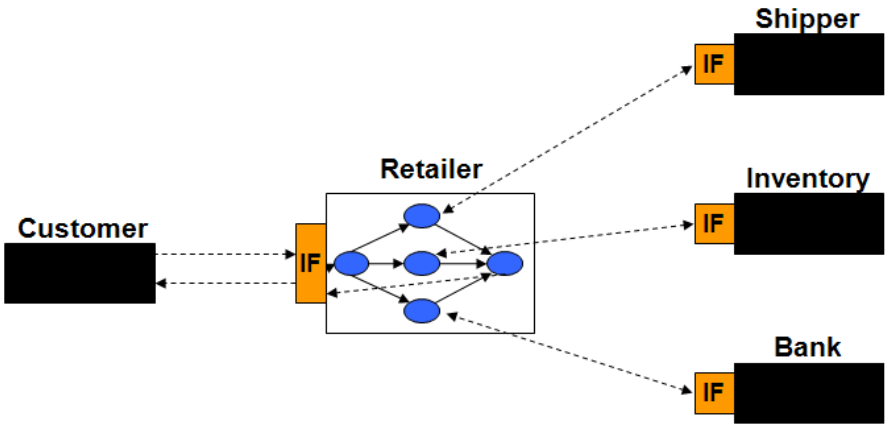


Figure 2.1: The Order Management Process

also that of credit cards that are issued by the bank. Hence, the operation for checking the balance of a client allows options in its input message: either a bank account or a credit card number may be queried.

The problem that would rise in current SOA is that either a human administrator catches this notification and starts implementing the change manually, or an error would occur at run-time when the upgrade at the bank has been implemented. In today's business environments, where SOA advocates dynamic establishment of cross-organizational relationships, this type of change happens frequently, see introduction to Chapter 1. It is therefore our purpose to enable automatic adaptation to changes in the environment.

2.3 Adaptation

Adaptation is a term that has been used for a long time and in many scientific disciplines. Adaptation has its roots in fundamental philosophical concepts such as change and time, which have been on a human's mind since Aristotle. Discussing these philosophical notions of change and time are beyond the scope of this thesis. This section provides an understanding of the concepts related to adaptation in software by presenting a definition and taxonomy for it.

2.3.1 Defining Adaptation

A lot of works on adaptation use an intuitive notion of adaptation without defining it, for example (Aksit & Choukair, 2003; McKinley et al., 2004b). A typical description of adaptation is the following: "a system is adaptive if it is sensitive to changes in its environment" (Zadeh, 1963). The goal of this section is to provide a comprehensive and precise definition of adaptation and anticipate its use in the scope of this dissertation.

Adaptation has four aspects:

Criteria: First aspect is that adaptation occurs in relation to a certain criteria (Zadeh, 1963). If a system needs to adapt then it needs to be aware of the change and also whether the response has any effect. To ensure this awareness in artificial systems, adaptation is often connected to the concept of performance (Ackfor & Emery, 1972; Sachs & Meditz, 1979; Kennedy, 2001). In relation to this performance often a notion of a norm or utility function is used to define a good or desired performance. Notably, in his seminal book (Ashby, 1960), Ashby defines adaptation as follows: *"a form of behavior is adaptive if it maintains the essential variables within physiological limits"*. Others (Ackfor & Emery, 1972) go further and state that some of the lost performance, caused by the change, must be regained in order to call it adaptive.

Environment: The second aspect of adaptation is the relation with the environment, also called context (Schilit et al., 1994; Lieberman & Selker, 2000). In our work we rely on system theory, similar to Sagasti (1970), to distinguish between environment and system. The roots of the word "adaptation" comes from Latin which means "to fit to", implying there is an entity that adapts and something the entity adapts to. Adaptation is therefore concerned with two concepts, namely an entity, i.e. the system, and it's environment. We formalize this intuition following a system theoretic approach (von Betalanffy, 1956) and define the following three concepts:

World: An entity which consists of two or more elements (E) and a non-empty set of relations (R) between the elements.

Environment: (E_e) A subset of elements of (E) such that the relationships between them are of no direct concern to the researcher.

		Origin	
		Environment (External)	*System (Internal)*
Response	*System (Darwinian)*	Environmental disturbance, system responds by modifying itself.	Disturbance originated within the object, system responds by modifying itself.
	Environment (Singerian)	Environmental disturbance, system responds by modifying its environment.	Disturbance originated within the object, system responds by modifying its environment.

Table 2.1: Classification Scheme of Adaptation

System: (E_o) A subset of elements of (E) such that the relationships between them are of direct concern to the researcher.

Based on the definitions of system and environment, a fourfold classification scheme of adaptation is created as displayed in Table 2.1 (adapted from (Sagasti, 1970)). It distinguishes between the origin and the response of the adaptation. The origin can be either external (environment) or internal (system) and the response singerian (environment) or darwinian (system). The names, darwinian and singerian come from the scientists who first discovered this type of adaptive behavior, respectively C. Darwin and E.A. Singer.

Perspective: The third aspect of adaptation is the perspective from which the adaptive behavior is regarded. Broy et al. (2009) take the perspective of the user of the system and define adaptive behavior as: *"Intuitively, a user experiences an adaptive system behavior, if the system's reaction resulting from his inputs is additionally determined by some information about the environment, i.e. implicit inputs."* They distinguish between two different types of adaptive behavior, namely transparent and non-transparent. Transparent adaptive behavior is experienced when the user is aware that besides the user's input, also input from the environment is used for determining the output of the system. In non-transparent adaptive behavior the user is not aware of the input from the environment. Following our system theoretic approach the user in our context is the researcher who studies the system.

Cyclic: The fourth aspect of adaptation is that it is generally considered as an ongoing process, typically depicted as a cycle or a loop. In this process, it contains three phases, namely detecting the change, deciding on how to tackle the change and execution any chosen action.

Given these four aspects of adaptation, we define a system to be adaptive as follows:

Definition 1. *(Adaptive)*
A system is called adaptive from the perspective of the researcher if it can automatically detect changes (within the system and its environment) in relation to criteria, decide whether and how to react on the change, and execute this decision.

In the following, we describe a taxonomy of adaptation from a computing perspective, which is based on the phases of the adaptation cycle, and discuss the different concepts behind them.

2.3.2 Taxonomy of Adaptation

We base our taxonomy of adaptation on other existing taxonomies for software engineering, software maintenance, software evolution and fault recovery found elsewhere in literature (Swanson, 1976; Rohr et al., 2006; Weigand & van den Heuvel, 2005; McKinley et al., 2004a; Buckley et al., 2005; Avizienis et al., 2004; Mariani, 2003; Bruning et al., 2007; Chan et al., 2007; Salehie & Tahvildari, 2009; Hielscher et al., 2009). Based on the system theoretic approach above, a taxonomy is presented in Figure 2.2. In this figure, both the process of adaptation is illustrated as well as the dimensions of adaptation for each phase of the process. Adaptation is typically regarded as an ongoing process and is therefore usually depicted as a cycle. This cyclic process contains three phases: *detecting*, *deciding* and *executing*. During the detection phase, a particular change is sensed. After a system is aware of the change, several plans to appropriately deal with the changed situation may be contemplated upon in the decision phase. In the last phase, the selected plan is executed to realize adaptation to the new situation. For each of the phases, a distinction can be made whether it concerns the environment or the system. As illustrated in Table 2.1, the cause and action can be used to classify the types of adaptation. Similarly, a distinction can be made based

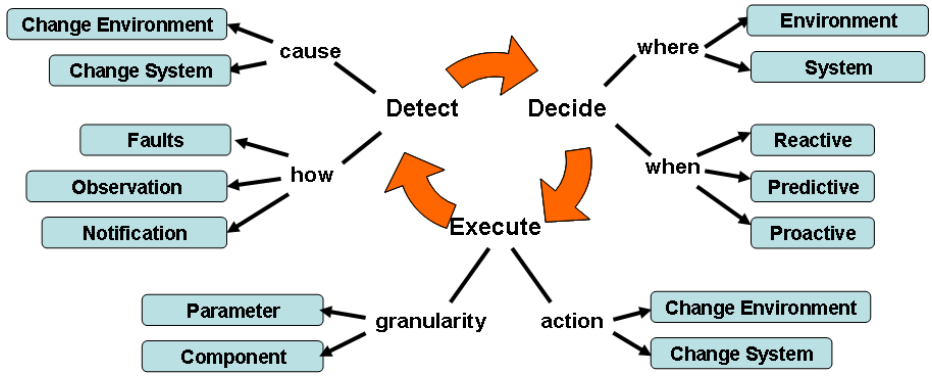


Figure 2.2: Taxonomy of Adaptation

on the question: where is the decision made on how to adapt. This decision can be taken either by the system itself or by an element in the environment.

In the following, we describe each of the dimensions of the taxonomy in more detail.

Detect

To adapt to a certain change, the system needs to be aware of it. We distinguish between three different means of how a system might become aware of changes (internal or in the environment):

Faults: One way of becoming aware of a change is through faults (or exceptions). If a changed system, or component, is used then this will result in a fault. For example, if a component expects an integer but receives a string then it returns a fault message. Fault taxonomies such as (Mariani, 2003; Bruning et al., 2007; Avizienis et al., 2004; Chan et al., 2007) describe the types of faults that can occur.

Observation: In case of observation, the environment and/or internal aspects of the system are observed for changes. We distinguish between two methods, namely monitoring and testing.

We define monitoring as the process of gathering information about the system or environment in order to detect changes. This definition is an adaptation of the definition provided by Hielscher et al. (2009). The advantage of monitoring is that if changes in the environment or in the system are localized, then faults may be prevented. Taxonomies of

monitoring have been provided for single entity systems (Delgado et al., 2004), and for distributed systems (Zanikolas & Sakellariou, 2005).

Testing differs from monitoring in that it actively inserts test data in the system or environment to see how it will respond to it. This method can be used to detect changes by comparing expected and actual response. Testing has also been called active monitoring (cf. (Cottrell, 2001)). A taxonomy of model-based testing was provided by Utting et al. (2006).

Notification: If a system changes then it can also send out a notification to every other system that makes use of it. Through a publish/subscribe communication protocol connected systems can receive a notification every time a related change is made, or is about to be made.

Decide

Commonly a causal approach is taken for defining adaptation. For instance, Sagasti (1970) states that a change has occurred and the system responds to this. In other settings, such as biology, the decision part of an adaptation process is not always apparent, or present. However, in artifacts such as software which action(s) to execute to adapt is a decision.

The question of when to adapt refers to timing. If a system responds too late (or too early) to a change, then this may have drastic effects. For example, if a system is designed to respond too late then that system might already have crashed. before it can respond. An illustration of this timing decision is given in Figure 2.3 (adapted from (Sachs, 1999)). This figure illustrates different categories of adaptation when a change causes a drop of performance. Each of the arrows indicates a category described in detail below. We draw an analogy with the types of software maintenance distinguished by Swanson (1976).

Reactive: If the performance of the system dropped under a certain level, then the system is triggered to respond. Arrow labeled number 3 represents reactive behavior. For example, when the response time of a server rises above a certain threshold, that server might start sharing the load with another server thereby attempting to lower the response time below the threshold. This type of adaptation is labeled "corrective" maintenance.

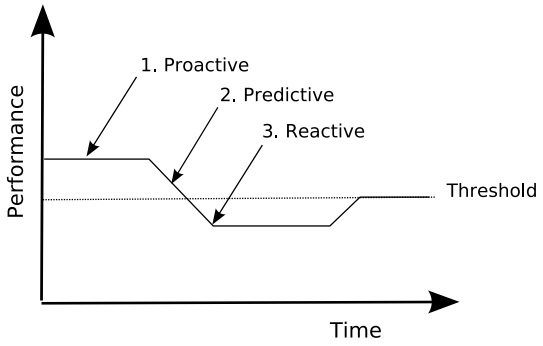


Figure 2.3: When to adapt

Predictive: If a certain state of the system or performance sensor indicates a future drop of performance, then the system might respond on that moment rather than waiting for the actual drop. We define this to be a predictive adaptation strategy. In Figure 2.3, predictive adaptation is indicated by the arrow labeled with number 2. An example of predictive adaptation is when a server can predict that the response time will go over the limit given the increase of response time until now. The server can predict that the limit will be reached in the future based on the rising response time now. Whether a change can be dealt with in a predictive or reactive manner is dependent on whether the change can be anticipated (Buckley et al., 2005), i.e., if there are any patterns that precede the change. This type of adaptation is also called "adaptive" maintenance.

Proactive: If the system has a normal performance level but chooses to change itself or the environment to gain a better performance then we call this proactive or goal-directed adaptation. This type of adaptation can occur at any point and therefore the arrow with number 1 in Figure 2.3, is just an example. An example of this type of adaptation is when the response time of a server is normal, but the server sees an opportunity to decrease the response time even further. For instance, by installing another encryption method it can save computation time and thereby response time. This type of adaptation is labeled "perfective" maintenance.

Execute

With regard to the execution, the question is whether the system or environment provides the means to adapt it, i.e., whether it is adaptable. We adopt here the definition of adaptability provided in (International Organization for Standardization and International Electrotechnical Comission, 2001) :

Definition 2. *(Adaptability)*
The capability of the software to be modified for different specified environments without applying actions or means other than those provided for this purpose by the software considered.

This definition is very broad and could be stated to be applicable for every software. Furthermore, for manual adaptation this definition should be extended to include factors such as time, costs and resources. The difference with adaptiveness in Definition 1, is that adaptability only contains the execution phase of adaptation, whereas adaptiveness contains all three phases. Therefore it does not specify how to detect a change or how to decide on an adaptation strategy. However, if software can not be adapted then it can not be made adaptive. Therefore, adaptability is a requirement of adaptive behavior.

In this thesis, we are interested in automated adaptation. Of importance is therefore the granularity and completeness of the means to alter the system. With completeness we mean whether everything in a software program can be altered or whether there are fixed elements that cannot be changed. For the granularity of the actions provided by the system, we follow the dichotomy of McKinley et al. (2004a,b):

Parameter adaptation: If existing variables can be modified such that it influences the dynamic behavior of the system then this is called parameter adaptation. The architecture or structure of the application is not affected by parameter adaptation, i.e. a strategy can be optimized but no new strategies can be adopted after implementation.

Compositional adaptation: If architectural or structural parts of the system can be altered then this is called compositional adaptation. This allows integrating new components or algorithms dealing with concerns unforeseen during the original design and construction.

In this section, we defined adaptation and explained the concepts related to it. We study adaptation in the context of software, more specific in Service-Oriented architecture. In the next section, we explain the core concepts of SOA. Using these concepts, we then determine how adaptive the traditional SOA is.

2.4 Service Oriented Architecture

Service Oriented Architecture has the goal to address the requirements of loosely coupled, standards-based, and protocol-independent distributed computing, linking the business processes and enterprise information systems isomorphically to each other (Papazoglou & van den Heuvel, 2007). Essential characteristics of services in a SOA are (Holley et al., 2003):

- All functions are considered services. This holds for business functions as well as system functions.
- Services are autonomous. The actual implementation of the functionality is encapsulated in a service, and is consequently invisible from the outside. Instead, the services are advertised in the interface of the service.
- Interfaces of the services are protocol-, network- and platform agnostic.

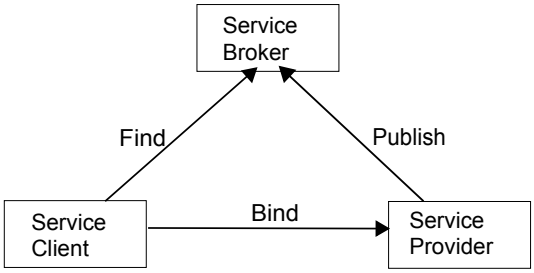


Figure 2.4: Service Brokerage

SOA supports two key roles: a service requestor (client) and service provider, which communicate via service requests. While SOA services are visible to the service client, their underlying realizations remain hidden and inaccessible. For the service provider however, the design of components,

their service exposure and management reflect key architecture and design decisions. Figure 2.4 depicts the standard SOA where a service broker serves as an intermediary interposed between service requesters and service providers. Under this configuration the broker typically offers a registry where the service providers advertise the definitions of services and where the service requestors may discover information about the services available. Once a requester has found suitable services, they may directly define bindings to the service realizations at the provider's site, and subsequently invoke them.

2.4.1 Core Concepts

While the standard SOA in this figure defines the main roles and the ways in which they may interact, it does not represent the conceptual structure of services, the first class citizens in SOA. To address this deficiency, we

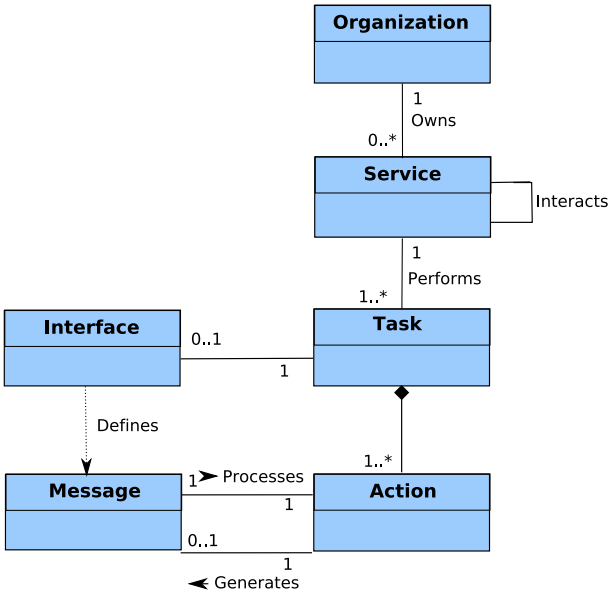


Figure 2.5: Ontology of a Service

have derived an ontology of services from existing specifications, standards and research papers, and in particular the WSA (Booth et al., 2004) and OASIS's reference model (OASIS, 2006). This ontology, depicted in Figure 2.5, is comprised of six essential concepts, which collectively define the core fabric of services in SOA:

- **Organization:** An organization (or person) denotes the concrete owner of an (abstract) service. As such, it materializes the linkage between the service, e.g., a Web service, and the organizational entity which bears responsibility for it in the real world.

- **Service:** Services are self-describing components that are capable of performing a task. A service can be nested, implying that a service can be composed of other services. Services that are assembled from other services are called composite (or aggregate) services, while atomic services refer to services that can not be further decomposed in finer-grained services. The relation in Figure 2.5 of the service to itself represents the interaction with other services, that can happen in each of the roles distinguished in Figure 2.4.

- **Action:** An action constitutes a discrete function that may be executed by the service. Examples of an action are processing or generating a message.

- **Task:** Several cohesive actions may be bundled into a task. Several criteria may be used to synthesize actions into a task, e.g., communicative- and functional cohesion. A functionally cohesive task should perform one and only one problem-related function and contain only actions necessary for that purpose, e.g., the services involved in the order management service. A communicatively cohesive task is one whose actions use the same set of input and output messages. The actions that collectively make up a task may be advertised in a service interface.

- **Message:** Services collaborate with each other by exchanging messages, each of which conveys one or more typed information elements. Messages may be correlated, e.g., in case of a two-way communication protocol, a send- and receive-message are associated with each other.

- **Interface:** A service interface specifies a task that the service can perform and defines the messages that a service can send and receive. Interfaces are the key instrument in SOA to ensure platform- and protocol independence, defining a service contract that forms a pair with an associated service implementation, and captures all platform-independent assertions. As such, the interface defines the identity of a

service and its invocation logistics. Other terminology that has been used for the same concept is service specification.

Note that this ontology does not contain the roles of the services in SOA, as they are already contained in the service brokerage triangle (see Figure 2.4).

2.4.2 Types of Changes

We are interested in the development of adaptive software, more specifically, in adaptive services. In order to make services adaptive, we must know what we are making services adaptive *to*. By identifying a specific type of change an adaptation strategy can be created and furthermore an service can be analyzed whether it is adaptive with regard to that type of change. For this purpose, in this section, we create a taxonomy of changes based on the core concepts of SOA described above.

For the identification of changes in the environment, we make a distinction between what is regarded as system and what as environment. In SOA the most prominent concept is the service, therefore we take this concept as the system. The concepts of task, action, message and interface (see Figure 2.5) are part of the service and therefore are considered as the system as well.

We focus on changes that originate in the environment of the service. Since we are regarding a framework for developing new services, we assume that the service will be implemented perfectly, e.g. no changes originate from the service itself by means of bugs. The environment of a service is comprised of two concepts, namely the organization that owns the service and the other services that the service may interact with. However, as we defined the concepts of task, action, message and interface to be part of a service, they are thus also part of the other services in the environment and thus are included in the taxonomy as well.

Based on the concepts of SOA described above, we define the following change taxonomy:

- **Task and Action:** The tasks and/or actions of a service can be changed. Although an interface might remain the same, the implementation behind it has been altered. An example of this type of change is when a sorting algorithm is replaced by another sorting algorithm, for instance bubblesort with quicksort. Semantic changes affecting a servicet's internals working fall under this category.

- **Message:** Changes on messages can either be that the data format has been changed or message(s) have been added or deleted. Typically this type of change is categorized under interoperability changes and affect the syntax, structure or semantics of the data in a message.

- **Interface:** Everything a service publishes about itself can be subject to change. Depending on what is published in the interface, a change can affect for instance tasks, actions, business protocols (that typically include a definition of the messages) or software properties such as Quality-of-Service. The release of a new interface is typically considered in the area of service versioning. Among others, a cause of a change of the interface can be the desire to remain compatible with new versions of service standards (Cisco, 2007).

- **Service:** A service is typically part of a network of services. Each of these services publishes an interface and changes affect these interfaces as described above. However, changes can also affect whole services. Services are typically part of a community of services (a set of services part of a registry or repository, like UDDI (Bellwood et al., 2004)). Given this community, new services will be added and older (unsuccessful) services will be removed, i.e., service retirement (Cisco, 2007).

- **Organization:** The organization that owns services will change, either due to changing business objectives or the service has been sold to another organization (Arsanjani, 2005). These changes will affect the implemented business policies and/or business rules that are part of the service. This type of changes has also been labeled as policy-induced changes (Papazoglou, 2008) or requirements change (Treiber et al., 2008a).

Note that the types of changes are not restricted to these concepts. Changes can be part of multiple concepts at the same time. For instance, a change in the data format of a message, is a change of a message, however this message is defined in the interface, and thus it is also a interface change. Furthermore, different types of changes have different impact on a service or service network. In Papazoglou (2008) a distinction is made whether a change has impact on only the service itself (cf (Wang & Capretz, 2009))

and maybe its clients or whether the change extends beyond the clients and propagates through the entire value-chain.

2.5 Adaptiveness in SOA

SOA is advocated to introduce flexibility and adaptivity. However, so far it was difficult to exactly pinpoint the reason why SOA is considered to be adaptive or why it should not be considered adaptive. The question to ask is: can SOA handle all the types of changes defined in the previous section? The criteria we use to answer this question is whether new concepts or relations need to be introduced or if changes can be dealt with within the scope of the existing concepts and relations. To represent SOA, we use the brokerage triangle and the core concepts defined in Section 2.4.1.

Traditional SOA

In SOA there is no concept of a manager or decision maker (Papazoglou, 2005). There is no concept other than the organization that is explicitly capable of determining adaptation strategies. Although adaptivity could be intended to be part of a concept itself, this is not apparent from the labels and descriptions of the concepts. The drawback of implicit adaptation is that intertwining adaptation logic with the business logic leads to poor scalability and maintainability (Salehie & Tahvildari, 2009).

Although SOA is not capable of dealing with changes automatically, it does facilitate for some adaptability. The separation between interface and implementation, regarding services as autonomous, and nesting of services to create service compositions are all designed to improve adaptability. Furthermore, SOA provides the ability for dealing with changes in the service community. The conceptual adaptability of SOA lies in its capability to discover services dynamically and make bindings to their implementations at runtime. However, service brokerage alone is not enough to create an adaptive architecture. Although new services can be found and bound at run-time, with or without plugins like aspects (Karastoyanova & Leymann, 2009), it will be hard to find a perfectly fitting service for a new or given service composition.

Approach	*Adaptation Phases*		
	Detect	*Decide*	*Execute*
SOA	-	-	✓
WSA	-	✓	✓
OASIS-RM	-	-	✓
WSMO	-	✓	✓
COSMO	-	✓	✓
SoaML	-	-	✓

Table 2.2: Comparison of Conceptual Approaches

Advances in SOA

To see, whether advances in SOA address these issues, we look at work done in the field of service modeling and service management. Service modeling aims at conceptually tackling all aspects part of the service and thereby providing an indication whether at design time adaptation is considered. Service management deals with all run-time aspects of the services.

Approaches that enhance the conceptual foundation of SOA are WSA, (Booth et al., 2004), OASIS-RM (OASIS, 2006), WSMO (Roman et al., 2005), COSMO (Quartel et al., 2007) and OMG's SoaML (OMG, 2009). Each of these approaches introduce a number of concepts to model a service and related aspects. We analyze these approaches based on the presence of concepts that specify or imply adaptive behavior. For this analysis, we use the phases of adaptation described in Section 2.3.1. The results of this analysis are displayed in Table 2.2. In this table, the conceptual approaches are represented including SOA, which is used as benchmark.

All approaches do not handle a full adaptation cycle, as none of them include a concept for detecting changes. The approaches differ most in other aspects such as expressivity. However, some approaches do contain some concepts for adaptation. WSA and SoaML have the concept of an agent. WSA states that Web services are implemented as agents and SoaML sees participants in a SOA as agents. Agents are commonly associated with adaptive behavior and it is therefore reasonable to assume that WSA and SoaML would incorporate adaptation, however this is not evident from the other given concepts and models. Similarly, WSA and COSMO both have the concept of a goal. Goals can help determine whether adaptation is required, based on whether the goal is reached or not. However both approaches do

Approach	*Adaptation Phases*			*Types of Changes*				
	Detect	*Decide*	*Execute*	*T/A*	*Msg*	*Interface*	*Service*	*Org*
SOA	-	-	✓	-	-	-	✓	-
WSDM	✓	✓	✓	-	-	-	-	-
SEMF	✓	-	-	✓	✓	✓	✓	✓
WSML	✓	✓	✓	-	-	-	✓	-
AWSE	✓	✓	✓	-	-	✓(QoS)	-	-
PAWS	✓	✓	✓	-	-	✓(QoS)	✓	-

Table 2.3: Comparison of Management Approaches

not specify how goals are used.

Next to these conceptual approaches, we compare approaches in service management. Service management is concerned with managing all run-time aspects of a service. Meaning that a management system controls and monitors the service and performs activities ranging from configuring the service, collecting metrics and tuning the service to ensure responsive execution (Papazoglou & van den Heuvel, 2005). This definition of service management is very similar to the definition of adaptation given in Section 2.3.1. The main difference is that service management is responsible for all aspects of the service, whereas adaptation is focussed on one aspect (criterion). In AC, this distinction is reflected in a range of self-* terms, for example self-optimizing and self-configuring, that together form self-management.

Next to service management a vast amount of work exists on service adaptation. However, each of these approaches tackles a specific type of change, for instance Quality-of-Service, or tackles only a specific aspect of the adaptation cycle, for instance monitoring (cf. (Baresi et al., 2009)). In short, these approaches do no provide insights on how the service in general should adapt to different types of changes. For this reason, we limit ourselves to efforts that state they are a management approach.

Table 2.3 provides a comparison in adaptivity of existing management approaches. We distinguish between two types of works in service management, namely standards and frameworks. Standards concerning the management of services include WS-Management (DMTF, 2008) and Web Service Distributed Management (WSDM) (Bullard et al., 2006). These approaches provide insights in what information is relevant for managing the service (WS-Management) and a general architecture on how services can be man-

aged (MOWS (OASIS, 2004)) and how services can be used for management (MUWS (Bullard & Vambenepe, 2006)). These standards provide the concepts of an explicit service manager in which the phases of adaptation are present, however they do not explain how this relates to the other concepts in SOA and what types of changes it can handle. They only provide architectural guidelines.

Service management frameworks include SEMF (Treiber et al., 2008b), WSML (Cibrán et al., 2007), AWSE (Tian et al., 2005) and PAWS (Ardagna et al., 2007). Each of these management frameworks focuses on a specfic aspect of adaptation. For example, SEMF provides an information model for managing and integrating all types of information related to services. This information model can be used to distill possible changes, however, the authors do not present a strategy on how to deal with changes. AWSE and PAWS are based on AC and conceptually capture a whole adaptation cycle, however both are focussed on Quality-of-Service and related adaptivity such as dynamic selection of services based on Quality-of-Service. All management frameworks describe how services can be managed and they handle different types of changes. However, none of them show a complete implementation or results that validate the claim that their management approach works.

Based on the adaptivity analysis above, we state that the SOA and advances in SOA do not support adaptive behavior (cf. (Di Nitto et al., 2008)). As a result, in practice, its key components are static in nature. Current service standards and platforms typically deliver static solutions, which are brittle and resistant to change.

2.6 Adaptive Service Oriented Architecture

To remedy these shortcomings and enable the adaptivity required in future business environments, we develop the Adaptive Service Oriented Architecture (ASOA). The ASOA we introduce here is based on SOA, however there are a number of alterations in the design (discussed further in this section). The adaptivity we seek in ASOA is delivered by adaptiveness of the individual services. We envision that, similarly to Autonomous Computing, the services act as autonomous units which are able to adapt themselves to their environment.

2.6.1 Adaptive Service Oriented Architecture: Basics

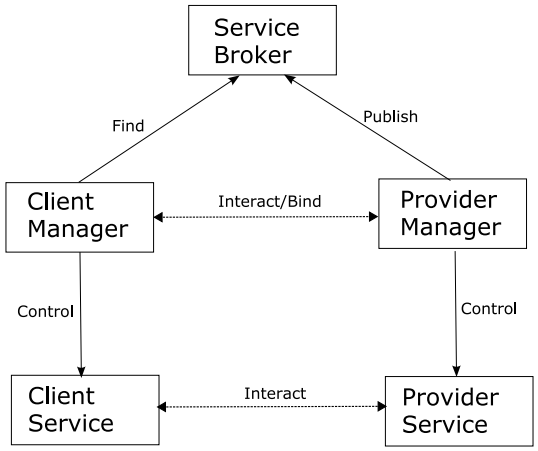


Figure 2.6: The Basic Adaptive Service Oriented Architecture

Figure 2.6 illustrates the basics of the ASOA. Like in SOA, service brokerage is used to find new services. The main difference is that a new role is introduced, that of the manager. Following the idea of Autonomic Computing at both the side of the provider and the requester a manager controls the services.

With the introduction of management in services, we make a distinction between the manageable (adaptable) service and the manager. The main advantage of this architecture is a clean separation of concerns; the service solely offers business functionality, while management and adaptation are the responsibility of the manager. Managers can interact with each other, for example to establish a contract or to notify changes (cf. Medjahed et al. (2004)).

We consider reconfiguration of a service to be a managerial task and not an operational, and therefore finding and integrating new services should be done by the manager. Figure 2.6 illustrates this with the connection from the service broker to the manager and not to the service.

We define the manager to be a service similar to the service that it manages. By defining the manager as a service, it exposes interfaces similar to the manageable service. The distinction between a standard service and a manager is that a manager has goals and enacts an adaptation cycle.

For the basis of the manager we borrow proven concepts from Agent

Technology. Agents are pieces of software able to make decisions and motivate these decisions. The characteristics of agents such as the pro-activeness make agents a suitable candidate to deal with unforeseen changes.

2.6.2 Concepts in ASOA

The concepts we described in the section on SOA together with concepts distilled from papers concerning management, like WSDM (OASIS, 2004; Bullard & Vambenepe, 2006) and Agents have resulted in the ontology illustrated in Figure 2.7. In this section, we explain our choices concerning these concepts and describe how they relate to each other.

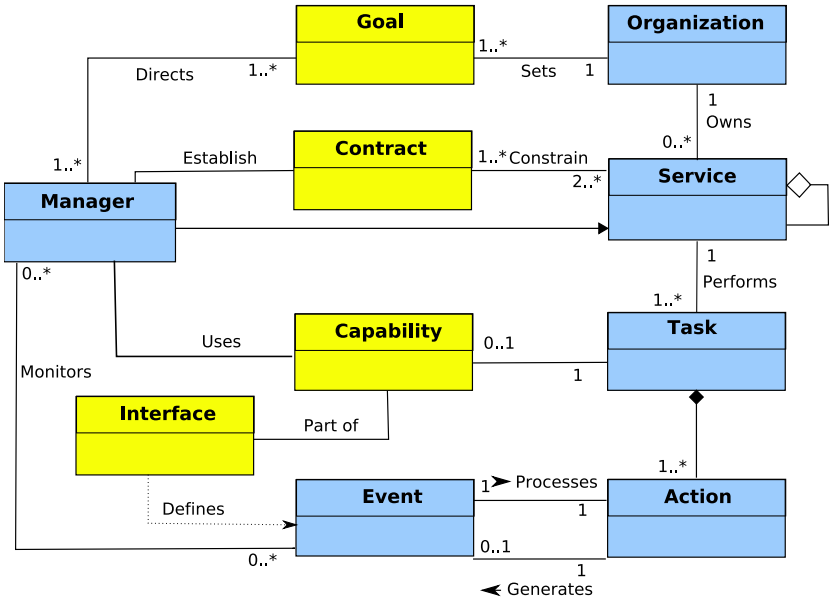


Figure 2.7: Ontology of an Adaptive Service Oriented Architecture

We distinguish between two types of concepts, namely *specifications* and *artifacts*. The specifications represent descriptions and components such as contracts, capabilities, and interfaces, and goals. The artifacts constitute the parts that are implemented such as the service and manager.

Figure 2.7 contains an **adaptation cycle** which is formed as follows: The manager listens to the events that are generated by and about the managed service. Based on these events, the manager *detects* changes, that for instance, affect the performance of the service. Depending on it's goal(s),

the manager *decides* whether and how to react to the change. The capabilities that the service provides are the means for the manager to *execute* these adaptations. The concepts introduced or altered (in relation to their definition in SOA) are explained in detail below:

- **Goal:** Software is build with a purpose (or goal). However in non-adaptive software the goal is typically implicit (Dardenne et al., 1993; Yu, 1997). An area where goals are explicitly used is software agents. One of the characteristics of an agent is proactiveness, implying that agents are goal-directed. Goals used for Agent programming have two aspects (Winikoff et al., 2002). The first aspect is that can be defined in a declarative manner. In this way, they describe the state of affairs which is sought by the agent. Declarative goals are required if agents need to reason about them. The second aspect is to define goals as procedural, meaning that a goal is defined as a set of procedures which is executed to achieve the goal. Both aspects are required for adaptation, as is explained in Section 2.7.

- **Contract:** The contract stipulates a mutual agreement between two or more services and defines prerequisites and results of particular service interaction. As part of the separation between managerial and operational aspects in ASOA, we see negotiation and agreement over a contract as a responsibility of the manager. The contract is implicitly included in WSA through the concept of "service semantics", but in the ASOA we need an explicit notion for restraining the adaptiveness of the service. Although the services should be adaptive and alter themselves to their environment, stability between parties is of critical importance to reduce uncertainty and establish trust relations. Among others, one aspect that is suggested to be captured in a contract is Quality-of-Service (Curbera, 2007).

- **Capability:** A capability is a task that is published in the interface. In ASOA, we make a distinction between two types of capabilities, namely operational and managerial. The operational capabilities are defined as in the standard SOA. The managerial capabilities, represent the means for adapting the service and to make it comply to the manager's intentions (Kreger et al., 2005). We distinguish between atomic and composite service. In the atomic service the manageability capabilities are limited to parameter adaptation and optimization of the

process, whereas in the composite service the manageability capabilities entail besides parameter adaptation, compositional adaptation and thus redesigning the (business) process. As operational and managerial capabilities have the same characteristics we do not make a separate concept for each type.

- **Event:** Events are messages that contain information about the systemt's functioning, and are used for the purpose of logging, alerting and monitoring. Events combined with event-correlation models form the basis on which the manager can detect changes, providing the foundation for reactive change management. Events are a common architectural style for distributed, loosely coupled and heterogeneous software (Rosenblum & Wolf, 1997; Muhl et al., 2006). As can be seen in Figure 2.7, we do not have the concept of message in the ASOA. The reason is that we assume that in asynchronous communication, messages can be equated with events. For the remainder of this thesis, we will use the term event and message interchangeably.

2.7 A Framework using Model Management

In the previous section, we described the concepts in ASOA. In this section, we sketch how we use Model Management to create a conceptual framework based on these concepts. Using this framework as foundation, we will provide in the following chapters the details on how to implement it for interoperability.

A number of enabling technologies are provisioned in literature for realizing adaptive services. Among others, enabling technologies are: Aspect-oriented Programming (Kiczales et al., 1997; Walker et al., 1999), Computational Reflection (Maes, 1987), Model Management (Bernstein et al., 2000; Bernstein, 2003), Machine Learning (Carbonell et al., 1983; Mitchell, 1997) and Agents (Wooldridge & Jennings, 1995; Wooldridge, 2001). In this paper, we primarily use Model Management but borrow proven concepts and methods from the other technologies as well.

As the name suggests, in model management the core concept is the model. With *model*, a complex structure is meant that represents a design artifact. The usage of models implies manipulation and transformation of one model to another model. Formal descriptions of such transformations

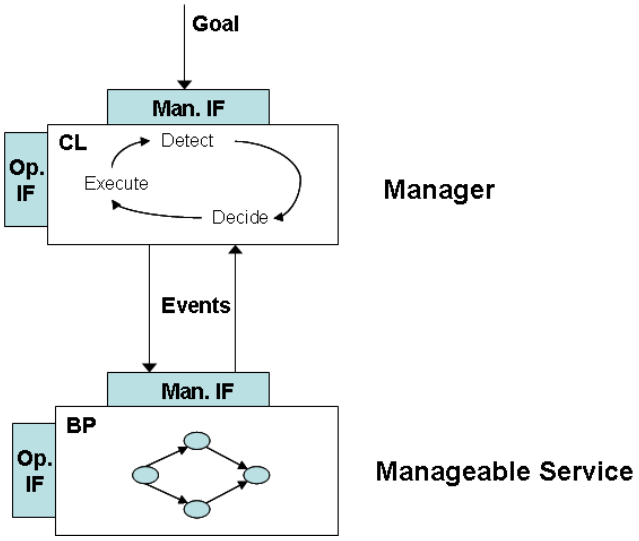


Figure 2.8: Manager and Manageable Service

are called *mappings*. Mappings are again a model. The key idea behind model management is to develop a set of algebraic *operators* that generalize the transformation operations.

Our motive for choosing model management, also called generic model management (Melnik et al., 2003b; Melnik, 2004), is its broad applicability. Although model management started in databases, it was also applied in other areas such as business processes (Madhusudan et al., 2004). Model management is an effort that can be placed in the context of Model-Driven engineering (MDE), where the philosophy is "everything is a model"(Bézivin, 2005). The other enabling technologies use models as well, for instance Neural Networks in Machine Learning, however, they do not advocate the development of generic operators for manipulating, transforming and creating mappings between models.

In our framework, we make the distinction between manageable (adaptable) service and manager (see Figure 2.8). The main advantage of this architecture is a clean separation of concerns: the service itself solely offers business functionality, while management and adaptation are the responsibility of the manager. We describe below both manager and manageable service using the concepts defined in Section 2.6. After this, we return to our example scenario and use it to illustrate how an adaptation process will execute in ASOA.

Concept	*Format*	*Description*
Action	$\mathrm{a}(v_1, .., v_n)$	every action is considered a function a where $v_1, .., v_n$ are variables
Task	$(< a_1, .., a_n >, \rho)$	a task is set of actions $< a_1, .., a_n >$ with an ordering ρ over those actions.
Event	$< rec, sen, data >$	an event is a model composed of a receiver rec, a sender sen and some $data$ contents .
Capability	$c = (< a_1, .., a_n >, \rho)$	a capability c is published task.
Interface	$(Op.IF, Man.IF)$	an interface contains two parts: an operational and a managerial interface.
Op.IF	$(< c_1^o, .., c_n^o >, uriMan)$	an operation interface contains a set of capabilities $< c_1^o, .., c_n^o >$ and an URI to the service manager $uriMan$.
Man.IF	$< c_1^m, .., c_r^m >$	a managerial interface contains a set of managerial capabilities $< c_1^m, .., c_r^m >$.

Table 2.4: ASOA model of a service

2.7.1 Manageable Service

We explain the manageable service through the use of the model of a service described in previous section. A formal representation and description for the concepts in that model is given in Table 2.4. Tasks represent a (business) process that is captured in for example, BPEL or a Petri-net. The data in events is typically modeled using XML Schema and the interface using WSDL. For the interface, we make a distinction between the operational interface (Op.IF) and managerial interface (Man.IF). The manageable service publishes an interface (Op.IF) containing the capabilities that it wants to advertise and a business protocol specifying the order of the messages involved. Next to these aspects, a URI to the manager of that service should be included. By publishing this endpoint, communication between managers is realized.

