Universität Stuttgart

KPI-Related Monitoring, Analysis, and Adaptation of Business Processes

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Branimir Wetzstein
aus Osijek, Kroatien

Hauptberichter: Prof. Dr. Dr. h.c. Frank Leymann

Mitberichter: Prof. Dr. Dimka Karastoyanova

Tag der mündlichen Prüfung: 31. Oktober 2016

Institut für Architektur von Anwendungssystemen
der Universität Stuttgart

2016

Contents

Abstract

In today's companies, business processes are increasingly supported by IT systems. They can be implemented as service orchestrations, for example in WS-BPEL, running on Business Process Management (BPM) systems. A service orchestration implements a business process by orchestrating a set of services. These services can be arbitrary IT functionality, human tasks, or again service orchestrations. Often, these business processes are implemented as part of business-to-business collaborations spanning several participating organizations. Service choreographies focus on modeling how processes of different participants interact in such collaborations.

An important aspect in BPM is performance management. Performance is measured in terms of Key Performance Indicators (KPIs), which reflect the achievement towards business goals. KPIs are based on domain-specific metrics typically reflecting the time, cost, and quality dimensions. Dealing with KPIs involves several phases, namely monitoring, analysis, and adaptation. In a first step, KPIs have to be monitored in order to evaluate the current process performance. In case monitoring shows negative results, there is a need for analyzing and understanding the reasons why KPI targets are not reached. Finally, after identifying the influential factors of KPIs, the processes have to be adapted in order to improve the performance.

This thesis presents an approach how KPIs can be monitored, analyzed, and

used for adaptation of processes. The concrete contributions of this thesis are: (i) an approach for monitoring of processes and their KPIs in service choreographies; (ii) a KPI dependency analysis approach based on classification learning which enables explaining how KPIs depend on a set of influential factors; (iii) a runtime adaptation approach which combines monitoring and KPI analysis in order to enable proactive adaptation of processes for improving the KPI performance; (iv) a prototypical implementation and experiment-based evaluation.

ZUSAMMENFASSUNG

Die Ausführung von Geschäftsprozessen wird heute zunehmend durch IT-Systeme unterstützt und auf Basis einer serviceorientierten Architektur umgesetzt. Die Prozesse werden dabei häufig als Service Orchestrierungen implementiert, z.B. in WS-BPEL. Eine Service Orchestrierung interagiert mit Services, die automatisiert oder durch Menschen ausgeführt werden, und wird durch eine Prozessausführungsumgebung ausgeführt. Darüber hinaus werden Geschäftsprozesse oft nicht in Isolation ausgeführt sondern interagieren mit weiteren Geschäftsprozessen, z.B. als Teil von Business-to-Business Beziehungen. Die Interaktionen der Prozesse werden dabei in Service Choreographien modelliert.

Ein wichtiger Aspekt des Geschäftsprozessmanagements ist die Optimierung der Prozesse in Bezug auf ihre Performance, die mit Hilfe von Key Performance Indicators (KPIs) gemessen wird. KPIs basieren auf Prozessmetriken, die typischerweise die Dimensionen Zeit, Kosten und Qualität abbilden, und evaluieren diese in Bezug auf die Erreichung von Unternehmenszielen. Die Optimierung der Prozesse in Bezug auf ihre KPIs umfasst mehrere Phasen. Im ersten Schritt müssen KPIs durch Monitoring der Prozesse zur Laufzeit erhoben werden. Falls die KPI Werte nicht zufriedenstellend sind, werden im nächsten Schritt die Faktoren analysiert, die die KPI Werte beeinflussen. Schließlich werden auf Basis dieser Analyse die Prozesse angepasst um die KPIs zu verbessern.