Following a model management approach we define a mapping between the operational interface and the implementation (task) of the manageable service. An advantage of this approach is that properties can be defined on this mapping such that the validity of the mapping can be guaranteed, i.e.,

the implementation conforms to the interface and vice versa. In our framework, this is realized by defining the capabilities published in operational and managerial interface to be equal to the set of tasks that the service is able to perform. Thus, let T_a denote the set of tasks of a service then $T_a \equiv < c_1^o, .., c_n^o > \cup < c_1^m, .., c_r^m >$. Note that although the service publishes tasks as capabilities, this does not necessarily mean that all actions performed to realize a task are published as well.

The managerial interface contains all aspects related to the manageability of the service. We make a distinction between two aspects of manageability of a service, namely ***adaptability*** and ***monitorability***. Both aspects are required to be present in the manageability interface (Man.IF).

Monitorability refers to the method how the manager will become aware that a change has occurred. This specifies whether the manager will receive information (events) from the manageable service, other managers, third party notification agents, or not at all.

The ***adaptability*** of the service refers to how the manager is able to adapt the service, i.e. the manageability capabilities. An important aspect of these capabilities are the operators that are defined on the model. In our framework, these operators consist of adding and removing concepts (and relations) in a model. An example of a set of operators for our ASOA service model is: addTask(), removeTask(), addAction(), removeAction(), addEvent(), removeEvent(). More formally defined: Let Σ be the finite alphabet of operators. A word of length $k \in N$ over Σ is a sequence of k symbols in Σ. Let Σ^* be the set of all words over Σ of length k for some $k \in N$.

Operators are commonly stored in a script. A script α is a sequence of operations that transforms one model into another, more formally defined as $\mathcal{M}_1 \stackrel{o_1}{\rightarrow} \mathcal{M}_2$ where $\mathcal{M}_2$ is the result of applying the basic operation o_1 to $\mathcal{M}_1$. For a sequence $\alpha = o_1, .., o_m$ of operations, we say $\mathcal{M}_1 \stackrel{\alpha}{\rightarrow} \mathcal{M}_{m+1}$ if there exist $\mathcal{M}_2, .., \mathcal{M}_m$ such that $\mathcal{M}_1 \stackrel{o_1}{\rightarrow} \mathcal{M}_2 \stackrel{o_2}{\rightarrow}, .. \stackrel{o_m}{\rightarrow} \mathcal{M}_{m+1}$.

Three properties of a set of basic operators are distinguished, namely *completeness*, *consistency* and *minimality* (Casati et al., 1998).

Completeness refers to the ability to transform a model of a type of model to another model of that same type. Based on this alphabet and model, we define when a set of operators is complete:

Definition 3. *(Completeness of operators)*
A set of operators Σ *for a type of model* $\mathbb{M}$ *is complete iff:*

$$\forall \mathcal{M}, \mathcal{M}' \in \mathbb{M}\ \exists \alpha \in \Sigma^*[\mathcal{M}' \xrightarrow{\alpha} \mathcal{M}]$$

Intuitively, Definition 3 means that if all concepts and relations of a type of model can be expressed using operators, then all instances of that type of model can be expressed using only these operators. Examples of types of models are Finite State Machines or Petri-nets. If the manageability capabilities include a set of operators that is complete, then we call the service *completely adaptable*. Having a completely adaptable type of model provides two advantages. The first advantage is it guarantees that if a change affects the model and if the solution can be expressed in the concepts of that model that a solution can be found. This is further explained for Section 2.7.2 on the manager. The second advantage is that all models of that type can be expressed using only operators, i.e., it can be expressed declaratively. This means that adaptation is equated with finding the right model. Advantage of an declarative model is that line-based versioning systems, such as Subversion [1] can be used to keep track of the change history. We describe an example of a completely adaptable service, i.e. the retailer in Chapter 3.

Consistency of a set of operators implies that after applying a script to a valid model again a valid model is returned. We define a model to be valid if it adheres to a set of properties, also called meta-model. For example a property for a finite state machine may be that every state must have either an incoming or an outgoing edge, i.e. every state in the finite state machine must be reachable. The set of properties defined for a model must hold for every created or adjusted model. The manageability interface should present the properties that must hold for a model. Let ϕ be a property then the notation $\mathcal{M} \models \phi$ means that the model $\mathcal{M}$ satisfies property ϕ.

Definition 4. *(Well-formedness of models)*
Let $< \phi_1, ..\phi_n >$ *be a set of properties, a model* $\mathcal{M}$ *is well-formed iff:*

$$\forall i \in \{1, .., n\} \forall \phi_i \in < \phi_1, ..\phi_n > [\mathcal{M} \models \phi_i]$$

[1] `http://subversion.tigris.org/`

In words, the above definition states that a model is *well-formed* if and only if it satisfies all properties.

Similar to well-formedness of models, we define well-formedness also for scripts.

Definition 5. *(Well-formedness of a script)*
Let $\mathcal{M}_{well}$ denote a well-formed model as defined in Definition 4. A script α is called well-formed if the following property holds:

$$\mathcal{M}_{well} \xrightarrow{\alpha} \mathcal{M}'_{well}$$

A script is well-formed if both the original model and the new model are well-formed.

Consistency can be guaranteed in two ways: either by specifying validation constraints as precondition for the operators (thereby ensuring that after every operator a valid model is achieved), or by specifying validation constraints that hold for the whole model and check these after all the operators of a script have been applied. Both approaches (if applied properly) yield the same result, namely a valid model.

Minimality of a set of operators refers to the ability of transforming models with a minimal set of operators. Because parameters can be used in the operators to create new instances of concepts, the minimal set of operators for a model contains a single operator, which contains all concepts as parameters. As there are different strategies for defining operators, minimality over different languages of operators is hard to impose. However, we do impose minimality of scripts of operators within the same operator language.

In order to have scripts that change a model, they should not be identity scripts. We define an identity script as follows:

Definition 6. *(Identity script)*
A script α is called an identity script (or identity transformation) if:

$$\mathcal{M} \xrightarrow{\alpha} \mathcal{M}.$$

A script of operators is an identity transformation if after applying the sequence of operators on a model, the result is again that same

model. An example of an identity transformation is if a change script contains the addition and removal of the same element. We do not impose minimality on a set of operators, however, we have for each concept exactly two operators, namely add and remove.

Definition 7. *(Minimal script)*
A script $\alpha \in \Sigma^*$ *is minimal (denoted as* α_{min}*) for transformation* $\mathcal{M} \xrightarrow{\alpha} \mathcal{M}'$ *iff:*

$$\neg\exists\delta \in \Sigma^*[|\delta| < |\alpha| \wedge \mathcal{M} \xrightarrow{\delta} \mathcal{M}']$$

Intuitively, the definition for minimality states that if a script is minimal then there does not exist another script that results in the same model and contains fewer operators.

Lemma 1. α *is minimal* $\Leftrightarrow \neg\exists\delta \in \Sigma^*[\delta \subseteq \alpha \wedge \mathcal{M} \xrightarrow{\delta} \mathcal{M}]$

Proof. We assume here that our language Σ contains only add and remove operators. The statement $\neg\exists\delta \in \Sigma^*[\delta \subseteq \alpha \wedge \mathcal{M} \xrightarrow{\delta} \mathcal{M}]$ states that "α does not contain an identity transformation".
$\Rightarrow$: Assume that α is minimal. As scripts contain only addition and removal of concepts, a script is minimal if all operators apply to different elements in the model. Thus, for all elements in the script only one operator per element is part of the script. If only one operator per element exists, this means that it can not contain an identity script as this would imply that the script contains two operators for one element.

$\Leftarrow$: Assume that α does not contain an identity transformation. This means that there are no two operators for the same concept, as this script may only contain addition and removal operators only elements can be introduced or removed. If an element is thus not added and also removed this implies that the script only contains operators for different elements, hence is minimal.

□

Lemma 1 states that if a script is minimal it can not contain, subscripts that are an identitity transformation and vice versa.

The managerial interface of a manageable service provides all functions related to adapting the service. In Table 2.5, we summarize the Manageability functions. An important function in the manageability interface is the

Function	*Output*	*Description*
getOperators()	Σ	returns the alphabet of operators Σ for the type of model used by the service.
getModel()	$\mathcal{M}$	returns the model currently used by the service (in the form of operators).
update(α)	boolean	updates the current model of the service by applying the operators of script α to it, with $\alpha \in \Sigma^*$.
getMetaModel()	$< \phi_1, ..\phi_n >$	returns a set of properties that must hold for a model to be well-formed.
valid($\mathcal{M}$)	boolean	determines whether a model is well-formed.

Table 2.5: Manageability Functions

update function. The manager will use this function to adapt the manageable service to changes. With $update_{\mathcal{M}}(\alpha)$ we denote the function that updates a model $\mathcal{M}$ by applying the operators in α to it. When it is clear from the context to which model the operators are applied we omit the model, like $update(\alpha)$.

We complete our description of a manageable service by providing an example specification of the retailer in our running example, see Table 2.6. In this specification we present the skeleton of the retailer, however this

Concept	*Example*
Actions	{receive(event), send(event),..}
Tasks	ordermanagement = {receive(order),send(checkAvailability)..}
Events	{order, checkAvailability,..,accept}
Capabilities	{ordermanagement}
Op. Interface	{ordermanagement},
	http://example.org/mymanager (URI manager)
Operators	{addTask(),removeTask(),..},
Meta-model	{consistent,..}
Man. Interface	see Table 2.5

Table 2.6: Specification for a Manageable Retailer

is not a complete description, for instance the ordering of the actions and messages is not apparent. In the next chapter we will describe the model we use for representing an adaptable orchestrator, and exemplify this in the

context of our running example, i.e. an adaptable retailer. Note that we state adaptable and not manageable because the monitoring aspect of the manageability interface is already set in this specification.

2.7.2 Manager

In ASOA, a manager is a service. Therefore we use the same concepts for representing it. We explain below for each of the concepts of the manager, how they are implemented.

Goal: The purpose of the manager is to adapt the manageable service in the context of a certain goal. A goal has been described as *a state of affairs that is to be reached*, i.e. it describes a desired situation (van Riemsdijk et al., 2008). In the context of Model Management, we define a goal to be a property over the model of the manageable service. The desired situation is then reached if the model satisfies the property. Following our notation for properties, we define a goal $\eth$ to be satisfied if $\mathcal{M} \models \eth$. As the manager will adapt the manageable service, $\mathcal{M}$ stands here for the model of the manageable service. The distinction between the goal and the properties of the meta-model of the manageable service is that the meta-model of the manageable service specifies fixed properties that must hold for every model. The goal on the other hand is specified over the model and may vary. Thus, the properties of the meta-model must always hold, and the goal property is desired to hold.

To see whether a manager can adapt the model of the manageable service to changes (for example, concerning interoperability), we define what can be handled in the following change categories.

Definition 8. *(Change categories)*
Let $\mathcal{M} \xrightarrow{\Delta} \mathcal{M}'$ be change where change script Δ is applied to $\mathcal{M}$ with as a result $\mathcal{M}'$ and let $\eth$ be the goal property defined for model $\mathcal{M}$. We define the following change categories:

Non-effective: $\mathcal{M}' \models \eth$

Solvable: $\mathcal{M}' \not\models \eth$ *and* $\exists\alpha[\mathcal{M}' \xrightarrow{\alpha} \mathcal{M}'' \wedge \mathcal{M}'' \models \eth]$

Problematic: $\mathcal{M}' \not\models \eth$ *and* $\neg\exists\alpha[\mathcal{M}' \xrightarrow{\alpha} \mathcal{M}'' \wedge \mathcal{M}'' \models \eth]$

The three categories above define the three types of action that the manager will perform when encountering changes. The first category, *Non-effective*, states that nothing needs to be done. The changes in this category do not affect the goal. The second category, *Solvable*, states that the changes affect the manageable service but that a script can be found such that the goal can be reached again. The third category, *Problematic*, states that a change cannot be solved and is therefore a problem. In this situation the manager escalates, i.e. sends a message to a (human) administrator stating that the manageable service does not comply with goal property and that (human) intervention is required.

Task: A goal specifies what should be achieved (declarative), the task on the other hand specifies how this goal should be reached (procedural). A goal should be woven with the process of the manager. For instance, to know whether a change has an impact on the goal property and whether a certain set of operators will solve any potential problems, i.e. the actions of the manager must be related to that goal. For different goals, different actions are needed. For instance, for interoperability different detection mechanisms are needed than for Quality-of-Service.

The manager enacts an adaptation cycle for adapting the manageable service. This process thus contains the three phases of adaptation, namely: detecting a change, deciding on an adaptation plan and executing this plan. In ASOA, this process is typically initiated by receiving an event. This event announces a change, for instance the bank will publish a new interface. The subsequent change is analyzed and diagnosed, for instance using a difference function to detect changes between two interfaces indicated by the event. This analysis should clarify whether something must be done to reach the specified goal, or whether the goal still holds. We make a distinction between changes that have no impact (nothing needs to be done), small impact (the changes are solvable by manipulating the model of the manageable service), and large impact (goal cannot be reached with any set of operators and the changes are not solvable by the manager). At the end of the adaptation cycle, the set of operators (found for any solvable change) is executed on the model of the manageable service, i.e. an adaptation cycle finishes with an $update(\alpha)$ action, where α is the set of operators to be executed.

Based on the process described for a manager above, we define a meta-model for an adaptation cycle. Properties for a typical adaptation cycle are: the process starts with the reception of an event indicating a change

and the process ends with either the execution of a set of operators on the manageable service or sending an event to higher level manager stating that the change can not be solved.

The manager need not be an atomic service. In this case, the manager is a composite service and relies on other services to realize parts of the adaptation process, e.g. for monitoring or diagnosis. This allows reuse of existing management functionality and a specialization of companies in function specific management services. For example business activity monitoring may be done by a service of Tibco [2], the decisions are made in a service provide by iLog [3]and the execution happens in an adaptable model in ActiveBPEL [4].

Furthermore, because this process is also a model, operators for adapting the adaptation cycle can be defined as well. This creates an adaptable adaptation cycle. Similar to operational services, changes will affect a manager. Therefore we employ the same method the make managers manageable. The manageability of managers follows from combining two ideas: 1) making services manageable, and 2) making the manager a service.

We discuss in the following chapters an example implementation for adapting an orchestrator to changes in the interfaces of its service providers. This process (adaptation cycle) is static and we assume that for this type of changes this process needs not be adapted. A different approach is to automatically derive this process, through goal-means reasoning or planning (van Riemsdijk & Wirsing, 2007). For the purpose of creating a generic adaptation cycle, we abstract from the possible architectures as for instance a deliberation cycle for Agents (Hindriks et al., 1999) or a control loop as provided by Control Theory (Kokar et al., 1999).

Operational Interface: The interface of the manager contains the operations that other managers will make use of. In ASOA, this includes an interaction protocol specifying how changes are communicated. For instance, managers are notified of a change through a publish/subscribe protocol. When a change occurs, or is planned, the manager of the manageable service will receive a notification of the manager of a service provider. In our running example, this translates into the situation where the manager of the retailer is subscribed to receive notifications of changes of the bank.

[2]http://www.tibco.com
[3]http://www.ilog.com
[4]http://www.activebpel.org

Function	*Output*	*Description*
getGoal()	$\eth$	returns the current goal-property
setGoal()	-	sets the goal-property

Table 2.7: Goal-related Functions

Managerial Interface: Because the manager is a service, a manager is required to define a managerial interface. This interface contains the operations as defined in the managerial interface of the manageable service, with the addition of two operations for setting and retrieving a goal. In our research, we restrict ourselves to study the situation where a manager has a single goal. The situation where multiple goals are handled at the same time is left for future work. Table 2.7 specifies the two operations related to the goal. Using this managerial interface, the model of the manager can be adapted in a way suitable for a higher level managers (or human administrators), thereby creating a potential hierarchy of managers.

An example specification of a manager is presented in Table 2.8. This manager is described in detail in Chapter 4 and 5.

Concept	*Example*
Actions	{receive(event), send(event),update(α),diff(),..}
Tasks	handleInteroperabilityChange = {receive(change),diff(),..}
Events	{change,..}
Goals	$Compatible(Process, Partner_1, .., Partner_n))$
Capabilities	{handleInteroperabilityChange}
Op. Interface	{handleInteroperabilityChange},
Operators	{addTask(v_1, v_2),removeTask(v_1),..},
Meta-model	{consistent,..}
Man. Interface	see Table 2.6

Table 2.8: Example manager for managing the retailer

2.7.3 Example: Adapting the Retailer

Consider the order management process of the retailer. The manager of the retailer, see Table 2.8, is subscribed to all the service managers of all the bank, shipper and inventory for notifications about change. After a period of

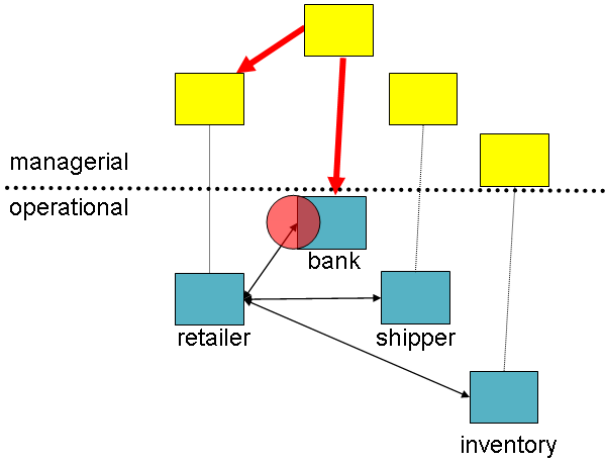


Figure 2.9: Change communication

time, the manager will receive a message (event) of the manager of the bank service that the bank service interface will change and that the old interface will be discarded. This notification will contain a date when the old interface can not be used anymore. An example of how a change is communicated in ASOA is depicted in Figure 2.9. To illustrate the adaptation process, Algorithm 2.1 provides an algorithmic representation for finding a solution.

The manager of the retailer receives an event concerning this change. It may receive this from the bank itself or through a third-party notification agent. This event contains either the new interface itself or a link to the new interface. The manager will then compare the old interface IF_B with the new interface IF'_B using a difference function $diff(IF_B, IF'_B)$ [5]. The result

Algorithm 2.1 Manager process to tackle Interface Changes

$receive(event_{IF'_B})$
$\Delta \leftarrow \text{diff}(IF_B, IF'_B)$
$\Delta_a \leftarrow \text{applicable}(\Delta, BP_R)$
if $\Delta_a \neq \emptyset$ **then**
  $\alpha \leftarrow \text{adapt}(\Delta_a, \eth, BP_R)$
  $update(\alpha, BP_R)$
end if

[5] In Model Management this function is called matching, however as our function determines the differences and not the similarities we prefer the term diff.

of this function is a set of operators (Δ), called a change script, containing the changes between the versions. Not all changes will result in incompatibilities and therefore the change script Δ is filtered for applicability in the *applicable* operator. The result of the *applicable* operator is a script Δ_a containing only operators that have an impact on the business process BP_R. If the change script containing all applicable operators Δ_a is not empty, then the *adapt* operator is executed to find a suitable adaptation script α. This script is then given to the *update* operator to create a new retailer. The details of this process and how these operators are implemented is discussed in Chapter 5.

2.8 Discussion

In this chapter, we explained our conceptual approach for realizing an Adaptive Service Oriented Architecture. By providing a taxonomy and defining adaptation and adaptability, we showed that standard SOA and the state-of-the art in SOA supports only limited adaptive behavior. Based on these shortcomings we created a conceptual model of an extended SOA (ASOA) which remedies this.

The two keys ideas behind ASOA are: 1) every service is managed by a manager, and 2) the manager is a service. Using Model Management, we provide a conceptual framework that guides the design of manageable services and managers. With a (completely) adaptable adaptation cycle, we provide the means to design a generic manager that can conceptually tackle any change in the environment or the service itself.

So far we demonstrated how ASOA works on a conceptual level. In the following chapters, we will go into detail how ASOA is realized and create a full model of the manager for interface changes. We use our example scenario throughout this dissertation to illustrate its workings.

Chapter 3

Modeling an Adaptable Service Orchestration

3.1 Introduction

In the previous chapter we described a conceptual framework using Model Management. As the name suggests, in model management the core concept is the model. In model management, with *model*, a complex structure is meant that represents a design artifact. The key idea behind model management is to develop a set of operators that can be used to manipulate models. In this chapter, we describe how we model an adaptable service orchestration and how we define operators on these models to capture changes and to adapt a model.

A distinction is made between two perspectives on a network of services, namely orchestration and choreography. In a service orchestration a central mediator is assumed to be present in the topology of services. This means that all messages are either sent or received by the orchestrator. The alternative perspective is that of a choreography. In a choreography there is no central mediator meaning that there exists at least one message which does not have the mediator as a sender and also not as a receiver. In our research, we study the effect of the environment (other services) on a service during run-time. In particular, we are interested in studying how an orchestrator can adapt when a service provider changes its interface. For instance, in our retailer example, the question is how a change in the interface of the bank affects the retailer. Figure 3.1 shows a managed orchestration following the

ideas of ASOA for our running example. More information on the difference and the relation between orchestration and choreography can be found in (Peltz, 2003; Daniel & Pernici, 2006).

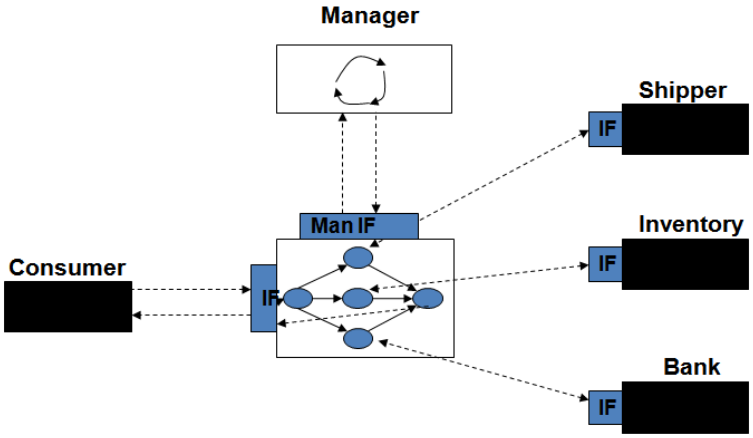


Figure 3.1: A Managed Orchestrator in a Service Orchestration

In our research, we are interested in interoperability between services. In particular, we are interested in how an orchestrator can remain compatible with its service providers as these service providers evolve. The things we need to model thus, are the service providers, in particular their service specification, the changes that can occur in those service specification, the orchestrator, and how the orchestrator can be adapted.

This chapter is organized as follows: In the next section (Section 3.2), we describe the model for service specifications. This model contains the assumptions of what services publish about themselves. Following our model management approach, we define operators on that model to capture changes in Section 3.3. In Section 3.4, we describe the underlying model of the orchestrator and how we model adaptation. After this, we describe in Section 3.5 a property that must hold for a network of service to be called an orchestration. We conclude this chapter in Section 3.6 with a discussion.

3.2 Service Specification

In Service Oriented Architecture (SOA), services can be created and maintained by different parties. If services are to cooperate successfully, such that

they can be combined to form an inter-organizational business process, then they must agree on certain aspects for example data format and Quality-of-Service. These aspects must be stipulated in an agreement. We assume that this agreement is unilaterally made by the service provider. The service provider (SP) presents the service and publishes information about it, whereas the service client (SC), by using the service, agrees with that published information. This means that the information published by SP should be sufficient for the service client to use it.

The published information is usually represented in the Web service stack, also called the interoperability stack, illustrated in Table 3.1. The lowest layer of the stack defines the way data is formatted, called eXtensible Markup Language (XML). XML is the global standard for encoding electronic documents. The layer on top of XML is XML Schema. XML Schema defines the data types and structure of XML documents. The ordering in which to send these documents (messages) is specified in a (Business) protocol, usually expressed in an abstract Business Process Execution Language (BPEL) process or a Finite Automaton (FA). The top layer of the interoperability stack is the interface. The interface contains all the lower layers and is described in Web Services Description Language (WSDL). In addition to these lower layers, the interface specifies the addresses to which to send the messages and the addresses where messages are expected. To incorporate semantics, extensions of WSDL and XML Schema have been proposed, such as SAWDSL (Farrell & Lausen, 2007), however, as we do not include semantics in this thesis, we do not discuss these further. In our research, we assume that service providers adhere to the languages specified in the interoperability stack. We study changes in the context of the type and protocol layer. Because we do not regard changes to data values, we use the term data for referring to types (XML Schema).

We describe below in more detail the two layers covered in this thesis,

Layer	*Published as*
Interface	WSDL
Protocol	BPEL/FA
Type	XML Schema
Data	XML

Table 3.1: Interoperability Stack

namely XML Schema and (Business) protocol.

3.2.1 XML Schema

Data types in messages are commonly specified using XML Schema and are typically part of a WSDL document. XML Schema defines built-in simple types (int, string etc) from which complex types can be built. For the reader familiar with XML Schema, we only consider complex types with element-only content, and do not consider facets, identity constraints, wildcards or substitution groups. We model XML Schema using a subset of Model Schema Language (MSL) (Brown et al., 2001), used before in (Fu et al., 2004d). MSL is a compact formal model covering the core ideas of XML Schema. Benefits of the formal notation is that it is concise and precise. This subset covers the most used features of XML Schema, thereby providing a small yet realistic formal representation of data, while avoiding the more complex and exotic features.

Let g denote the grammar and $g_1, .., g_k$ represent different (ordered) MSL types. In this grammar, d stands for the atomic data type such as string or integer. Furthermore, let $\tau[g]$ be a tag τ with as child g and let x and y integers where $x \leq y$ and $y > 0$. With $g\{x, y\}$ a sequence of g is denoted which has at least size x and at most y. Where the cardinality is not important, we omit it. An ordered sequence from 1 to k is denoted by $g_1, .., g_k$; $g_1|..|g_k$ indicates the choice amongst types g_1 to g_k.

Definition 9. *(MSL type)*
Grammar g is defined as follows:

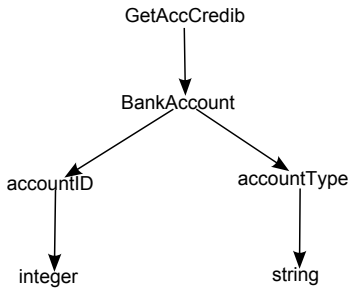
$$g \rightarrow d,\ \tau[g],\ g\{m, n\},\ g_1, ..., g_k,\ g_1|...|g_k.$$

Often when we refer to a grammar, we use a type tree to represent it. Given a MSL Type, it is straightforward to derive the corresponding type tree. In a type tree, each node is labeled with a MSL Type. If a MSL Type is present which contains multiple layers of MSL Types, then each node is a MSL Type subexpression of that MSL Type. Constraints on a MSL Type, such as the occurrence or choice, are indicated behind the label of the node. In the context of our running example, an example of a XML Schema, corresponding MSL, and type tree are given for a credibility request message

in Figure 3.2. This message is sent from the retailer to the bank to check whether a customer is creditworthy.

```
<xs:element name="GetAccCredib"
                          type="bank:BankAccount" />
<xs:complexType name="BankAccount">
   <xs:sequence>
     <xs:element name="accountID" type="xs:int"/>
     <xs:element name="accountType" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

```
GetAccCredib [
   BankAccount [
     accountID [int],
     accountType [string]
   ]
]
```



(c)

Figure 3.2: Message content description (a) in XML Schema (b) MSL (c) corresponding type tree

In a type tree, we often want to know whether a node is a leaf of that tree. Therefore, we define the boolean function **leaf**(g), which given a grammar-construct, returns "true" if it is a leaf (thus of the form $g \rightarrow d$) and "false" otherwise. We use e where we indicate a leaf-node. Furthermore, we define a boolean function **opt**(g), which returns "true" if the grammar construct (typically a leaf-node) is part of a choice, i.e., is optional.

As we are only interested in types and not in the actual values of the data, in the following whenever we use the term "data" we refer to the type level.

3.2.2 Business Protocol

The business protocol, also called protocol or conversation protocol (Benatallah et al., 2004b), specifies the order in which the messages may be invoked.

In very simple services with only one operation, i.e., stateless informational services, there is no need for a protocol, however, as services become more complex (due to composition or by adding more functionality), the

ordering of the operations, and thereby messages, may be of importance or even crucial for the successful execution of the operations. The commonly used example to explain the necessity of ordering is the purchase order, where a service requires that an order must be placed first before a payment can be made.

Because of the possible dependencies between operations, the order should be known to the client. Discovering this ordering of operations only from the WSDL is not an option since all possible permutations have to be checked to find the right combination. Since WSDL 2.0, protocols are incorporated in WSDL and are called Message Exchange Patterns. For a more detailed discussion on the need and advantages of business protocols the interested reader is referred to (Singh et al., 2004; Benatallah et al., 2004b; Alonso et al., 2004).

Finite Automata (FA) are commonly used as the formalism for modeling the protocols (cf. (Brand & Zafiropulo, 1983; Benatallah et al., 2004b)). Advantages of FA are the intuitive graphical notation and the large body of work behind it. We define a business protocol as follows:

Definition 10. *(Business protocol)*
A protocol is a tuple $P =< S, s_0, F, M, T >$*, where*

- S *is the set of states of the protocol,*
- $s_0 \in S$ *is the initial state,*
- M *is the set of messages supported by the service. Messages are defined as* $m =< rec, sen, g >$ *where* rec *indicates the receiver of the message,* sen *is the sender of the message and* g *is the MSL type of the message, defined in Def 9,*
- $T \in S^2 \times M$ *is the set of transitions. Each transition* $t =< s_f, m, s_t >$ *has a source state* s_f*, a target state* s_t *and either an input or output message* m *that is consumed or produced during this transition, and*
- $F \subseteq S$ *represents the finite set of final states.*

In our definition of a business protocol we include both receiver and sender of the message in the transition. For readability, we often omit the receiver and sender information and provide only "?" if a message is received

or "!" sent from the perspective of the owner of the protocol, for example $?m$ denotes a message that is to be received. With M^{in} we denote the set of received, and to be received messages, defined as $\{M^{in} : ?m \in M\}$. Similarly, we denote the set of outgoing messages as M^{out}, thus: $M = M^{in} \cup M^{out}$.

The execution of a protocol follows the usual rules for deterministic finite automata (Hopcroft et al., 2000). An execution path denotes the transitions in such a path, like $< t_1, t_2 >$ where first transition t_1 was executed, followed by t_2. We denote an execution path (or transition path) with tex_P^n of length $n \in \mathbb{N}$ on protocol P. Given a function $\sigma(t_i)$ which, given a transition, returns the associated message, we can likewise get a message path $< m_1, m_2 >$. A message path mex_P^n of length $n \in \mathbb{N}$ on protocol P is defined as

$$mex_P^n :=< m^1, .., m^n > .$$

The set MEX_P^n contains all message paths of length n, and MEX_P contains all accepted message paths.

Furthermore, let M be the alphabet of message. A *word of length* $n \in \mathbb{N}$ over M is a sequence of n messages in M. Let M^* be the set of all words over M of length n over some $n \in \mathbb{N}$. A language is a subset of M^*. Given a protocol P, the language of P denoted as $\mathcal{L}(P) = MEX_P$. All words in $\mathcal{L}(P)$ are message paths as defined above.

An example of a protocol represented as a FA is depicted in Figure 3.3. In this example protocol, the bank supports two paths, in the left path a

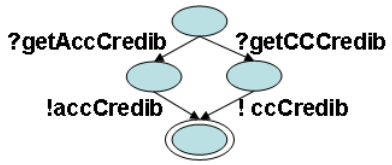


Figure 3.3: Example Protocol of the Bank service

request for credibility of a bank account is asked and sent back. In the right path, a credibility for credit cards is handled. The choice between the paths indicates that either a bank account or a credit card is checked, but not both.

Layer	*Concept*	*Operator*	*Description*
Type	Element	addType(τ, d, p)	add element with tag τ, datatype d following path p
		removeType(τ, p)	remove element with tag τ following path p
	Constraints	updateCC(x', y', e, p)	set cardinality to min x' and max to y'
		updateSC(c^s, p)	set structural constraint c^s
Protocol	State	addState(s_i)	add a state
		removeState(s_i)	remove a state.
	Transition	addTrans(s_i, m, s_j)	add transition from state s_i to state s_j with message m
		removeTrans(s_i, m, s_j)	remove transition s_i, s_j, m
	Message	addMsg(rec, sen, g)	add message with receiver rec and sender sen and type g
		removeMsg(rec, sen, g)	remove message rec, sen, g

Table 3.2: Basic Change Operators

3.3 Changes in a Service Specification

The changes applied to a service must be recognized by the manager in order to incorporate these changes in the business process. Our approach is to regard the service specifications as models and use model management operators to realize and recognize changes. Model management primitives or operators commonly consist of addition, removal and update. Other operators, such as insert, move and copy, are less frequently used. Table 3.2 presents the basic change operators we define for the type- and protocol layer. Strictly speaking, the update operators in Table 3.2 are not basic operators, as they can be composed from an add and remove operator. However, as the constraints may not be empty (a type must always have a cardinality and be part of a structure) we categorized these here as basic.

For each of the operators the semantics can be defined formally. For example a transition rule for the addition of a (normal) state to the protocol is as follows.

$$\frac{\text{addType}(s_i)}{< S, s_0, F, M, T > \rightarrow < S \cup s_i, s_0, F, M, T >}$$

Operator	*Description*
updateType(d, p)	update element with datatype d following path p
moveType(e, p_f, p_t)	move element e within message from location following path p_f to the new location following path p_t
moveMsg(m, t_i, t_j)	move message m from transition t_i to t_j
moveTypeMsg(e, p_i, m_i, m_j, p_j)	move element e from message m_i to message m_j
addTree(T_x, p)	add subtree T_x following path p
removeTree(T_x, p)	remove subtree T_x following path p
moveTreeMsg(T_x, p_i, m_i, m_j, p_j)	move subtree T_x from message m_i to message m_j

Table 3.3: Composite Operators

This rule states that if s_i is to be added (nominator) then the protocol is adjusted such that the s_i is added to the set of states S (denominator). For the other operators a similar line of reasoning can be deployed. The complete operation semantics for the basic change operators can be found in Appendix B.

Next to basic operators, composite operators can be defined as in Table 3.3. Next to these move operators, the same operators can be defined for subtrees; we describe here only "add" and "remove", but these also apply to the move operators.

The operators are commonly stored in a script, called change-script. A change-script Δ (also called edit-script or delta-script) is a sequence of operators that transform one model into another, more formally defined as $\mathcal{P}_1 \xrightarrow{o_1} \mathcal{P}_2$ where $\mathcal{P}_2$ is the result of applying the operator o_1 to $\mathcal{P}_1$. For a sequence $\Delta = o_1, .., o_m$ of basic operations, we say $\mathcal{P}_1 \xrightarrow{\Delta} \mathcal{P}_{m+1}$ if there exist $\mathcal{P}_2, .., \mathcal{P}_m$ such that $\mathcal{P}_1 \xrightarrow{o_1} \mathcal{P}_2 \xrightarrow{o_2}, .. \xrightarrow{o_m} \mathcal{P}_{m+1}$.

Three properties of a set of operators are distinguished, namely completeness, minimality and consistency (Casati et al., 1998), see Section 2.7 of Chapter 2. Completeness refers to the ability to transform a model of a class of models to another model of that same class.

Theorem 1. *The set of operators defined in Table 3.2 for protocols is complete.*

Proof. It needs to be shown that given a class of models $\mathbb{P}$, any model $P \in \mathbb{P}$ can be transformed into any other model $P' \in \mathbb{P}$ using only the operators of Table 3.2(see Definition 3 in Chapter 2).

We show here the proof for protocols without the type layer. However, for the type layer the same line of reasoning can be deployed. Let $\mathcal{P} =< S, s_0, F, M, T >$ and $\mathcal{P}' =< S', s_0', F', M', T' >$ be two randomly chosen models from $\mathbb{P}$. To prove that they can be transformed in each other, we must show that $\mathcal{P} \xrightarrow{\alpha} \mathcal{P}'$, the proof of the reverse, $\mathcal{P} \xrightarrow{\gamma} \mathcal{P}'$, is analogue [1]. In order to transform a model we must show that there is a script Δ that transform them. Given the specification of the models, the algorithm that creates such as script Δ is defined in Algorithm 2. Using this algorithm

Algorithm 3.1 diff(Business Protocol P, Business Protocol P')

$\Delta \leftarrow \emptyset$
$\forall s \in S \backslash S' : \Delta \leftarrow \Delta \cup removeState(s)$
$\forall s \in S' \backslash S : \Delta \leftarrow \Delta \cup addState(s)$
$\forall (s_i, s_j, m) \in T \backslash T' : \Delta \leftarrow \Delta \cup removeTrans(s_i, s_j, m)$
$\forall (s_i, s_j, m) \in T' \backslash T : \Delta \leftarrow \Delta \cup addTrans(s_i, s_j, m)$
$\forall (rec, sen, g) \in M \backslash M' : \Delta \leftarrow \Delta \cup removeMsg(rec, sen, g)$
$\forall (rec, sen, g) \in M' \backslash M : \Delta \leftarrow \Delta \cup addMsg(rec, sen, g)$
return Δ

(called diff), we get a script Δ that removes everything from $\mathcal{P}$ that is not in $\mathcal{P}'$ and adds everything that is in $\mathcal{P}'$ that is not in $\mathcal{P}$. This proofs $\mathcal{P} \xrightarrow{\Delta} \mathcal{P}'$. □

Minimality of a set of operators refers to the ability of transforming models with a minimal set of operators. The minimality of a set of operators is guaranteed if for every construct/concept in the model exactly two operator exists, namely the add and remove. Note, that through the use of parameters operators minimality can not be stated in absolute terms.

The consistency of a set of operators implies that after applying a script to a valid model again a valid model is returned. As stated in the previous

[1] It furthermore follows that γ is the inverse of α, thus $\gamma = \alpha^{-1}$.

chapter, consistency can be guaranteed in two ways: either by specifying validation constraints as precondition for the operators (thereby ensuring that after every operator a valid model is achieved), or by specifying validation constraints that hold for the whole model and check these after all the operators of a script have been applied. Both approaches (if applied properly) yield the same result, namely a valid model. In this thesis we will specify properties over the whole model. The reason is that in this way it creates less dependability on how the operators are specified. No matter how the operators are specified, the properties must hold for the model.

3.3.1 Mismatches

The operators defined in the previous section are used in our research to represent changes in the service specification of services. These changes are mismatches between what was and what is. In literature, different mismatches have been identified that can occur between services. To align our research with these previous efforts, we map our operators to these mismatches. The mismatches are defined as patterns of change operators. We use work done by Benatallah et al. (2005) for the protocol layer, Ponnekanti & Fox (2004) for the data layer and van den Heuvel (2007) for both layers. With notable exception of mismatches identified on data values and semantics, we use all mismatches from the above mentioned literature.

Although it is not our goal to provide an exhaustive overview of all mismatches identified in research, the selection of mismatches of the mentioned work encompasses the majority of interoperability problems. The mismatches and corresponding change operators are shown in Table 3.4. Most mismatches, such as Extra Field, Missing Field, Extra Message and Missing Message consist of a single basic operator. Other mismatches, such as Message Ordering, consist of a composite operator such as "move". The more complex patterns, such as Message Split and Message Merge, are described in detail below.

- *Message Merge:* If the content of several messages (≥ 2) is moved to one other message. This can be seen as x move operations between different messages (thus, multiple move(T_e, m_i, m_j, y)), where x is the number of elements that is to be moved. This mismatch, identified by Benatallah et al. (2005) on protocol level, has the special condition

Layer	*Mismatch*	*Operators*
Type	Extra Field	addType(e, d, y)
	Missing Field	removeType(e)
	Wrong Type	updateType(d, p)
	Cardinality Mismatch	updateCC(min, max, p)
Protocol	Extra Message	addMsg(m, t)
	Missing Message	removeMsg(m)
	Message Ordering	moveMsg(m, t_i, t_j)
	Message Split	addMsg(m_j), moveTypeMsg$(e, p_i, m_i, m_j, p_j)^x$
	Message Merge	removeMsg(m_i), moveTypeMsg$(e, p_i, m_i, m_j, p_j)^x$

Table 3.4: Mismatches and corresponding operators

that all the content of a message has been moved and thus the message is missing (removeMsg(m_i)).

- *Message Split:* If the content of one message is spread over several (≥ 2) messages. Similar to the Message Merge, this can be regarded as a sequence of "move element" operators between messages. The mismatch (Benatallah et al., 2005) is defined on protocol level and states that a message must have been added. We regard these two mismatches as special cases of the inter-message moves operator.

Note that we have neither included the Facet Mismatch (Ponnekanti & Fox, 2004) nor the Parameter Constraints (Benatallah et al., 2005), for we do not include values or constraints on values. Semantic issues or naming problems, that are part of the Signature Mismatch, are neither considered.

3.4 Orchestrator

The current industry standard for representing service orchestrations is the Business Process Execution Language for Web Services (BPEL4WS or BPEL for short) (Andrews et al., 2003). In our model management approach, BPEL is a model and model management operators can be defined over the concepts in BPEL, as suggested in (Hiel et al., 2008a).

However, BPEL suffers from two drawbacks. First, BPEL by itself does not provide any operational semantics and therefore it is not possible to show what a model exactly means. To remedy this, many researchers provided formalizations of BPEL (among others, Process Algebra (Salaün et al., 2004; Liu et al., 2007), Automata (Fu et al., 2004a; Pistore et al., 2005b) and Petri Nets (Hinz et al., 2005; Lohmann et al., 2009)). For an overview of formalisms proposed for modeling BPEL, we refer the interested reader to (van Breugel & Koshkina, 2006).

Second, BPEL contains a lot of different concepts and relations which potentially create complex models (Cardoso, 2006). To capture all these concepts and relations in operators and maintain consistency is therefore a difficult task. In this thesis, we therefore do not focus on the industry standard itself, but on one of the formalizations, namely Guarded Automata (Fu et al., 2004a). As can be deduced from the name, a guarded automaton is a finite state automaton with guards. An advantage of using automata is that they can be represented as graphical models. Graphical models are intuitive and easy to grasp. This makes them well suited for illustrating how changes affect a business process and how a business process can be adjusted to adapt to these changes.

An orchestrator enacts a business process. We are interested in interoperability and study how changes in service providers impacts data and data dependencies in the business process. These data dependencies we capture in data mappings, mappings for short. We define an orchestrator as follows:

Definition 11. *(Orchestrator)*
An orchestrator is a tuple $\mathcal{O} =< \mathcal{BP}, \mathcal{MP} >$ *where* $\mathcal{BP}$ *is a business process and* $\mathcal{MP}$ *is a set of mappings.*

We model the business process of the orchestrator using a guarded automaton. We give here first a formal description of the guarded automaton, after which we describe the guards and mappings in more detail in Section 3.4.1 and 3.4.2 respectively.

Definition 12. *(Business process)*
A business process is the tuple: $< M, S, s_0, F, T >$ *where* M, S, s_o, F *are defined as in Definition 10. Each transition* $t \in T$ *of the guarded automaton has a source state* $s_1 \in S$ *and a destination state* $s_2 \in S$ *and is in one of the following three forms:*

1. *local-transition,* $t =< s_1; (\epsilon, \varrho); s_2 >$*, where* ϱ *is the transition guard. The transition changes the state of the automaton from* s_1 *to* s_2*.*

2. *receive-transition,* $t =< s_1; (?m, \varrho); s_2 >$*, where* $m \in M$*. The transition changes the state of the automaton from* s_1 *to* s_2*.*

3. *send-transition,* $t =< s_1; (!m; \varrho); s_2 >$*, where* $m \in M$ *and* ϱ *is the transition guard. The transition changes the state of the automaton from* s_1 *to* s_2*.*

For readability purposes, we use the notation for transitions in Definition 12 also for guardless automata, the guard in these transitions should be read as being true.

For intermediaries such as the orchestrator, the business process consists of exchanged messages (business protocols) and business logic. The protocol of the orchestrator represents the communication with the environment of the service, similar to the published protocols of the service providers. The protocol of the orchestrator is the composition of the protocols of the service providers. How to compose an orchestrator, and corresponding protocol, is discussed in Chapter 4. Figure 3.4 shows an example part of the orchestration for our retailer example.

This example deals with an order management process that is structured as follows. After having received a purchase order from a customer, the retailer executes three tasks. Firstly, the retailer ascertains that sufficient parts are in stock by contacting the inventory service. Secondly, the retailer inquires a shipper, whether he can deliver the parts to the customer before the requested date. Lastly, the retailer checks the creditworthiness of the customer. For this purpose, he invokes an external web-service offered by a trusted third party, here a bank. Once these tasks are completed, the order management process is concluded by sending an acceptance or rejection to the customer.

As can be seen in Figure 3.4, for the protocol it suffices to use a finite state machine. However, for representing business logic this is not enough. For business logic, we therefore make use of the guards of the guarded automata.

3.4.1 Guards

As stated in Definition 12, a guarded automata has guards on the transitions. Guards express constraints on the data and provide the expressivity to rep-

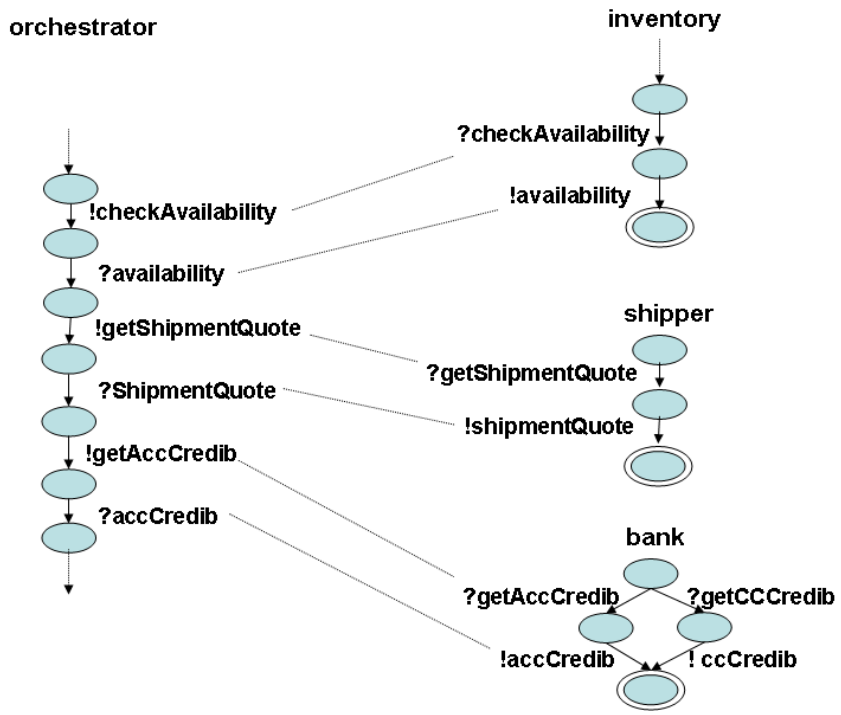


Figure 3.4: Partial Orchestration of the Order Management Process

resent business rules (as will be described in Chapter 4). A guard consist of a condition and a set of assignments. A guard is written as $\varrho \equiv (\varrho_1 \Rightarrow \varrho_2)$, where ϱ_1 is a condition expressed as a boolean XPath expression (W3C, 1999) on the message classes (as well as any local variables), and ϱ_2 is an assignment of the form $p \leftarrow exp$ where p is a XPath location path and exp is an XPath location path or an XPath expression.

XPath is commonly used to traverse and manipulate XML documents (in our case type trees). We express these conditions and assignments in a subset of XPath. This subset of XPath contains the core functionality of XPath and allows us to demonstrate how business rules can be used in a business process. This fragment of XPath (Fu et al., 2004d) consists of the following operators: child axis (/), descendant axis (//), self-reference (.), parent-reference (..), basic type test (b()), node name test (t), wildcard (*), and predicate ([]).

Definition 13. *(XPath)*
An XPath expression is defined with the following grammar

$$exp \rightarrow p \mid exp\ op\ exp$$
$$p \rightarrow r \mid /r \mid //r$$
$$r \rightarrow s \mid r/s \mid r//s$$
$$s \rightarrow . \mid .. \mid n?([exp]) \mid position() \mid last()$$
$$n \rightarrow b() \mid t \mid *$$

where $n?$ denotes n or "empty string" and $([exp])$ denotes zero or more repetitions of $[exp]$. In the syntax rules of Definition 13, an expression on basic types (such as boolean, integer, and string) can be constructed by combining XPath location paths (represented by p) and operators on basic types (represented by op). There are two types of location paths: relative location paths and absolute location paths. An absolute location path starts with / or //. A relative location path (represented by r) consists of a list of steps (represented by s) which are connected with / or //. The steps in a relative location path are evaluated from left to right. A step can be a self-reference (.), a parent-reference (..), or a more complex form which consists of a node test (n) and a sequence of predicates of the form $[exp]$. A node test n has three possible forms: type test ($b()$), name test (t), and wildcard match (.). Finally, a step can be a function call such as $position()$ or $last()$ with the following restriction: Function calls can only appear as the last step of a location path.

We assume that the operation op is either a relational operator ($=, \neq, <, >, \leq, \geq$), an arithmetic operator (+,-,*, /,%), or a boolean operator ($\vee, \wedge, \neg$). Note that, when used as a condition, a boolean XPath expression evaluates to "true" if its result set contains at least one true value.

Consider the type tree in Figure 3.2(c) and the following examples of xpath expressions:

- //BankAccount/accountType,
- /AccCredib/BankAccount [accountID > 10].

The results are respectively, the string holding the accountType information and true or false (depending on the value of the accountID). For more information and complete semantics of this fragment of XPath we refer the interested reader to (Fu et al., 2004d).

3.4.2 Mappings

Data received by the orchestrator can either be stored, used for inference, or directly used for input of another service. In the last case, data from a message is mapped/assigned to the structure of another message. We use data mappings to relate data provided and required for messages. These mappings, in model management called morphisms (Melnik et al., 2003b), are assignments of the form $p \leftarrow p'$ where both sides are location path expressions. Included in a location path is the message m in which the path must hold and the leaf-node e to which it will assign or retrieve a value. For the readability of our analysis we will make them explicit in the context of mappings. With $\mathbf{valid}(p, m, e)$, we denote that the path p in the type tree of message m ending at element e is valid. Because the element e is included in the path expression, we will, where needed, also use the short notation $\mathbf{valid}(p, m)$.

Definition 14. *(Mapping)*
A mapping is defined as:

$$(m, p, e\{x, y\}) \overset{u}{\leftarrow} (m', p', e'\{k, l\}).$$

This mapping states that in message m following path p, the element e gets the value of the element e' of message m' following path p' with a cardinality of u. We define the cardinality u as a range of values. If the cardinality is 1, we omit the cardinality. In other mapping representations, the cardinality and the element are often included in the query itself, however, for readability of our analysis we make them explicit. With $\mathcal{MP}$ the set of mappings (assignments) of the orchestrator is denoted. This set is obtained by collecting all the assignments from the guard of every transition in a guarded automaton.

Definition 15. *(Set of mappings)*
The set of mappings $\mathcal{MP}$ is defined as:

$$\begin{aligned}\{(m, p, e\{x, y\}) \overset{u}{\leftarrow} (m', p', e'\{k, l\}) : {} & !m \in M, e \in g_m, \boldsymbol{valid}(p, m, e), \\ & ?m' \in M, \boldsymbol{valid}(p', m', e'), e' \in g_{m'}, \\ & e \simeq e', \boldsymbol{opt}(e) = \boldsymbol{opt}(e'), x \leq k\},\end{aligned}$$

with the match-operator '$\simeq$' denoting whether a leaf-node matches another leaf-node, which means comparing names and data-type. For matching, we assume usage of the same namespace to avoid naming conflicts. Note that in this thesis we only work on type level, and therefore do not match data values. With g_m we denote the type tree of message m. With $\mathcal{MP}^*$, the set of all possible mappings is denoted.