In dieser Arbeit wird ein integrierter Ansatz für das Monitoring, die Analyse

und automatisierte Adaption von Prozessen mit dem Ziel der Optimierung hinsichtlich der KPIs vorgestellt. Die Beiträge der Arbeit sind wie folgt: (i) ein Ansatz zum Monitoring von KPIs über einzelne Prozesse hinweg in Service Choreographien, (ii) ein Ansatz zur Analyse von beeinflussenden Faktoren von KPIs auf Basis von Entscheidungsbäumen, (iii) ein Ansatz zur automatisierten, proaktiven Adaption von Prozessen zur Laufzeit auf Basis des Monitorings und der KPI Analyse, (iv) eine prototypische Implementierung und experimentelle Evaluierung.

CHAPTER 1

INTRODUCTION

1.1 Application Area

Business processes in companies are increasingly supported by IT systems today. This support ranges from automated business functions as part of business processes, to fully automated business processes running as workflows. Considering the whole lifecycle of business processes, a new discipline has emerged, Business Process Management (BPM), which encompasses methods, models, and tools for managing business processes in and across organizations [Wes07]. Recently, BPM has been supported by an integrated set of tools, so called Business Process Management Systems (BPMS), supporting the process lifecycle in a unified manner.

Typically, the business process lifecycle begins with the modeling phase when a business analyst analyzes business processes in the company and creates business process models using a process modeling notation such as Business Process Model and Notation (BPMN) [OMG11]. In this context, one can distinguish between an orchestration model, which presents the business process from the point of view of one partner and a choreography model, which is used for modeling business-to-business collaborations focusing on interactions between

orchestrations of different partners. If the process model is to be enacted on a BPMS, it is transformed and refined to an executable workflow model. In the deployment phase, the workflow model is deployed to the workflow engine (part of the BPMS), which executes the process by delegating process tasks to humans and applications.

Service-Oriented Architecture (SOA) is the most recent architecture paradigm for implementing enterprise integration solutions [PTDL07]. SOA is supporting BPM by exposing functionality as services, which are used for implementing activities in business processes and business processes again can be exposed as services. The Web services platform is a concrete technology which can be used to implement a SOA [WCL+05]. It consists of a set of specifications, such as Web Services Description Language (WSDL) for describing service interfaces and Web Services Business Process Execution Language (WS-BPEL, or BPEL for short) for orchestration of Web services [OAS07]. In that context, service orchestrations are used for implementing business processes. For example, business processes can be modeled using BPMN and then mapped to BPEL for execution. The BPEL process engine executes the BPEL process model by delegating the process tasks to Web services. While a service orchestration implements an executable private process model implemented by each participant, a service choreography models the publicly visible processes and message exchanges between participants from a global viewpoint [Pel03]. BPEL4Chor is a BPEL extension for modeling service choreographies [DKLW07].

To summarize, this thesis focuses on business processes which are enacted on a BPMS and are realized based on the SOA architecture paradigm, in particular using Web service technologies as implementation platform.

1.2 Motivation

An important aspect in BPM, and the focus of this work, is performance management of business processes.

The motivation is provided based on a purchase order processing scenario, which is also used throughout the thesis for explaining the concepts based on examples. A high-level overview of the business process is shown in Figure 1.1.

It is a choreography between a customer, a reseller, and a shipper. Other involved participants such as suppliers, internal services (e.g., warehouse service of reseller) are not shown in the figure for simplicity reasons.

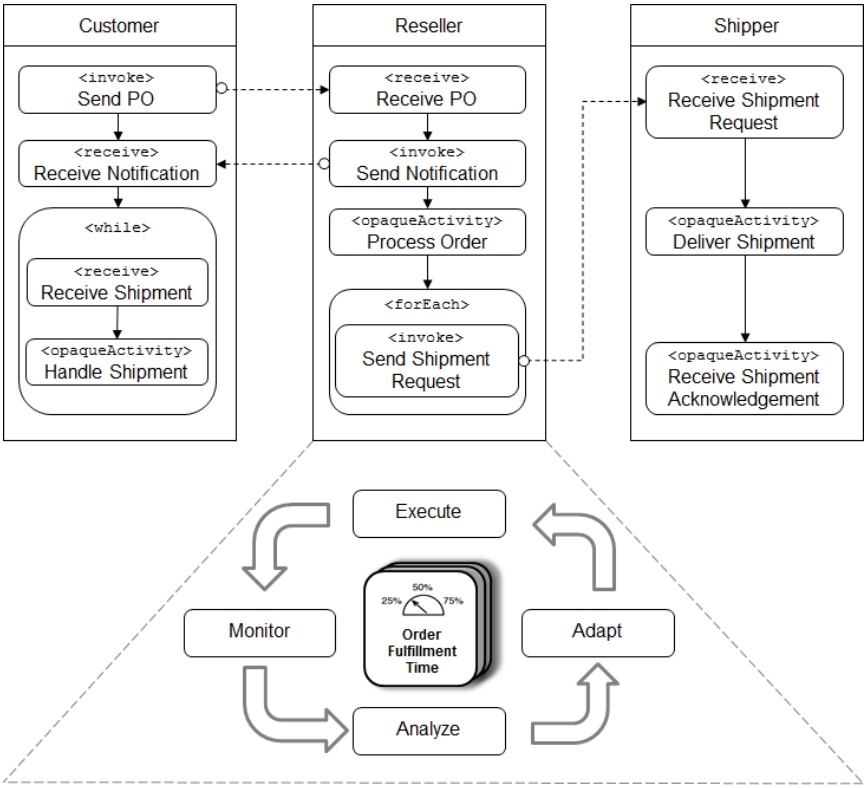

Figure 1.1: Purchase Order Processing Scenario

The scenario is as follows. The customer sends a purchase order to the reseller. The reseller checks in his warehouse whether and by when he can fulfill the order and sends a notification to the customer. The reseller then orders the requested product items at suppliers if they are not in stock, packages the products and hands them over to the shipment service. The order can be split in several shipments. The reseller bills the customer. The shipper delivers

the products to the customer.

Figure 1.1 depicts from the perspective of the reseller, the phases relevant for management of process performance. As the process is *executed*, the Key Performance Indicators (KPIs) are first *monitored*, then *analyzed*, and finally the process is *adapted*, if needed, to improve the performance.

In the following, the motivation and the research questions are described for each of those three phases. Also the contribution of the thesis is sketched shortly for each phase and then presented in detail in the next section.

Monitoring. Companies are interested in *monitoring* the performance of their business processes. They specify and monitor KPIs, which are the most important metrics showing the achievement of business goals. Process-related KPIs are measured based on metrics typically related to time, cost, and quality dimensions. Exemplary KPIs in the above scenario are order fulfillment time and number of orders processed from stock. A KPI definition specifies how the underlying metric has to be calculated and a target function which maps the metric values to a set of categories which enable interpretation of the metric value in relation to business goals (e.g., green, yellow, red).

KPI monitoring can already be performed on process level, e.g., by using state-of-the-art Business Activity Monitoring (BAM) technology, which is often provided as part of BPMS systems. Such systems enable event-based monitoring of service orchestrations in a near real-time fashion. Besides monitoring of KPIs for service orchestrations, monitoring across business processes in service choreographies is getting more and more important. Due to outsourcing, there are more and more business processes which are spread across organizational boundaries. In that case, a company is still interested in monitoring of outsourced process fragments. There is also a trend that companies provide monitoring data of their business processes to some extent to other companies as a feature. For example, shipment services today already provide data on some steps of their business processes giving information on the location of the shipment to the customer. In the scenario, for example, for calculating the order fulfillment time the reseller needs monitoring information from the shipper or customer on when the shipment was delivered. That information is

not available in the reseller process.

There is, thus, a need for monitoring of processes across single orchestrations in service choreographies. The challenges include how to create monitor models and corresponding monitoring interfaces which contain only those monitoring information the participants are willing to provide and how to correlate monitoring information of the different processes in order to calculate composite properties needed for calculation of KPIs. The monitoring approach developed in this thesis supports the definition of monitor models based on service choreography descriptions. Such a monitor model is a monitoring contract between choreography participants which specifies the monitored resources and their properties each participant has to provide in his monitoring interface.