```
(AccCredib,/bankaccount,accountID <- Order,/bankaccount,accountID)
(AccCredib,/bankaccount,accountType <- Order,/bankaccount,accountType)
```

Figure 3.5: Example mappings

In Figure 3.5 an example of a set of mappings for the AccCredib message is shown. In this example set, the account information such as accountID and accountType is mapped from the Order message to the GetCredibility message which will be sent to the bank.

3.4.3 Adaptation Operators

Similar to the operators defined for protocols, we define operators for a business process. We use these operators for realizing adaptation. If a manager needs to adapt a service, it needs to have the tools to do so. Guarded automata are an extension of normal automata and therefore many of the operators specified for business protocols can be applied to business processes. However, in addition, we need operators for adapting the guards and mappings of the orchestrator. We define these operators as follows.

The change operators defined in Table 3.2, together with the operators in Table 3.5 form the adaptation operators in this thesis. For the simplicity of the discussion, we omit here the operators for capturing changes in and manipulation of Xpath expressions. We focus on interoperability between services and leave adaptivity to changing business requirements (expressed in xpath) for future work.

Note that the update operators in Table 3.5 can be considered as redundant composite operators since their behavior can be simulated by using only add and remove. However, for readability purposes, we prefer to use them explicitly. Following the same line of reasoning we will often use the move operator as well.

Concept	*Operator*	*Description*
Mapping	addMap$((m, p, e) \leftarrow (m', p', e'))$	add new mapping
	removeMap(m, p, e)	remove existing mapping
	updateRHS(m, p, e)	update right-hand side
	updateLHS(m, p, e)	update left-hand side.
	updateCard(x, y)	update cardinality
Condition	addCond(exp, ϱ)	add condition exp to guard ϱ
	removeCond(exp, ϱ)	remove condition from guard ϱ
Assignment	addAssign$(p \leftarrow exp, \varrho)$	add assignment $p \leftarrow exp$ to guard ϱ
	removeAssign$(p \leftarrow exp, \varrho)$	remove assignment $p \leftarrow exp$ from guard ϱ

Table 3.5: Operators for Mappings and Guards

Similar to the change-script for capturing changes, we define the adaptation-script for adaptation. The definition is identical, except that we denote the basic operator with a and the script with σ, thus: $\mathcal{MP} \xrightarrow{\sigma} \mathcal{MP}'$.

3.5 Orchestration Properties

We defined models and the operators for manipulating these models in the previous sections. However, we have not specified the properties that must hold for these models. In this section we describe the meta-model (collection of properties) that must hold for an orchestration as a whole and for the components in this orchestration.

3.5.1 Orchestration

The models described above define how services and the orchestrator are represented. However, these models do not specify what defines an orchestration. In a service orchestration a central mediator is assumed to be present in the topology of services. This means that all messages are either send or received by the orchestrator. To check whether a given service composition is a service orchestration and not a choreography, we define an orchestration schema as follows:

Definition 16. *(Orchestration schema)*
An orchestration schema is a tuple: $(\mathcal{P}, M)$ *where* $\mathcal{P} =< P_1, .., P_n >$ *is the finite set of peers and* M *is the finite set of messages satisfying the following:*

$$M = \bigcup_{j \in \{1..n\}} M_j \text{ and,}$$

$$\forall < rec_1, sen_1, g_1 >, < rec_2, sen_2, g_2 > \in M$$
$$[rec_1 = rec_2 \vee sen_1 = sen_2 \vee rec_1 = sen_2 \vee rec_2 = sen_1]$$

In this definition, we state that the message set of an orchestration is the union of the message sets of the services it communicates with. For each two messages part of this set, they must have a sender or receiver in common (the orchestrator). Our definition is a more restrictive version of the definition of a choreography by Bultan et al. (2003).

An example of an orchestration schema is depicted in Figure 3.6, in this figure the retailer is the orchestrator. All messages either go to, or originate from the retailer.

In an orchestration, we are often interested in the behavior of an individual service within the context of the orchestrator. For example for determining if a change to a service has an impact on the orchestrator (see Section 5.3 in Chapter 5). We therefore define the projection operator π. For an orchestration schema $C = (\mathcal{P}, M)$, given a word $w \in M^*, \pi_i(w)$ denotes the projection of w to the alphabet M_i of the peer P_i i.e. $\pi_i(w)$ is a subsequence of w obtained from w by removing all the messages which are not in M_i. When the projection operation is applied to a set of words the result is the set of words generated by application of the operation to each word in the set.

Next to the orchestration schema, we assume that in an orchestration there is only one party that can initiate the process. For instance, in our running example, the customer initiates the process by sending a request for a catalog to the retailer.

3.5.2 Components

Next to the schema which defines what constitutes as an orchestration, we have a number of properties that must hold for each component in this orchestration.

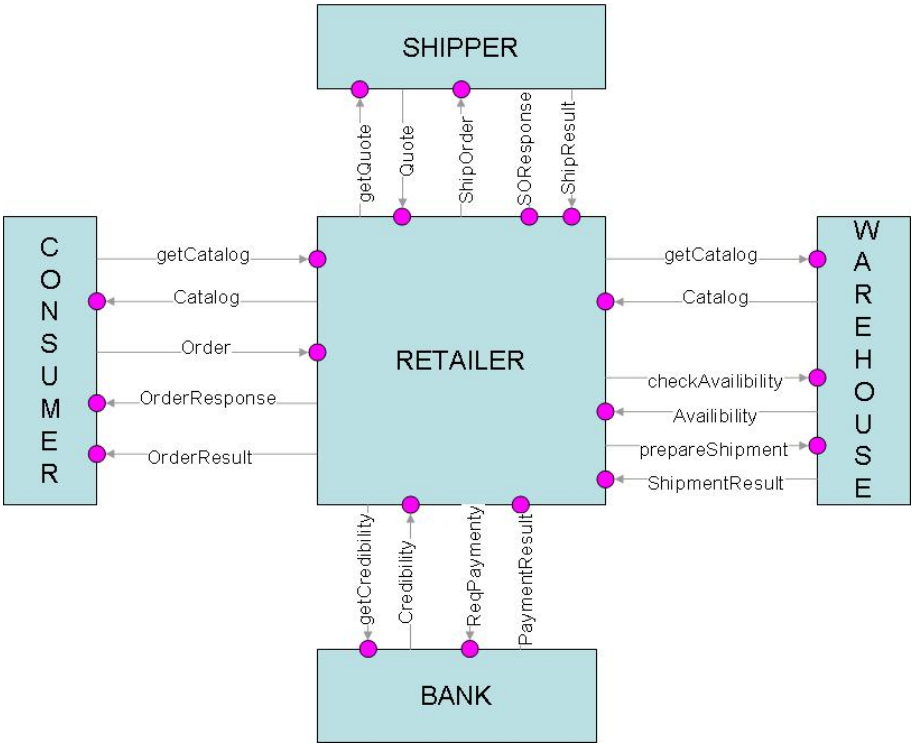


Figure 3.6: Example Orchestration Schema

We assume that each finite automaton (FA), for instance the business protocols and business process, in the orchestration is deterministic. If a FA is nondeterministic then it can be converted to an equivalent deterministic FA (Hopcroft et al., 2000). Another assumption is that there is no nonproductive state in the FA's used, i.e., there exists a path from every state to an accepting state. Furthermore, we require that every finite state automaton has no incoming transition to the initial state and no outgoing transition from any final state.

3.6 Discussion

In this chapter, we described the models for representing service specifications as well as a model for representing an orchestrator. Based on these models, we defined operators for capturing changes and effectuating adaptation. Together these models and operators form the foundation of a manage-

able service orchestration. To control this orchestration, and thereby making it adaptive, we describe the internal workings of a manager in the chapters 4 and 5.

One of the limitations of our research captured by the definition of our models is that we deal with syntactic and structural interoperability only. We do not handle semantic interoperability.

In our service orchestration we assume that the participating parties have a shared ontology. In short, we assume that names have the same meaning throughout the orchestration. How to come to this ontology and how this should be enforced is beyond the scope of thesis. Our approach can be extended to deal with semantics by extending the matching operators to include ontologies.

Chapter 4

Automatic Composition of a Hybrid Orchestrator

4.1 Introduction

A service that uses functionality from other services is called composite. The process of developing a composite service is called service composition or synthesis. Since the emergence of Service Oriented Computing a lot of attention is going to service composition and the many aspects related to it. An important motive behind this research is that the ability of nesting services fulfills one the promises of SOA, namely rapid development of new services.

We distinguish between two elements of service composition, namely the *existing services* and the *specification of the business requirements* of the new, to be composed, service.

The *existing services* represent the functionality that can be re-used. However, these services also provide constraints on what compositions are possible. Services publish information about themselves how they are to be used. This information, such as the contents of messages and the ordering of these messages, restrict the set of possible orchestrators.

Next to the existing services, the *business requirements* of the, to be constructed, service must be specified. What is specified differs per approach. We distinguish between two different automatic composition approaches, namely action-based and AI-planning based. Action-based approaches (e.g. (Berardi et al., 2003; Gerede et al., 2004)) specify an action-structure as a

automaton and focus on the delegation of actions to different services (subcontractors). All actions are specified and the goal is to find a suitable delegation. AI-planning based approaches (e.g., (Ponnekanti & Fox, 2002; Pistore et al., 2005a)) on the other hand specify only the desired result, typically in terms of data, and use planning to reach this goal.

In this chapter we outline how, using our model for a business process, business requirements are represented in a business process and how such a business process can be automatically constructed using our *compose* operator. Our approach for service composition distinguishes from current state-of-the-art in three aspects:

1. **Protocol-based:** Most automatic composition approaches act from a action-based or AI-planning based perspective where the focus is either on actions or data. However, the building blocks for synthesis, the interfaces of other services, are interaction-oriented. Interfaces specify not only the data of messages but also the ordering in which to send these messages (see also Section 3.2 of Chapter 3). Therefore, we advocate that a more intuitive and useful perspective is that of the interaction. Our vantage point is to start with a set of business protocols and the goal is to assemble these into an orchestrator. This furthermore allows us to analyze how changes in the interaction affect the business process (described in Chapter 5).

2. **Business rules:** A lack of current synthesis approaches is that they only use services for the composition process. A problem due to this is, that without business logic or business rules the composition must be an exact match with what is offered, i.e., the internal structure of the orchestrator must exist completely in the interfaces of the service providers. We provide here a hybrid approach where we incorporate business rules in the synthesis process. These business rules are the "glue" between services and embody the added value of the orchestrator.

   An additional advantage of our approach is that business rules remain clearly visible in the orchestration, therewith not losing their identity and allowing for better adaptability and maintainability.

3. **Policy-based selection:** The goal of existing automatic composition approaches is to demonstrate whether an orchestrator exists and if it

exists, show how this orchestrator is structured. As there may exist many different structures that satisfy the composition requirements, these approaches often incorporate implicit assumptions about how the orchestrator should be structured (for instance, by specifying a goal structure).

In our approach, we aim to make these assumptions explicit by defining policies. Policies provide the means to select the most desired orchestrator among a set of valid alternatives. We identify key decision points where an user can define a policy. These policies also contain the possibility to choose for a *semi*-automatic synthesis process.

Our research covers two levels in interoperability, namely type and protocol level (see Section 3.2 of Chapter 3). For an orchestrator to be interoperable it needs to be compatible on both levels. In this chapter, we define the goal property that guarantees that our composed business process of the orchestrator is correct with respect to interoperability. This means that for all messages sent in this orchestration the right data is available and that thus no exceptions/incompatibilities will occur.

Next to being data-consistent, our constructed orchestrator supports asynchronous communication. Systems that use synchronous communication are more expensive to implement and more tightly coupled. An advantage of these systems is, however, that formal analysis methods exist that can be applied to achieve reliable software. In order to create more loosely coupled services contemporary approaches are more and more using event-based (asynchronous) communication semantics. A problem with asynchronous communication is that it makes formal analysis more difficult (Betin-Can et al., 2005). In order to use formal analysis *and* support asynchronous communication, we conform to properties which guarantee that synchronous communication semantics equals asynchronous communication semantics.

This chapter is organized as follows: In the next section (Section 4.2), we describe the overall process of composing an orchestrator and the components that we use in this process. After this, we present in Section 4.3 the properties that must hold to guarantee interoperability for an orchestrator. Whether or not an orchestrator can be constructed is determined in Section 4.4. Because we want a minimal orchestrator and also want it to work with asynchronous communication semantics, we reduce the set of possible

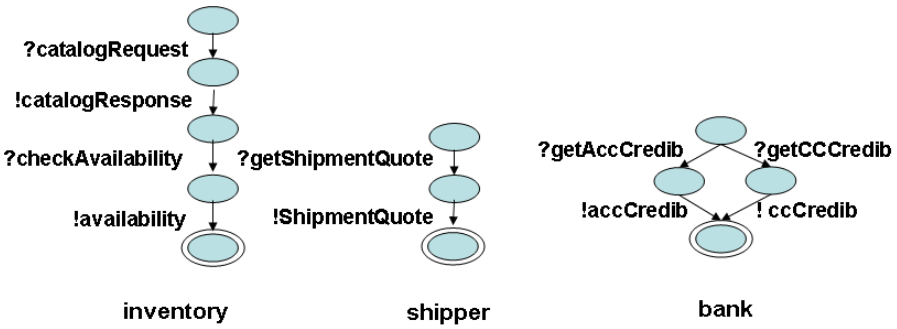


Figure 4.1: Inventory, Shipper and Bank service

orchestrators in Section 4.5. The desired orchestrator is selected using user defined policies in Section 4.6. The last section (Section 4.7) concludes this chapter with a discussion.

4.2 Composition Process

In our example scenario, an organization wishes to create a new intermediary service which handles order management. In this organization, the IT department is slowly adopting a service-oriented approach to its information systems. They already converted a legacy system used for inventory management into a service. The other services to be used in this new composite service are already published by the bank and shipper. Figure 4.1 illustrates the business protocols (defined in Section 3.2.2) of the involved services.

In Figure 4.2 the composition process is illustrated. It consists of two phases, namely synthesis and selection. In the synthesis phase the services together with the business rules and target business protocol are combined. The result of the synthesis phase is a set of valid alternative orchestrators. In the selection phase, from this set the most preferred orchestrator is chosen based on policies. We describe below in detail the three models we use for representing the requirements and wishes of an organization, namely target business protocol, business rules and policies.

4.2.1 Target Business Protocol

The target business protocol, also called the goal business protocol, represents the communication between the customer and the, to be composed,

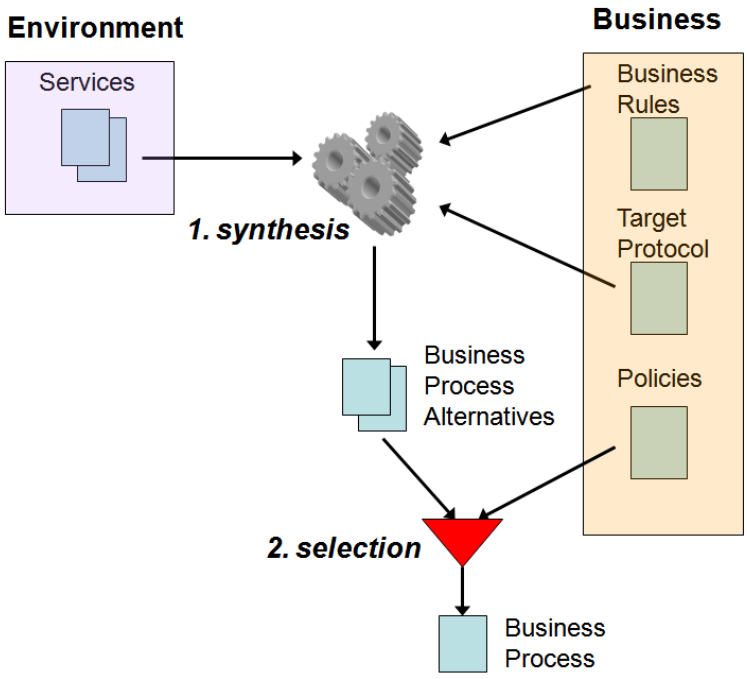


Figure 4.2: Hybrid Service Composition

orchestrator. Like the business protocols specified by the service providers this is a bilateral communication protocol.

The difference between our target business protocol and the target business process that is commonly used in automatic composition is that our approach allows for more flexibility. In the target business process, see for instance Berardi et al. (2003), Pathak et al. (2008) or Gerede et al. (2004), all actions are specified and the goal is to delegate all actions of the target specification to services that are capable of executing these actions. In our approach, by using a target business protocol instead of a target business process we specify only the external behavior and thereby create more flexibility in the structure of the internal business process.

In Figure 4.3 we specify the target protocol for the retailer. This protocol is structured as follows: First, the retailer expects a catalog request from the customer, after receiving this request, the retailer sends back the catalog and waits for an order from that customer. Based on this order, the retailer either accepts or rejects the order by sending an accept message or reject message respectively.

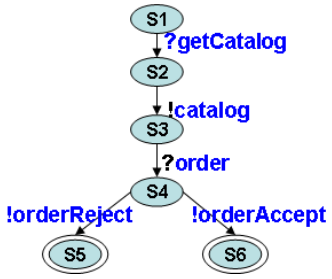


Figure 4.3: Target protocol of the retailer

4.2.2 Business Rules

As described in the introduction to this chapter, if only services are used in the synthesis process, then the internal structure of the orchestrator must be a complete match with what is offered by these services. To provide an organization with the means to assemble a service that is more than a collection of other services we incorporate business rules. The standard definition of a business rule (c.f. (Charfi & Mezini, 2004; Rosenberg & Dustdar, 2005; Cibrán et al., 2007; van Eijndhoven et al., 2008)) is defined by the Business Rules Group as: "a statement that defines or constraints some aspect of the business. It is intended to asset business structure or to control or influence the behavior of business" (BRG, 2000).

Incorporating rules in a service composition process requires business rules to be expressed in an executable format. We presume that an analysis phase has been done and that a set of business rules in the form of if/then statements is the result of that phase. We also assume that this analysis encompasses aspects such as further decomposition into atomic parts and whether rules cannot work together (conflicts). We focus on the implementation phase where emphasis is put on how to incorporate business rules in the automatic composition process.

Several classification schemas exist for business rules. Although multiple types of rules have been distinguished, such as constraint rules, action-enabler rules, computation rules and inference rules (von Halle, 2001), in this thesis we primarily study inference rules, also called decision rules. An inference rule is defined as "a statement that tests conditions and upon finding them true, establishes the truth of a new fact" (von Halle, 2001). Inference rules represent the (automated) decisions made in a business process and

capture an important aspect of business logic. In this thesis, we focus on maintaining consistency with respect to data (information) and the decisions made based on that data. For this reason, we limit ourselves to inference rules and leave the other types of rules for future work.

In our running example, we use two business rules. In the specification of the target protocol of the retailer (Figure 4.3), the retailer sends either an orderReject or an orderAccept message to the customer. The decision whether an order should be accepted or rejected is a business rule. The business rules for our running example, in text and XPath notation are:

1. If the product is available and the customer's creditworthiness is okay and the shipper sent a shipmentquote, then the customer's order is accepted.

   if $availability/Availability[available = true]$ **and**
   $shipmentQuote/ShipmentQuote[shipmentID > 0]$ **and**
   $(accCredib/AccCredib[balance > 0]$ **then** $//acceptance := true$

2. If either the product is not available, the customer's balance is not good or the shipper cannot ship, then the order is rejected.

   if $availability/Availability[available = false]$ **or**
   $accCredib/AccCredib[balance < 0]$ **or**
   $shipmentQuote/ShipmentQuote[shipmentID <= 0]$
   then $//rejected := true$

Although we focus on inference rules, our approach can easily be extended to cover the other types. Limitations on what kind of rules can be expressed, can be drawn from the representation of the rules, in our case the subset of XPath described in Section 3.4.1.

Translating Business Rules in Guarded Automata

We assume that business rules are available in an executable format (Xpath), however, they cannot directly be used for synthesis. In Chapter 3, we expressed business protocols and the business process using Finite Automata (FA), therefore if business rules are to be incorporated in the composition process, they must be expressed using FA as well. The advantage of translating business rules in automata is that it allows us to build further on

existing automatic synthesis approaches based on automata, especially (Berardi et al., 2003; Gerede et al., 2004)).

We formally define a business rule as follows:

Definition 17. *(Business rule)*
A business rule is a guarded automaton $< M, S, s_0, F, T >$ *where* $M = \{m\}$, $S = \{s_1, s_2\}$, $s_0 = s_1$, $F = \{s_2\}$ *and* $T = \{t\}$ *and the transition is of the form:* $t =< s_1; (m; \varrho); s_2 >$, *where* ϱ *is the transition guard. The transition changes the state of the automaton from* s_1 *to* s_2. *The message is defined as* $m =< rec, sen, g >$ *where* $rec = sen$ *is the orchestrator and* g *is the MSL type of the message, defined in Definition 9.*

Recall from Section 3.4.1 that we model a guard as $\varrho \equiv (\varrho_1 \Rightarrow \varrho_2)$, where ϱ_1 is a boolean XPath expression (W3C, 1999) on the message classes, and ϱ_2 is an assignment of the form $p \leftarrow exp$ where p is a XPath location path and exp is an XPath location path or an XPath expression. Using this format, we can directly translate a business rule to a guarded automaton containing only one transition, which consists of the condition of the rule (mapped to the guard ϱ_1) and the action of the rule (mapped to the assignment ϱ_2).

In Figure 4.4, in WSAT syntax (Fu et al., 2004e), we illustrate the business rule for accepting an order. The full description of the guarded automata representing both business rules can be found in Appendix C.

```
States{sbr21,sbr22},
InitialState {sbr21},
FinalStates{sbr22},
TransitionRelation{
     t_br21{ sbr21-> sbr22: accept,
          Guard{ $availability/Availability[available = true] and
                 $accCredib/AccCredib[balance > 0] and
                 $shipmentQuote/ShipmentQuote[shipmentID > 0]
                   =>
                   $accept[ //acceptance:= true]}
      }
 }
```

Figure 4.4: Business rule for accepting an order

When business rules are used in a composition process, they are often

not recognizable in the resulting model. As we describe in the following, by representing the business rules as one transition automata, we do not lose the track of the rules during the composition process and are able to locate where the rules are triggered.

4.2.3 Policies

In some situations, there are multiple variants of orchestrators that can be composed from the set of services, business rules and target protocol. In such a situation, a choice has to be made which alternative is preferred over another. To make such a choice we use policies. Policies allow the developer to select the best orchestrator for what he defines as best. Policies are used where a choice is to be made between valid alternatives. Policies specify the precondition and post-condition of a choice, i.e. providing a set of alternatives and the selected candidate. A policy can be implemented in different ways. We provide the developer with a few example strategies for making the choice, but also allow the developer to define his own implementation for making the choice. We define below three example strategies that can always be applied .

First Occurrence: The first option is taken.

Random: The option is randomly chosen.

Interactive: The decision is presented to the developer, who then must decide which option is chosen. If this strategy is chosen then the selection phase will be semi-automatic. How to present this choice in a user-friendly way is beyond the scope of this thesis.

4.3 Orchestrator Properties

Using the components discussed in the previous section, we synthesize an orchestrator. However, not all possible orchestrators constructed with these components will be guaranteed to work without incompatibilities. In order to successfully compose an orchestrator that is guaranteed to be interoperable, we define in this section a number of properties. These properties provide the means to be able to measure whether an orchestrator is compatible. Furthermore, using these properties we can determine whether changes

affect the orchestrator and whether an adaptation script solves emerging incompatibilities. The following properties indicate whether an orchestrator is internally consistent with respect to type, mappings and business rules.

4.3.1 Type

Before a type can be send, it needs to be received first. For this we define whether a type is provided, see definition below.

Definition 18. *(Provided)*
A type $g \in g_{!m_j}$ is provided (denoted by provided(g)) iff:

$$\forall < m_0, .., !m_j > \in MEX^j \; \exists ?m_i \in < m_0, .., !m_j > \exists g' \in g_{?m_i} \; [g \simeq g']$$

For an ordered sequence,

$$provided(g_1, .., g_k) = provided(g_1) \wedge .. \wedge provided(g_k)$$

holds, and for a choice of sequences,

$$provided(g_1|..|g_k) = provided(g_1) \vee .. \vee provided(g_k)$$

holds. We define a message m to be provided, denoted provided(m), if its type g_m is provided.

Intuitively, Definition 18 means that the type (data) must be available (provided) to the orchestrator before it can be used in a message. For instance, the retailer can only send the bank-account information from a customer to the bank after it has received this information from that customer.

Definition 19. *(Data-consistency)*
We define a service with business process $\mathcal{BP}$ to be data-consistent (denoted by d-consistent($\mathcal{BP}$)) iff:

$$\forall !m \in M \; [provided(m)].$$

If for all outgoing messages of the orchestrator the data is available then we call the orchestrator data-consistent.

4.3.2 Mappings

Data (types) received by the orchestrator can either be stored, used for inference, or directly used for the input of another service. In the last case, the data from a message is mapped/assigned to the structure of another message. We use data mappings to connect types that are provided to types that are required.

Definition 20. *(Mapping-sufficient)*
A set of mappings $\mathcal{MP}$ is sufficient for a type $g\{x, y\}$ (denoted by m-sufficient$(\mathcal{MP}, g\{x, y\})$) iff:

$$\exists(m_j, p, g\{x, y\}) \leftarrow (m_i, p', g'\{k, l\}) \in \mathcal{MP} \ \wedge$$
$$\neg\exists(m_j, p, g\{x, y\}) \leftarrow (m_x, p'', g''\{k, l\}) \in \mathcal{MP}[m_i \neq m_x \wedge p \neq p'']$$

For an ordered sequence,

$$\textit{m-sufficient}(g_1, .., g_k) = \textit{m-sufficient}(g_1) \wedge .. \wedge \textit{m-sufficient}(g_k)$$

holds, and for a sequences of choices,

$$\textit{m-sufficient}(g_1|..|g_k) = \textit{m-sufficient}(g_1) \otimes .. \otimes \textit{m-sufficient}(g_k),$$

where $\otimes$ is the boolean xor-operator. We define a message m to be sufficiently mapped (denoted m-sufficient$(\mathcal{MP}, m)$) if its type g_m is sufficiently mapped.

Intuitively, a sequence is sufficiently mapped if all parts are sufficiently mapped. For a choice construct, there must be exactly one part that must be sufficiently mapped. The difference with the definition of provided (Definition 18) is that provided means that data must be available before it can be used (mapped), whereas mapped-sufficiently specifies exactly what data must be put into the messages. Thus, for instance, an element can be available many times, however, if that element is required only once then it should be mapped only once. How to deal with the situation when a type is provided multiple times is discussed in Section 4.6.

Definition 21. *(Mapping-completeness)*
We define a service to be mapping-complete (denoted by m-complete$(\mathcal{BP}, \mathcal{MP})$) iff the following property holds:

$$\forall!m \in M \ [\textit{provided}(m) \wedge \textit{m-sufficient}(\mathcal{MP}, m)]$$

Intuitively, the definition of mapping-completeness means that we call a service to be mapping-complete if for all outgoing messages the types are available (provided) and adequately mapped. If a service is mapping-complete, then this service is also data-consistent.

Lemma 2. *m-complete*$(\mathcal{BP}, \mathcal{MP}) \Rightarrow$ *d-consistent*$(\mathcal{BP})$.

The proof for this lemma follows directly from Definition 21 and 19. Intuitively, this lemma states that if a service is mapping complete (has correct mappings for all outgoing messages) then it must by data consistent (has all data needed for those mappings).

Lemma 3. $\neg$*provided*$(m) \Rightarrow \neg$*m-sufficient*(m).

The proof of this lemma follows directly from Lemma 2. Intuitively, this lemma means that if data is not received then it can not be used in a mapping.

Lemma 4. *If d-consistent*$(\mathcal{BP}')$ *and not m-complete*$(\mathcal{BP}', \mathcal{MP})$*, then* $\exists\sigma[\mathcal{MP} \xrightarrow{\sigma} \mathcal{MP}' \wedge$ *m-complete*$(\mathcal{BP}', \mathcal{MP}')]$.

Proof. From the definition of data consistency and mapping completeness it follows that d-consistent$(\mathcal{BP}')$ but not m-complete$(\mathcal{BP}', \mathcal{MP})$ means: $\exists m \in M[$*provided*$(m) \wedge \neg$m-sufficient$(\mathcal{MP}, m)]$. Hence, an adaptation script σ can be found (using operators as described in Table) such that after applying, a valid mapping (Definition 14) exists such that:

$$(m_j, p, e\{x, y\}) \xleftarrow{u} (m_i, p', e'\{k, l\}).$$

□

This lemma states that if a business process is data consistent then a set of operators can be found to make it's mapping complete.

4.3.3 Business Rules

Similar as for mappings, for business rules we check whether data (type) is available. For business rules this means checking if enough data is available in order to be able to evaluate the condition(s). Because we work on type level, and not on instance level, we cannot check whether a business rule

returns a "true" or "false", however based on the types available it is possible to deduce whether a business rule might fire.

I.e., we check whether sufficient information is received to be able to evaluate parts of the condition of the business rule. If enough information has been received such that the business rule could in theory be triggered, then it can be evaluated.

Definition 22. *(Can evaluate)*
The boolean function $\boldsymbol{eval}(exp)$ *is defined on the boolean XPath expressions used for the conditions:*

$$\boldsymbol{eval}(exp) := \begin{cases} \forall < m_0, ..?m_i, ..m_j > \in MEX^j & \\ \qquad\qquad [valid(exp, m_i)] & \text{if } exp \text{ is a location path;} \\ eval(exp^1) \wedge eval(exp^2) & \text{if } exp = (exp^1 \wedge exp^2); \\ eval(exp^1) \vee eval(exp^2) & \text{if } exp = (exp^1 \vee exp^2); \\ \neg eval(exp^1) & \text{if } exp = \neg exp^1. \\ false & \text{otherwise.} \end{cases}$$

We define that a guard ϱ *can be evaluated, denoted* $eval(\varrho)$*, if its conditions (boolean expression) are provided.*

To check whether a rule can be evaluated, we only need the data (XPath path expressions) in the conditions. Together with the logical connectors $(\neg, \vee, \wedge)$ it is possible to deduce whether a rule can be triggered. For example, the business rule for rejecting an order (discussed in Section 4.2.2) contains two or-connectors, this rule can therefore be evaluated after the orchestrator receives either one of these three messages (aval, credib, shipQuote).

4.4 Orchestration Existence

An important question in automatic composition is whether a composite service, in our case an orchestrator, exists. In this section, we use standard automata techniques to answer this question. The idea is to construct an automaton which contains all possible valid execution paths (called a product machine), thus reducing the question of whether an orchestrator exists to whether a product machine can be created. If an orchestrator exists then the direct follow-up question is how to construct it. We answer this question in Section 4.6.

Definition 23. *(Composite service)*
A composite service is a tuple $< (\mathcal{P}, M), \mathcal{B}, p_d, p_1, .., p_n >$ *where* $(\mathcal{P}, M)$ *is an orchestration schema, and* $\mathcal{B} =< b_1, .., b_r >$ *is a set of business rules. Each* p_i, $i \in \{1, .., n\}$, *is the business protocol of the corresponding peer* $P_i \in \mathcal{P}$, $|\mathcal{P}| = n + 1$, *and* p_d *is the implementation of the target protocol* $P_d \in \mathcal{P}$.

Because all components of the composite service are expressed as finite state machines, we use the notation that $\mathcal{A}$ represents any FA in a composite service. For our running example, the number of FA's in a composite service is $|\mathcal{A}|$ is 6, with 2 business rules, 3 services and 1 target protocol. As defined in Definition 10 in Chapter 3, let the set of states, transitions and messages of an FA $\mathcal{A}_i$ be labeled with subscript i with $i \in \{P_d, P_1, .., P_n, B_1, ..., B_r\}$. For instance, if T is the set of transitions of the target protocol P_d, then we denote it as T_{P_d}.

Definition 24. *(Configuration)*
Let $\mathcal{S} =< (\mathcal{P}, M), \mathcal{B}, p_d, p_1, .., p_n >$ *be a composite service. Then a configuration of* $\mathcal{S}$ *is a tuple* $< s_{P_d}, s_{P_1}, .., s_{P_n}, s_{B_1}, .., s_{B_r} >$, *where* s_{P_d} *is the state of the target protocol* P_d, s_{P_i}, $1 \leq i \leq n$, *is the state of service* P_i, *and* s_{B_j}, $1 \leq j \leq r$ *is the state of business rule* B_j.

At every configuration a number of transitions can be executed. However, not all transitions can be chosen when regarding interoperability. For example, a transition may send a message for which the data is not available (provided). To ensure that this type of situation does not occur, we define the enabled function.

Definition 25. *(Enabled)*
The boolean function ***enabled***(t) *is defined on a transition as follows:*

$$\textbf{\textit{enabled}}(t) := \begin{cases} true & \text{if } !m_t : t \in T_{P_i} \text{ with } 1 \leq i \leq n,\ i \neq d \\ & \text{or } ?m_t : t \in T_{P_d}; \\ provided(!m_t) & \text{if } t \in T_{P_d}; \\ provided(?m_t) & \text{if } t \in T_{P_i} \text{ with } 1 \leq i \leq n,\ i \neq d; \\ eval(\varrho_t) & \text{if } t \in T_B; \\ false & \text{otherwise.} \end{cases}$$

The enabled function decides whether a transition can be executed. For sending a message this means that its type must be provided. Because we are in the process of constructing the orchestrator, the ? and ! are from

the perspective of the owning participant. For instance, in the fourth line of Definition 25 $?m_t$ means that the service will receive a message from the orchestrator. For business rules we check whether there is sufficient information available in order to evaluate it.

Definition 26. *(Enabled derivation relation)*
For two configurations $\gamma =< s_{P_d}, s_{P_1}, .., s_{P_n}, s_{B_1}, .., s_{B_r} >$ *and* $\gamma' =< s'_{P_d}, s'_{P_1}, .., s'_{P_n}, s'_{B_1}, .., s'_{B_r} >$*, we say that* γ *derives* γ' *(denoted as* $\gamma \rightarrow \gamma'$*), if one of the following two conditions holds:*

- *Automaton* $\mathcal{A}_i$ *performs a* ***send*** *action, i.e. there exist* i *and a transition* $t = (s_i; (!m; \varrho); s'_i)$ *such that* $t \in T_i$*,* ***enabled****(t), and* $s'_k = s_k$ *for each* $k \neq i$*.*

- *Automaton* $\mathcal{A}_i$ *performs a* ***receive*** *action, i.e. there exist* i *and a transition* $t = (s_i; (?m; \varrho); s'_i)$ *such that* $t \in T_i$ *and* ***enabled****(t), and* $s'_k = s_k$ *for each* $k \neq i$*.*

In the derivation relation we state how configurations are related to each other, with the enforcement that transitions are enabled.

Definition 27. *Let* $\mathcal{S} =< (\mathcal{P}, M), \mathcal{B}, p_d, p_1, .., p_n >$ *be a composite service, a configuration sequence* $\gamma = (\gamma_0, \gamma_1, .., \gamma_k)$ *is a* ***run*** *of* $\mathcal{S}$ *if it satisfies the first two of the following three conditions, and* γ *is a* ***complete run*** *if it satisfies all three conditions.*

1. $\gamma_0 =< s^0_{P_d}, s^0_1, .., s^0_n, s^0_{B_1}, .., s^0_{B_r} >$ *where* $s^0_{P_d}$ *is the initial state of the target protocol,* s^0_i *is the initial state of peer* P_i *for each* $i \in \{1..n\}$ *and* $s^0_{B_1}$ *is the initial state of business rule* B_1 *for each* $i \in \{1, .., r\}$*,*

2. *for each* $j \in \{0, .., k-1\}, \gamma_j \rightarrow \gamma_{j+1}$*,*

3. $\gamma_k =< s^F_{P_d}, s^F_1, .., s^F_n, s^F_{B_1}, .., s^F_{B_r} >$ *where* $s^F_{P_d} \in F_{P_d}$*, and for each service* P_i*,* $s^F_i \in F_{P_i}$*, and each business rule* B_j*,* $s^F_{B_j} \in F_{B_j}$*.*

Note that in the final configuration we distinguish between automata (services and business rules) that participate in the composition based on whether they are in the initial state or not. We assume that the initial state does not have incoming arcs and thus once an automaton participates it cannot withdraw.

Lemma 5. *Let $\mathcal{S} =< (\mathcal{P}, M), \mathcal{A}_1, .., \mathcal{A}_n >$ be a composite service. Given a word $w \in M^*$, if for each $i \in \{1, .., n\}, \pi_i(w)$ is a complete run of $\mathcal{A}_i$, then w is a complete run in $\mathcal{S}$.*

Proof. Let $w =< m_1, .., m_k >$. Since for each $i \in \{1, .., n\}$, the projection $\pi_i(w)$ is a complete execution there exists a corresponding complete run mex_i for $\pi_i(w)$. We show that w is a complete run of $\mathcal{S}$ by constructing a complete run which simulates each mex_i. The construction has k phases. In each phase j, let $< m_1, ..., m_{j-1} >$ be the message path until $j-1$. We simulate the transmission of message m_j where only the sender and receiver of m_j are involved. We start with the sender m_j, we execute the transition and the next action should be either a receive message m'_j or a sending of a message m'_j where $j' > j$. Then we turn to the receiver m_j and execute the receiving action of that message. Then we turn to the automaton associated with message m_{j+1}.

To prove the correctness of the above process, we need to show that after each phase the simulation can always continue, and that at the end of the simulation, the configuration is consistent with the states of each peer at the end of that peer's run. They are guaranteed by the following induction assumption: prior to phase j of the simulation, the following statement (denoted Q) is true: for each automaton $\mathcal{A}_i$, its complete local run is simulated up to a state s_i where s_i is either a final state or there is a next **enabled** send or receive message action in the local run of $\mathcal{A}_i$. When Q holds at the beginning of the phase j where $j \in \{1..k\}$, the simulation at phase j guarantees that Q holds at the end of phase j (i.e. the beginning of phase $j+1$). □

A composite service $\mathcal{S}$ as described above can be expressed in a guarded automaton using the enabled derivation relation.

Definition 28. *(Enabled Guarded Automaton)*
Given a composite service $\mathcal{S} =< (\mathcal{P}, M), \mathcal{A}_1, .., \mathcal{A}_n >$ an equivalent Enabled Guarded Automaton (EGA) is a tuple $< M_{\mathcal{S}}, S_{\mathcal{S}}, s^0_{\mathcal{S}}, F_{\mathcal{S}}, \delta >$ where

- *$S_{\mathcal{S}}$ is the set of configurations (Definition 24),*
- *$s^0_{\mathcal{S}} \in S_{\mathcal{S}}$ is the initial configuration (Definition 27 condition (1)),*
- *$F_{\mathcal{S}} \subseteq S_{\mathcal{S}}$ represents the finite set of final configurations (Definition 27 condition (3))*

- $M_{\mathcal{S}} = M$ *is the set of messages,*
- δ *is the enabled derivation relation (Definition 26)*

In the following, with $MEX_{\mathcal{S}}$ we denote the set of message exchange paths accepted by the enabled guarded automaton (MEX is defined in Section 3.2.2).

Lemma 6. *$MEX_{\mathcal{S}}$ is data-consistent.*

Proof. Let EGA be a guarded automaton with as language $MEX_{\mathcal{S}}$ and let $!m_x \in M$. It holds that every transition of EGA must be enabled (Definition 26). We distinguish between two cases $!m_x$ is send to service or to the customer. In both cases, Definition 25 states that $!m_x$ is provided (line 3 and 4 respectively). □

As every transition is enabled in this automaton, meaning that for every transition the data is available, this automaton is guaranteed to be able to interoperate with the services providers used in it. Although the automaton only contains transitions for which there is enough data, this does not necessarily mean that an orchestrator exists. The criteria by which we define that an orchestrator exists are as follows.

Definition 29. *(Orchestrator existence)*
Let $\mathcal{BP}$ be a business process and $MEX_{\mathcal{BP}}$ be its set of message exchange paths, then an orchestrator $\mathcal{O}$ exists if the following properties hold:

(1) $\forall < rec, sen, g_1 > \in M[rec = \mathcal{O} \vee sen = \mathcal{O}]$

(2) $MEX_{P_d} = \pi_{P_d}(MEX_{\mathcal{BP}})$

In this definition the first property (1) states that every message must either be sent or received by the orchestrator. The second property (2) states that the target business protocol must be part of the behavior.

Theorem 2. *Let $\mathcal{S} =< (\mathcal{P}, M), \mathcal{B}, p_d, p_1, .., p_{n-1} >$ be a composite service and $MEX_{P_d} = \pi_{P_d}(MEX_{\mathcal{S}})$ then there exists an orchestrator $\mathcal{O}$.*

Proof. The composite service $\mathcal{S}$ is a business process. We further need to show that the properties of Definition 29 are satisfied. Property (1) is satisfied by the definition of a composite service (Definition 23) and property (2) is given in the theorem itself. □

4.5 Orchestrator Reduction

The composite service described in the previous section contains all possible runs. However, the model has two shortcomings: (a) the composition model only holds for synchronous communication, and (b) the model is not minimal. We describe and handle these two shortcomings in this section.

4.5.1 Synchronizable

Results for automatic composition typically only hold for models (such as finite state machines) that use synchronous communication. In response to this, Fu et al. (2004a,b,c) introduced synchronizability properties that guarantee, if they hold, that asynchronous communication semantics can be equated with synchronous communication semantics. By adhering to these properties we ensure that our orchestration is able to work with event-based communication. For the interested reader, a full description of these properties and corresponding proofs can be found in (Fu, 2004).

Definition 30. *(Synchronizable)*
Let $\mathcal{S} =< (\mathcal{P}, M), \mathcal{B}, p_d, p_1, .., p_{n-1} >$ *be a composite service.* $\mathcal{S}$ *is said to be synchronizable if it produces the same set of message paths under both the asynchronous and the synchronous communication semantics.*

To ensure synchronizability two sufficient conditions were introduced, called the synchronous compatible and the autonomous condition.

Definition 31. *(Synchronous compatible)*
A composite service $\mathcal{S} =< (\mathcal{P}, M), \mathcal{A}_1, .., \mathcal{A}_n >$ *is synchronous compatible if for each word* $w \in M^*$ *of length* k*, and each message* $m \in M_a^{out} \cap M_b^{in}$ *for* $a, b \in \{1..n\}$*,:*

$$(\forall i \in 1..n \pi_i(w) \in mex_i^k)) \wedge \pi_a(wm) \in mex_a^{k+1}) \Rightarrow \pi_b(wm) \in mex_b^{k+1}.$$

Synchronous compatible condition states that at every state if a peer can send a message, then that receiver must have a receiving action.

Lemma 7. $MEX_{\mathcal{S}}$ *satisfies the synchronous compatible property.*

Lemma 7 can be deduced from the fact that we are using synchronous models for our composition.

Definition 32. *(Autonomous)*
A composite service $\mathcal{S} =< (\mathcal{P}, M), \mathcal{A}_1, .., \mathcal{A}_n >$ *is autonomous if for each peer* $\mathcal{A}_i$*, and for each word* $w \in M_i^*$*, exactly one of the following three statements holds:*

(a) w *is accepted by* $\mathcal{A}_i$*,*

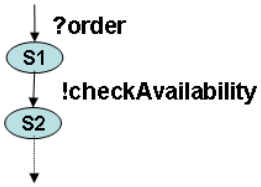
(b) there exists $\beta \in M_i^{in}$ *such that* $w\beta \in mex_i$*,*

(c) there exists $\alpha \in M_i^{out}$ *such that* $w\alpha \in mex_i$*.*

A web service composition is autonomous if each peer, at any moment, can do only *one* of the following: 1) terminate (it has reached a final state), 2) send a message, or 3) receive a message. Note that this property should not only hold for the composite service, but also for every peer in the composition, thus also the orchestrator.

Lemma 8. $MEX_{\mathcal{S}}$ *does not satisfy the autonomous property.*

Proof. Counterexample: We show that the set of enabled transitions in some configuration contains send and receive transitions from one party. In the context of our running example, from the perspective of the retailer, the retailer receives the order of the customer (as specified in the target protocol (Figure 4.5[1])). See Appendix A for a full description of the service specifications. This order message contains enough information such that both checkAvailability and getShipQuote message are enabled in state S1. After sending the checkAvailability message (thus in state S2), there are again two transitions enabled, namely receiving the Availability message from the inventory and sending the getShipQuote to the shipper. Thus here the autonomous property does not hold.



State	*Enabled Transitions*
S1	!checkAvailability, !getShipmentQuote
S2	?availability, !getShipmentQuote

Figure 4.5: Part of the retailer (left) and corresponding enabled transitions (right)

□

[1]For simplification of this example, we have left the bank out

4.5.2 Minimal

The composite service defined in the previous section contains unnecessary long message paths which are redundant. For instance, in our retailer example a valid sequence is depicted in Figure 4.6.

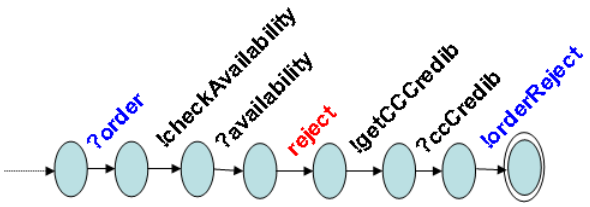


Figure 4.6: Example of a partial not-minimal message path

In this sequence, after the business rule for rejecting the order has been executed (reject) first another service is called (the bank for credibility information) after which the retailer sends the orderReject message to the customer. After the triggering of the business rule there is no need to request other services because there is sufficient information for sending the orderReject message.

Because we want to prevent invoking other services unless it is necessary, we aim to keep the orchestrator minimal.

Definition 33. *(Minimal)*
A sequence of messages mex is minimal if a transition of the target is executed at the (earliest) point it can be executed.

Lemma 9. *MEX_S is not minimal.*

The example described above serves as a counterexample to the hypothesis that MEX_S is minimal.

4.5.3 Transition Selection

In order to deal with minimality and synchronizability, we define in this section a selection function and a new derivation relation. Because both the function and the derivation relation reason about the set of enabled transitions, we first define the **enabledSet**(γ) function. This function returns the set of all enabled transitions for a configuration γ.

Definition 34. *(EnabledSet)*
Let $\gamma =< s_{P_d}, s_{P_1}, .., s_{P_n}, s_{B_1}, .., s_{B_r} >$ *be a configuration. Given* γ*, we define the function* $\boldsymbol{enabledSet} : \gamma \rightarrow \mathcal{E}$ *(where* $\mathcal{E}$ *is a set of transitions) as follows:*

$$\boldsymbol{enabledSet}(\gamma) =$$

$$\bigcup_{i \in \{P_d, P_1, .., P_n, B_1, ..., B_r\}} \{< s_i, m, s_2 > :< s_i, m, s_2 > \in T_i, \boldsymbol{enabled}(< s_i, m, s_2 >)\}$$

Note that if a transition is enabled it stays enabled until it is executed. For instance, if enough data is present to send a credibility check to the bank on time 0, then this data is also present at time 1. The relaxation of this assumption on the monotonicity of the data is left for future work.

The following function selects the set of executable transition that will be executed.

Definition 35. *(Select)*
Let $\mathcal{E}$ *be the set of all enabled transitions for all automata of a configuration* γ*. Given* $\mathcal{E}$*, we define the function* $\boldsymbol{select} : \mathcal{E} \rightarrow \mathcal{E}'$ *(where* $\mathcal{E}'$ *and* $\mathcal{E}$ *are set of transitions with* $\mathcal{E}' \subseteq \mathcal{E}$*) as follows:*

$$\boldsymbol{select}(\mathcal{E}) = \begin{cases} t_b & \text{if } \exists t_b =< s_1, s_2, ?m >\in \mathcal{E}[t_b \in T_{P_i} \wedge i \neq d]; \\ t_x & \text{if } t_x \in T_{P_d} \wedge t_x \in \mathcal{E}; \\ \mathcal{E} & \text{otherwise.} \end{cases}$$

The first line of the function checks whether there is a transition with an incoming message. The second line checks whether there are any transitions possible for the target protocol. If so then we choose only these as possible transition to execute.

Based on the above described selection of enabled transition, we define a derivation relation that is well-formed.

Definition 36. *(Reduced derivation relation)*
Let the set of states, etc of an automaton $\mathcal{A}_i$ *be labeled with subscript* i *with* $i \in \{P_d, P_1, .., P_n, B_1, ..., B_r\}$*. For two configurations*
$\gamma =< s_{P_d}, s_{P_1}, .., s_{P_n}, s_{B_1}, .., s_{B_r} >$ *and* $\gamma' =< s'_{P_d}, s'_{P_1}, .., s'_{P_n}, s'_{B_1}, .., s'_{B_r} >$*,*
γ *derives* γ'*, if one of the following two conditions holds:*

- *Automaton* $\mathcal{A}_i$ *performs a* ***send*** *action, i.e. there exist* i *and a transition* $t = (s_i; (!m; \varrho); s'_i)$ *such that* $t \in T_i$*,* $s'_k = s_k$ *for each* $k \neq i$ *and* $t \in \boldsymbol{select}(\mathcal{E})$

- *Automaton $\mathcal{A}_i$ performs a* ***receive*** *action, i.e. there exist i and a transition $t = (s_i; (?m; \varrho); s'_i)$ such that $t \in T_i$ and $s'_k = s_k$ for each $k \neq i$, and $t \in \textbf{\textit{select}}(\mathcal{E})$*

This derivation relation takes into account both synchronizability as well as the minimality by using the **select** function.

Lemma 10. *Let $\mathcal{S}$ be a composite service using the enabled derivation relation (Definition 26) and $\mathcal{S}_{red}$ a composite service using the Reduced derivation relation. Then $MEX_{\mathcal{S}_{red}} \subseteq MEX_{\mathcal{S}}$.*

The language of the composite service created by the algorithm is a subset of the language of the composite service described above. The proof of this lemma can be easily shown by looking at the **select**($\mathcal{E}$) function, which is defined to give a subset of the set that was given as input.

Theorem 3. *$MEX_{\mathcal{S}_{red}}$ satisfies the autonomous property.*

Proof. Lemma 8 states that $MEX_{\mathcal{S}}$ does not satisfy the autonomous property. The difference between $MEX_{\mathcal{S}_{red}}$ and $MEX_{\mathcal{S}}$ is the derivation relation, more specific the set of enabled transitions that is executed. Thus, to prove that $MEX_{\mathcal{S}_{red}}$ does satisfy the autonomous property, we need to show by using the **select** function that a set that violates the autonomous property is never returned. We prove this by induction on the possible set of enabled transitions $\mathcal{E}_i$. Let $\mathcal{E}_i$ denote the set of enabled transitions obtained from the function *enabled*(γ_i) at time i. In this set there are four types of transitions, namely t_d^{in}, t_d^{out} denote incoming and outgoing message of the target protocol, t_S^{in}, t_S^{out} denotes incoming and outgoing messages of a service. We make two observations: First, based on our assumption that only one party has the initiative, it follows that $t_d^{in} \otimes t_d^{out} \otimes t_S^{in} \otimes t_S^{out}$ at time $i = 0$. Second, if a transition is enabled at time x, then it will also be enabled at time $x + 1$, unless executed.

Given theses types we distinguish between seven situations:

(1) $t_d^{in} \otimes t_d^{out} \otimes t_S^{in} \otimes t_S^{out}$: if only one type is element of $\mathcal{E}$ then the **select** function simply returns that type.

(2) $t_S^{out} \wedge t_d^{out}$: **select** function states that t_d^{out} is returned.

(3) $t_S^{out} \wedge t_S^{in}$: **select** function states that t_S^{in} is returned.

(4) $t_S^{out} \wedge t_d^{in}$: **select** function states that t_d^{in} is returned.

(5) $t_d^{in} \wedge \ t_d^{out}$: This situation can never occur, because the autonomous property must hold for each protocol (also for the target protocol).

(6) $t_S^{in} \wedge \ t_d^{in}$: This situation can not occur. At time $i = 0$, this situation cannot occur due to our assumption of one initial party. Thus there must be a time $i-1$, where either $t_S^{out} \in \mathcal{E}_{i-1}$ or $t_d^{out} \in \mathcal{E}_{i-1}$. However, if a transition is enabled at time i then it was also enabled at time $i-1$, thus we get the two following possible situations at time $i-1$: $t_S^{out} \wedge t_d^{in}$ or $t_S^{in} \wedge t_d^{out}$. However, in $t_S^{out} \wedge t_d^{in}$ the transition t_d^{in} would have been chosen, see (3). Also the possible situation $t_S^{in} \wedge t_d^{out}$ is not possible, see (7).

(7) $t_S^{in} \wedge t_d^{out}$: This situation can not occur. Similar reasoning as for the previous situation. Following our assumption that only one service will take the initiative at first, it follows that another message was sent out in the configuration before. We get two possible situation at time $i-1$: $t_S^{out} \wedge t_d^{out}$ or $t_S^{in} \wedge t_d^{in}$. For $t_S^{out} \wedge t_d^{out}$, see (2). For the other situation $t_S^{in} \wedge t_d^{in}$, we get into a loop where every time $i-1$ the other situation ((6) and (7) alternately) should hold. This loop continues until $i = 0$ where either (6) or (7) holds. However, at time $i = 0$ both situations can not hold.

All other possible combinations (and those including business rules) contain a situation as described above. From the above situations, it follows that $MEX_{\mathcal{S}_{red}}$ satisfies the autonomous property since every possible output of the **select** function produces a set that adheres to this property.

□

Theorem 4. *$MEX_{\mathcal{S}_{red}}$ is minimal.*

Proof. What we need to proof is that $MEX_{\mathcal{S}_{red}}$ does not contain any sequences, where at any point another transition could have been chosen instead of a transition by the target protocol. Assume that $MEX_{\mathcal{S}_{red}}$ is not minimal. Thus there must exists at least one $< t_1, ..t_n > \in MEX_{\mathcal{S}_{red}}$, such that at time $1 < x < n$ another transition t could be chosen such that $t \notin P_d$. However, since the well-formed derivation relation states that all chosen transitions must come out of the set produced by $\mathbf{select}(\mathcal{E}, t_l)$, it

follows that if a transition $t \in P_d$ exists, that this is executed (see Theorem 3). This is a contradiction and thus $MEX_{\mathcal{S}_{red}}$ is minimal. □

In the following we will describe how an orchestrator can be constructed with additional constraints to create a single orchestrator, e.g. in the case where multiple valid orchestrators exist how only one is selected.

In the initial configuration, all the services and business rules start in their initial state, the state of the orchestrator is set to the initial state of the target protocol.

4.6 Policy-based Orchestrator Selection

The product machine described above contains all possible alternatives for a business process. We assume that a business process is enacted in one particular way and not every time executed in a different variation. Therefore, we select only one of the possible alternative orchestrators. Algorithm 4.1 illustrates the general approach to composition. It represents our **compose** operator. In this algorithm, we first construct the product machine

Algorithm 4.1 Construction of an orchestrator: Compose()

```
Input: < P, B, P_d, θ >
Construct MEX_{S_red}
if MEX_{S_red} is empty then
  return error /* there is no orchestrator */
else
  select orchestrator O based on policies θ
  return O
end if
```

containing all alternatives, after which we select an alternative according to policies.

We distinguish between policies concerning two concepts, namely transitions and mappings. For each policy, we provide examples on how these policies can be implemented.

4.6.1 Transition Policy

We handle only bilateral business protocols. This one-to-one specification leads to greater flexibility (and possible alternatives) in our orchestrator. This flexibility is an effect of the fact that the individual protocols do not specify the interaction with other parties (as is assumed in a choreography perspective). This flexibility results in a choice between possible transitions (messages to be send).

An example of this flexibility in our running example is as follows: After receiving an order, the retailer has enough information to send three messages: to the bank for a credit-check, to the inventory to check the availability and to the shipper for a shipping-request. The order in which to send the messages is not specified in the protocol of each individual service because they are specified between two parties only. Therefore the retailer has the luxury to choose the order in which to send these messages. This also holds if multiple services offer the same functionality, for instance, if there were two banks who both offer a credibility check.

Definition 37. *(Transition policy)*
We define a transition policy as ***transPol:*** $T_{pol} \rightarrow \{t\}$ *where* T_{pol} *is set of transitions defined as* $\{t :!m_t, t \in T_{P_i}$ *with* $1 \leq i \leq n\}$ *and* $t \in T_{pol}$ *is a transition.*

The transition policy selects the transition that is most desired at a specific point. Next to the general applicable strategies for policies (specified in Section 4.2.3), we provide here three more examples:

- **Preferred Service:** An ordering is provided specifying which services are preferred over others.
- **Most data returned:** the option is taken which provides the most data in their return message(s).
- **Shortest path:** Albeit in principle all paths are minimal (see Section 4.5.2) this does not mean that every alternative orchestrator has paths of the same length. Therefore, to get the minimal orchestrator it is possible to calculate the paths to all final states.

Note, that this policy only selects between transitions containing messages that are to be send to services, and that it therefore does not affect other transitions like business rules or target protocol.

4.6.2 Mapping Policies

Concerning mappings, we make a distinction between two choices, namely if a message allows multiple combinations of data and if data is provided multiple times.

Type Options

Our model for expressing the type layer includes required and optional data in messages. The choice structure $g_1|...|g_k$ presented in Definition 9 of Chapter 3 allows different type structures to satisfy the structural constraints of a message. If a choice is present in a message and this message needs to be send, then the orchestrator needs to make a choice in which format (type structure) to send it.

For determining how data in a message should be mapped, we study the possible combinations that data can be put in a message. This means that we determine all possible combinations of leaf-nodes that satisfy the structural requirements of a message. These combinations we call the type options of a message.

If a leafnode is provided then it can be used for a mapping (see Lemma 4). The orchestrator should be mapping complete, therefore every message must be sufficiently mapped. We define a function ***createMapSet***: $L \rightarrow MP$ which takes as input a set of provided leafnodes L and returns a set of mappings (MP). Based on this function, we define a type-option as follows:

Definition 38. *(Type option)*
Let L_m be the set of leafnodes in message m. A type option is then defined as:

$$l_d \subseteq L_m[m\text{-}\textit{sufficient}(\textit{createMapSet}(l_d), m)]$$

Intuitively, a type option is a set of leaf-nodes that if put in the right format satisfies the structural constraints of a MSL Type. For example, the shipment request message illustrated in Figure 4.7 contains an or-connector. This message is satisfied by two different sets of leaf-nodes: {productID, quantity, streetname, zipcode, city, country} and {productID, quantity, referenceID}.

Based on the information that is available, we can determine whether the message is provided (at least one type option is present/available) and whether we have to choose between type-options, based on a mapping-policy.

```
GetShipmentQuote[
    productList[
        productID[xsd:int],
        quantity [xsd:int]
    ]{1,10},
    address[
        streetName [xsd:string],
        zipcode [xsd:string],
        city    [xsd:string],
        country [xsd:string]
    ] |
    referenceID [xsd:int]
],
```

Figure 4.7: Shipment request message

Definition 39. *(Type option policy)*
We define a type option policy as $\boldsymbol{TypeOptPol} : L \rightarrow l$ *where* L *is a set of type options and* l_d *is a data option.*

We define here two example implementations for dealing with this choice:

- **Most Mapped:** The option is taken with the most mappings, thus the option is chosen in which the most data is send.
- **Last Received:** The option is taken which contains information from the last received message.

Mapping

The type option policy chooses between different sets of leafnodes that satisfy the structural constraints of a message. Next to this, there exists a possibility that a leaf is provided multiple times. For example, a productID may be received in a quote message and in an order message. When this situation occurs a decision needs to be made which productID is to be used (mapped).

Definition 40. *(Mapping policy)*
We define a mapping policy as $\boldsymbol{mapPol} : \mathcal{MP} \rightarrow mp$ *where* $\mathcal{MP}$ *is set of mappings and* $mp \in \mathcal{MP}$ *is the chosen mapping.*

An example of how this policy can be implemented is to chose the mappings that belong to a message which is used for other mappings. The number of mappings to a certain message can distinguish a message as being a more important message.

The choice which element is best for a mapping is closely related to the matching of element. In this thesis, complex matching of elements is beyond the scope, however, we would like to point out that for this decision other approaches like for instance a semantic approach can be used to extend our work.

4.6.3 Policy-based Selection Algorithm

Using the policies defined in the previous section, an algorithm to select an orchestrator from the product machine is presented in Algorithm 4.2.

Algorithm 4.2 Selecting an orchestrator

Input: $< \gamma, \theta >$
$\alpha \leftarrow \emptyset$
$\mathcal{E} \leftarrow enabled(\gamma)$
$\mathcal{E}_t \leftarrow selectTransitions(\mathcal{E}, \theta)$
for each $< s_1, m, \varrho, s_2 > \in \mathcal{E}_t$ **do**
  $\alpha \leftarrow \alpha \cup \text{addTrans}(s_{\mathcal{O}}, m, \varrho, s_2)$
  $\alpha \leftarrow \alpha \cup \text{addState}(s_1)$
  $\alpha \leftarrow \alpha \cup \text{addMsg}(m)$
  if !m **then**
    $mp \leftarrow selectMappings(\mathcal{O}, \theta)$
    for each $(m, p, e) \leftarrow (m', p', e') \in mp$ **do**
      $\alpha \leftarrow \alpha \cup \text{addMap}((m, p, e) \leftarrow (m', p', e'))$
    end for
  end if
  $\gamma \xrightarrow{<s_1, m, \varrho, s_2>} \gamma'$
  $\mathcal{O} \xrightarrow{\alpha} \mathcal{O}'$
  Selecting an orchestrator $< \mathcal{O}', \gamma', \theta >$
end for

This recursive algorithm works as follows: The algorithm takes as input: an orchestrator (empty in the beginning), the configuration of $MEX_{\mathcal{S}_{red}}$ and the set of policies. From the set of enabled transitions at that configuration, a subset is selected using the *transPol* policy. For each transition in the set of selected transitions a script (containing operators defined in the previous chapter) is created for adding the appropriate state,transition and

policy selected mappings. This procedure is repeated until there are no more transitions to choose, i.e., final states have been reached.

Lemma 11. *The orchestrator $\mathcal{O}$ constructed by Algorithm 4.2 is mapping-complete.*

Proof. For every policy it holds that $\mathcal{E}' \subseteq \mathcal{E}$ therefore it follows that $MEX_{\mathcal{O}} \subseteq MEX_{\mathcal{S}_{red}} \subseteq MEX_{\mathcal{S}}$. Since $MEX_{\mathcal{S}}$ is data-consistent (Lemma 6) this implies that $MEX_{\mathcal{O}}$ is data-consistent. Lemma 4 shows that data-consistency implies that a set of mappings can be found, the selectMappings function called in Algorithm 2 returns the preferred data option based on the mapping policy. Thus, the orchestrator $\mathcal{O}$ is mapping-complete. □

Theorem 5. *The orchestrator $\mathcal{O}$ found by Algorithm 4.2 is well-formed.*

Proof follows directly from Lemma 11 and Theorem 2.

4.6.4 Example: Constructing a Retailer

Returning to our running example of the retailer. The full picture of the orchestration, with business rules, is depicted in Figure 4.8. In this figure three types of transitions are present, namely: transitions that indicate communication with services, transitions of the target protocol, and transitions specifying decisions made using business rules. For the business rules, the reject business rule is triggered three times. Each time the business rule is triggered, it is executed because more information is available for evaluating the condition. For the reject business rule, this happens after receiving the availability, after receiving the credibility and after receiving the shipment quote. The other business rule, the acceptance of an order, in contrast, is only triggered once.

All transitions of the target protocol are represented in the orchestrator therefore we can state that the orchestrator exerts the desired behavior. The specification of this orchestrator is given in Appendix D.

4.7 Discussion

In this chapter, we introduced our *compose* operator. Where traditional automatic composition processes only consist of business processes, we use additional business requirements such as *business rules* and *policies*. We

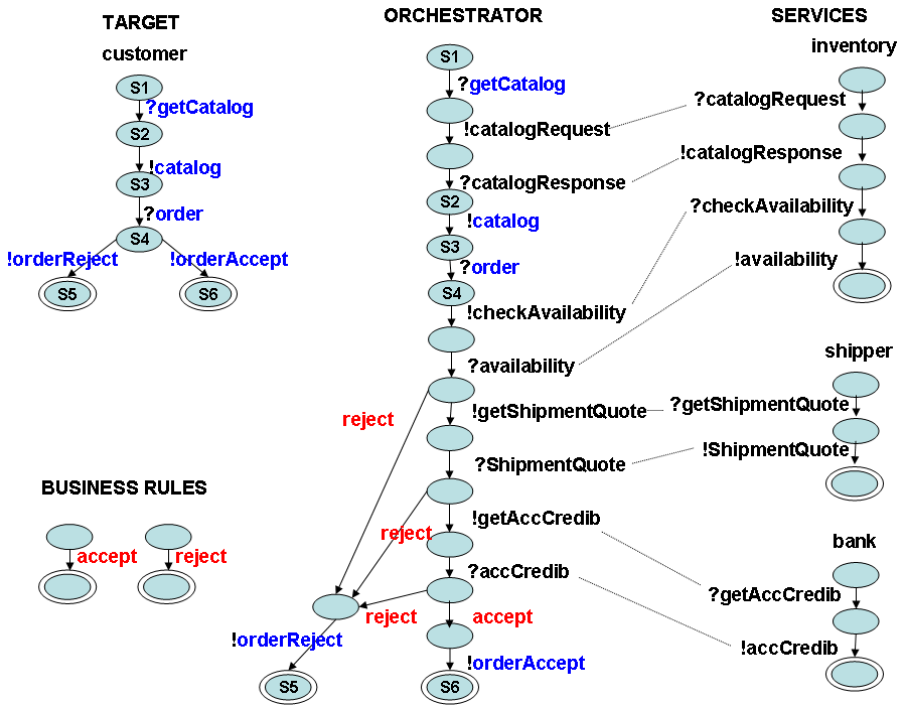


Figure 4.8: The Orchestration for the retailer

synthesize an orchestrator based on the business protocols published in the interfaces of the service providers. Our synthesis satisfies additional constraints, such that the orchestrator can be employed using *both synchronous and asynchronous communication semantics.*

One of the distinctive features of our composition approach was the usage of business rules. By translating business rules to guarded automata, we are able to incorporate them in the synthesis process. We modeled each business rule as a separate GA. However, for the expression of more complex business constraints more complex GA's can be used. If business rules are specified separately then the situation may occur that multiple rules can fire at the same time. If this situation is not desired then business rules should be glued together to form a single automaton, which represents the whole business logic.

A question or assumption that must be made for the orchestration is whether the names of the states and transitions of the different business protocols (to be part of the composition) are unique for the whole orchestration. For this dissertation, we assume that they are. In practical scenarios, it suf-

fices that the name of the state as well as the name of the peer is remembered along with the name of the state in the orchestrator. Uniqueness of names makes a difference for adaptation, as it is easier to localize changes when states or transitions are removed/added.

We advocate that the models of the components used for the construction of the orchestrator are kept for later reference, for instance keeping business rules in a rule management system. If business rules change in the organization then these changes are easily adapted in the orchestration. Otherwise, there is the risk of creating one monolithic process where the origin of the business rules cannot be traced back.

Although we adhere to the synchronizability properties and thereby ensure that our orchestrator can work with both synchronous as with asynchronous communication semantics; this in the end comes down to waiting for the other party to send a message (if this is indicated so by that party's protocol). Extending this would imply further study in asynchronous communication semantics.

Chapter 5

Automatically Adapting an Orchestrator

5.1 Introduction

In the previous chapter we described how to compose a service orchestrator. This orchestration functions correctly given the properties for interoperability as well as for the business requirements, i.e., business rules and target business protocol. However, even if software could be free of defects in the beginning, it is not possible to use it forever without modification (Parnas, 1994). The environment of the orchestrator, consisting of other services and the organization, will change. In this chapter we present a method for dealing with evolving service providers. More specific, we describe how to adapt an orchestrator with as goal to maintain interoperability.

As services evolve, changes to the interface of that service are inevitable (see Chapter 1). By publishing a new version of the interface changes are presented, and thereby propagated, to its service clients. In our research, we study how to adapt to these changes in such a way that changes do not propagate further to connected services. This is called evolution transparency or interoperability preservation (Andrikopoulos et al., 2009).

Whether a change to the interface of a service results in incompatibilities and whether an incompatibility can be solved, depends on the flexibility of the orchestration. The flexibility of the orchestrator depends on three aspects:

Partial usage of functionality: Ponnekanti & Fox (2004) show that in practice changes to a service often do not directly result in incompatibilities because the functionality was not used. A change to the functionality of a service provider that is not used by the orchestrator has no impact on the orchestrator.

Redundancy of information: A change may have the result that certain information is no longer available to the orchestrator. As this information may need to be send to another service, this would result in an incompatibility. However, in an orchestration information can be provided by multiple messages and by multiple parties. Therefore if the required information is removed then there is a chance that this information is still provided by another message.

Bilateral business protocols: Bilateral protocols specify the ordering of messages between one service provider and one service client. In our specification of an orchestration we only use bilateral protocols. Therefore there are no (direct) dependencies between service providers of the orchestrator. Because there are no dependencies between service providers, the orchestrator is able to choose the order in which to invoke these service providers.

In this chapter, we exploit each of these aspects in order to solve incompatibilities. In Chapter 2 we discussed the structure of an abstract algorithm for adapting the retailer, re-illustrated in Algorithm 5.1. We describe in this chapter the details of this algorithm and discuss each of the functions.

We use the adaptation process (Algorithm 2.1) to structure this chapter. We show how to detect changes and introduce the diff-operator in Section 5.2. Based on a change script we analyze whether changes are applicable (have impact) in Section 5.3. The decision process on how to adapt to any changes that have impact on the orchestrator is described in Section 5.4. This section includes the definition of three change categories (non-effective, solvable and problematic) and our notion of an incompatibility. After this, we explain in Section 5.5 our prototype. We conclude this chapter in Section 5.6.

Parts of this chapter have been published in (Hiel & Weigand, 2009; van den Heuvel et al., 2007).

Algorithm 5.1 Adaptation Process