Analysis. In process performance management, monitoring of KPIs is just the first step. This is because when things go wrong, i.e., monitoring shows that KPI targets are not met, in the next step the goal is to *analyze* the most influential factors which lead to those target violations. For example, the reseller wants to know why the order fulfillment time for some orders has been longer than accepted. Due to the fact that KPIs are complex properties that rely upon a wide range of factors, the analysis is not straightforward. In the scenario, the order fulfillment time may be influenced by many different factors, such as duration of sub-processes and activities, response time and availability of used services, ordered products and their properties such as number of ordered items, product type and size, cost of delivery service, and availability of IT infrastructure. All those factors influence the KPI in some way. The difficulty in analysis comes from the fact that those factors behave in a different way for different process instances. For example, partners such as the supplier and shipper can behave differently based on product types or some external context factors, such as a specific day of week or the weather. All those factors and a combination of those can lead to late delivery of orders.

In order to deal with the analysis problem, typically one would build a data mart and then manually pose analysis questions in terms of OLAP queries. That approach is however, time-consuming, costly, and does not allow automated

analysis. The approach in this thesis utilizes data mining techniques, in particular decision trees, for KPI analysis. At modeling time, for each KPI a set of property definitions is generated which pose potential influential factors for that KPI. At process runtime, those properties are monitored. For performing the analysis, a classification learning problem is constructed, whereby each process instance is classified according to its KPI value and the other properties are used as features. As a result a decision tree, the so-called KPI dependency tree, is built, which explains how the KPI depends on the lower-level influential factors. This knowledge can then be used to adapt the process in order to improve the KPIs.

Adaptation. The last step of the lifecycle is to adapt the process based on the analysis in order to improve its performance. When adapting processes, one can distinguish between process model adaptation, where the adapted model affects all future process instances, and process instance adaptation, where only particular process instances are adapted at process runtime. Furthermore, one can distinguish between reactive and proactive adaptation. In reactive adaptation, the adaptation is triggered after a certain event takes place, e.g., the KPI target has been violated. In proactive adaptation, one tries first to predict the undesirable event and then adapt the process in order to prevent it.

In this thesis, the focus is on proactive adaptation of running process instances. The goal is thereby to adapt running process instances in order to improve their performance considering the specified KPIs. For example, in the scenario after the warehouse check one might predict based on the requested product types that the order fulfillment time will be red for the particular process instance. One could then decide to adapt that process instance by selecting and binding a faster supplier or shipper. That assumes, of course, that corresponding adaptation mechanisms are available and that adaptation alternatives exist for the particular process, e.g., that alternative shippers and suppliers are available. Also, it should be taken into account that typically several KPIs are defined, so improving one KPI could lead to deterioration of another. For example, selecting a premium shipment service improves the shipment duration and thus the order fulfillment time, however has a negative impact on a KPI reflecting

the costs.

To summarize, the goal is to enable a proactive runtime adaptation of process instances in order to improve the specified KPIs. The adaptation approach of the thesis uses the monitoring and analysis framework in order to perform runtime adaptation of processes. Thereby, the KPI values are predicted based on KPI dependency trees and then an adaptation strategy is identified and selected based on predefined adaptation alternatives and preferences. Thereby, no new adaptation mechanisms (e.g., dynamic service binding, process fragment substitution) are developed, but existing ones are used as part of the adaptation framework.

1.3 Contributions

As motivated in the previous section, the main goal of this work is to enable monitoring of business process performance in choreographies in terms of KPIs, analyzing the reasons for KPIs not meeting their targets and their explanation, and corresponding proactive runtime adaptation of process instances which leads to an improvement of process performance.

The contributions of this thesis are as follows:

Contribution 1: Business Process Monitoring in Service Choreographies

A monitoring approach is presented which enables process monitoring in service choreographies.

The developed monitoring metamodel supports the definition of monitor models in terms of monitored resources and properties based on service choreography models. It allows to selectively define which monitoring information is to be provided by the participants in the choreography and represents a monitoring contract between the participants. Monitored properties include basic properties such as execution state of process activities needed for process tracking and composite properties needed for calculation of KPIs, evaluated based on events stemming from different processes using complex event processing.