```
receive(event_{P'_c})
Δ ← diff(P_c, P'_c)
Δ_a ← applicable(Δ, π_{P_c}(BP))
if Δ_a ≠ ∅ then
  α ← adapt(Δ_a, G, BP)
  if α = ∅ then
    send(event_{P'_c}, problematic) //escalate: unsolvable problem
  else
    update(α, BP) //adapt the orchestrator
  end if
end if
```

5.2 Change Detection

An adaptation process is initiated by receiving an event indicating the presence of a change. The manager will then detect what has changed, i.e., it will capture the changes in a change script. Change detection deals with comparing models, and many approaches have been discussed for different types of models, among others XML (Cobena et al., 2002; Wang et al., 2003), DTD (Leonardi et al., 2007), Graphs (Wang et al., 1995; Bunke, 2000) and (Business) Protocols (Yellin & Strom, 1997; Benatallah et al., 2006).

For determining the differences in our models we employ a set theoretic approach, however, in principle any other formal technique could have been used. The changed models are other services, thus we are comparing are business protocols. Business protocols were defined (Section 3.2.2) using automata, thus we compare the sets that hold the states, transitions and messages. As stated in the description of the operators in Chapter 3, we only detect additions and removals; composite operators can be constructed from these basic operators. The implementation of the diff operator is shown below (Algorithm 5.2).

Note that this algorithm is the same algorithm as used for the proof of Theorem 1 in Chapter 3. The change script found by the algorithm has the property: $P_c \xrightarrow{\Delta} P'_c$. Hence, if the change script Δ is applied to the business protocol P_c then the result is P'_c. The subscript c denotes the service provider owning that business protocol. For instance, in our running example c can either be the bank, the inventory or the shipper.

Algorithm 5.2 diff(Business Protocol P_c, Business Protocol P_c')

$\Delta \leftarrow \emptyset$
$\forall s \in S \backslash S' : \Delta \leftarrow \Delta \cup \text{removeState}(s)$
$\forall s \in S' \backslash S : \Delta \leftarrow \Delta \cup \text{addState}(s)$
$\forall (s_i, s_j, m) \in T \backslash T' : \Delta \leftarrow \Delta \cup \text{removeTrans}(s_i, s_j, m)$
$\forall (s_i, s_j, m) \in T' \backslash T : \Delta \leftarrow \Delta \cup \text{addTrans}(s_i, s_j, m)$
$\forall (rec, sen, g) \in M \backslash M' : \Delta \leftarrow \Delta \cup \text{removeMsg}(rec, sen, g)$
$\forall (rec, sen, g) \in M' \backslash M : \Delta \leftarrow \Delta \cup \text{addMsg}(rec, sen, g)$
return Δ

Lemma 12. *Change script Δ is well-formed.*

Proof. Following Definition 5 in Chapter 3, a script is well-formed if both the models P_c and P_c' are well-formed. Well-formedness of the models is satisfied by the assumption that a service provider provides well-formed models. Hence, Δ is well-formed. □

Lemma 13. *Change script Δ is minimal.*

Proof. Following Definition 7 in Chapter 3, a script is minimal if it does not contain any identity transformations. Since for each element of each set we can have at most one operator, it is not possible to create an identity transformation sequence. Hence, Δ is minimal. □

5.3 Applicability

Not every change results in an incompatibility. The reason is that not all changes affect the functionality used by the orchestrator. Ponnekanti & Fox (2004) state that only a small fraction of all changes result in incompatibilities since only a small part of the functionality offered by Web services is used. If a change has impact on the orchestrator we call it applicable.

To see whether the changes captured in change script Δ affect the orchestrator, we determine for each operator o in that script if it is applicable. Because a change to a service provider cannot directly affect the interaction of the orchestrator with other services, we isolate the interaction with that service. To determine applicability we therefore check whether the operator affects the projection of the changed service on the orchestrator $\pi_{P_c}(\mathcal{BP})$. In other words, if the operator affects the message sequence (path) used by

Concept	*Operator o*	*Applicable iff:*
Transition	addTrans(s_f, m, s_t)	$s_f \in S \vee s_t \in S \vee$
		$\exists$addTrans(s_{ix}, m_x, s_{jx}) $\in \Delta_a$
		$[s_f = s_{ix} \vee s_f = s_{jx} \vee s_t = s_{ix} \vee s_t = s_{jx}]$
	removeTrans(s_f, m, s_t)	$< s_f, m, s_t > \in T$
State	addState(s_i)	$\exists$addTrans(s_f, m, s_t) $\in \Delta_a [s_i = s_f \vee s_i = s_t]$
	removeState(s_1)	$s_1 \in S$
Message	addMsg(m_1)	$\exists$addTrans(s_f, m, s_t) $\in \Delta_a [m_i = m]$
	removeMsg(m_1)	$m_1 \in M$
Type	addType(e, d, p)	$\exists m \in M$[valid(p, m, e)] $\vee$
		$\exists$addTrans(s_f, m, s_t) $\in \Delta_a$[valid(p, m, e)]
	removeType(e, p)	$\exists m \in M$[valid(p, m, e)]
Cardinality Const.	updateCC(x', y', e, p)	$\exists m \in M$[valid(p, m, e)] $\vee$
		$\exists$addTrans(s_f, m, s_t) $\in \Delta_a$[valid(p, m, e)]
Structural Const.	updateSC(c^s,p)	$\exists m \in M$[valid(p, m, e)] $\vee$
		$\exists$addTrans(s_f, m, s_t) $\in \Delta_a$[valid(p, m, e)]

Table 5.1: Applicability of operators

the orchestrator, then it is applicable. Note, that if it is assumed that the orchestrator is *fully compatible*(Benatallah et al., 2006), meaning that all executions of P_c can interoperate with $\mathcal{BP}$, thus $\pi_{P_c}(\mathcal{BP}) = P_c$, then every operator, and therefore every change, is applicable.

In general, the addition of an element, such as a message, state or transition, is applicable if the element has a relation with an existing transition, state, message, or if it refers to a transition that has been just added. Removal of an element is applicable if that element exists in the orchestrator. For example, the operator removeMsg(m) is applicable only if that message m is received or sent by the orchestrator. Below we define the function ***applicableOp*** which given an operator o, an applicable change script Δ_a, and the projection of the changed protocol on the business process $\pi_{P_c}(\mathcal{BP})$, determines whether the operator is applicable or not.

Definition 41. *(ApplicableOp)*
The boolean function ***applicableOp***$(o, \Delta_a, \pi_{P_c}(\mathcal{BP}))$ *is defined on an operator as follows:*

$$\textbf{\textit{applicableOp}}(o, \Delta_a, \pi_{P_c}(\mathcal{BP})) := \begin{cases} true & \text{if } o \text{ in Table 5.1 holds;} \\ false & \text{otherwise.} \end{cases}$$

The ***applicableOp*** operator refers to Table 5.1. In this table, for every change operator we give the condition on which that operator is applicable (see column on the far right). For instance, the addition of a transition operator (addTrans) is applicable if one of the states s_f or s_t is part of the protocol ($s_f \in S \vee s_t \in S$), meaning that a new transition is added onto an existing state, or if another transition was already added and this transition continues that new transition path ($\exists \text{addTrans}(s_{ix}, m_x, s_{jx}) \in \Delta_a [s_f = s_{ix} \vee s_f = s_{jx} \vee s_t = s_{ix} \vee s_t = s_{jx}]$).

The ***applicableOp*** operator determines the applicability of one operator. As change scripts in general consist of multiple operators, we therefore define in Algorithm 5.3 an operator for filtering all applicable operators of a script. The output of this algorithm is a script containing only applicable operators. If this script does not contain any operators (is empty), then the change is not applicable to the orchestrator.

Algorithm 5.3 applicable(Change-script Δ, Model $\mathcal{M}$)

```
Δ_a = ∅
for each o ∈ Δ do
  if applicableOp(o, Δ_a, M) then
    Δ_a := Δ_a ∪ o
  end if
end for
return Δ_a
```

Lemma 14. *Change script Δ_a is well-formed.*

Proof. To prove well-formedness (Definition 5), we show that both the originating and resulting model are well-formed, thus given $P \xrightarrow{\Delta_a} P'$, that P and P' are well-formed.

Due to Lemma 5 in Chapter 4, it holds that $\pi_{\mathcal{BP}}(P)$ is well-formed if $\mathcal{BP}$ and P are well-formed. From Lemma 12 it follows that P is well-formed and Lemma 11 in Chapter 4 implies that $\mathcal{BP}$ is well-formed. Thus it also holds that $\pi_{\mathcal{BP}}(P')$ is well-formed. As $\pi_{\mathcal{BP}}(P) \xrightarrow{\Delta_a} \pi_{\mathcal{BP}}(P')$, therefore Δ_a is well-formed. □

Lemma 15. *Change script Δ_a is minimal.*

Proof. To prove minimality (Definition 7), we show that Δ_a does not contain any identity transformation sequence. Δ is minimal (Lemma 13), and since $\Delta_a \subseteq \Delta$, therefore Δ_a does not contain any identity transformation sequences. □

5.4 Adapting

Given an applicable change script Δ_a, the manager needs to determine how to respond to these changes. This decision process, depicted as the adapt operator in Algorithm 5.4, serves two purposes. The first purpose is to determine whether changes can be solved, and the second one is to construct an adequate adaptation script in case changes can be solved.

Algorithm 5.4 adapt(Change-script Δ_a, Orchestrator $< \mathcal{BP}, \mathcal{MP} >$)

```
< BP, MP > --Δa--> < BP', MP >
I, α ← remap(Δa, < BP', MP >)
if I = ∅ then
  return α
else
  α ← reorder(BP, I)
  if I = ∅ then
    return α
  else
    α ← recompose(BP)
    return α
  end if
end if
```

The first line in the algorithm applies the applicable script to the model of the orchestrator. Since the change operators do not affect the mappings, we can safely state that the mappings before applying the script are the same as after applying the script, i.e. $< \mathcal{BP}, \mathcal{MP} > \xrightarrow{\Delta_a} < \mathcal{BP}', \mathcal{MP} >$.

With this altered business process, we search for a set of operators α such that after applying this set the business process is again mapping-complete (see Definition 21 in Chapter 4). The steps of the algorithm, i.e., *remap,*

reorder and *recompose* represent the different functions we employ for finding an adaptation script. We discuss each function in detail in sections 5.4.2, 5.4.3 and 5.4.4 respectively. Before explaining the implementation of these operators we define what can be solved in general and what we define as an incompatibility.

5.4.1 Change Categories and Incompatibilities

In this section, we discuss a categorization of changes, making a distinction between non-effective, solvable, and problematic changes. After this, we define incompatibilities. The definitions given in this section use the definitions of mapping-completeness, data-consistency and provided discussed in Section 4.3 of Chapter 4.

Change Categories

To see whether a manager can adapt a business process to changes, we define what can be handled. Given the orchestrator $\mathcal{O} =< \mathcal{BP}', \mathcal{MP} >$, we define the following categories of change:

1. **Non-effective:** If *m-complete*$(\mathcal{BP}', \mathcal{MP})$, then no adaptation necessary.

   Proof. Assume *m-complete*$(\mathcal{BP}', \mathcal{MP})$ and adaptation script σ to be empty. Then, with $\mathcal{MP} \xrightarrow{\sigma} \mathcal{MP}'$, it holds that: $\mathcal{MP} \equiv \mathcal{MP}'$, thus *m-complete*$(\mathcal{BP}', \mathcal{MP}')$. □

2. **Solvable:** If *d-consistent*$(\mathcal{BP}')$ and not *m-complete*$(\mathcal{BP}', \mathcal{MP})$, then $\exists\sigma[\mathcal{MP} \xrightarrow{\sigma} \mathcal{MP}' \wedge$ *m-complete*$(\mathcal{BP}', \mathcal{MP}')]$.

   Proof. From the definition of data consistency and mapping completeness it follows that *d-consistent*$(\mathcal{BP}')$ but not *m-complete*$(\mathcal{BP}', \mathcal{MP})$ means: $\exists m \in M[$*provided*$(m) \wedge \neg$*m-sufficient*$(\mathcal{MP}, m)]$. Hence, after applying σ, a valid mapping exists such that:

   $$(m_j, p, e\{x, y\}) \xleftarrow{u} (m_i, p', e'\{k, l\}).$$

   □

3. **Problematic:** If not *d-consistent*($\mathcal{BP}'$), then no possible mapping(s) can be found.

   Proof. Observe that $\neg$*d-consistent*($\mathcal{BP}'$) can be rewritten as

   $$\begin{aligned}&\exists! m \in M\ \exists e\{x,y\} \in g(m)\\ &\quad \exists < m_0, .., ?m_i, .., !m_j > \in MEX^j\\ &\quad\quad \forall e'\{k,l\} \in \{g(?m_0), ..., g(?m_i) : i < j\}\\ &\quad\quad\quad [x > k \vee e \not\simeq e'].\end{aligned}$$

   Therefore, because of Definition 15, no mapping is possible. □

The three categories above define the three types of action that the manager will perform when encountering changes. The first category, **Non-effective**, states that nothing needs to be done such that the orchestration remains compatible. The changes in this category do not affect the interoperability. The second category, **Solvable**, states that the changes affect the business process of the orchestrator but that mappings can be found such that the orchestrator is compatible again. The third category, **Problematic**, states that a change cannot be solved and is therefore a problem. In this situation the manager escalates, i.e. sends a message to a human administrator stating that the orchestrator will not be compatible and that human intervention is required.

Lemma 16.

$$\forall o \in \Delta[\neg \textit{applicable}(o)] \Rightarrow \textit{non-effective}(\Delta)$$

Proof. If all operators in the change-script are not applicable, then there does not exist an operator that affects the orchestrator. Thus Δ_a is empty. Thus, after applying Δ_a, the orchestrator is still *m-complete*($\mathcal{O}$), see proof **Non-effective**. □

Note that if all operators are not applicable, this implies non-effective. The other way around does not hold. If a script falls under the category of non-effective, this does not imply that all operators are not applicable. There may exist operators in that script that affect the protocol but do not cause any incompatibilities. For example, the addition of an element to the type of an incoming message of the orchestrator can be applicable, however this change falls under the non-effective category, as is shown in Section 5.4.2.

Incompatibilities

Not all changes create problems. Problems concerning interoperability are incompatibilities. We define an incompatibility as:

Definition 42. *(Incompatibility)*
An incompatibility exists iff

$$\exists !m \in M[\neg m\text{-}\mathit{sufficient}(!m)]$$

Intuitively an incompatibility is everything that causes the service not to be mapping complete. An incompatibility is caused if for a message a mapping is not valid or missing. Changes can cause multiple incompatibilities, therefore, in the following we often use the set of incompatibilities (defined below).

Definition 43. *(Set of incompatibilities)*
The set of incompatibilities $\mathcal{I}$ is defined as

$$\{!m : \neg m\text{-}\mathit{sufficient}(!m)\}$$

5.4.2 Remapping

If changes affect the orchestrator, i.e., they are applicable, then the manager will search for an adaptation script. Remapping is a method for altering the mappings without affecting the structure of the orchestrator. Remapping performs two tasks, it looks whether there are any incompatibilities, and in case there are, it determines whether they can be solved by looking for alternative mappings.

Algorithm 5.5 presents the implementation of the remap operator. First the projection of the change business protocol on the business process is constructed. We use the projection because only messages in that construct may have changed. In the for-loop we go through every transition of the projection and determine whether the message of that transition has been changed. When the data is still provided, we apply the solution as shown in Table 5.2. If data is not provided, this means that there is an incompatibility that can not be solved by remapping.

Theorem 6. *The remap operator solves all incompatibilities iff the change is solvable.*

Algorithm 5.5 remap(Change-script Δ_a, Orchestrator $< \mathcal{BP}', \mathcal{MP} >$)

$\mathcal{BP}_{proj} \leftarrow \pi_{P_c}(\mathcal{BP}')$
$\mathcal{I} \leftarrow \emptyset$
for each $< s_f, m, g, s_t > \in tex_{\mathcal{BP}_{proj}}$ **do**
  if *provided*(m) **then**
    for each $o \in \pi_m(\Delta)$ **do**
      $\alpha_{rem} += a$ as in Table 5.2
    end for
  else
    $\mathcal{I} = \mathcal{I} \cup m$
  end if
end for
return $\mathcal{I}, \alpha_{rem}$

Proof. From the proof of the solvable category, we get that there exists an adaptation script if data is provided. For every operator in the change-script, an adaptation operator can be found as in Table 5.2. To proof the correctness of a script, we need to proof the correctness of every operator. Here we provide the proof of two operators, namely move and remove. For the remaining operators the proof can be constructed following the same line of reasoning. The changes in the proof occur in the context of a message m.
move(e, p_f, p_t):
input:
$\exists(m, p_f, e) \leftarrow (m_j, p_j, e) \in \mathcal{MP}[\neg valid(m, p_f, e)] \Rightarrow \neg$m-complete$(\mathcal{BP}', \mathcal{MP})$.
d-consistent$(\mathcal{BP}')$ (cat 2): $a = updateLHS(m, p_t, e)$.
output:
$\neg\exists(m_l, p_l, e_l) \leftarrow (m_r, p_r, e_r)[m_l = m \wedge p_l = p_f \wedge e_l = e]$
$\Rightarrow$m-complete$(\mathcal{BP}', \mathcal{MP})$ (cat 1), else d-consistent$(\mathcal{BP}')$
(cat 2): $a = updateRHS(m, p_t, e)$.

remove(e, p):
input: $\exists(m, p, e) \leftarrow (m_j, p_j, e) \in \mathcal{MP}\ [e \notin g(m)] \Rightarrow \neg$m-complete$(\mathcal{BP}', \mathcal{MP})$.
Still d-consistent$(\mathcal{BP}')$ (cat 2): a $= removeMap((m, p, e) \leftarrow (m_j, p_j, e))$
output: If $\neg\exists(m_l, p_l, e_l) \leftarrow (m_r, p_r, e_r) \in \mathcal{MP}[e_r = e]$
$\Rightarrow$ m-complete$(\mathcal{BP}', \mathcal{MP})$ (cat. 1), else if $provided(e, m) \Rightarrow$ d-consistent$(\mathcal{BP}')$
(cat 2): $a = updateRHS(m_j, p_j, e)$,
else (cat 3). □

Operator	*Solution*	
	Input	*Output*
addType(e, d, p)	$addMap((m,p,e) \leftarrow (m_j,p_j,e))$	-
removeType(e,p)	$removeMap(m,p,e)$	$updateRHS(m_j,p_j,e)$
moveType(e,p_f,p_t)	$updateLHS(m,p_t,e)$	$updateRHS(m,p_t,e)$
updateType(d, p)	$updateLHS(m,p,e)$	$updateRHS(m,p,e)$
updateCC(x',y',e,p)	$updateCard(k,y')$	$updateCard(x',l)$
updateSC(opt, p)	$removeMap(m,p,e)$	-
updateSC(seq, p)	$addMap((m,p,e) \leftarrow (m,p,e))$	-
addMsg(m)	$addMap((m,p,e) \leftarrow (m,p,e))$	-
removeMsg(m)	$removeMap(m,p,e)$	$updateRHS(m_j,p_j,e)$ [a]
moveMsg(m,t_i,t_j)	$updateLHS(m,p,e)$	$updateRHS(m,p,e)$ [a]

[a] $foreach\ e \in g(m)$

Table 5.2: Solutions for Change Operators and mismatches

Theorem 6 states that the remap operator solves all incompatibilities unless the data (type) is not provided. Therefore it is interesting to look in which situations data is provided and in which situations it is not. In Table 5.3 a categorization is shown for all operators and mismatches. The categories 1, 2 and 3, stand for non-effective, solvable and problematic, respectively. This categorization shows which operators can be solved by remapping and which can not. For the situations where the operator falls under the solvable category (category 2), the solution for that operator in Table 5.2 can be used.

Concept	*Operator*	*Category*	
		Input	*Output*
Type	addType(e, d, p)	2/3	1
	removeType(e, p)	2	1/2/3
	moveType(e, p_f, p_t)	2	2
	updateType(d, p)	2/3	1/2/3
Constraints	updateCC(x', y', e, p)	1/2/3	1/2/3
	updateSC(opt, p)	1/2	1
	updateSC(seq, p)	2/3	1
Message	addMsg(m)	2/3	1
	removeMsg(m)	2	2/3
	moveMsg(m)	2/3	1/2/3

Table 5.3: Categorization of Change Operators and mismatches

Lemma 17. *Adaptation script α_{rem} is well-formed.*

Proof. Well-formedness of the models is guaranteed as the script creates a mapping-complete orchestrator (Theorem 6). □

Lemma 18. *Adaptation script α_{rem} is minimal.*

Proof. α_{rem} cannot contain an identity transformation since (1) Δ_a does not contain an identity transformation(Lemma 15) and (2) the remap operator creates only one adaptation operator per change operator. □

To illustrate the process of remapping, we work out two examples below.

Examples: Message Merge and Message Split

Consider the following example: The bank changes its protocol from a two message interaction, Figure 5.1(a), to a four message interaction, Figure 5.1(b). The difference is that in the short version the name of the accountholder was asked together with the rest of the information, whereas in the long version the bank requests this information separately. The change from (a) to (b) is called Message Split and from (b) to (a) Message Merge (Benatallah et al., 2005). The manager receives the notification of this change and detects the changes, with the change-script for Merge (Figure 5.2) and Split (Figure 5.3) as a result.

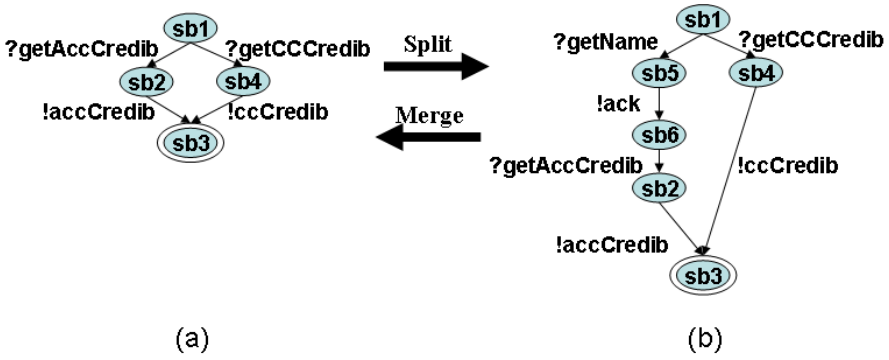


Figure 5.1: Change Example: (a) Bank short version (b) Bank long version

As can be seen in Figure 5.1, all operators in the change-script(s) are applicable, thus $\Delta_a = \Delta$. After filtering for applicability, the manager checks whether the messages of the used transitions are still provided. In the cases above no data was removed, only the name of the accountholder was moved, and thus we can conclude that the data is still provided. The mismatches are therefore solvable by using remapping. The adaptation-script for solving the Message Merge is shown in Figure 5.4, for Message Split in Figure 5.5.

5.4.3 Reordering

For the operators in Table 5.3 where a category 3 (problematic) is indicated, there exists a situation in which the data is needed but not provided:

$$\exists m \in \mathcal{I}[\neg \text{provided}(m)].$$

However, not provided means that in the *current* structure of the orchestrator that message is not provided. Because by remapping we do not alter the structure of the orchestrator, this does not mean that interoperability can not be maintained. By looking at alternative valid structures, it is still possible to solve incompatibilities. We use two functions to search for alternatives, either by reordering, discussed in this section, or by recomposition, described in the next section.

In orchestrations where protocols are specified bilateral, an innate flexibility exists as there are no direct relations between messages of different services. This flexibility allows for alternative orchestrators, similarly to what was discussed in the previous chapter. Reordering exploits this flexibility by

```
removeTrans(sb1,getName,sb5);
removeTrans(sb5,ack,sb6);
removeTrans(sb6,getAccCredib,sb2);
addTrans(sb1,getAccCredib,sb2);
removeState(sb5);
removeState(sb6);
removeMsg(Bank,Retailer,getName);
removeMsg(Retailer,Bank,ack);
removeType(getName,null);
removeType(ack,null);
addType(name,string,/GetAccCredib);
```

Figure 5.2: Change script Merge

```
removeTrans(sb1,getAccCredib,sb2);
addTrans(sb1,getName,sb5);
addTrans(sb5,ack,sb6);
addTrans(sb6,getAccCredib,sb2);
addState(sb5);
addState(sb6);
addMsg(Bank,Retailer,getName);
addMsg(Retailer,Bank,ack);
addType(getName,,/);
addType(name,string,/getName);
addType(ack,,/);
addType(void,string,/ack);
removeType(name,/GetAccCredib);
```

Figure 5.3: Change script Split

```
addMap(t_b1,/getAccCredib/name,/order/customer/name);
```

Figure 5.4: Adaptation script Merge

looking whether messages can be shuffled in such a way that incompatibilities can be solved.

To determine whether messages can be reordered, we check whether messages are provided at different points in the business process (protocol). Therefore we extend the definition of *provided* given in Section 4.3 of Chapter 4.

Definition 44. *(Provided$_i$)*
Let $< m_1, .., m_n >$ be a message sequence. With provided$_i(m)$, where $i \in \{1, .., n\}$, we denote that after the i'th message the data of the message m is provided.

The difference with the definition of *provided* is that *provided$_i$* looks at message sequences without regarding the message's position. In *provided* the position of the message is fixed in the sequence, however, when looking if we can change this order, we look whether the data of the message is provided at different positions.

```
addMap(t_b3,/getName/name,/order/customer/name);
removeMap(t_b1,/getAccCredib/name);
```

Figure 5.5: Adaptation script Split

Lemma 19. *$provided_i(m) \Rightarrow provided_{i+1}(m)$.*

The proof for this Lemma follows directly from our non-monotonicity assumption on data. This assumption implies that data values will not change, therefore if something is provided at one point, it will always be provided.

Definition 45. *(Can be reordered)*
Let $m_j^{reo} \in M_{P_r}$ be the message of service P_r that is considered for reordering, then m_j^{reo} can be reordered if the following properties hold:

(1) $\exists x \in \{1, .., z\} \forall < m_0, .., m_z > \in MEX[provided_x(m^{reo})]$

(2) $\forall m_l \in \pi_{P_r}(< m_{j+1}, .., m_x >)[provided_{x+l}(m_l)]$

(3) $\forall m_l \in \pi_{P_i}(< m_{j+1}, .., m_x >)[provided_j(m_l) \wedge i \neq r]$

Intuitively, the three properties in Definition 45 state that (1) there is a point further on the path such that at that point the message is provided, (2) all the messages in between of the same service can be placed behind that point as well, and (3) for all other messages of other services the messages should be provided earlier.

For creating a script for reordering, we look at the sequence that has to be moved.

Definition 46. *(Reorder script)*
Let $< t_0, .., t_i, .., t_j, .., t_x, .., t_z >$ be a valid transition sequence of the orchestrator and let $t_i =< s_1, m^{reo}, s_2 >$ and $\pi_{P_r}(< t_i, .., t_j, .., t_x >) =< t_i, .., t_j >$ be the projection of the service on the sequence of messages. Then a script can be created as follows:
$\alpha_{reo} \leftarrow \alpha_{reo} \cup removeTransition(s_1, m_{j+1}, s_2)$
$\alpha_{reo} \leftarrow \alpha_{reo} \cup addTrans(s_{1_{t_i}}, m_{j+1}, s_2)$
$\alpha_{reo} \leftarrow \alpha_{reo} \cup removeTrans(s_1, m_i, s_2)$
$\alpha_{reo} \leftarrow \alpha_{reo} \cup addTrans(s_{2_{t_x}}, m_i, s_2)$

$\alpha_{reo} \leftarrow \alpha_{reo} \cup removeTrans(s_1, m_{x+1}, s_2)$
$\alpha_{reo} \leftarrow \alpha_{reo} \cup addTrans(s_{1_{t_i}}, m_{x+1}, s_2)$.

The first two lines link the messages after the sequence to be reordered to the origin of that sequence. The second two lines put the reorder sequence behind the providing transition t_x and the last two lines link everything behind that transition to the end of the reorder sequence. Note that after transition t_x all messages are still provided as reordering does not remove any messages from the sequence.

Algorithm 5.6 reorder(Business Process $\mathcal{BP}$, Incompatibilities $\mathcal{I}$)

```
solved ← ∅
for each m ∈ I do
  if m can be reordered (Definition 45 ) then
    α_reo ← see Definition 46
    solved ← solved ∪ m
  end if
end for
if I = solved then
  return α_reo
end if
```

Lemma 20. *Adaptation script α_{reo} is well-formed.*

Proof. To prove well-formedness (Definition 5), we prove both $\mathcal{BP}$ and $\mathcal{BP}'$ are well-formed. Well-formedness of the script follows from the well-formedness of the models if the script is applied, thus $\mathcal{BP} \overset{\alpha_{reo}}{\rightarrow} \mathcal{BP}'$. $\mathcal{BP}$ is well-formed (Lemma 14). For BP' we show that the new sequences created are also valid. Let $< t_0, .., t_{i-1}, t_i, .., t_j, t_{j+1}, .., t_x, t_{x+1}, .., t_z >$ be a valid sequence, then $< t_0, .., t_{i-1}, t_{j+1}, .., t_x, t_i, .., t_j, t_{x+1}, .., t_z >$ is the sequence after reordering. First, observe that the sequence before t_i, thus $t_0, .., t_{i-1}$, remains the same, and is therefore provided. The same holds for the messages after t_x, thus $t_{x+1}.., t_z$. That $t_i, .., t_j$ is provided after t_x is guaranteed by Definition 45, Property (1) and (2), and that $t_{j+1}, .., t_x$ is provided after t_{i-1} is guaranteed by Definition 45 Property (3). Therefore, Lemma 5 from the previous chapter holds for BP', thus $\mathcal{BP}'$ is well-formed. □

Lemma 21. *Adaptation script α_{reo} is minimal.*

Proof. To prove minimality (Definition 7), we prove that α_{reo} does not contain an identity transformation sequence. An identity transformation can occur in the script for moving a single message or in a script when multiple messages are moved. From Definition 46 we can see that moving a single sequence (message) does not contain any identity transformation sequence. That a script containing the operators for moving multiple message does not contain a identity transformation sequence, we prove by contradiction. Assume that adaptation script α does contain a identity transformation sequence. This means that a message is moved further in a path and then back to its original place. If this message is moved this means that it was not provided at its original position, $\neg provided_i(m)$. However, it is then moved back and this implies $provided_i(m)$. Hence, α_{reo} cannot contain any identity transformation sequence. □

When trying to reorder, we are interested in what problems can be solved by shuffling the order of the messages. To analyze this, we look at the set of alternative orchestrators which use the same set of messages. An alternative that uses the same set of messages, we call a message equivalent alternative. We define the set of message equivalent alternatives as follows:

Definition 47. *(Set of message equivalent alternatives)*
Let $\mathcal{S}'_{red} =< (\mathcal{P}, M), \mathcal{B}, p_d, p_c, p_1, .., p_{n-1} >$ be the new composite service incorporating the changed business protocol with as language $MEX_{\mathcal{S}'_{red}}$. The set of message equivalent alternatives $MEX^*_{\mathcal{BP}}$ is defined as:

$$MEX^*_{\mathcal{BP}} = \{MEX : \ \forall < m_1, .., m_z > \in MEX \\ \forall m \in < m_1, .., m_z > [m \in M_{\mathcal{BP}}] \ , \\ MEX \subseteq MEX_{\mathcal{S}'_{red}}\}.$$

Intuitively, the Definition 47 describes the set of alternative valid orchestrators. Every orchestrator that uses the same set of messages as the current (temporary) orchestrator is considered an alternative.

Theorem 7. *Reordering solves all incompatibilities iff*

$$\exists MEX \in MEX^*_{\mathcal{BP}}[\textit{d-consistent}(\mathcal{BP})].$$

Proof. The statement "reordering solves all incompatibilities" implies that there exists a non-empty set of incompatibilities $\mathcal{I} \neq \emptyset$. This implies not *m-complete*($\mathcal{BP}$), and that the reorder operator creates an adaptation script such that after applying it $\mathcal{BP} \overset{\alpha_{reo}}{\rightarrow} \mathcal{BP}'$, that $\mathcal{BP}'$ is mapping-complete. Since Theorem 6 shows that remapping can be employed for any data consistent business process, it suffices to show that after $\mathcal{BP} \overset{\alpha_{reo}}{\rightarrow} \mathcal{BP}'$, $\mathcal{BP}'$ is data-consistent.
$\Leftarrow$: Assume $\exists MEX \in MEX^*_{\mathcal{BP}'}[$*d-consistent*$(\mathcal{BP}')]$. We show that reordering will find that MEX. We distinguish two cases: $\mathcal{I} = \emptyset$ or $\mathcal{I} \neq \emptyset$. First, $\mathcal{I} = \emptyset$, then *d-consistent*($\mathcal{BP}'$) and the reorder operator returns an empty adaptation script. Second, $\mathcal{I} \neq \emptyset$, because there exists a MEX such that *d-consistent*($\mathcal{BP}$), it holds $\forall m \in \mathcal{I}[provided_i(m)]$ (property 1 of Definition 45). Since it holds that $\mathcal{I} \neq \emptyset$, this implies that $\exists m \in \mathcal{I}[\neg provided(m)]$, however, since *d-consistent*($\mathcal{BP}$) holds, there must exist a MEX such that messages are reordered and implying $\forall m \in \mathcal{I}[provided_i(m)]$. From Lemma 19 it follows that other messages can be moved to the front (Definition 45 property (2)). Definition 45 property (3) is satisfied by Lemma 19, and hence $\forall m \in \mathcal{I}$ the messages can be reordered.
The reverse "$\Rightarrow$" follows directly from Lemma 20. □

Example: Additional Inventory Constraints

Consider the orchestrator for the retailer constructed in Chapter 4 (displayed in Figure 4.8). In this orchestrator, we first check the availability of the products, then request a shipment quote, and last check the credibility of the customer, see Figure 5.9(a). However, imagine that the people behind the inventory service have often prepared products for shipment but in some instances they were not picked up. For this reason, they updated the service with an additional constraint on the checkAvailability message, as illustrated in Figure 5.6. This constraint entails that a shipmentID must be sent along with the request for availability, which is an effort to guarantee that the products are picked up.

The change script for this change only contains one operator, namely the addition of type shipmentID, see Figure 5.7. This change results in an incompatibility which can not be resolved using remapping since the shipmentID is not provided at the time it is needed. Reordering looks whether this shipmentID is provided at a later point. In this case, it is provided after

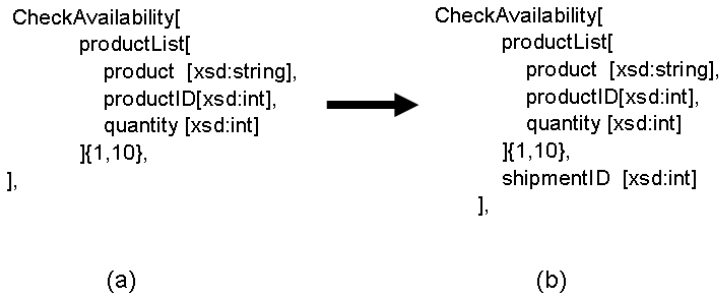


Figure 5.6: CheckAvailability message (a) original and (b) with shipmentID

the orchestrator receives the ShipQuote message.

```
addType(shipmentID,integer,/CheckAvailability);
```

Figure 5.7: Change script Additional Constraints

The adaptation script for solving this incompatibility is shown in Figure 5.8. In this script, the first two lines connect the shipment messages to the first point after the order is received. The second two lines, connect the communication of the inventory to the last message of the shipper and the third two lines (5 and 6) connect the messages after that to the inventory. The last line adds the mapping to the shipmentID. The resulting orchestrator, after applying this script, is shown in Figure 5.9(b). In this figure the order of messages has been changed such that the availability request to the inventory is placed after the communication with the shipper.

```
removeTrans(si5, getShipmentQuote, sh2);
addTrans(s4,getShipmentQuote ,sh2);
removeTrans(s4,checkAvailability,si4);
addTrans(sh3,checkAvailability,si4);
removeTrans(sh3,getAccCredib,sb2);
addTrans(si5,getAccCredib,sh2);
addMap(/CheckAvailability/shipmentID,/ShipmentQuote/shipmentID);
```

Figure 5.8: Adaptation script Additional Constraints

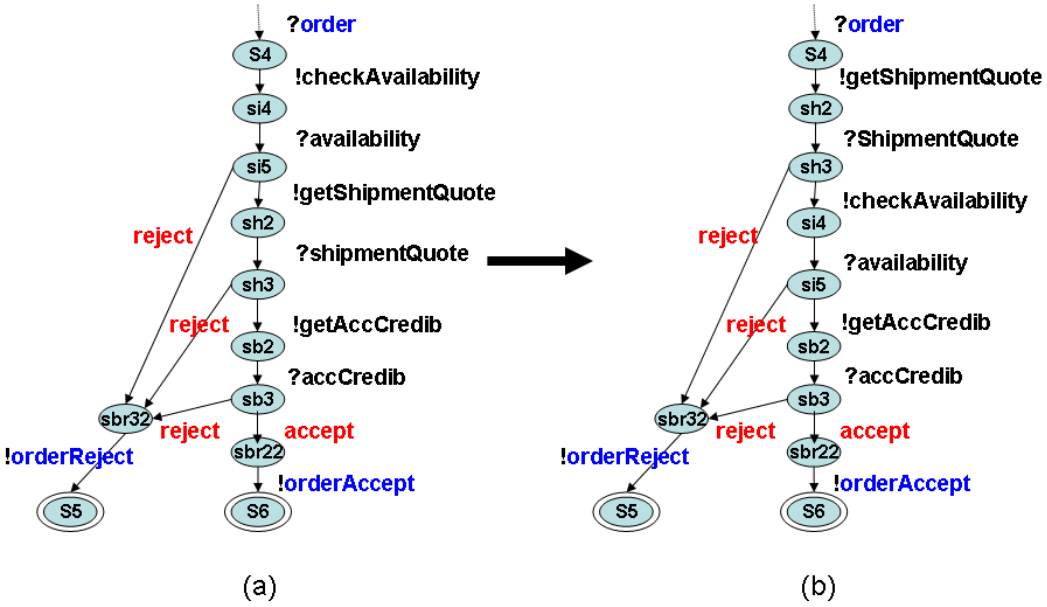


Figure 5.9: Orchestrator (a) before and (b) after adapting

5.4.4 Recomposition

Reordering solves incompatibilities by using the flexibility of the business process. However, if data is not provided at a later point, then reordering can not solve this. The last step in trying to solve this kind of incompatibility is through recomposition. Recomposition looks whether the missing information is provided in paths that were not considered in the initial composition. Because changes are introduced, other message paths may become feasible alternatives. Recomposition, as the name implies, is based on the composition approach discussed in Chapter 4.

Algorithm 5.7 recompose(Orchestrator $\mathcal{O}$, Services $\mathcal{P}$)

$\mathcal{O}' \leftarrow compose(\mathcal{P}, B, P_d, \theta)$
if $\mathcal{O}' = \emptyset$ **then**
  return error /* can not be solved */
end if
$\alpha_{com} \leftarrow diff(\mathcal{O}', \mathcal{O})$
return α_{com}

Algorithm 5.7 shows the implementation of the recompose operator. The

first line initiates the compose operator (discussed in Chapter 4). The main difference between composition and recomposition is the *diff* function. The compose operator generates a new orchestrator, however as the manageable service (the orchestrator) should be updated by applying a script of operators (see Chapter 2), we obtain this script by using the *diff* function.

Lemma 22. *Adaptation script α_{com} obtained by recomposing is well-formed.*

The proof of this Lemma follows directly from Theorem 5 of the previous chapter and Lemma 12.

Lemma 23. *Adaptation script α_{com} obtained by recomposing is minimal.*

The proof of this Lemma follows directly from Theorem 5 of the previous chapter and Lemma 13.

Theorem 8. *Let $\mathcal{S}' =< (\mathcal{P}, M), \mathcal{B}, p_d, p_c, p_1, .., p_{n-1} >$ be the new composite service with as language $MEX_{S'_{red}}$. Recomposition solves all incompatibilities iff*

$$\exists MEX[MEX \subseteq MEX_{S'_{red}}].$$

The proof of this theorem follows directly from Theorem 2 in Chapter 4.

Example: Bank removes functionality

The bank in our running example used to offer support for checking the credibility of bankaccounts of its customers. However, imagine that due to some reason, for instance a security leak, this functionality is (temporarily) removed. The new and old business protocol of the bank can be seen in Figure 5.10.

The manager of the retailer receives this notification, thereby initiating the adaptation process. The manager determines that these changes are applicable and furthermore that they cannot be solved by either remapping or reordering. Recomposition solves these problems in two cases, namely if there is another path that provides the data through another party or if the path containing the incompatibility can be avoided. In this example, it is possible to avoid using the message path for bankaccounts by using the alternative path for creditcards.

To migrate from the old version to the new version the manager creates an adaptation script by executing the diff-function on the new and old

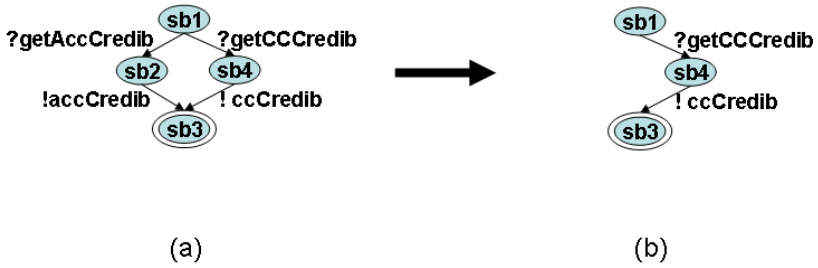


Figure 5.10: Change Example: (a) Bank with and (b) without bankaccount support

orchestrator. The adaptation script for handling this situation is given in Figure 5.11. This adaptation script contains all operators to remove the usage of the bankaccount and add the support of the creditcard at the same point.

The reverse situation, where a path is added, has like the addition of an incoming message no direct effect on the orchestrator (non-effective).

5.5 Prototype

The algorithms described in this and the previous chapter were implemented in a prototype. The prototype was used to generate all change and adaptation scripts presented in this thesis. This prototype serves as a proof-of-concept implementation of a manager for tackling interoperability problems. Through this implementation we show that our framework can be used to develop model-based adaptive behavior in services.

The prototype contains two functionalities, namely composition and adaptation. The architecture for service composition is illustrated in Figure 4.2 and described in Algorithm 3. The architecture for the second functionality, adaptation of an orchestrator, is shown in Figure 5.12. In Figure adaptation is illustrated with references to the related algorithms. Given a changed service, the prototype creates the change script as described in Algorithm 6, performs an impact analysis (shown in Algorithm 7), and based on the changes that have an impact, and attempts to construct an adaptation script by remapping (Algorithm 9), reordering (Algorithm 10) and recomposition (Algorithm 11).

```
removeTrans(sh3,getAccCredib,sb4);
removeTrans(sb4,accCredib,sb3);
addTrans(sb1,getCCCredib,sb2);
addTrans(sb2,ccCredib,sb3);
removeState(sb2);
addState(sb4);
removeMsg(Retailer,Bank,getAccCredib);
removeMsg(Bank,Retailer,accCredib);
addMsg(Retailer,Bank,getCCCredib);
addMsg(Bank,Retailer,ccCredib);
removeType(GetAccCredib,);
removeType(AccCredib,);
addType(GetCCCredib,,);
addType(creditcard,,/GetCCCredib);
addType(number,int,/GetCCCredib/creditcard);
addType(expiryMonth,int,/GetCCCredib/creditcard);
addType(CCCredib,,);
addType(balance,int,/CCCredib);
removeMap(/GetAccCredib/bankaccount/accountID,/order/payment/
bankaccount/accountID);
removeMap(/GetAccCredib/bankaccount/accountType,/order/payment/
bankaccount/accountType);
addMap(/GetCCCredib/creditcard/number,/order/payment/creditcard/
number);
addMap(/GetCCCredib/creditcard/expiryMonth,/order/payment/creditcard/
expiryMonth);
```

Figure 5.11: Adaptation Script for replacing the bank's path

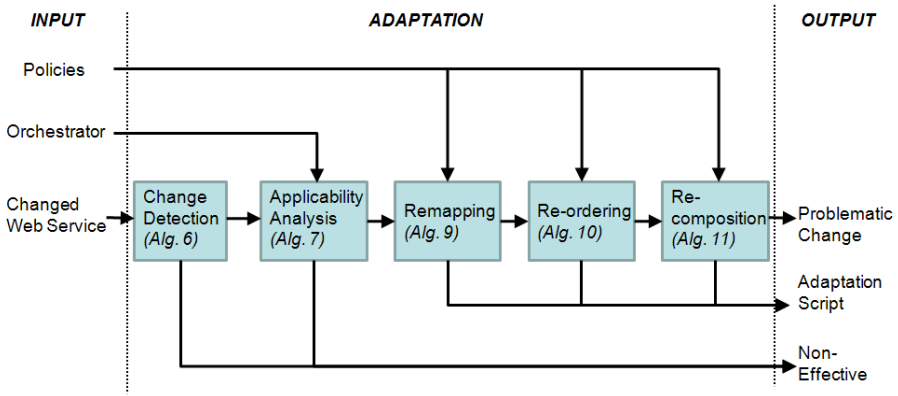


Figure 5.12: Architecture

The prototype is built on top of a extended version of the Web Service Analysis Toolkit (WSAT) by Fu et al. (2004e). This toolkit provides the means to analyze composite Web services by translating BPEL via Guarded Automata into Promela, the language for the model checker SPIN (Holzmann, 2004). For our purposes we used and extended the functionality the authors implemented for Guarded Finite State Automata, as well as the properties they defined for supporting asynchronous communication semantics.

A distinction is made between a bottom-up and a top-down specification. Top down means that an orchestration is specified as one business process where all services are integrated. On the other hand, the bottom up perspective specifies an orchestration as a collection of services. In our work we study changes in services and how these affect an orchestrator, therefore we prefer the bottom-up approach. The grammar in Figure 5.13 specifies the bottom-up perspective, the top-down perspective is given in (Fu, 2004). In Figure 5.13 the italic words are grammar constructs (terminals or non-terminals), the non-italic words represent text used in the specification files.

The syntax defined by the grammar in Figure 5.13 was used to define all input: services (Appendix A), business rules and target protocol (Appendix C) as well as for the output namely the newly composed or adapted orchestrator (Appendix D).

Spec → { *Schema* , *Peerset* }
Schema → Schema{ PeerList{ *StringList* }, TypeList{ *MSLList* }, MessageList{ *MessageList* }}
MessageList → *Message* | *Message* , *MessageList*
Message → *name* { *source* –> *destination* : *type* }
Peerset → *Peer* | *Peer* , *Peerset*
Peer → *name* { States{ *StringList* } , InitialState{ *StringList* }, FinalStates{ *StringList* } , TransitionRelation{ *TransitionList* }}
TransitionList → *Transition* | *Transition* , *TransitionList*
Transition → *name* { *source* –> *destination* : *message* , *Guard* }
Guard → Guard{ *XPathExp* => *Update* }
Update → *name* { *AssignList* }
AssignList → Assign | Assign , AssignList
Assign → XPathExp := XPathExp

Figure 5.13: Syntax of Service Orchestrations

5.6 Discussion

In this chapter, we described our approach for dealing with changes in business protocols of service providers. We showed how changes are captured in a change script, how to determine whether changes are applicable, and we introduced three operators, to (semi-)automatically solve any solvable change.

The three operators utilize all the flexibility that an orchestration may have. The remap operator identifies incompatibilities and solves those incompatibilities if data is provided in the same structure. The reordering operator determines whether incompatibilities can be solved by looking at alternative orchestrators that use the same set of messages (functionalities), and provides an adaptation script for those that can be solved. The recom-

position operator solves incompatibilities that require alternative paths and alternative service providers in the orchestration.

Although in this chapter we only adapted the orchestrator to changes in versions of the same service provider, our approach, without modification, can also be applied when a service is replaced by another. Instead of comparing versions, a new service is given as input and the result is an adaptation script that indicates whether the orchestrator can be adapted such that it uses this new service.

A limitation of our approach is that we tackle one change at a time. In other words, if a queue of change notifications would exists then our approach would try to tackle them one at a time. However in some cases it may require tackling multiple notifications at the same time to solve all incompatibilities. For instance, if a new legislation rule (such as the Sarbanes-Oxley Act (2002)) applies to multiple providers at once, then they may require additional messages that other providers deliver. However, since our approach tackles only one change a time, it will not find a solution in this situation.

Note that although we employed a formal approach, we did not provide complexity results. It is our goal to show that our approach works and is correct, and not to illustrate how much computation is needed to realize it. Our work shows that adaptation works correctly, future work will have the task of making it more efficient.

Chapter 6

Related Work

6.1 Introduction

In this thesis, we study service adaptation. We split related work into three categories, namely work on service adaptation in general, conceptual work (ASOA), and applied work on interoperability. Table 6.1 maps the distinguished fields of knowledge described in this chapter to these three categories. Furthermore, Table 6.1 describes which sections of this chapter are most relevant to which other chapter(s) in this thesis.

6.2 Service Management

Web services used to realize a business process may be provided by several, external parties. This implies that management of Web services goes beyond traditional system management, in that it must be able to deal with changes

Category	*Related Work*	*Section*	*Relevant Chapter(s)*
Conceptual	Service Management	6.2	2
	Model Management	6.3	2 & 3
Interoperability	Service Interoperability	6.4	3,4 & 5
	Service Composition	6.5	4 & 5
Adaptation	Workflow Evolution	6.6	5
	Service Adaptation	6.7	2 & 5

Table 6.1: Categorization of related work

originating outside their domain (hence, in external Web services). To ensure that these Web services perform adequately, Web service management is needed. A survey on Web service management is provided by Papazoglou & van den Heuvel (2005).

To guarantee interoperability between management solutions, a dedicated task force in OASIS defined the Web Services Distributed Management (WSDM) specification. WSDM consists of two orthogonal parts, namely management of Web services (MOWS) (OASIS, 2004) and management using Web services (MUWS) (Bullard & Vambenepe, 2006). MUWS addresses management of IT resources by defining a set of manageability interfaces, while MOWS may be perceived as an application of MUWS focusing on the management of the Web services themselves. For a similar purpose the WS-Management standard was created, however it lacks an explicit concept of relationships needed for inter-organizational changes. In our work, we use concepts and ideas from both WSDM and WS-Management in order to realize the ASOA.

An approach that studies the relation between Autonomic Computing (AC) and WSDM is presented in (Martin et al., 2007). In this paper the authors re-implement a prototype system called Autonomic Web Services Environment (AWSE)(Tian et al., 2005) following the WSDM standard. This comparison provides useful insights in the practical and architectural problems that occur when implementing WSDM. However, in AWSE the authors make a distinction between a performance interface and a goal interface. The performance interface exposes methods for the meta-data and the goal provides methods to query. However, what a goal constitutes is not defined. In our work, we define the goal to be a property of a model, which may be performance.

Another framework concerning Web services drawing inspiration from Autonomic Computing is PAWS (Ardagna et al., 2007). This framework for flexible an adaptive execution of managed Web service-based business processes covers both design-time and run-time aspects of adaptive services. The presented case study is based only on Quality-of-Service and no other aspects of software are described.

Casati et al. (2003) view Web service management from a business perspective. In this paper, the authors advocate a holistic approach which includes business protocols as well as metrics such as Quality-of-Service. However, the approach described remains conceptual and no implementation or

validation results are shown.

In (Cox & Kreger, 2005) the authors describe requirements of managing services within the context of the whole life cycle of a service. The authors make a distinction between development, testing and production requirements and in addition distinguish layers such as the business process layer, the service layer and the IT infrastructure. As stated this work provides only requirements.

A conceptual management framework that is aimed at service evolution is the Service Evolution Management Framework (SEMF) by Treiber et al. (2008b,a). The framework is build on top of a conceptual model distinguishing between factors of influence (causes of changes), for instance the developer or the hosting environment and information sources such as Quality-of-Service or documents. In the context of our definition of adaptation, the framework aims to develop a framework for detecting/monitoring the evolution of changes and does not tackle automatically adapting to changes.

Another approach aimed at tackling the conceptual scope of service evolution management is (Andrikopoulos et al., 2008). The authors specify characteristics of service evolution management such as identification of changes, propagation analysis, validation and compliance, version control and instance migration. Furthermore, a Service Specification Reference Model and operators for capturing changes are introduced. However, the work remains on an abstract level and it does not describe how adaptation of a service is enacted.

In (Ludwig et al., 2009; Wassermann et al., 2009), the authors discuss the issues and challenges in cross-domain change management. Change management deals with the process of implementation a change in a complex IT environment where it is assumed that many dependencies between components are unknown and that the process is done (largely) by hand. The approach proposed complements our approach in that they tackle the question of how changes are notified and how different parties can work together in order to create a new version of the inter-organizational business process. From an adaptation perspective, the authors provide a framework for monitoring and change detection. However, the decision of how to adapt and whether the software is adaptable is not discussed.

Another work on dealing with the management of service evolution is by von Susani & Dugerdil (2009). The authors suggest that due to the dependencies between services, the migration of a service to a new version should be scheduled. The work provides insights in how changes will be

communicated concerning a termination time for the old version and with possible overlap with the new version. Although the suggestions in the work have merit, no results of following this approach are shown.

An approach that describes an implementation is (Cibrán et al., 2007). In this work the authors introduce the Web Service Management Layer (WSML). This is a layer placed between client and external Web services. It supports some management functions, such as service selection, billing, accounting and transactions. The authors rely on an Aspect-Oriented Programming language to interweave management aspect with the client application. Concerning adaptation the approach deals only with service selection. Drawbacks of using Aspect-Oriented Programming for realizing adaptive behavior are discussed in Section 6.7.

Another approach containing a prototypical implementation is (Liu, 2009). Similar to our approach the author expresses that management should use models and that managing is a process. The author proposes three strategies of dealing with changes, namely: heuristic, policies and machine learning. Unlike our implementation aimed at interoperability, the authors target Quality-of-Service.

Our contribution: Management approaches, with the notable exception of (Cibrán et al., 2007) and (Liu, 2009), present only conceptual frameworks (or requirements) and deliver no proof by means of an implementation. Management deals with all aspects of a service and is therefore cumbersome to implement. This in contrast to adaptation which is typically regarded in the context of a single criterium. In this thesis, we provide both a conceptual framework for dealing with any type of change, and describe a proof-of-concept prototype focussed on an important functional aspect of services, namely interoperability.

6.3 Model Management

In this thesis we employ a model-driven approach to handle service evolution. Model-driven engineering (MDE) is the unification of initiatives that aim to improve software developments by employing high-level, domain-specific, models in the implementation, integration, maintenance and testing of software systems. It is stated that where before the main idea was that "everything is an object", in MDE the main idea is "everything is a model" (Bézivin, 2005).

Most prominent among MDE initiatives is the OMG's Model-driven Architecture (MDA) (OMG, 2003). MDA suggest the usage of multiple models at different levels of abstraction and the transformation from one model to another, mostly from a Platform Independent Model (PIM) to a Platform Specific Model (PSM). A key idea behind MDA is that it makes use of OMG's standards like Meta-object Facility (MOF), Unified Modeling Language (UML) and Object Constraint Language (OCL).

MDA is not the only initiative for a model-based approach, other efforts include Microsoft's Software Factories (Greenfield & Short, 2004) and Generic Model Management (Bernstein et al., 2000; Bernstein, 2003).

Approaches based on model management are RONDO (Melnik et al., 2003b,a), AMW (Fabro et al., 2005) and GeRoMeSuite (Kensche et al., 2007). These approaches introduce new generic operators but work only on database schemat's.

Our contribution: Our motive for not using MDA is that model management suggests the usage of operators for manipulating models and mappings. This idea allows (automatically) adapting a model to a change as well as setting the boundaries to which changes can be adapted. This approach is therefore more intuitive in the context of service evolution. Unlike previous Model Management approaches, we describe an approach for using Model Management for Business Processes rather than for database schemat's.

6.4 Service Interoperability

Interoperability is an important aspect of software especially in distributed systems. Evolution of individual services is therefore an important problem. Interoperability is of recognized importance in academia and industry. In industry an organization dealing with interoperability is the Web Service Interoperability organization (WS-I). The WS-I is an open industry organization with as goal to establish best practices for web services interoperability. Among others, the WS-I provides an extensive Supply Chain Management Scenario with sample application implementations from many different vendors (Web Services Interoperability Organization, 2007). In research, projects like Interop (Berre et al., 2004) were launched in order to leverage efforts dealing with interoperability.

Figure 6.1 illustrates the different concepts related to interoperability. We discuss each of these four concepts with related work in detail below.

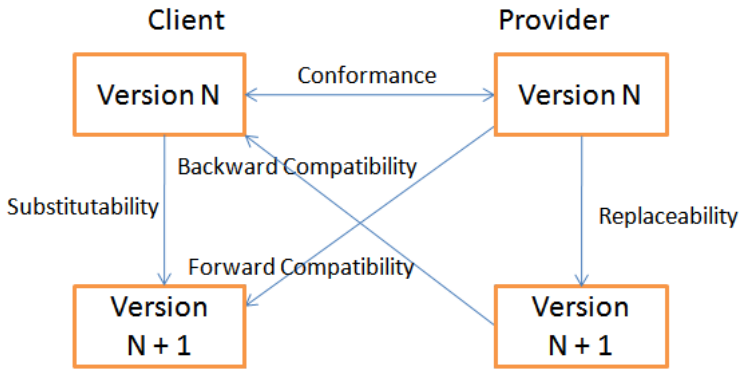


Figure 6.1: Concepts related to Interoperability

6.4.1 Compatibility

To determine whether two services can work together is called compatibility analysis. As stated in Section 3.2 of Chapter 3, we cover only data (types) and business protocols (also called behavior) in this thesis. In relating to other work we therefore focus on work done that covers one or both of these aspects. In compatibility analysis a distinction is made in the perspective from which the compatibility is regarded. We distinguish between general, backward and forward compatibility analysis.

General

In the general case, compatibility is regarded from a third-party perspective. An example of research conducted in the general case is provide by Martens (2003). In this paper, the author uses Petri nets to formally verify behavioral compatibility of Web services.

Another approach using Petri nets for formally verifying compatibility was provided by De Backer et al. (2009). In this paper a new paradigm for modelling interacting business processes is introduced. The authors provide interesting insights in the relation between interaction aspects and business process aspects in collaborative e-business. A drawback of the approach (which also holds for Martens (2003)) is that no data (types) are handled.

Andrikopoulos et al. (2009) tackle service evolution from a contractual perspective between two participants (service provider and service consumer). Their goal is to allow independent evolution of loosely coupled services in

a transparent manner. To reach this goal, the authors introduce an exposition/expectation view that in addition to the traditional required/provided view describes in more complete terms how changes affect a contract.

Ponnekanti & Fox (2004) focus on mismatches on the data layer (WSDL + XML Schema). They systematically categorize possible mismatches and use real world cases to see when problems occur. They state that most services use only a fraction of the other service functionality, so although a lot of incompatibilities can in their view not be solved in most cases it is not a problem, because it was not used.

Business protocols are described in a business protocol definition and from this definition multiple instances can be generated and run at the same time. In (Ryu et al., 2007, 2008) the authors study how changes impact the definition of protocols and provide change operators for manipulating protocols. They study how to update running instances given certain changes, as is studied in Workflow Evolution (see Section 6.6). Similar to our manager, the authors propose a protocol evolution manager for handling these changes. However, unlike our approach, the authors do not handle data and remain on the behavioral layer for interoperability.

Aït-Bachir et al. (2009) introduce a detection algorithm for comparing two business protocols specified as Finite State Machines for incompatibilities based on the addition, removal and modification of an operation. However, similar to the work mentioned above the authors do not handle data.

Backward Compatibility

Backward compatibility means that a newer version of the service provider can be deployed without breaking interoperability with the client. I.e. how a service provider should change, such that a client can still use the newer version of the service (Becker et al., 2008). Backward compatibility is a property that is reachable through extensibility and is frequently discussed in the context of Web service versioning. Recent work on backwards Compatibility is, among others, (Becker et al., 2008; Fang et al., 2007; Kaminski et al., 2006a)

Becker et al. (2008) focus on backward compatibility and provide a framework for automatically analyzing changes in between versions.

Fang et al. (2007) state which changes cause non-backward compatibility in the categories: implementation, interface and binding changes.

They extend SOA for handling versioning, meaning that they extend WSDL and UDDI. They also advocate that the UDDI be equipped with a publish/subscribe such that clients of a service are notified about changes.

Kaminski et al. (2006b,a) present a design technique called a "Chain of Adapters". They guarantee backward compatibility by chaining adapters such that the original version is in tact. This approach is based on extensibility and therefore can only be used for a limited number of times. In addition to this, a service compatible with an earlier version has to talk to a service later in the chain taking more time to communicate.

Forward Compatibility

Forward compatibility means that a newer version of the client can be deployed in a way without breaking the interoperability with existing providers. Not many approaches deal with forward compatibility, worth mentioning is Orchard (2006). Orchard (2006) states that forward compatibility is partially achieved by ignoring received data that is unknown to the recipient. Furthermore Orchard states that both backward and forward compatibility are achievable only through extensibility. As argued in Chapter 1 extensibility can only be used a limited number of times, and therefore is only a partial solution at best for guaranteeing interoperability.

6.4.2 Conformance

If the interaction between a service provider and a service client is to go according to an agreed protocol then both services should conform to this protocol. Testing whether a participant is conforming to a certain protocol or contract is called a conformance test. Conformance is especially important in a choreography perspective where properties of the whole choreography can only be guaranteed if each participant conforms to the agreed protocol and/or contract. Example of work done on conformance is by Baldoni et al. (2009). Conformance is closely related to compliance where a business process must adhere to regulations. An example of research conducted in that area was done by Liu et al. (2007) where the authors create a framework that uses model checking for checking compliance.

6.4.3 Replaceability

If two services have equivalent protocols, that is, if they can support the same set of conversations then these services can replace each other. This means that they can talk to the same clients. Replaceability analysis involves finding the set of conversations that both protocols can express. Replaceability is an important aspect of adapters. Recent work on adapters includes (Brogi et al., 2004; Benatallah et al., 2005; Nezhad et al., 2007; Taher et al., 2009).

Zhou et al. (2008a,b) describe an approach for automatically creating an adapter, they use dependency graphs to model data dependencies. They compare execution traces (called scenarios) to see whether two public processes are compatible. In their approach, they use the term similarity instead of replaceability.

Nezhad et al. (2007) semi-automatically detect mismatches between service interfaces, looking at both data and protocol. They further create an adapter based on what they call a mismatch tree. This tree contains possible deadlocks that might occur and the authors use it in order to generate a compatible adapter.

An important work on the management of business protocols is that of Benatallah et al. (2004b,a, 2006). In this work they describe management operators for analyzing compatibility and replaceability. They introduce operators such as *Difference*, *Intersection* and *Compatible composition* for managing protocols. The main difference between their work and ours is that they focus on protocols, without data, between two parties, whereas we look at orchestrations (multiple parties) including the data layer.

In (Ponge et al., 2007) the authors also study compatibility and replaceability of timed protocols. The authors provide operators such as difference, intersection and projection and formally verify their properties.

6.4.4 Substitutability

Replaceability determines whether two services can replace each other from the perspective of the service provider. Substitutability on the other hand regards replacement of services from the perspective of the client. It determines whether a service used in a service composition can be substituted by another service that provides the same functionality. Our approach for adapting a service orchestrator described in Chapter 5 also covers substitutability.

In (Antonellis et al., 2006) the authors state that approaches like BPEL do not provide any real substitutability, as concrete services must be specified in the process definition. Therefore a substitute would have to be a hundred percent match with the substituted service, which is nearly impossible. In their framework, they suggest to use an abstract service description such that multiple concrete services match the description. Difference with our framework is that the Web services involved do not have business protocols but operate on a request-response basis. Furthermore, no formal validation of their architecture is provided.

Ernst et al. (2006) study the detection of substitutability and composability of Web services. In this practical oriented work, the authors published Web services and look whether the input and output can be chained (composability) or whether they show similarities (substitutability). Same as (Antonellis et al., 2006), the approach works with concrete Web services (no formal foundation) and works only with Web services having a request-response protocol.

A formal approach based on mu-calculus was provided by Pathak et al. (2007). They specify a property over the composition and determine whether substituting a service in that composition still satisfies that property. A drawback of their approach is that the authors do no consider data parameters, i.e., messages being exchanged by the services.

Our contribution: As discussed in the first chapter of this dissertation (Section 1.1), both backward and forward compatibility only prevent and do not solve incompatibilities. Furthermore, adapters provide a solution to incompatibilities, but suffer from the drawback that they regard services as black box and therefore can solve only a limited range of problems. To overcome these shortcomings, we describe an approach how an orchestrator can be made adaptive to overcome incompatibilities. The approach determines whether changes are applicable, i.e., checks whether a service provider's interface is backward compatible. To solve incompatibilities, we describe how to adapt using three operators, which can also be used to determine substitutability of services. Regarding adaptive behavior we guarantee replaceability for solvable incompatibilities. Unlike most approaches on interoperability, which typically focus on one layer, we handle both data (type) and business protocol layer.

6.5 Service Composition

A service that is implemented by combining functionality from different services is called composite. The process of developing a composite service is called service composition (or synthesis). Since the emergence of Service Oriented Computing a lot of attention has been going to service composition and the aspects related to it. A motive for this research is that this nesting of services allows rapid development of new services. Some surveys of the field are: (Agarwal et al., 2008; Dustdar & Schreiner, 2005; Hull & Su, 2005; Milanovic & Malek, 2004; Rao & Su, 2004; ter Beek et al., 2007; Baryannis et al., 2008) A categorization of service composition was provided by Agarwal et al. (2008).

We make a distinction between techniques based on automata, rule-based and those that combine automata and rules. Although other distinctions and categorizations exists, for our work these are the most related.

Automata-based

Work using automata (also called labeled transition systems) has been done by Berardi et al. (2003, 2005a,b); Calvanese et al. (2008) on the frameworks called Roman and Colombo. In their work they assume that services export their behavior in the form of finite automata and that the user specifies a goal service (also an automaton). The result of the synthesis process is a mediator which is also specified as an automaton. They encode the problem using Propositional Dynamic Logic (PDL)(Harel et al., 2000) reducing the synthesis process to a satisfiability problem. The verification of properties, such as the existence of the composition can thereby be formally proven. They have an action-based perspective on the composition.

Fu et al. (2004a,b,c,d) use guarded automata with queues for modeling services with asynchronous communication. They provide properties for ensuring that composition is still decidable after composing. Their framework is aimed to provide model checking possibilities to service compositions. Although not directly aimed at creating compositions, the results of their work are important for web service composition.

Pathak et al. (2006a,b, 2008) introduce MoSCoE, a framework that builds on insights from Colombo. They state that the framework handles incompletely specified target services which contain functions. These functions are

in the composite service then translated into an exchange of messages. One of the advantages of their approach is that the causes of failure to compose can be communicated back to the user who can then reformulate the goal specification in an iterative fashion.

Gerede et al. (2004) also extend the Roman model to study for what they call lookaheads. The problem they solve is that when running a service it should look ahead to see the future dependencies on the action that he delegates to a certain service now. For instance, a service that executes an action m, must execute afterwards an action n. Thus if the orchestrator chooses to let that service execute action m then it is obligated to let it also perform action n. If this is not desired then the orchestrator must look ahead to see future dependencies.

Other formal models that have been used for service composition are, among others: Petri-nets (Hamadi & Benatallah, 2003) and Process Algebra (Yi & Kochut, 2004). Petri-nets are more expressive than automata, for instance, petri-nets can represent parallelism whereas automata can not. However, for the purpose of explaining our conceptual approach (applicable to any model), automata, because they are less expressive, serve better to illustrate models and model adaptation. Similarly to Petri-nets, Process Algebra are also more expressive but lack a graphical representation.

Rule-based

Orriëns et al. (2003); Orriëns (2007) employ a model driven approach to dynamic service composition by using business rules. By viewing the composition as a model they raise the level of abstraction thereby creating a potentially more flexible and agile service compostion.

Where the rules used by Orriens guide the development process of a service composition, rule-based approaches have also been suggested as an orchestration model. In their book Alonso et al. (2004), describe in the chapter on service composition also a rule-based orchestration. They discuss event-action (EA) and event-condition-action (ECA) rules as a mean for orchestration. Since rule-based models are inherently less structured, they advocate to model orchestrations only that have few constraints among activities and thus were the number of rules are few.

Another usage of a rule-based approach has been in AI-planning (cf. (Medjahed et al., 2003)). The key feature of their composition approach are

the composability rules. These rules compare the syntactic and semantic features of Web services. Based on this comparison it is decided whether the services are composable.
Ponnekanti & Fox (2002) developed a developer toolkit called SWORD, which also uses rule-based planning. They model services by its preconditions and postconditions. To create a composite service, the user only specifies the initial and final state. Plan generation is done by a rule-based expert system.

The rule-based planning approaches take into account the semantics of the data. Although this is not handled directly in this thesis, other work that is worth mentioning is METEOR-S (Aggarwal et al., 2004).

One of the major drawbacks of these systems is that from a set of rules it is hard, for human users, to figure out what the resulting process flow is. This is due to the fact that rule sets may involve loops, may have side effects on the underlying data, and that rules might be nested and therefore tend to be hard to understand.

Hybrid-based

A number of hybrid approaches have been suggested that combine business process (mostly BPEL) and business rules. For instance, Charfi & Mezini (2004) model business rules as aspects and use an aspect-oriented version of BPEL to weave these aspects in the process. Other use ECA-rules (van Eijndhoven et al., 2008; Baresi et al., 2007b) to create more flexibility. A drawback of all these approaches is that they only provide methods on how to model business rules together with business processes. They do not provide any insights on how an automated synthesis process should go and what can be used.

Colombo et al. (2006) also combine BPEL with rules. They use ECA rules to guide the binding and reconfiguration when their properties do not match the environment requirements or when context changes.

In their approach of suggesting to use Finite State machines for modeling business protocols (instead of BPEL), Benatallah et al. (2004b) also propose that conversation management can be enabled with a control table containing a set of ECA-rules. However, they do not go into detail how this could be realized.

An approach that suggests to use ECA-rules for capturing adaptation

logic is provided by Sheng et al. (2009). In this paper, the authors describe a semi-automatic approach for web service composition where they use ECA-rules to specify control-policies.

Our contribution: Approaches on hybrid business processes present models that can be used to represent business rules within processes. However, how to synthesize these processes from different components was not done. On the other hand, the approaches on automatic service composition, do not incorporate business rules in the synthesis process. In this thesis, we combined these approaches and present an approach for modeling *and* composing a hybrid business process. In order to make the softare more manageable, we also incorporate policies that allow users (developers) to chose between alternative business processes.

6.6 Workflow Evolution

Workflow is well represented in the field of Service Oriented Computing, particularly in relation to service orchestration and service composition. Services fit nicely in the vision of workflow because of their distributed, heterogeneous, component-based nature. One of the drawbacks of workflow management systems was that their process definitions had different proprietary formats and were difficult to share (Georgakopoulos et al., 1995). The advantage of services is that they have a standardized interface (WSDL) and also the BPEL standard was aimed to bridge this gap (Khalaf et al., 2006).

For a survey of existing commercial workflow management systems, we refer the reader to Ader (2004). A few overviews of scientific work on the field are: (Georgakopoulos et al., 1995; Stohr & Zhao, 2001).

In Chapter 3 we discussed when a set of operators is complete for a given type of model. However, if a construct is not part of the type of model (language) then something cannot be expressed. For instance, if a model has no concept for XOR, then XOR cannot be used. In workflow management, what can be expressed in a workflow language has been extensively investigated (van der Aalst et al., 2003; Russell et al., 2004, 2006). They use these patterns to make a comparison between different management systems in terms of expressivity in a process definition.

How changes affect a workflow management system is called workflow evolution or adaptive workflow systems. In workflow management two types of change are distinguished; a change can affect either a process definition

or a process instance. An important part of the work done on workflow evolution is when a process definition changes, how these changes propagate to the running instances of that process definition (Bandinelli et al., 1994; Ellis et al., 1995; Casati et al., 1996; Rinderle et al., 2004; Weske, 2004). Not all instances of running processes can be migrated when a change occurs and a question is therefore how to deal with them. This is also studied in the context of BPEL specified processes (Reichert & Rinderle, 2006).

Other work aims at providing the user/administrator with more flexibility during the execution of workflow instances. For instance, when exceptions occur a user might need to deviate from the process described by the process definition. Reichert & Dadam (1998) provide a solution based on operators which are formally specified. Main difference with our approach is that after applying an operator the model should be correct whereas we demand that the model is correct after applying all operators in a script. Others who also specify operators, are (Casati et al., 1996; Reichert & Dadam, 1998)) for control aspect and (Rinderle & Reichert, 2006) for data.

Based on the work of workflow patterns, change patterns have also been identified for comparing adaptive workflow management systems (Weber et al., 2008)

One approach for creating more flexibility in a process is by leaving parts unspecified. This approach is taken by Sadiq et al. (2001), who introduce so called pockets of flexibility. These pockets provide a template-based choice on how to fill the gaps of this process at run-time, given a set of activities. Later the authors extend this to incorporate constraints in (Sadiq et al., 2005).

Another approach based on constraints is (Pesic et al., 2007). Based on their declarative model for workflow (Pesic & van der Aalst, 2006), the constraints specify what is allowed in a model, similar to our model properties. An advantage of the constraint-based approach is that it allows flexibility by leaving parts unspecified, similar to Sadiq et al. (2005).

Other approaches to make workflow management systems more adaptive/adaptable include incorporating rules. Either by using rules for capturing control structure (Muth et al., 1997; Hull et al., 1999; Joeris & Herzog, 1999; Bae et al., 2004) or for capturing adaptation (Müller et al., 2004).

Hamadi & Benatallah (2004, 2005); Hamadi et al. (2008) introduce a self-adaptive recovery net (SARN), which is an extended Petri net model for specifying exceptional behavior at design time. This model can then adapt

the structure of the Petri net model at run-time (instances).

A drawback of most approaches is that they act from a centralized engine . Park & Kim (2009) suggest to use publish/subscribe protocol to exchange events such that users have their own workflow definition and that every workflow can therefore become self-managing. However, not many details about how this vision would be realized was presented.

Another approach for making workflows adaptive was to incorporate (conversational) case-based reasoning which look at previous encountered similar problems and uses these to solve new problems (Weber et al., 2006).

Our contribution: Workflow evolution approaches focus mainly on process related aspects while largely neglecting interoperability problems due to changes. Unlike these approaches, we describe how interoperability affects a business process and show how to adapt to maintain interoperability.

6.7 Service Adaptation

In categorizing work done on service adaptation, we make a distinction between approaches based on aspect-oriented programming, approaches based on self-adaptation and extensions of the SOA aimed at (self-)adaptation.

Aspect-oriented

Aspect-Oriented Programming (AOP) is currently a popular approach in research for achieving adaptability and adaptation. Among others, recent work includes: (Kongdenfha et al., 2006; Cibrán et al., 2007; Moser et al., 2008; Charfi et al., 2009; Karastoyanova & Leymann, 2009).

Many AOP approaches extend BPEL and use an additional language or engine for realizing adaptation, for example (Charfi et al., 2009). Other work, for example Karastoyanova & Leymann (2009) propose an approach based on Aspect-Oriented Programming which uses only BPEL constructs to realize adaptation.

In (Moser et al., 2008) an aspect-oriented approach is described for monitoring and selecting/substituting web services. The framework, called VieDAME, is focussed on Quality-of-Service aspects. The authors extend the ActiveBPEL engine to incorporate what they call the Interception and Adaptation Layer (IAL). The framework allows dynamic substitution of services and they fore-

see that adapters, called transformers, are placed when services are not syntactically compatible.

Aspects in AOP approaches for service adaptation are based on the idea of a separation of concern between business- and adaptation logic. The adaptation that is captured in the logic is in most of these approaches based on the service selection and service substitution, allowing hot-swapping of services. A shortcoming of using an aspect oriented approach is that adaptability of the software is limited to what is captured by the aspects. Our model-management approach, where the models can be completely adaptable, does not suffer from this shortcoming.

Extensions of SOA

Next to these approaches for handling specific problems, some architectures and extensions to SOA have been introduced. Conceptual extensions of SOA, like WSMF (Fensel & Bussler, 2002), WSMO (Roman et al., 2005), Colombo (Curbera et al., 2005), and COSMO (Quartel et al., 2007) have been discussed in Chapter 2, and will not be further discussed here.

An older work on creating a dynamic SOA (DySOA) was provided by Siljee et al. (2005). DySOA is an architectural extension for service-based application systems. It contains components for monitoring, analysis, evaluation and configuration of an application. The work is aimed at Quality-of-Service and does not specify how functional aspects like interoperability should be handled.

Tanksali (2006) coined the term of an Adaptive Service Oriented Architecture. In this work Tanksali advocates that services should become intelligent, possibly incorporating techniques from Artificial Intelligence. How this intelligent service should be realized, by which technique, for instance, Agents, Aspect-oriented Programming or a manager is not specified. Furthermore, the work presents only a vision and it not validated in theory or practice.

Another extension of SOA is described in (Rolland et al., 2007). In this paper it is argued that traditional SOA is function-driven and that many aspects, such as the interfaces in WSDL, are hard to understand by business people. The authors propose an intentional-driven SOA, called ISOA. The brokerage triangle is adapted for intentions and the authors propose an Intentional Service Model for describing each service. Agents are suggested

to be used for realizing an ISOA. Although they make a valid point that adaptation should be guided by the business, they do not show these agents are implemented and how different types of changes can be handled.

In the paper introducing Web based Internet-accessible Service (WebBIS) (Medjahed et al., 2004) the authors propose a declarative language for composing Web services. They define meta-services (similar to our managers), that notify concerning changes about the availability of the service. Lack of this framework is that it does not tackle data.

Self-adaptation

With the introduction of Autonomic Computing (AC) a lot of terms with "self-" were launched, such as self-configuring, self-healing, self-optimizing and self-protecting. Self-adaptation is required to realize each of these properties. Although the addition of "self-" to the word adaptation makes it explicit that the software changes itself, this is often already implicit in the notion of adaptation used. As we consider that a manager exists that adapts the manageable service, we prefer to use the word adaptive over self-adaptive in this thesis.

For a recent survey on self-adaptive software the interested reader is referred to (Salehie & Tahvildari, 2009). For the work done on self-adaptation, we limit ourselves to work done in relation to (Web) services.

In (Dustdar et al., 2009) several research challenges for self-adaptive service-oriented systems are presented, such as compliance and run-time management of requirements. For each of the challenges a solution direction is given. Similar to our approach the authors suggest a model-driven approach in tackling these challenges. Although the paper provides interesting insights in self-adaptive software engineering, it does not provide concrete results.

Denaro et al. (2006); Tosi et al. (2009) describe in their work an approach for the design of self-adaptive service-oriented applications. They identify different (semantic) mismatches and propose that software architects design, test and create adaptation strategies to tackle them. The testing is done via assertions which are weaved in the application. If an assertion is true, this means that a predefined adaptation strategy is executed. This approach is very similar to the aspect-oriented approach of Baresi et al. (2007a). The authors are practical oriented but provide only preliminary results.

Another work on self-adaptation was provided by Dorn et al. (2009). The authors focus on the problem of selecting the best service to forward a request to. Other aspects, such as service evolution are not handled.

In (Gjørven et al., 2008) self-adaptation is introduced in SOA through adaptation middleware. This middleware is designed for integrating and exploiting technology-specific adaptation mechanisms. The work is based on component models and the authors aim to tackle adaptation on the service interface layer as well as on the application layer. Although the work provides interesting insights in adaptive middleware, it does not provide results in how changes are tackled.

Baresi et al. (2007a,b, 2009) use a rule-engine for creating a self-healing composition, similar to the hybrid composition approaches discussed in Section 6.5. The authors achieve self-healing by defining two languages; WsCol for specifying constraints on the execution of BPEL processes and WsReL to state recovery strategies. A drawback of their approach is that these two new languages must be used in order to achieve self-healing.

Our contribution: Adaptation approaches, such as described above, focus on a specific change. For this change an implementation is given which usually can not be used for other changes. Only the conceptual adaptation cycle is generic to these approaches. We describe a conceptual framework for developing services for any change and furthermore demonstrate this approach by implementing a prototype for a specific problem, namely changes affecting interoperability.

Chapter 7

Conclusion

7.1 Introduction

Organizations wish to be able to easily cooperate with other companies and still be flexible. The IT infrastructure used by these companies should facilitate these wishes. Service-Oriented Architecture (SOA) and Autonomic Computing (AC) were introduced in order to realize such an infrastructure, however both have their shortcomings and do not fulfil these wishes. AC is visionary and also SOA does not provide concrete solutions for designing adaptive software.

This dissertation addresses that problem and presents an approach for incorporating (self-)adaptive behavior in software. A conceptual foundation of adaptation is provided and SOA is extended to incorporate adaptive behavior, called Adaptive Service Oriented Architecture (ASOA). To demonstrate the feasibility of this conceptual framework, we implement it to address a crucial aspect of distributed systems, namely interoperability. We study the situation of a service orchestration where a service orchestrator wishes to adapt itself when its service providers evolve.

In this chapter the research results of the previous chapters are summarized and insights regarding the contributions of the research are provided. Furthermore, directions of future research are presented for realizing (self-) adaptive software.

7.2 Research questions and answers

In Chapter 1, the research questions were split in a group concerned with the conceptual aspects of software adaptation and in a group related to interoperability. We follow the same distinction here.

Conceptual

In Chapter 2 we answered the following research questions.

1. *What types of changes occur in a Service-Oriented Architecture?*
   We defined what constitutes the core concept of SOA, namely a service. Taking a system theoretic approach, we treated the service as system and all other related concepts as environment. All concepts of the environment are subject to change and can influence the service. For instance, the organization which owns the service may impose different requirements, or the other services may publish new, and altered, interfaces. The concepts we distinguished, and thus the type of changes, were described in Section 2.4.2, and they were: actions (and tasks), messages, interface, service, and organization.

2. *How can a (composite) service be made (self-)adaptive?*
   To make a service adaptive it needs adaptation logic, also called adaptation strategy. Adaptation logic describes how the service adapts, in terms of our taxonomy of adaptation, how to make the decision. We incorporated the adaptation logic in a separate entity, named the manager, thereby creating a separation of concerns between adaptation logic and business logic. Motive for this separation is that intertwining adaptation logic and business logic leads to an increase of software maintenance (Salehie & Tahvildari, 2009). To further develop a conceptual framework for adaptive services, we used Model Management.

   Our motive for choosing model management, also called generic model management (Melnik et al., 2003b; Melnik, 2004), is its broad applicability. Model management proposes the development of generic operators for manipulating, transforming and creating mappings between models. By using model management, we developed a framework for tackling any type of change.

2.1 *How can a service be made manageable?*
In Section 2.7.1, we defined *manageable* as the ability of software to be monitored as well as being adapted. In terms of our taxonomy of adaptation, this means that mechanisms for detecting changes as well as for executing adaptation plans should be facilitated.
In our Model Management approach, we defined the process of a service to be a model. On this model change operators can be defined such that the model can be adapted. Furthermore, every model consists of concepts and relations; if all concepts and all relations are captured using operators, then a model can be completely adaptable. If a concept of monitoring exists in the model, then a model is completely manageable.
In order to enable the interaction between a manager and a manageable service, the manageable service publishes a managerial interface. This interface provides all functions for managing the service, thus retrieving the current model and change operators, setting a goal, and updating the current model with a script containing change operators.

2.2 *How to design a manager?*
In our framework, a manager is a service. Therefore everything we defined for a manageable service applies for a manager as well. Additional constraints for the manager are that the process of a manager entails an adaptation cycle. The question of when to adapt and how to know whether adaptation is successful, is answered by the incorporation of a goal. We defined a goal to be a property of a model. Because we define a goal as a property, we can verify whether a goal is reached by determining whether the property holds for the model. To demonstrate the validity and illustrate the working of the manager, we presented an implementation of a manager designed to deal with changes that affect interoperability (Chapters 3,4 and 5).

3. *How can a Service-Oriented Architecture be extended to handle (self-)adaptive services?*
We introduced a number of new concepts and altered another concept of the core conceptual model of SOA. We created a conceptual model

and illustrated how concepts are related. The new concepts were Manager, Goal, Capability and Contract. The altered concept was Message, which we converted into Event. With the addition of these new and altered concepts, the conceptual model contained an adaptation cycle and provided the conceptual foundation for adaptation and service management. This extension we called the Adaptive Service-Oriented Architecture (ASOA).

We validated ASOA in two ways. The first way was to show that our conceptual framework does not suffer from the shortcomings of traditional SOA. We realized this in three steps. The first step was by defining what constitutes adaptive behavior based on existing literature. The second step was to demonstrate that services built in SOA *do not* possess adaptive behavior. And the third step was to show that services in ASOA *do* posses adaptive behavior. The second way of validation was that we demonstrated that (self-)adaptation, as a prototypical implementation of ASOA, serves as a solution for interoperability problems in a service orchestration.

Applied to Interoperability

Concerning interoperability, we dealt with changes to the interfaces of service providers and set as a goal to be able to maintain interoperability in a service orchestration. In relation to this goal, we posed the following questions.

4. *How can we model changes to a service interface?*
   Similar to the manageable service, we defined the interface of a service to be a model (Chapter 3). We distilled the concepts and relations, and defined change operators for adding and removing these concepts and relations. By using these operators we were able to capture every change in the interface, under the assumption that the model which is used as the interface was known. To relate our approach to others, we demonstrated how known mismatches in business protocols are represented using our operators.

5. *What changes result in incompatibilities in a service orchestration?*
   Not all changes result in incompatibilities. Whether a change can be solved or not was described in Section 5.4.1 of Chapter 5. An incompatibility can be solved if there exists an adaptation script for it, or if the changes are not applicable. To determine whether a change affects

the business process of an orchestrator, we introduced the applicability operator in Section 5.3 of Chapter 5. This operator determined whether a set of changes has an impact on business processes. For finding the right adaptation script, we introduced the adapt operator.

6. *How to automatically adapt the service orchestrator in order to maintain interoperability, without changing its interface to its clients?*
   This question can be rephrased to: How to design a manager using our conceptual framework for automatically maintaining interoperability in a service orchestration, without changing the interface of the orchestrator while service providers evolve. An abstract process description and design of the manager was described in Section 2.7.2 of Chapter 2. This process was explained in detail in Chapter 5. The main component of this process was the adapt operator. The adapt operator combines three operators, namely: remapping, reordering and recomposition. Each of these operators has a different impact on the business process and a different range of problems that it can solve. Remapping adjusts the mapping of the orchestrator, but does not change the process. Reordering exploits any flexibility in the ordering of messages and attempts to reorder messages to solve incompatibilities. The recomposition operator searches for an alternative composition that incorporates the changed service. As the name of the "recomposition operator" indicates, it is based on the composition approach described in Chapter 4. Using these three operators, we not only can decide whether and how an incompatibility can be solved, but they can also be used to analyze whether a service can be substituted for another.

We validated the framework for interoperability in two ways. The first way was by formally verifying that our adaptation solution was correct with respect to guaranteeing interoperability. The second way was by constructing a prototype that implemented this solution.

To summarize, this dissertation introduced Adaptive Service Oriented Architecture (ASOA). We provided a definition and taxonomy of adaptation, and used it to analyze and extend SOA. With ASOA as foundation, we developed a conceptual framework, using Model Management for developing services capable of dealing with changes in their environment. We expressed how a service can be made manageable and also how a manager can be designed. This conceptual framework was implemented in a prototype

for handling an important class of problems in distributed systems, namely interoperability.

7.3 Future work

We answered the main research questions posed in this thesis. However, while answering these questions many other questions arose. In this section we describe the directions of future research important for realizing (self-) adaptive software.

Maintaining identity: The implementation of our conceptual framework focused on interoperability and how the environment (other services) influence an orchestrator. An aspect that we have not dealt with is the consideration how far the influence of the environment can go. The question is how to maintain the identity of the service despite adaptation (Regev et al., 2007). An example of a scenario where this question is relevant is when a change in the interface of the bank would require the retailer to change a business rule in order to comply. As more business rules are changed due to the desire to maintain interoperability, the question rises whether the retailer is still a retailer. In this thesis, we adapted only the interaction of the orchestrator and did not change the business rules. Therefore we can state that although services provider may be changed or replaced, the orchestrator remains a retailer until a problematic change occurs. In general, for each model used for the manageable service, it must be specified what can be adapted, by whom, and to what extend.

Multiple criteria: Interoperability is but one of the many aspects of software. Other aspects include Quality-of-Service, Reliability etc. If self-managing software is to be successful, then all aspects should be regarded to create an adaptation strategy. An example of an approach indicating some problems with handling multiple criteria is Cheng et al. (2006). An important aspect regarding multiple criteria is how criteria are related. It is crucial to find out how different information sources and stakeholders are related to each other.

Management of self-adaptive software: An important aspect of software is that, although in service orchestrations not always evident, in the end software interacts with humans. Adaptation has an aura of mystique around it and developers are hesitant to deploy it in real products. Software should remain understandable for the developers as well as for the users.

Self-adaptive software should reflect the wishes of developer (and users) in such a way, that incorrect or undesirable adaptation will not occur. We call this process of managing self-adaptive software *Managed Self-Adaptation*. In this thesis, a few mechanisms were introduced, such as policies and an escalation procedure that exemplifies this management. The policies reflect preferences used for choices and the escalation procedure contacts a human administrator when the manager fails to find an adaptation strategy.

However, to ensure the success of (self-) adaptive software in the field of software engineering, more management capabilities for self-adaptive software should be incorporated. An example of a management capability are interruption mechanisms. For instance, a developer should be able to turn adaptation off if he wishes to study a problem.

Model evolution: We used a model-driven approach and used model management for manipulating these models. However, these models may be subject to change themselves, for instance to improve the expressivity of the modeling language (van Deursen et al., 2007). Therefore, as new versions of the models are used to deploy new systems, older versions may have to be migrated to the newer version of the model.

Solution propagation: In the introduction to this thesis, we described that changes can propagate through a network of services. Making services adaptive helps to decrease the frequency of these propagations. However, similar to problems caused by changes, solutions for adapting to these changes can also be propagated. Mechanisms that can be used for this purpose are case-based reasoning or imitation (Hiel & Weigand, 2006).

These directions of future research indicate that there remains a lot of work to be done for realizing self-adaptive services. As self-adaptive services will become more important in the future (Papazoglou et al., 2007; Di Nitto et al., 2008; Dustdar et al., 2009), it is expected that more researchers will get involved. It will be interesting to follow the development of (self-)adaptive software in the future in both academia and industry.

Appendix A

Specification of Services

This appendix contains all the specifications of the example services throughout the thesis. They are in the format described in Section 5.5.

A.1 Shipper