The monitoring approach uses BPEL4Chor as the choreography language and is based on Web Services Distributed Management (WSDM) as monitoring technology. Therefore, BPEL4Chor runtime entities are mapped to a set of resource types, which can be monitored by a new set of monitoring capabilities introduced as an extension to the WSDM framework. A corresponding method for developing monitoring solutions based on the metamodel is described.

Contribution 2: Analysis of Influential Factors of Process Performance

The process monitoring approach provides values which are used to evaluate KPIs showing how business processes perform. Beyond that, if KPI targets are not met, further analysis has to be done explaining why. In this context, the KPI Dependency Analysis approach is presented which explains the influential factors the KPI performance depends on.

A KPI analysis metamodel is presented which supports modeling of KPIs, potential influential factors and analysis tasks. The so created analysis model is integrated with the monitor model in order to ensure that KPIs and influential factors are monitored. A corresponding method for performing the KPI dependency analysis is described.

For the analysis, established decision tree algorithms are used. Data preparation for the decision tree mining is performed in an automated way based on the analysis model. The resulting decision tree, the so-called KPI dependency tree, shows how a KPI depends on the combinations of influential factors. The KPI dependency tree can be used as basis for adapting the process.

Contribution 3: Runtime Adaptation based on KPI Dependency Analysis

The KPI dependency analysis shows the main influential factors of the KPI performance and in particular those influential factor combinations which lead to bad KPI values. It can also be used for predicting the KPI values of running process instances. This contribution shows how the process can be proactively adapted at runtime based on the dependency trees so that KPIs are improved.

The approach includes an adaptation metamodel which allows defining adaptation subjects and related adaptation alternatives which can be used at runtime to adapt the process by using available adaptation mechanisms. Furthermore, preferences and constraints related to KPI values and other properties can be specified to lead the adaptation. Based on the adaptation metamodel, algorithms are presented which extract adaptation requirements from the prediction results and group available adaptation alternatives into adaptation strategies to satisfy those requirements. A selection strategy for adaptation strategies based on the preferences model is presented.

This contribution brings together the monitoring (Contribution 1) and analysis (Contribution 2) and extends it by the adaptation aspect, thus developing a framework for KPI-related proactive adaptation of process instances.

Contribution 4: Prototypical Realization and Scenario-based Evaluation
In order to demonstrate the realizability of the proposed concepts, a prototype has been developed. It consists of a monitoring, analysis, and adaptation framework, which has been implemented on top of an existing BPEL engine, a CEP framework, and a data mining library. The purchase order scenario has been implemented based on BPEL and has been used for experiment-based evaluation of the approach.

1.4 Structure of the Thesis

The thesis is structured as follows. Chapter 2 presents the background and related work needed for understanding the concepts developed in this thesis. The first part of the chapter gives an overview of the concepts and technologies of the BPM and SOA domains. The second part focuses on process performance monitoring, analysis, and adaptation, which is the research domain of this thesis. Related research approaches are presented and their relations to the approach of the thesis are discussed.

Chapter 3 presents the first part in the overall approach focusing on monitoring of business processes in service choreographies. After presenting the motivation and the objectives based on a scenario, an overview of the approach is given and the method is described. Then in the following, the monitoring metamodel is presented in detail, showing how service choreographies can be mapped to monitorable resources and their properties. In particular, the definition of composite events and custom properties is presented, which is the basis for the following parts of the approach.

Chapter 4 explains how based on the monitoring results, the performance of business processes can be analyzed. After giving an overview of the approach and some background information on employed machine learning techniques, in the first part of the chapter an analysis metamodel is presented which allows modeling of KPIs and analysis tasks. The second part then shows how the influential factors of KPIs can be analyzed using decision tree techniques for creating KPI dependency trees.

In Chapter 5 monitoring and analysis are combined for supporting runtime adaptation of processes in order to optimize process performance. First, the overall process is presented explaining all phases from modeling to runtime monitoring and adaptation. Then, the adaptation metamodel and the runtime adaptation approach are presented in detail.