```
Bottomup {
    Schema{
        PeerList{Shipper,Retailer},
        TypeList{

        GetShipmentQuote[
            productList[
                productID[xsd:int],
                quantity [xsd:int]
            ]{1,10},
            address[
                streetName [xsd:string],
                zipcode [xsd:string],
                city    [xsd:string],
                country [xsd:string]
            ]
        ],

        ShipmentQuote[
            shipmentID  [xsd:int],
            shipmentDate [xsd:string]
```

```
        ],

        },
        MessageList{
          getShipmentQuote { Retailer -> Shipper : GetShipmentQuote},
          shipmentQuote { Shipper -> Retailer : ShipmentQuote}
        }
    },

    Peerset{
    Shipper{
        States{sh1,sh2,sh3},
        InitialState {sh1},
        FinalStates{sh3},
        TransitionRelation{
            t_sh1{ sh1 -> sh2 : getShipmentQuote,
                Guard{ true }
            },

            t_sh2{ sh2 -> sh3 : shipmentQuote,
                Guard{ true }
            }
        }
    }
} }
```

A.2 Inventory

```
Bottomup {
    Schema{
        PeerList{Inventory, Retailer},
        TypeList{

        CatalogRequest[
            getCatalogRequest[xsd:bool]
        ],

        CatalogResponse[
```

```
        productList[
            product  [xsd:string],
            productID[xsd:int],
            price    [xsd:int]
        ]{1,10}
    ],

    CheckAvailability[
        productList[
            product  [xsd:string],
            productID[xsd:int],
            quantity [xsd:int]
        ]{1,10}
    ],

    Availability[
        productID    [xsd:int],
        available [xsd:bool]
    ]

    },

    MessageList{
     catalogRequest { Retailer -> Inventory : CatalogRequest},
     catalogResponse { Inventory -> Retailer : CatalogResponse},
     checkAvailability { Retailer -> Inventory : CheckAvailability},
     availability { Inventory -> Retailer : Availability }
    }
},

%-  set of guarded automata
Peerset{

    Inventory{
        States{si1,si2,si3,si4,si5},
        InitialState {si1},
        FinalStates{si5},
        TransitionRelation{
            ti1{ si1 -> si2 : catalogRequest,
```

```
                Guard{ true }
            },

            ti2{ si2 -> si3 : catalogResponse,
                Guard{ true }
            },

            ti3{ si3 -> si4 : checkAvailability,
                Guard{ true }
            },

            ti4{ si4 -> si5 : availability,
                Guard{ true }
            }
        }
    }
}
```

A.3 Bank

```
Bottomup {
    Schema{
        PeerList{Bank,Retailer},
        TypeList{

        GetAccCredib[
            bankaccount[
                accountID  [xsd:int],
                accountType[xsd:int]
            ]
        ],

        AccCredib[
            balance[xsd:int]
        ],

        GetCCCredib[
            creditcard[
```

```
            number [xsd:int],
            expiryMonth [xsd:int],
            expiryMonth [xsd:int]
        ]
    ],

    CCCredib[
        balance[xsd:int]
    ]

    },
    MessageList{
        getAccCredib { Retailer -> Bank: GetAccCredib},
        accCredib { Bank -> Retailer: AccCredib},
        getCCCard { Retailer -> Bank: GetCCCredib},
        ccCredib { Bank -> Retailer: CCCredib}
    }
},

Peerset{
        Bank{
        States{sb1,sb2,sb3,sb4},
        InitialState {sb1},
        FinalStates{sb3},
        TransitionRelation{
            t_b1{ sb1 -> sb2 : getAccCredib,
                Guard{ true }
            },

            t_b2{ sb2 -> sb3 : accCredib,
                Guard{ true }
            },

            t_b3{ sb1 -> sb4 : getCCCredib,
                Guard{ true }
            },

            t_b4{ sb4 -> sb3 : ccCredib,
                Guard{ true }
```

```
                }
              }
            }
          }
}
```

Appendix B

Operation Semantics of Change Operators

This appendix contains all the operation semantics for all basic change operators specified in Table 3.2 in Chapter 3.

B.1 Protocol

Addition of a state:

$$\frac{\text{addState}(s_i)}{< S, s_0, F, M, T > \rightarrow < S \cup s_i, s_0, F, M, T >}$$

Removal of a state:

$$\frac{\text{removeState}(s_i)}{< S, s_0, F, M, T > \rightarrow < S - s_i, s_0, F, M, T >}$$

Addition of a transition:

$$\frac{\text{addTrans}(s_i, m, s_j)}{< S, s_0, F, M, T > \rightarrow < S, s_0, F, M, T \cup (s_i, s_j, m) >}$$

Removal of a transition:

$$\frac{\text{removeTrans}(s_i, m, s_j)}{< S, s_0, F, M, T > \rightarrow < S, s_0, F, M, T - (s_i, s_j, m) >}$$

Addition of a message:

$$\frac{\text{addMsg}(rec, sen, g)}{< S, s_0, F, M, T > \rightarrow < S, s_0, F, M \cup (rec, sen, g), T >}$$

Removal of a message:

$$\frac{\text{removeMsg}(rec, sen, g)}{< S, s_0, F, M, T > \rightarrow < S, s_0, F, M - (rec, sen, g), T >}$$

Note that there are three types of states in a graph, namely initial, final and normal. The final states and initial state are respectively a subset and an element of the set of states, for this reason they are not specified explicitly here.

B.2 Type

In the above we have given the operation semantics from a set-theoretic perspective. For type we do the same. However, because we defined it before as a grammar, we can not use the above directly. We therefore express a grammar as a single set containing only types (E). An type is then specified as: $e = (t, d, p, x, y, c)$ where t is the tag (name) of the element, d the datatype, p denotes the location between other elements (as defined by Xpath in Section 3.4.1), the minimum and maximum cardinality (x and y respectively) and $c \in \{seq, choice\}$ a structural constraint which can be either a sequence or a choice. Using this set, we define the operators as follows:
Addition of a type:

$$\frac{\text{addType}(\tau, d, p)}{E \rightarrow E \cup (\tau, d, p, 1, 1, seq)}$$

When adding a type, the constraints gets the standard values for the constraints, which is 1 for the cardinality and seq for the structural constraint.
Removal of a type:

$$\frac{\text{removeType}(\tau, p)}{E \cup (\tau, d, p, x, y, c) \rightarrow E \backslash (\tau, d, p, x, y, c)}$$

Update cardinality:

$$\frac{\text{updateCC}(x, y, p)}{\{E \cup (\tau, d, p, x', y', c)\} \rightarrow \{E \cup (\tau, d, p, x, y, c)\} \backslash (\tau, d, p, x', y', c)}$$

Update structural constraint:

$$\frac{\text{updateSC}(c^s, p)}{\{E \cup (\tau, d, p, x, y, c)\} \rightarrow \{E \cup (\tau, d, p, x, y, c^s)\} \backslash (\tau, d, p, x, y, c)}$$

Appendix C

Specification of Business Requirements

This appendix contains the specification of the example business rules and target protocol used throughout the thesis. They are in the format described in Section 5.5.

C.1 Business Rules

```
Bottomup {
    Schema{
        PeerList{BR2,BR3},
        TypeList{

        Acceptance[
            acceptance[xsd:bool]
        ],

        Rejected[
            rejected[xsd:bool]
        ]

        },
        MessageList{
            accept { BR2-> BR2: Acceptance},
            reject { BR3-> BR3: Rejected}
        }
```

```
},

Peerset{
    BR2{
        States{sbr21,sbr22},
        InitialState {sbr21},
        FinalStates{sbr22},
        TransitionRelation{
         t_br21{ sbr21-> sbr22: accept,
          Guard{ $availability/Availability[available = true] and
                  $getAccCredib/GetAccCredib[balance > 0] and
                  $shipmentQuote/ShipmentQuote[shipmentID > 0] =>
                 $accept[
                   //acceptance:= true
                 ]}
           }

        }
    },

    BR3{
        States{sbr31,sbr32},
        InitialState {sbr31},
        FinalStates{sbr32},
        TransitionRelation{
         t_br31{ sbr31-> sbr32: reject,
          Guard{ $availability/Availability[available = false] or
                 $getAccCredib/GetAccCredib[balance < 0] or
                 $shipmentQuote/ShipmentQuote[shipmentID <= 0] =>
                 $reject[
                   //rejected:= true
                 ]}
            }
        }
    }


}
```

C.2 Target Protocol

```
Bottomup {
    Schema{
        PeerList{Customer,Retailer},
        TypeList{

        GetCatalog[
            getCatalogRequest[xsd:bool]
        ],

        Catalog[
            productList[
                product  [xsd:string],
                productID[xsd:int],
                price    [xsd:int]
            ]{1,10}
        ],

        Order[
            customer[
                name [xsd:string],
                address[
                    streetName [xsd:string],
                    zipcode [xsd:string],
                    city    [xsd:string],
                    country [xsd:string]
                ],
                bankaccount[
                    accountID  [xsd:int],
                    accountType[xsd:int]
                ]
            ],
            productList[
                product [xsd:string],
                productID[xsd:int],
                price    [xsd:int],
                quantity [xsd:int]
            ]{1,10}
```

```
        ],

        OrderAccept[
            acceptance [xsd:bool]
        ],

        OrderReject[
            rejected [xsd:bool]
        ]

        },
        MessageList{
            getCatalog { Customer -> Retailer : GetCatalog },
            catalog { Retailer -> Customer : Catalog },
            order { Customer -> Retailer : Order },
            orderAccept { Retailer -> Customer : OrderAccept },
            orderReject { Retailer -> Customer : OrderReject }
        }
    },

    Peerset{
        Customer{
            States{sc1,sc2,sc3,sc4,sc5,sc6},
            InitialState {sc1},
            FinalStates{sc5,sc6},
            TransitionRelation{
                t_c1{ sc1 -> sc2 : getCatalog,
                    Guard{ true }
                },

                t_c2{ sc2 -> sc3 : catalog,
                    Guard{ true }
                },

                t_c3{ sc3 -> sc4 : order,
                    Guard{ true }
                },

                t_c4{ sc4 -> sc5 : orderReject,
```

```
                    Guard{ $reject/Rejected[rejected = true]}
                },

                t_c5{ sc4 -> sc6 : orderAccept,
                    Guard{ $accept/Acceptance[acceptance = true]}
                }
        }
    }
} }
```

Appendix D

Specification of the Retailer

This appendix contains of the our example orchestrator, the retailer, as created by the composition operator discussed in Chapter 4. It is in the format described in Section 5.5. Note that we omit here the specifications of services and the typelist as they are identical to the ones that were given in Appendix A and Appendix C.

```
Bottomup{
    Schema{
        PeerList { Retailer, Customer, Shipper, Bank, Inventory}

        TypeList {}

        MessageList {
          accept { Retailer -> Retailer: Acceptance},
          shipmentQuote { Shipper-> Retailer: ShipmentQuote},
          getShipmentQuote {Retailer-> Shipper: GetShipmentQuote},
          getAccCredib { Bank-> Retailer: GetAccCredib},
          accCredib { Retailer-> Bank: AccCredib},
          reject { Retailer-> Retailer: Rejected},
          availability { Inventory-> Retailer: Availability},
          checkAvailability { Retailer-> Inventory: CheckAvailability},
          catalogResponse { Inventory-> Retailer: CatalogResponse},
          catalogRequest {Retailer-> Inventory: CatalogRequest},
          getCatalog {Customer-> Retailer: GetCatalog},
          catalog { Retailer-> Customer: Catalog},
          order { Customer-> Retailer: Order},
          orderAccept {Retailer-> Customer: OrderAccept},
```

```
      orderReject { Retailer-> Customer: OrderReject}
    }
},

Peerset{
    Retailer{
        States{sc1, sc2, si2, si3, sc3, sc4, si4, si5, sbr32,
                              sc5, sb2, sb3, sh2, sh3, sbr22,sc6}
        InitialState{sc1}
        FinalStates{sc5, sc6}
        TransitionRelation{
            t_c1{ sc1 -> sc2: catalogRequest
                Guard { true }
            },

            ti1{ sc2 -> si2: getCatalog
                Guard { true=>
                          getCatalog [
                           /GetCatalogRequest/getCatalogRequest :=
                                   /GetCatalog/getCatalogRequest
                          ]
                }
            },

            ti2{ si2 -> si3: catalogResponse
                Guard { true }
            },

            t_c2{ si3 -> sc3: catalog
                Guard { true=>
                          catalog [
                            /Catalog/productList/product :=
                                /CatalogResponse/productList/product
                            /Catalog/productList/productID :=
                                /CatalogResponse/productList/productID
                            /Catalog/productList/price :=
                                /CatalogResponse/productList/price
                          ]
                }
```

```
},

t_c3{ sc3 -> sc4: order
    Guard { true }
},

ti3{ sc4 -> si4: checkAvailability
    Guard { true=>
              checkAvailability [
                /CheckAvailability/productList/product :=
                    /CatalogResponse/productList/product
                /CheckAvailability/productList/productID :=
                    /CatalogResponse/productList/productID
                /CheckAvailability/productList/quantity :=
                    /Order/productList/quantity
              ]
    }
},

ti4{ si4 -> si5: availability
        Guard { true }
},

ot_7{ si5 -> sbr32: reject
    Guard { (( $availability/Availability
                          [( available == false) ] ||
                $accCredib/AccCredib
                          [( balance < 0) ]) ||
                $shipmentQuote/ShipmentQuote
                          [( shipmentID <= 0) ]) =>
                  reject [
                      //rejected :=  true
                  ]
    }
},

t_c4{ sbr32 -> sc5: orderReject
  Guard { $reject/Rejected [( rejected == true) ]=>
          orderReject [
```

```
            /OrderReject/rejected := /Rejected/rejected
          ]
  }
},

t_b1{ si5 -> sb2: getAccCredib
  Guard { true=>
           getAccCredib [
              /GetAccCredib/bankaccount/accountID :=
                  /Order/payment/bankaccount/accountID
              /GetAccCredib/bankaccount/accountType :=
                  /Order/payment/bankaccount/accountType
              /GetAccCredib/name := /Order/customer/name
           ]
  }
},

t_b2{ sb2 -> sb3: accCredib
    Guard { true }
},

ot_11{ sb3 -> sbr32: reject
  Guard { (( $availability/Availability
                        [( available == false) ] ||
                $accCredib/AccCredib
                        [( balance < 0) ]) ||
                $shipmentQuote/ShipmentQuote
                        [( shipmentID <= 0) ]) =>
                  reject [
                     //rejected :=  true
                  ]
  }
},

t_sh1{ sb3 -> sh2: getShipmentQuote
  Guard { true=>
           getShipmentQuote [
             /GetShipmentQuote/productList/productID :=
               /CatalogResponse/productList/productID
```

```
                /GetShipmentQuote/productList/quantity :=
                  /Order/productList/quantity
                /GetShipmentQuote/address/streetName :=
                  /Order/customer/address/streetName
                /GetShipmentQuote/address/zipcode :=
                  /Order/customer/address/zipcode
                /GetShipmentQuote/address/city :=
                  /Order/customer/address/city
                /GetShipmentQuote/address/country :=
                  /Order/customer/address/country
                ]
    }
},

t_sh2{ sh2 -> sh3: shipmentQuote
  Guard { true }
},

ot_14{ sh3 -> sbr22: accept
  Guard {(($availability/Availability
                    [( availibility == true) ] &&
            $accCredib/AccCredib
                    [( balance < 0) ]) &&
            $shipmentQuote/ShipmentQuote
                    [( shipmentID < 0) ])=>
               accept [
                    //acceptance :=  true
               ]
    }
},

t_c5{ sbr22 -> sc6: orderAccept
  Guard { $accept/Acceptance
                    [( acceptance == true) ]=>
               orderAccept [
                    /OrderAccept/acceptance :=
                        /Acceptance/acceptance
          ]
  }
```

```
            },

            ot_16{ sh3 -> sbr32: reject
              Guard { (( $availability/Availability
                                    [( available == false) ] ||
                           $accCredib/AccCredib
                                    [( balance < 0) ]) ||
                           $shipmentQuote/ShipmentQuote
                                    [( shipmentID <= 0) ]) =>
                            reject [
                                //rejected := true
                        ]
                 }
            }
        }
    }
}}
```

Bibliography

Ackfor, R. & Emery, F. (1972). *On Purposeful Systems.* Adline Atherton.

Ader, M. (2004). *Workflow Comparative Study.* Waria.

Agarwal, V., Chafle, G., Mittal, S., & Srivastava, B. (2008). Understanding Approaches for Web Service Composition and Execution. In *Compute '08: Proceedings of the 1st Bangalore annual Compute conference* (pp. 1–8).

Aggarwal, R., Verma, K., Miller, J., & Milnor, W. (2004). Constraint Driven Web Service Composition in METEOR-S. In *SCC '04: Proceedings of IEEE International Conference on Services Computing* (pp. 23–30).

Aït-Bachir, A., Dumas, M., & Fauvet, M.-C. (2009). Detecting Behavioural Incompatibilities between Pairs of Services. In *WESOA '08: Fourth International Workshop on Engineering Service-Oriented Applications* (pp. 79–90).

Aksit, M. & Choukair, Z. (2003). Dynamic, Adaptive and Reconfigurable Systems Overview and Prospective Vision. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems* (pp. 84–90).

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2004). *Web Services: Concepts, Architectures and Applications.* Springer-Verlag.

Andrews, T., Dholakia, H., Goland, Y., Klein, J., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., & Weerawarana, S. (2003). Business Process Execution Language for Web Services, Version 1.1.

Andrikopoulos, V., Benbernou, S., & Papazoglou, M. P. (2008). Managing the Evolution of Service Specifications. In *CAISE '08:Proceedings of the*

20th international conference on Advanced Information Systems Engineering (pp. 359–374).

Andrikopoulos, V., Benbernou, S., & Papazoglou, M. P. (2009). Evolving Services from a Contractual Perspective. In *CAISE '09: Proceedings of the 21st International Conference on Advanced Information Systems Engineering* (pp. 290–304).

Anthony, R. J. (2009). Policy-based Autonomic Computing with Integral Support for Self-Stabilisation. *International Journal of Autonomic Computing*, 1(1), 1–33.

Antonellis, V. D., Melchiori, M., Santis, L. D., Mecella, M., Mussi, E., Pernici, B., & Plebani, P. (2006). A Layered Architecture For Flexible Web Service Invocation. *Software: Practice and Experience*, 36(2), 191–223.

Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., & Plebani, P. (2007). PAWS: A Framework for Executing Adaptive Web-Service Processes. *IEEE Software*, 24(6), 39–46.

Arsanjani, A. (2005). Toward a Pattern Language for Service-Oriented Architecture and Integration, Part 2: Service Composition. `http://www.ibm.com/developerworks/webservices/library/ws-soa-soi2/`.

Ashby, W. R. (1960). *Design for a Brain, 2nd Edition*. Chapman & Hall.

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), 11–33.

Bae, J., Bae, H., Kang, S.-H., & Kim, Y. (2004). Automatic Control of Workflow Processes Using ECA Rules. *IEEE Transactions on Knowledge and Data Engineering*, 16(8), 1010–1023.

Baldoni, M., Baroglio, C., Chopra, A. K., Desai, N., Patti, V., & Singh, M. P. (2009). Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In *AAMAS '09: Proceedings of the 8th International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 843–850).

Bandinelli, S., Nitto, E. D., & Fuggetta, A. (1994). Policies and Mechanisms to Support Process Evolution in PSEEs. In *ICSP3 : Proceedings of the 3rd IEEE International Conference on the Software Process* (pp. 9–20).

Baresi, L., Guinea, S., & Pasquale, L. (2007a). Self-healing BPEL Processes with Dynamo and the JBoss Rule Engine. In *ESSPE '07: International Workshop on Engineering of Software Services for Pervasive Environments* (pp. 11–20).

Baresi, L., Guinea, S., Pistore, M., & Trainotti, M. (2009). Dynamo + Astro: An Integrated Approach for BPEL Monitoring. In *ICWS '09:Proceedings of the 2009 IEEE International Conference on Web Services* (pp. 230–237).

Baresi, L., Nitto, E. D., Ghezzi, C., & Guinea, S. (2007b). A Framework for the Deployment of Adaptable Web Service Compositions. *Service Oriented Computing and Applications*, 1(1), 75–91.

Baryannis, G., Carro, M., Danylevych, O., Dustdar, S., Karastoyanova, D., Kritikos, K., Leinter, P., Rosenberg, F., & Wetzstein, B. (2008). *Overview of the State of the Art in Composition and Coordination of Services.* Deliverable PO-JRA-2.2.1, S-Cube Consortium.

Becker, K., Lopes, A., Milojicic, D. S., Pruyne, J., & Singhal, S. (2008). Automatically Determining Compatibility of Evolving Services. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services* (pp. 161–168).

Bellwood, T., Capell, S., Clement, L., Colgrave, J., Dovey, M. J., Feygin, D., Hately, A., Kochman, R., Macias, P., Novotny, M., Paolucci, M., von Riegen, C., Rogers, T., Sycara, K., Wenzel, P., & Wu, Z. (2004). Universal Description Discovery Integration (Version 3). `http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm`.

Benatallah, B., Casati, F., Grigori, D., Nezhad, H. R. M., & Toumani, F. (2005). Developing Adapters for Web Services Integration. In *CAISE '05: Proceedings of the 17th International Conference on Advanced Information Systems Engineering* (pp. 415–429).

Benatallah, B., Casati, F., & Toumani, F. (2004a). Analysis and Management of Web Service Protocols. In *ER '04: Proceedings of the 23rd International Conference on Conceptual Modeling* (pp. 524–541).

Benatallah, B., Casati, F., & Toumani, F. (2004b). Web Service Conversation Modeling: A Cornerstone for E-Business Automation. *IEEE Computer*, 8(1), 46–54.

Benatallah, B., Casati, F., & Toumani, F. (2006). Representing, Analysing and Managing Web service Protocols. *Data and Knowledge Engineering*, 58, 347–357.

Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., & Mecella, M. (2003). Automatic Composition of e-Services that Export their Behavior. In *ICSOC '03: Proceedings of the 1st International Conference on Service Oriented Computing* (pp. 43–58).

Berardi, D., Calvanese, D., Giacomo, G. D., Hull, R., & Mecella, M. (2005a). Automatic Composition of Transition-based Semantic Web Services with Messaging. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Databases* (pp. 613–624).

Berardi, D., Calvanese, D., Giacomo, G. D., Hull, R., & Mecella, M. (2005b). Automatic Composition of Web Services in Colombo. In *SEBD '05: Proceedings of the Thirteenth Italian Symposium on Advanced Database Systems* (pp. 8–15).

Bernstein, P. A. (2003). Applying Model Management to Classical Meta Data Problems. In *CIDR '03: Proceedings of the First Biennial Conference on Innovative Data Systems Research*.

Bernstein, P. A., Halevy, A. Y., & Pottinger, R. A. (2000). A Vision for Management of Complex Models. *ACM SIGMOD Record*, 29(4), 55–63.

Berre, A.-J., Hahn, A., Akehurst, D., Bezivin, J., Tsalgatidou, A., Vermaut, F., Kutvonen, L., & Linington, P. F. (2004). State-of-the-art for Interoperability Architecture Approaches. EU INTEROP Network of Excellence Deliverable D9.1.

Betin-Can, A., Bultan, T., & Fu, X. (2005). Design for Verification for Asynchronously Communicating Web Services. In *WWW '05: Proceedings of the 14th international conference on World Wide Web* (pp. 750–759).

Bézivin, J. (2005). On the Unification Power of Models. *Software and System Modeling*, 4(2), 171–188.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., & Orchard, D. (2004). *Web Services Architecture.* Working notes, W3C.

Brand, D. & Zafiropulo, P. (1983). On Communicating Finite-State Machines. *Journal of the ACM*, 30(2), 323–342.

BRG (2000). Defining Business Rules, What are They Really? `http://www.businessrulesgroup.org/first_paper/br01c0.htm`.

Brogi, A., Canal, C., Pimentel, E., & Vallecillo, A. (2004). Formalizing Web Service Choreographies. *Electronic Notes in Theoretical Computer Science*, 105(10), 73–94.

Brown, A., Fuchs, M., Robie, J., & Wadler, P. (2001). MSL. A Model for W3C XML Schema. In *WWW '01: Proceedings of the 10th International Conference on World Wide Web* (pp. 191–200).

Broy, M., Leuxner, C., Sitou, W., Spanfelner, B., & Winter, S. (2009). Formalizing the Notion of Adaptive System Behavior. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing* (pp. 1029–1033).

Bruning, S., Weissleder, S., & Malek, M. (2007). A Fault Taxonomy for Service-Oriented Architecture. In *HASE '07: Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium* (pp. 367–368).

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel, G. (2005). Towards a Taxonomy of Software Change. *Journal of Software Maintenance And Evolution: Research and Practice*, 17, 309–332.

Bullard, V., Murray, B., & Wilson, K. (2006). *An Introduction to WSDM.* Committee Draft. wsdm-1.0-intro-primer-cd-01, OASIS.

Bullard, V. & Vambenepe, W. (2006). *Web Services Distributed Management: Management Using Web Services (MUWS1.1) Part 1.* Oasis committee draft, OASIS.

Bultan, T., Fu, X., Hull, R., & Su, J. (2003). Conversation Specification: a New Approach to Design and Analysis of e-Service Composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web* (pp. 403–410).

Bunke, H. (2000). Graph Matching: Theoretical Foundations, Algorithms, and Applications. In *VI '2000: Proceedings of the International Conference on Vision Interface* (pp. 82–88).

Calvanese, D., Giacomo, G. D., Lenzerini, M., Mecella, M., & Patrizi, F. (2008). Automatic Service Composition and Synthesis: the Roman Model. *IEEE Data Engineering Bulletin*, 31(3), 18–22.

Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (1983). Machine Learning: A Historical and Methodological Analysis. *AI Magazine*, 4(3), 69–79.

Cardoso, J. (2006). Complexity Analysis of BPEL Web Processes. *Software Process: Improvement and Practice*, 12(1), 35–49.

Casati, F., Ceri, S., Pernici, B., & Pozzi, G. (1996). Workflow Evolution. In *ER '96: Proceedings of the 15th International Conference on Conceptual Modeling* (pp. 438–455).

Casati, F., Ceri, S., Pernici, B., & Pozzi, G. (1998). Workflow Evolution. *Data & Knowledge Engineering*, 24(3), 211–238.

Casati, F., Shan, E., Dayal, U., & Shan, M.-C. (2003). Business-Oriented Management of Web Services. *Communications of the ACM*, 46(10), 55–60.

Chan, K. S., Bishop, J., Steyn, J., Baresi, L., & Guinea, S. (2007). A Fault Taxonomy for Web Service Composition. In *WESOA '07: Third International Workshop on Engineering Service-Oriented Applications: Analysis, Design, and Composition* (pp. 363–375).

Charfi, A., Dinkelaker, T., & Mezini, M. (2009). A Plug-in Architecture for Self-Adaptive Web Service Compositions. In *ICWS '09: Proceedings of 7th International Conference on Web Services* (pp. 35–42).

Charfi, A. & Mezini, M. (2004). Hybrid Web Service Composition: Business Processes meet Business Rules. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing* (pp. 30–38).

Cheng, S.-W., Garlan, D., & Schmerl, B. (2006). Architecture-Based Self-Adaptation in the Presence of Multiple Objectives. In *SEAMS '06: Proceedings of the 2006 International Workshop on Self-adaptation and Self-Managing Systems* (pp. 2–8).

Cibrán, M. A., Verheecke, B., Vanderperren, W., Suvée, D., & Jonckers, V. (2007). Aspect-oriented Programming for Dynamic Web Service Selection, Integration and Management. *World Wide Web*, 10(3), 211–242.

Cisco (2007). *Service Virtualization: Managing Change in a Service-Oriented Architecture*. White paper, CISCO.

Cobena, G., Abiteboul, S., & Marian, A. (2002). Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)* (pp. 41–52).

Colombo, M., Nitto, E. D., & Mauri, M. (2006). SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In *ICSOC '06: Proceedings of the 4th International Conference on Service-Oriented Computing* (pp. 191–202).

Cottrell, L. (2001). Passive vs Active Monitoring. `http://www.slac.stanford.edu/comp/net/wan-mon/passive-vs-active.html`.

Cox, D. E. & Kreger, H. (2005). Management of the Service-Oriented Architecture Life Cycle. *IBM Systems Journal*, 44(4), 709–726.

Curbera, F. (2007). Component Contracts in Service-Oriented Architectures. *Computer*, 40(11), 74–80.

Curbera, F., Duftler, M. J., Khalaf, R., Nagy, W. A., Mukhi, N., & Weerawarana, S. (2005). Colombo: Lightweight Middleware for Service-Oriented Computing. *IBM Systems Journal*, 44(4), 799–820.

Daniel, F. & Pernici, B. (2006). Insights into Web Service Orchestration and Choreography. *International Journal of E-Business Research*, 2(1), 58–77.

Dardenne, A., van Lamsweerde, A., & Fickas, S. (1993). Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20(1-2), 3–50.

De Backer, M., Snoeck, M., Monsieur, G., Lemahieu, W., & Dedene, G. (2009). A Scenario-based Verification Technique to Assess the Compatibility of Collaborative Business Processes. *Data and Knowledge Engineering*, 68(6), 531–551.

Delgado, N., Gates, A. Q., & Roach, S. (2004). A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Transactions on Software Engineering*, 30(12), 859–872.

Denaro, G., Pezze, M., Tosi, D., & Schilling, D. (2006). Towards Self-Adaptive Service-Oriented Architectures. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, Analysis, and Verification of Web Services and Applications* (pp. 10–16).

Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., & Pohl, K. (2008). A Journey to Highly Dynamic, Self-Adaptive Service-Based Applications. *Automated Software Engineering*, 15(3-4), 313–341.

DMTF (2008). *Web Services for Management (WS-Management) Specification.* Final Standard DSP0226, Distributed Management Task Force.

Dorn, C., Schall, D., & Dustdar, S. (2009). A Model and Algorithm for Self-Adaptation in Service-oriented Systems. In *ECOWS '09: Proceedings of the European Conference on Web Services.*

Dustdar, S., Goeschka, K. M., Truong, H.-L., & Zdun, U. (2009). Self-Adaptation Techniques for Complex Service-oriented Systems. In *Proceedings of the 5th IEEE International Conference on Next Generation Web Services Practices.*

Dustdar, S. & Schreiner, W. (2005). A Survey on Web Services Composition. *International Journal on Web and Grid Services*, 1(1), 1–30.

Ellis, C., Keddara, K., & Rozenberg, G. (1995). Dynamic Change within Workflow Systems. In *COCS '95: Proceedings of Conference on Organizational Computing Systems* (pp. 10–21).

Ernst, M. D., Lencevicius, R., & Perkins, J. H. (2006). Detection of Web Service Substitutability and Composability. In *WS-MaTe '06: Proceeding of the International Workshop on Web Services — Modeling and Testing* (pp. 123–135).

Fabro, M. D. D., Bézivin, J., Jouault, F., & Valduriez, P. (2005). Applying Generic Model Management to Data Mapping. In *BDA '05:Proceedings of the Journées Bases de Données Avancées*.

Fallside, D. & Walmsley, P. (2004). *XML Schema Part 0: Primer Second Edition*. W3C.

Fang, R., Lam, L., Fong, L., Frank, D., Vignola, C., Chen, Y., & Du, N. (2007). A Version-aware Approach for Web Service Directory. In *ICWS '07: Proceedings of the IEEE International Conference on Web Services* (pp. 406–413).

Farrell, J. & Lausen, H. (2007). *Semantic Annotations for WSDL and XML Schema*. W3C Recommendation, W3C.

Fensel, D. & Bussler, C. (2002). The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2), 113–137.

Fu, X. (2004). *Formal Specification and Verification of Asynchronously Communicating Web Services*. PhD thesis, University of California.

Fu, X., Bultan, T., & Su, J. (2004a). Analysis of Interacting BPEL Web Services. In *WWW '04: Proceedings of the 13th International Conference on World Wide Web* (pp. 621–630).

Fu, X., Bultan, T., & Su, J. (2004b). Conversation Protocols: a Formalism for Specification and Verification of Reactive Electronic Services. *Theoretical Computer Science*, 328(1-2), 19–37.

Fu, X., Bultan, T., & Su, J. (2004c). *Model Checking Interactions of Composite Web Services*. Technical report, University of California at Santa Barbara.

Fu, X., Bultan, T., & Su, J. (2004d). Model Checking XML Manipulating Software. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 252–262).

Fu, X., Bultan, T., & Su, J. (2004e). WSAT: A Tool for Formal Analysis of Web Services. In *CAV '04: Proceedings of the 16th International Conference on Computer Aided Verification* (pp. 510–514).

Ganek, A. G. & Corbi, T. A. (2003). The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1), 5–18.

Georgakopoulos, D., Hornick, M., & Sheth, A. (1995). An Overview of Workflow Management: from Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2), 119–153.

Gerede, Ç. E., Hull, R., Ibarra, O. H., & Su, J. (2004). Automated Composition of e-Services: Lookaheads. In *ICSOC '04: Proceedings of the 2nd International Conference on Service Oriented Computing* (pp. 252–262).

Gjørven, E., Rouvoy, R., & Eliassen, F. (2008). Cross-Layer Self-Adaptation of Service-Oriented Architectures. In *MW4SOC '08: Proceedings of the 3rd Workshop on Middleware for Service Oriented Computing* (pp. 37–42).

Greenfield, J. & Short, K. (2004). *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. John Wiley & Sons.

Hamadi, R. & Benatallah, B. (2003). A Petri Net-based Model for Web Service Composition. In *ADC '03: Proceedings of the 14th Australasian Database Conference* (pp. 191–200).

Hamadi, R. & Benatallah, B. (2004). Recovery Nets: Towards Self-Adaptive Workfow Systems. In *WISE '04: Proceedings of the 5th International Conference on Web Information Systems Engineering* (pp. 439–453).

Hamadi, R. & Benatallah, B. (2005). Dynamic Restructuring of Recovery Nets. In *ADC '05: Proceedings of the 16th Australasian Database Conference* (pp. 37–46).

Hamadi, R., Benatallah, B., & Medjahed, B. (2008). Self-adapting Recovery Nets for Policy-driven Exception Handling in Business Processes. *Distributed and Parallel Databases*, 23(1), 1–44.

Harel, D., Kozen, D., & Tiuryn, J. (2000). *Dynamic Logic*. MIT Press.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *Management Information Systems Quarterly*, 28, 75–105.

Hiel, M. & Weigand, H. (2006). Requirements On the Use Of Goal-directed Imitation for Self-adaptation. In *SAACS '06: Proceedings of the 4th International Workshop on Self-Adaptive and Autonomic Computing Systems* (pp. 98–103).

Hiel, M. & Weigand, H. (2009). Interoperability Changes in an Adaptive Service Orchestration. In *ICWS '09: Proceedings of 7th International Conference on Web Services* (pp. 351–358).

Hiel, M., Weigand, H., & van den Heuvel, W.-J. (2008a). An Adaptive Service Oriented Architecture. In *IESA '08: Proceedings of the 4th International Conference Interoperability for Enterprise Software and Applications* (pp. 197–208). (Best Young Researcher Paper Award).

Hiel, M., Weigand, H., & van den Heuvel, W.-J. (2008b). An Adaptive Service Oriented Architecture. *International Journal of Interoperability in Business Information Systems*, 2(3), 37–51.

Hielscher, J., Metzger, A., & Kazhamiakin, R. (2009). *Taxonomy of Adaptation Principles and Mechanisms*. Contractual Deliverable CD-JRA-1.2.2, S-Cube Consortium.

Hindriks, K. V., de Boer, F. S., van der Hoek, W., & Meyer, J.-J. C. (1999). Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems Journal*, 2, 357–401.

Hinz, S., Schmidt, K., & Stahl, C. (2005). Transforming BPEL to Petri Nets. In *BPM' 05: Proceedings of the Third International Conference on Business Process Management* (pp. 220–235).

Holley, K., Channabasavaiah, K., & Tuggle, J. (2003). Migrating to a Service-Oriented Architecture, IBM DeveloperWorks.

Holzmann, G. J. (2004). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley.

Horn, P. (2001). *Autonomic Computing: IBM's Perspective on the State of Information Technology*. Manifesto, IBM Research.

Hull, R., Llirbat, F., Siman, E., Su, J., Dong, G., Kumar, B., & Zhou, G. (1999). Declarative Workflows that Support Easy Modification and Dynamic Browsing. In *WACC '99: Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration* (pp. 69–78).

Hull, R. & Su, J. (2005). Tools for Composite Web Services: A Short Overview. *SIGMOD Record*, 34(2), 86–95.

IBM (2003). An Architectural Blueprint for Autonomic Computing.

International Organization for Standardization and International Electrotechnical Comission (2001). Software engineering – Product quality – Part 1: Quality Model. ISO/IEC 9126-1:2001.

Joeris, G. & Herzog, O. (1999). Towards Flexible and High-Level Modeling and Enacting of Processes. In *CAiSE '99: Proceedings of the 11th International Conference on Advanced Information Systems Engineering* (pp. 88–102).

Jones, C. (2006). The Economics of Software Maintenance In The Twenty First Century. Unpublished Manuscript.

Kaminski, P., Litoiu, M., & Müller, H. (2006a). A Design Technique for Evolving Web Services. In *CASCON '06: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research* (pp.̃23).

Kaminski, P., Müller, H., & Litoiu, M. (2006b). A Design for Adaptive Web Service Evolution. In *SEAMS '06: Proceedings of the 2006 international Workshop on Self-Adaptation and Self-Managing Systems* (pp. 86–92).

Karastoyanova, D. & Leymann, F. (2009). BPELŠnŠAspects: Adapting Service Orchestration Logic. In *ICWS '09: Proceedings of 7th International Conference on Web Services*

Kennedy, J, R. E. (2001). *Swarm Intelligence*. Morgan Kaufmann.

Kenney, F. (2008). Ahh Shucks, SOA is a Failure. `http://blogs.gartner.com/frank_kenney/2008/11/12/ahh-shucks-soa-is-a-failure/`.

Kensche, D., Quix, C., Li, X., & Li, Y. (2007). GeRoMeSuite: a System for Holistic Generic Model Management. In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases* (pp. 1322–1325).

Kephart, J. & Chess, D. (2003). The Vision of Autonomic Computing. *Computer*, 36(1), 41 – 50.

Khalaf, R., Keller, A., & Leymann, F. (2006). Business Processes for Web Services: Principles and Applications. *IBM Systems Journal*, 45(2), 425–446.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-Oriented Programming. In *ECOOP '97: Proceedings of the European Conference on Object-Oriented Programming* (pp. 220–242).

Knoll, K. & Jarvenpaa, S. L. (1994). Information Technology Alignment or "fit" in Highly Turbulent Environments: the Concept of Flexibility. In *SIGCPR '94: Proceedings of the 1994 Computer Personnel Research Conference on Reinventing IS: Managing Information Technology in Changing Organizations* (pp. 1–14).

Kokar, M. M., Baclawski, K., & Eracar, Y. A. (1999). Control Theory-Based Foundations of Self-Controlling Software. *IEEE Intelligent Systems*, 14(3), 37–45.

Kongdenfha, W., Saint-Paul, R., Benatallah, B., & Casati, F. (2006). An Aspect-Oriented Framework for Service Adaptation. In *ICSOC '06: Proceeding of the 4th International Conference of Service Oriented Computing* (pp. 15–26).

Kreger, Sedukhin, V., Graham, & Murray (2005). Management Using Web Services a Proposed Architecture and Roadmap. `ftp://www6.software.ibm.com/software/developer/library/ws-mroadmap.pdf`.

Lehman, M. M. (1980). On Understanding Laws, Evolution and Conservation in the Large-Program Life Cycle. *Journal of Systems and Software*, 1, 213–221.

Leonardi, E., Hoai, T. T., Bhowmick, S. S., & Madria, S. (2007). DTD-Diff: A Change Detection Algorithm for DTDs. *Data & Knowledge Engineering*, 61(2), 384–402.

Lieberman, H. & Selker, T. (2000). Out of Context: Computer Systems that Adapt to, and Learn from, Context. *IBM Systems Journal*, 39(3-4), 617–632.

Liu, Y. (2009). A Process Modeling-based Approach for Web Service Management. In *ICWS '09: Proceedings of 7th International Conference on Web Services* (pp. 928–935).

Liu, Y., Müller, S., & Xu, K. (2007). A Static Compliance-Checking Framework for Business Process Models. *IBM Systems Journal*, 46(2), 335–361.

Lohmann, N., Verbeek, E., Ouyang, C., Stahl, C., & van der Aalst, W. M. P. (2009). Comparing and Evaluating Petri Net Semantics for BPEL. *International Journal of Business Process Integration and Management*, 4, 60–73.

Ludwig, H., Laredo, J., Bhattacharya, K., Pasquale, L., & Wassermann, B. (2009). REST-Based Management of Loosely Coupled Services. In *WWW '09: Proceedings of the 18th International Conference on World Wide Web* (pp. 931–940).

Madhusudan, T., Zhao, J. L., & Marshall, B. (2004). A Case-Based Reasoning Framework for Workflow Model Management. *Data & Knowledge Engineering*, 50(1), 87–115.

Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *OOPSLA '87: Proceedings of the ACM Conference on Object-Oriented Languages* (pp. 147 – 155).

Manes, A. T. (2009). SOA is Dead; Long Live Services. `http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html`.

Mariani, L. (2003). A Fault Taxonomy for Component-Based Software. In *TACoS'03: Proceedings of the International Workshop on Test and Analysis of Component-Based Systems (Satellite Event of ETAPS 2003)* (pp. 55–65).

Martens, A. (2003). On Compatibility of Web Services. *Petri Net Newsletter, Special Interest Groups on Petri Nets and Related Systems Models, Oct. 2003, Gesellschaft für Informatik e.V.*, 65, 12–20.

Martin, P., Powley, W., Wilson, K., Tian, W., Xu, T., & Zebedee, J. (2007). The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services. In *SEAMS '07: Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems* (pp.9̃).

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., & Cheng, B. H. C. (2004a). *A Taxonomy of Compositional Adaptation*. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, Michigan.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., & Cheng, B. H. C. (2004b). Composing Adaptive Software. *IEEE Computer*, 737(7), 56–64.

Medjahed, B., Benatallah, B., Bouguettaya, A., & Elmagarmid, A. K. (2004). Webbis: An Infrastructure For Agile Integration Of Web Services. *International Journal of Cooperative Information Systems*, 13(2), 121–158.

Medjahed, B., Bouguettaya, A., & Elmagarmid, A. K. (2003). Composing Web Services on the Semantic Web. *The International Journal on Very Large Data Bases*, 12(4), 333–351.

Melnik, S. (2004). *Generic Model Management: Concepts and Algorithms*. PhD thesis, University of Leipzig.

Melnik, S., Rahm, E., & Bernstein, P. A. (2003a). Developing Metadata-Intensive Applications with Rondo. *Journal of Web Semantics*, 1(1), 47–74.

Melnik, S., Rahm, E., & Bernstein, P. A. (2003b). Rondo: a Programming Platform for Generic Model Management. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (pp. 193–204).

Milanovic, N. & Malek, M. (2004). Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6), 51–59.

Mitchell, T. (1997). *Machine Learning.* McGraw Hill.

Moser, O., Rosenberg, F., & Dustdar, S. (2008). Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *WWW '08: Proceeding of the 17th international conference on World Wide Web* (pp. 815–824).

Muhl, G., Fiege, L., & Pietzuch, P. R. (2006). *Distributed Event-Based Systems.* Springer.

Müller, R., Greiner, U., & Rahm, E. (2004). AGENT WORK: a Workflow System Supporting Rule-Based Workflow Adaptation. *Data & Knowledge Engineering*, 51(2), 223–256.

Muth, P., Wodtke, D., Weissenfels, J., Weikum, G., & Kotz-Dittrich, A. (1997). Enterprise-Wide Workflow Management Based on State and Activity Charts. In *NATO Advanced Study Institute on Workflow Management Systems and Interoperability.*

Nezhad, H. R. M., Benatallah, B., Martens, A., Curbera, F., & Casati, F. (2007). Semi-Automated Adaptation of Service Interactions. In *WWW '07: Proceedings of the 16th International Conference on World Wide Web* (pp. 993–1002).

OASIS (2004). *Web Services Distributed Management: Management of Web Services (WSDM-MOWS) 1.0.* Oasis committee draft, Organization for the Advancement of Structured Information Standards.

OASIS (2006). *Reference Model for Service Oriented Architecture.* Official oasis standard, Organization for the Advancement of Structured Information Standards.

O'Brien, L., Merson, P., & Bass, L. (2007). Quality Attributes for Service-Oriented Architectures. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments* (pp.$\tilde{3}$).

OMG (2003). *MDA Guide Version 1.0.1.* Technical report, Object Management Group.

OMG (2009). *Service oriented architecture Modeling Language (SoaML) Version 1.0 - Beta 1.* FTF Beta 1 ptc/2009-04-01, Object Management Group.

Orchard, D. (2006). A Theory of Compatible Versions. `http://www.xml.com/pub/a/2006/12/20/a-theory-of-compatible-versions.html`.

Orriëns, B. (2007). *On The Development and Management of Adaptive Business Collaborations*. PhD thesis, Tilburg University.

Orriëns, B., Yang, J., & Papazoglou, M. P. (2003). Model Driven Service Composition. In *ICSOC' 03: Proceedings of the 1st International Conference on Service Oriented Computing* (pp. 75–90).

Papazoglou, M. (2008). The Challenges of Service Evolution. In *CAiSE '08: Proceedings of The 20th International Conference on Advanced Information Systems Engineering* (pp. 1–15).

Papazoglou, M. P. (2003). Service-Oriented Computing: Concepts, Characteristics and Directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering* (pp. 3–12).

Papazoglou, M. P. (2005). Extending the Service Oriented Architecture. *Business Integration Journal*, (pp. 18–21).

Papazoglou, M. P. & Georgakapoulos, G. (2003). Introduction to the Special Issue about Service-Oriented Computing. *Communications of the ACM*, 46(10), 24–28.

Papazoglou, M. P. & Ribbers, P. M. (2006). *e-Business: Organizational and Technical Foundations*. Wiley.

Papazoglou, M. P., Traverso, P., Dustdar, S., & Leymann, F. (2007). Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 7, 64 – 71.

Papazoglou, M. P. & van den Heuvel, W. (2007). Service Oriented Architectures: Approaches, Technologies and Research Issues. *The International Journal on Very Large Data Bases*, 16(3), 389–415.

Papazoglou, M. P. & van den Heuvel, W.-J. (2005). Web Services Management: A Survey. *IEEE Internet Computing*, (pp. 58–64).

Park, J. & Kim, K. (2009). Hyperlinking the Work for Self-Management of Flexible Workflows. *Communications of the ACM*, 52(6), 113–117.

Parnas, D. L. (1994). Software Aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering* (pp. 279–287).

Pathak, J., Basu, S., & Honavar, V. (2007). On Context-Specific Substitutability of Web Services. In *ICWS'07: Proceedings of the IEEE International Conference on Web Services* (pp. 192–199).

Pathak, J., Basu, S., Lutz, R., & Honavar, V. (2006a). Parallel Web Service Composition in MoSCoE: A Choreography-based Approach. In *ECOWS '06: Proceedings of the 4th IEEE European Conference on Web Services* (pp. 3–12).

Pathak, J., Basu, S., Lutz, R., & Honavar, V. (2006b). Selecting and Composing Web Services through Iterative Reformulation of Functional Specifications. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence* (pp. 445–454).

Pathak, J., Basu, S., Lutz, R. R., & Honavar, V. (2008). MOSCOE: an Approach for Composing Web Services through Iterative Reformulation of Functional Specifications. *International Journal on Artificial Intelligence Tools*, 17, 109–138.

Peltz, C. (2003). Web Services Orchestration and Choreography. *IEEE Computer*, 36(10), 46–52.

Pesic, M., Schonenberg, M. H., Sidorova, N., & van der Aalst, W. M. P. (2007). Constraint-Based Workflow Models: Change Made Easy. In *Proceedings of the OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007* (pp. 77–94).

Pesic, M. & van der Aalst, W. M. P. (2006). A Declarative Approach for Flexible Business Processes Management. In *BPM '06: Proceedings of the Business Process Management Workshops* (pp. 169–180).

Pistore, M., Marconi, A., Bertoli, P., & Traverso, P. (2005a). Automated Composition of Web Services by Planning at the Knowledge Level. In *IJCAI'05: Proceedings of the 19th International Joint Conference on Artificial intelligence* (pp. 1252–1259).

Pistore, M., Traverso, P., Bertoli, P., & Marconi, A. (2005b). Automated Synthesis of Composite BPEL4WS Web Services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services* (pp. 293–301).

Ponge, J., Toumani, F., Benatallah, B., & Casati, F. (2007). Fine-grained Compatibility and Replaceability Analysis of Timed Web Service Protocols. In *Proceedings of the 26th International Conference on Conceptual Modeling (ER'07)*.

Ponnekanti, S. R. & Fox, A. (2002). SWORD: A Developer Toolkit for Web Service Composition. In *WWW '02: Proceedings of the 11th International World Wide Web Conference*.

Ponnekanti, S. R. & Fox, A. (2004). Interoperability Among Independently Evolving Web Services. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware* (pp. 331–351).

Quartel, D. A., Steen, M. W., Pokraev, S., & Sinderen, M. J. (2007). COSMO: A Conceptual Framework for Service Modelling and Refinement. *Information Systems Frontiers*, 9(2-3), 225–244.

Rao, J. & Su, X. (2004). A Survey of Automated Web Service Composition Methods. In *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition* (pp. 43–54).

Regev, G., Bider, I., & Wegmann, A. (2007). Defining Business Process Flexibility with the Help of Invariants. *Software Process: Improvement and Practice*, 12(1), 65–79.

Reichert, M. & Dadam, P. (1998). Adept{flex}-Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems*, 10(2), 93–129.

Reichert, M. & Rinderle, S. (2006). On Design Principles for Realizing Adaptive Service Flows with BPEL. In *EMISA '06: Proceedings of the Workshop "Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen"* (pp. 133–146).

Rinderle, S. & Reichert, M. (2006). Data-Driven Process Control and Exception Handling in Process Management Systems. In *CAiSE '06: Proceedings of the 18th International Conference on Advanced Information Systems Engineering* (pp. 273–287).

Rinderle, S., Reichert, M., & Dadam, P. (2004). Correctness Criteria for Dynamic Changes in Workflow Systems: a Survey. *Data & Knowledge Engineering*, 50, 9–34.

Rohr, M., Giesecke, S., Hasselbring, W., Hiel, M., van den Heuvel, W.-J., & Weigand, H. (2006). A Classification Scheme for Self-adaptation Research. In *SOAS '06: Proceedings of the International Conference on Self-Organization and Autonomous Systems in Computing and Communications*.

Rolland, C., Kaabi, R. S., & Kraïem, N. (2007). On ISOA: Intentional Services Oriented Architecture. In *CAiSE'07: Proceedings of the 19th International Conference on Advanced Information Systems Engineering* (pp. 158–172).

Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., & Fensel, D. (2005). Web Service Modeling Ontology. *Applied Ontology*, 1, 77 – 106.

Rosenberg, F. & Dustdar, S. (2005). Business Rules Integration in BPEL - A Service-Oriented Approach. In *CEC '05: Proceedings of the Seventh IEEE International Conference on E-Commerce Technology* (pp. 476–479). Washington, DC, USA: IEEE Computer Society.

Rosenblum, D. S. & Wolf, A. L. (1997). A Design Framework for Internet-Scale Event Observation and Notification. *ACM SIGSOFT Software Engineering Notes*, 22(6), 344–360.

Russell, N., ter Hofstede, A., Edmond, D., & van der Aalst, W. (2004). *Workflow Data Patterns*. QUT Technical report FIT-TR-2004-01, Queensland University of Technology.

Russell, N., ter Hofstede, A., van der Aalst, W., & Mulyar, N. (2006). *Workflow Control-Flow Patterns : A Revised View*. BPM Center Report BPM-06-22, BPMcenter.org.

Ryu, S. H., Casati, F., Skogsrud, H., Benatallah, B., & Saint-Paul, R. (2008). Supporting The Dynamic Evolution of Web Service Protocols in Service Oriented Architectures. *ACM Transaction on the Web*, 2(2), 1–46.

Ryu, S. H., Saint-Paul, R., Benatallah, B., & Casati, F. (2007). A Framework for Managing the Evolution of Business Protocols in Web Services. In *APCCM '07: Proceedings of the 4th Asia-Pacific Conference on Comceptual Modelling* (pp. 49–59).

Sachs, W. M. (1999). *Adaptation Revisited: Concepts for Proactive Management of Change*. Working papers, Reims Management School.

Sachs, W. M. & Meditz, M. L. (1979). A Concept of Active Adaptation. *Human Relations*, 32(12), 1081–1093.

Sadiq, S. W., Orlowska, M. E., & Sadiq, W. (2005). Specification and Validation of Process Constraints for Flexible Workflows. *Information Systems*, 30(5), 349–378.

Sadiq, S. W., Sadiq, W., & Orlowska, M. E. (2001). Pockets of Flexibility in Workflow Specification. In *ER '01: Proceedings of the 20th International Conference on Conceptual Modeling* (pp. 513–526).

Sagasti, F. (1970). A Conceptual and Taxonomic Framework for the Analysis of Adaptive Behavior. *General Systems*, XV, 151–160.

Salaün, G., Bordeaux, L., & Schaerf, M. (2004). Describing and Reasoning on Web Services using Process Algebra. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services* (pp. 43–55).

Salehie, M. & Tahvildari, L. (2009). Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 1–42.

Sarbanes-Oxley Act (2002). Public Law 107-204 (116 Statute 745),United States Senate and House of Reprensentatives in Congress.

Schilit, B. N., Adams, N., & Want, R. (1994). Context-Aware Computing Applications. In *WMCSA '94: Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications* (pp. 89–101).

Sheng, Q. Z., Benatallah, B., Maamar, Z., & Ngu, A. H. H. (2009). Configurable Composition and Adaptive Provisioning of Web Services. *IEEE Trans. Serv. Comput.*, 2(1), 34–49.

Siljee, J., Bosloper, I., Nijhuis, J., & Hammer, D. (2005). DySOA: Making Service Systems Self-Adaptive. In B. Benatallah, F. Casati, & P. Traverso (Eds.), *ICSOC'05: Proceedings of the 3th International Conference on Service-Oriented Computing*, volume 3826 of *Lecture Notes in Computer Science (LNCS)* (pp. 255–268).: Springer.

Simon, H. (1996). *The Sciences of the Artificial*. MIT Press.

Singh, M. P., Chopra, A. K., Desai, N., & Mallya, A. U. (2004). Protocols for Processes: Programming in the Large for Open Systems. *ACM SIGPLAN Notices*, 39(12), 73–83.

Stohr, E. A. & Zhao, J. L. (2001). Workflow Automation: Overview and Research Issues. *Information Systems Frontiers*, 3(3), 281–296.

Swanson, E. B. (1976). The Dimensions of Maintenance. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering* (pp. 492–497).

Taher, Y., Ait-Bachir, A., Fauvet, M.-C., & Benslimane, D. (2009). Diagnosing Incompatibilities in Web Service Interactions for Automatic Generation of Adapters. In *AINA '09: Proceedings of the 2009 International Conference on Advanced Information Networking and Applications* (pp. 652–659).

Tanksali, S. (2006). *Adaptive Service Oriented Architecture*. Technical report, Integration Consortium.

ter Beek, M., Bucchiarone, A., & Gnesi, S. (2007). Web Service Composition Approaches: From Industrial Standards to Formal Methods. In *Proceedings of the 2nd International Conference on Internet and Web Applications and Services* (pp.1̃5).

Tian, W., Zulkernine, F. H., Zebedee, J., Powley, W., & Martin, P. (2005). Architecture for an Autonomic Web Services Environment. In *WSMDEIS '05: Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Services* (pp. 32–44).

Tosi, D., Denaro, G., & Pezze, M. (2009). Towards Autonomic Service-Oriented Applications. *International Journal of Autonomic Computing*, 1(1), 58–80.

Treiber, M., Truong, H. L., & Dustdar, S. (2008a). On Analyzing Evolutionary Changes of Web Services. In *Proceedings of the ICSOC 2008 Workshops* (pp. 284–297).

Treiber, M., Truong, H. L., & Dustdar, S. (2008b). SEMF - Service Evolution Management Framework. In *SEAA '08: Proceeding of the 34th Euromicro Conference on Software Engineering and Advanced Applications* (pp. 329–336).

Utting, M., Pretschner, A., & Legeard, B. (2006). *A Taxonomy of Model-based Testing*. Working Paper uow-cs-wp-2006-04, The University of Waikato, New Zealand.

van Breugel, F. & Koshkina, M. (2006). *Models and Verification of BPEL*. Technical report, York University.

van den Heuvel, W.-J. (2007). *Aligning Modern Business Processes and Legacy Systems: A Component-Based Perspective*. MIT Press.

van den Heuvel, W.-J., Weigand, H., & Hiel, M. (2007). Configurable Adapters: the Substrate of Self-Adaptive Web Services. In *ICEC '07: Proceedings of the 9th International Conference on Electronic Commerce* (pp. 127–134).

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., & Barros, A. (2003). Workflow Patterns. *Distributed and Parallel Databases*, 14(3), 5–51.

van Deursen, A., Visser, E., & Warmer, J. (2007). Model-Driven Software Evolution: A Research Agenda. In D. Tamzalit (Ed.), *MoDSE '07: Proceeding of the CSMR Workshop on Model-Driven Software Evolution* (pp. 41–49). Amsterdam, The Netherlands.

van Eijndhoven, T., Iacob, M.-E., & Ponisio, M. L. (2008). Achieving Business Process Flexibility with Business Rules. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference* (pp. 95–104).

van Riemsdijk, M. B., Dastani, M., & Winikoff, M. (2008). Goals in Agent Systems: a Unifying Framework. In *AAMAS '08: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 713–720).

van Riemsdijk, M. B. & Wirsing, M. (2007). Goal-Oriented and Procedural Service Orchestration: A Formal Comparison. In *MALLOW '07: Proceedings of the Multi-Agent Logics, Languages, and Organisations Federated Workshops* (pp. 3–18).

von Betalanffy, L. (1956). General Systems Theory. *General Systems*, 1.

von Halle, B. (2001). *Business Rules Applied: Building Better Systems using the Business Rules Approach*. Wiley.

von Susani, O. & Dugerdil, P. (2009). Cross-Organizational Service Evolution Management. In *ITNG '09: Proceedings of the 2009 6th International Conference on Information Technology: New Generations* (pp. 332–337).

W3C (1999). XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`.

Walker, R. J., Baniassad, E. L. A., & Murphy, G. C. (1999). An Initial Assessment of Aspect-Oriented Programming. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering* (pp. 120–130).

Wang, J. T. L., Zhang, K., & Chirn, G.-W. (1995). Algorithms for Approximate Graph Matching. *Information SciencesŮInformatics and Computer Science: An International Journal*, 82(1-2), 45–74.

Wang, S. & Capretz, M. A. M. (2009). A Dependency Impact Analysis Model for Web Services Evolution. In *ICWS '09: Proceedings of 7th International Conference on Web Services* (pp. 359–365).

Wang, Y., DeWitt, D. J., & yi Cai, J. (2003). X-Diff: An Effective Change Detection Algorithm for XML Documents. In *ICDE '03: Proceedings of the 19th International Conference on Data Engineering* (pp. 519–530).

Wassermann, B., Ludwig, H., Laredo, J., Bhattacharya, K., & Pasquale, L. (2009). Distributed Cross-Domain Change Management. In *ICWS '09:*

Proceedings of the 2009 IEEE International Conference on Web Services (pp. 59–66).

Web Services Interoperability Organization (2007). Sample applications. `http://www.ws-i.org/deliverables/workinggroup.aspx?wg=sampleapps`.

Weber, B., Reichert, M., & Rinderle, S. (2008). Change Patterns and Change Support Features - Enhancing Flexibility in Process-Aware Information Systems. *Data & knowledge engineering*, 66(3), 438–466.

Weber, B., Reichert, M., & Wild, W. (2006). Case-Base Maintenance for CCBR-Based Process Evolution. In *Proceedings of the 8th European Conference on Advances in Case-Based Reasoning* (pp. 106–120).

Weigand, H. & van den Heuvel, W.-J. (2005). Leveraging Autonomic Collaborative Processes with Imitation. In *ASMEA '05: Proceedings of the CAiSE '05 Workshops* (pp. 155–169).

Weske, M. (2004). Formal Foundation and Conceptual Design of Dynamic Adaptations in a Workflow Management System. In *HICCS '04: Proceedings of the 34th Hawaii International Conference on System Sciences*.

Wieringa, R. J. & Heerkens, J. M. G. (2006). The Methodological Soundness of Requirements Engineering Papers: A Conceptual Framework and Two Case Studies. *Requirements Engineering*, 11(4), 295–307.

Winikoff, M., Padgham, L., Harland, J., & Thangarajah, J. (2002). Declarative & Procedural Goals in Intelligent Agent Systems. In *KR '02: Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning* (pp. 470–481).

Wooldridge, M. J. (2001). *Introduction to Multiagent Systems*. John Wiley & Sons, Inc.

Wooldridge, M. J. & Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2), 115–152.

Yellin, D. M. & Strom, R. E. (1997). Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2), 292–333.

Yi, X. & Kochut, K. J. (2004). A CP-nets-based Design and Verification Framework for Web Services Composition. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services* (pp. 756 – 760).

Yu, E. S. (1997). Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering* (pp. 226–235).

Zadeh, L. (1963). On the Definition of Adaptivity. *Proceedings IEEE (Correspondence)*, 51, 469.

Zanikolas, S. & Sakellariou, R. (2005). A Taxonomy of Grid Monitoring Systems. *Future Generation Computer Systems*, 21(1), 163–188.

Zhou, Z., Bhiri, S., Gaaloul, W., Shu, L., Vasiliu, L., & Hauswirth, M. (2008a). Developing Process Mediator for Web Service Interactions. In *ICWS '08: Proceedings of the IEEE International Conference on Web Services* (pp. 828–829).

Zhou, Z., Ning, K., Bhiri, S., Vasiliu, L., Shu, L., & Hauswirth, M. (2008b). Behavioral Analysis of Web Services for Supporting Mediated Service Interoperations. In *ICEC '08: Proceedings of the 10th International Conference on Electronic Commerce* (pp. 1–10).

SIKS Dissertation Series

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 Perspective.*

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), *Redesign 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 Coup** (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), *Onotology-Based Information Sharing In Weakly Structured Environments.*

2003-02 || **Jan Broersen** (VU), *Modal Action Logics for Reasoning About Reactive Systems.*

2003-03 || **Martijn Schuemie** (TUD), *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy.*

2003-04 || **Milan Petkovic** (UT), *Content-Based Video Retrieval Supported by Database Technology.*

2003-05 || **Jos Lehmann** (UVA), *Causation in Artificial Intelligence and Law - A modelling approach.*

2003-06 || **Boris van Schooten** (UT), *Development and specification of virtual environments.*

2003-07 || **Machiel Jansen** (UvA), *Formal Explorations of Knowledge Intensive Tasks.*

2003-08 || **Yongping Ran** (UM), *Repair Based Scheduling.*

2003-09 || **Rens Kortmann** (UM), *The resolution of visually guided behaviour.*

2003-10 || **Andreas Lincke** (UvT), *Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture.*

2003-11 || **Simon Keizer** (UT), *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks.*

2003-12 || **Roeland Ordelman** (UT), *Dutch speech recognition in multimedia information retrieval.*

2003-13 || **Jeroen Donkers** (UM), *Nosce Hostem - Searching with Opponent Models.*

2003-14 || **Stijn Hoppenbrouwers** (KUN), *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations.*

2003-15 || **Mathijs de Weerdt** (TUD), *Plan Merging in Multi-Agent Systems.*

2003-16 || **Menzo Windhouwer** (CWI), *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses.*

2003-17 || **David Jansen** (UT), *Extensions of Statecharts with Probability, Time, and Stochastic Timing.*

2003-18 || **Levente Kocsis** (UM), *Learning Search Decisions.*

2004

2004-01 || **Virginia Dignum** (UU), *A Model for Organizational Interaction: Based on Agents, Founded in Logic.*

2004-02 || **Lai Xu** (UvT), *Monitoring Multi-party Contracts for E-business.*

2004-03 || **Perry Groot** (VU), *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving.*

2004-04 || **Chris van Aart** (UVA), *Organizational Principles for Multi-Agent Architectures.*

2004-05 || **Viara Popova** (EUR), *Knowledge discovery and monotonicity.*

2004-06 || **Bart-Jan Hommes** (TUD), *The Evaluation of Business Process Modeling Techniques.*

2004-07 || **Elise Boltjes** (UM), *Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes.*

2004-08 || **Joop Verbeek** (UM), *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise.*

2004-09 || **Martin Caminada** (VU), *For the Sake of the Argument; explorations into argument-based reasoning.*

2004-10 || **Suzanne Kabel** (UVA), *Knowledge-rich indexing of learning-objects.*

2004-11 || **Michel Klein** (VU), *Change Management for Distributed Ontologies.*

2004-12 || **The Duy Bui** (UT), *Creating emotions and facial expressions for embodied agents.*

2004-13 || **Wojciech Jamroga** (UT), *Using Multiple Models of Reality: On Agents who Know how to Play.*

2004-14 || **Paul Harrenstein** (UU), *Logic in Conflict. Logical Explorations in Strategic Equilibrium.*

2004-15 || **Arno Knobbe** (UU), *Multi-Relational Data Mining.*

2004-16 || **Federico Divina** (VU), *Hybrid Genetic Relational Search for Inductive Learning.*

2004-17 || **Mark Winands** (UM), *Informed Search in Complex Games.*

2004-18 || **Vania Bessa Machado** (UvA), *Supporting the Construction of Qualitative Knowledge Models.*

2004-19 || **Thijs Westerveld** (UT), *Using generative probabilistic models for multimedia retrieval.*

2004-20 || **Madelon Evers** (Nyenrode), *Learning from Design: facilitating multidisciplinary design teams.*

2005

2005-01 || **Floor Verdenius** (UVA), *Methodological Aspects of Designing Induction-Based Applications.*

2005-02 || **Erik van der Werf** (UM), *AI techniques for the game of Go.*

2005-03 || **Franc Grootjen** (RUN), *A Pragmatic Approach to the Conceptualisation of Language.*

2005-04 || **Nirvana Meratnia** (UT), *Towards Database Support for Moving Object data.*

2005-05 || **Gabriel Infante-Lopez** (UVA), *Two-Level Probabilistic Grammars for Natural Language Parsing.*

2005-06 || **Pieter Spronck** (UM), *Adaptive Game AI.*

2005-07 || **Flavius Frasincar** (TUE), *Hypermedia Presentation Generation for Semantic Web Information Systems.*

2005-08 || **Richard Vdovjak** (TUE), *A Model-driven Approach for Building Distributed Ontology-based Web Applications.*

2005-09 || **Jeen Broekstra** (VU), *Storage, Querying and Inferencing for Semantic Web Languages.*

2005-10 || **Anders Bouwer** (UVA), *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments.*

2005-11 || **Elth Ogston** (VU), *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search.*

2005-12 || **Csaba Boer** (EUR), *Distributed Simulation in Industry.*

2005-13 || **Fred Hamburg** (UL), *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen.*

2005-14 || **Borys Omelayenko** (VU), *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics.*

2005-15 || **Tibor Bosse** (VU), *Analysis of the Dynamics of Cognitive Processes.*

2005-16 || **Joris Graaumans** (UU), *Usability of XML Query Languages.*

2005-17 || **Boris Shishkov** (TUD), *Software Specification Based on Re-usable Business Components.*

2005-18 || **Danielle Sent** (UU), *Test-selection strategies for probabilistic networks.*

2005-19 || **Michel van Dartel** (UM), *Situated Representation.*

2005-20 || **Cristina Coteanu** (UL), *Cyber Consumer Law, State of the Art and Perspectives.*

2005-21 || **Wijnand Derks** (UT), *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics.*

2006

2006-01 || **Samuil Angelov** (TUE), *Foundations of B2B Electronic Contracting.*

2006-02 || **Cristina Chisalita** (VU), *Contextual issues in the design and use of information technology in organizations.*

2006-03 || **Noor Christoph** (UVA), *The role of metacognitive skills in learning to solve problems.*

2006-04 || **Marta Sabou** (VU), *Building Web Service Ontologies.*

2006-05 || **Cees Pierik** (UU), *Validation Techniques for Object-Oriented Proof Outlines.*

2006-06 || **Ziv Baida** (VU), *Software-aided Service Bundling – Intelligent Methods & Tools for Graphical Service Modeling.*

2006-07 || **Marko Smiljanic** (UT), *XML schema matching – balancing efficiency and effectiveness by means of clustering.*

2006-08 || **Eelco Herder** (UT), *Forward, Back and Home Again – Analyzing User Behavior on the Web.*

2006-09 || **Mohamed Wahdan** (UM), *Automatic Formulation of the Auditor's Opinion.*

2006-10 || **Ronny Siebes** (VU), *Semantic Routing in Peer-to-Peer Systems.*

2006-11 || **Joeri van Ruth** (UT), *Flattening Queries over Nested Data Types.*

2006-12 || **Bert Bongers** (VU), *Interactivation – Towards an e-cology of people, our technological environment, and the arts.*

2006-13 || **Henk-Jan Lebbink** (UU), *Dialogue and Decision Games for Information Exchanging Agents.*

2006-14 || **Johan Hoorn** (VU), *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change.*

2006-15 || **Rainer Malik** (UU), *CONAN: Text Mining in the Biomedical Domain.*

2006-16 || **Carsten Riggelsen** (UU), *Approximation Methods for Efficient Learning of Bayesian Networks.*

2006-17 || **Stacey Nagata** (UU), *User Assistance for Multitasking with Interruptions on a Mobile Device.*

2006-18 || **Valentin Zhizhkun** (UVA), *Graph transformation for Natural Language Processing.*

2006-19 || **Birna van Riemsdijk** (UU), *Cognitive Agent Programming: A Semantic Approach.*

2006-20 || **Marina Velikova** (UvT), *Monotone models for prediction in data mining.*

2006-21 || **Bas van Gils** (RUN), *Aptness on the Web.*

2006-22 || **Paul de Vrieze** (RUN), *Fundaments of Adaptive Personalisation.*

2006-23 || **Ion Juvina** (UU), *Development of Cognitive Model for Navigating on the Web.*

2006-24 || **Laura Hollink** (VU), *Semantic Annotation for Retrieval of Visual Resources.*

2006-25 || **Madalina Drugan** (UU), *Conditional log-likelihood MDL and Evolutionary MCMC.*

2006-26 || **Vojkan Mihajlovic** (UT), *Score Region Algebra: A Flexible Framework for Structured Information Retrieval.*

2006-27 || **Stefano Bocconi** (CWI), *Vox Populi: generating video documentaries from semantically annotated media repositories.*

2006-28 || **Borkur Sigurbjornsson** (UVA), *Focused Information Access using XML Element Retrieval.*

2007

2007-01 || **Kees Leune** (UvT), *Access Control and Service-Oriented Architectures.*

2007-02 || **Wouter Teepe** (RUG), *Reconciling Information Exchange and Confidentiality: A Formal Approach.*

2007-03 || **Peter Mika** (VU), *Social Networks and the Semantic Web.*

2007-04 || **Jurriaan van Diggelen** (UU), *Achieving Semantic Interoperability in Multi-agent Systems: A Dialogue-based Approach.*

2007-05 || **Bart Schermer** (UL), *Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance.*

2007-06 || **Gilad Mishne** (UVA), *Applied Text Analytics for Blogs.*

2007-07 || **Natasa Jovanovic** (UT), *To Who 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-Universitaet zu Berlin), *Perating 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 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-Universitaet 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), *A 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), *Politiële 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** (TUD), *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 || **Viktor 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.*