Chapter 6 presents the implementation and the evaluation of the approach. In the first part, the prototypical implementation of the monitoring, analysis, and adaptation is described. Then, the results of an experiment-based evaluation are discussed.

Finally, in the last chapter the contributions of the thesis are summarized and an outlook is given, discussing how the concepts of the thesis can be improved and extended in future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter introduces background concepts, technologies, and related work needed for understanding the contributions of the thesis as presented in the following chapters.

Section 2.1 gives an introduction into the areas of BPM and SOA. It explains the business process lifecycle, provides an overview of the main Web service technologies and focuses then on service orchestrations and service choreographies.

In the next two sections, monitoring, analysis, and adaptation aspects are described in detail, presenting the relevant research approaches and comparing them to the approach of this thesis. Section 2.2 gives an overview of monitoring approaches in BPM and SOA, and explains concepts and technologies related to event processing and BAM. In particular, related research approaches in the area of BPEL monitoring and cross-organizational monitoring are presented. Section 2.3 presents related process performance analysis techniques that go beyond monitoring and process adaptation approaches, in particular those that

combine monitoring, analysis, and adaptation techniques.

2.1 BPM, SOA, and Web Services

This thesis deals with business processes that are realized based on a SOA, in particular using Web services as a concrete SOA implementation technology. In the following, an overview of BPM and SOA concepts is given and the main Web service technologies relevant for this thesis are introduced.

2.1.1 Business Process Management

Business Process Management (BPM) is a discipline which deals with concepts, methods, and techniques to support the business process lifecycle [Wes07]. A *business process* consists of a set of activities which jointly realize a business goal. A business process is enacted by a single organization, but it may interact with business processes of other organizations. A *business process model* defines the execution constraints between process activities. Execution constraints are specified by defining sequencing of activities, decision points, parallel execution, exception handling and so on. There are different languages for specifying business process models (e.g., BPMN [OMG11], EPC [STA05]). A business process model serves as a blueprint for *business process instances*, which represent concrete executions based on the business process model (e.g., processing of a concrete purchase order). In the following, the term *business process* may refer to either a model or an instance, depending on the context.

A business process can be enacted manually or automatically on a software system. In the former case, the business process model is mainly used for documentation and analysis purposes. In the latter case the business process model is deployed on a software system (a.k.a. a workflow engine), which executes the business process instances by delegating work to humans and automated IT applications. Business processes which are executed by software systems are also known as *workflows* [LR00]. In BPM, workflow engines are part of a bigger software system known as *Business Process Management System (BPMS)*. The goal of the BPMS is to support all phases of the business process

lifecycle.

A business process is often not performed in isolation, but is interacting with other business processes, potentially crossing organizational boundaries. Thereby, one distinguishes between a process *orchestration* and a process *choreography*. A process orchestration can be controlled and executed by a central software component which orchestrates the activity executions. A process choreography models the interactions between several process orchestrations. There is no central component which can execute a choreography and involved process orchestrations can be executed by different organizations. A choreography can focus solely on interactions between orchestrations, but can also include internal behavior of the orchestrations which goes beyond interactions. In both cases, the choreography models *public processes* of the participants, while an executable process orchestration models the *private process* of a participant.

The *business process lifecycle* consists of several phases [Wes07]. In the *design and analysis phase* the lifecycle begins by the identification of business processes in an organization and the creation of business process models which are to be realized. In addition, one specifies non-functional aspects of the process, such as performance goals, compliance rules, and security aspects. In the *configuration phase* the business process is implemented. In case of manual processes, this might result in a set of rules and policies which the employees have to follow. If the process is to be executed by a process engine, then it has to be made executable. This involves first selecting the implementation platform. If the implementation of the process follows the SOA paradigm, the process model is realized as a service orchestration (cf. Section 2.1.4). In the next step technical details have to be added to the business process model, which often means transforming the business process model to a workflow representation of the process engine. Non-functional aspects have to be correspondingly mapped to implementation artifacts, e.g., measurement directives for performance metrics. Finally, the process is tested and deployed on the process engine. In the *enactment phase*, process instances are executed. The execution is controlled by the process engine. As the process is executed, the engine typically publishes execution events, which are stored in an audit log and can be used by a monitoring tool to show the execution status of the process instances

and evaluate non-functional aspects. In the *evaluation phase*, business process models are evaluated with the goal of process optimization. The information stored in the audit logs, for example, can be used for assessing the performance of the business process and analyzing optimization potentials. For example, the process model or the resources used by the process can be changed.

This thesis focuses on business processes which are realized as orchestrations in the context of a SOA (cf. Section 2.1.2), and which can interact with other orchestrations in choreographies.

2.1.2 Service-Oriented Architecture

The *Service-Oriented Architecture (SOA)* is an architectural paradigm for the creation of software applications [Erl05, PTDL07]. A *service-based application (SBA)* uses *services* as building blocks. A service has a well-defined service interface and a service implementation.

The *service interface* defines how to interact with the service and consists of a functional description, typically specified in terms of operations with inputs and outputs and technical information on protocols to be used to interact with the service, and optionally of a description of its non-functional properties. The *service implementation* can be realized using arbitrary technologies. Thereby, one can distinguish between atomic services and composite services. An *atomic service* acts as a wrapper of a software component (or a set of components) implemented using an arbitrary component implementation technology (e.g., Java EE, .NET) and programming language (e.g., Java, C++). Such an atomic service does not use other services defined in a service-based application. A *composite service* (a.k.a. service composition, service aggregation) implements a new service by composing already existing services, which can be either atomic or composite. In a wider sense, there are different types of service compositions such as service orchestration, service choreography, service coordination, and service wiring [KL03]. In a narrower sense, a service composition denotes a service orchestration (cf. Section 2.1.4). For the service interface description and the implementation of composite services, typically special models and languages are provided (as in the case of Web services, cf. Section 2.1.3), while

atomic services are typically implemented using 3GL languages and can be used in a SOA by defining a service interface for them.

A *service provider* first implements a service he wants to offer to service requesters. This can involve composing services from other providers and it can also involve creating a choreography model thus agreeing with requesters and providers on the message interactions. Then he creates a service interface description for the service and publishes this service interface description to a *service registry* or provides it directly to the service requesters. A *service requester* obtains the service interface description and *binds* against the service. One can thereby distinguish between *static binding* and *dynamic binding*. In static binding, at design time or at deployment time the concrete service has to be selected; typically then binding information (e.g., stubs for interaction with the services) is generated. In dynamic binding, the decision which service is to be used is prolonged to the runtime thus gaining in flexibility which service is to be invoked. The invocation of a service is supported by a *service bus* [Ley05, Cha04], which handles the technical interaction details in service interactions. In some cases, the service requester and service provider create a Service Level Agreement (SLA), which is a contract specifying the functional and non-functional properties the service should provide, and the consequences in case of SLA violations.

SOA supports the realization of business processes. Firstly, process orchestrations can use services exposed in a SOA for implementing automated process activities. Services in this context typically encapsulate business functions, which are often reusable and can be used in several business processes. Secondly, process orchestrations can be implemented as service orchestrations and executed in a corresponding workflow engine.

2.1.3 Web Services

The Web service (WS) stack [WCL+05, Pap08] is a particular implementation of a SOA. It consists of a set of standardized specifications and technologies which support the realization of service-based applications. In the WS stack, services are called Web services. The WS stack consists of several specifications

for defining different aspects needed for creation of service-based applications: service interface description, service interaction, service discovery, service composition, support for transactions, security, management, and SLAs. In the following, a short overview of the key WS specifications is provided.

The functional interface of a Web service is specified using the Web Service Description Language (WSDL) [W3C01], which is the basic technology of the WS stack. A WSDL 1.1 description is an XML-based document specifying (i) the *abstract interface* (a.k.a. `portType` in WSDL 1.1) of a Web service in terms of operations, and input and output messages of these operations; (ii) a concrete *binding*, which implements and provides the abstract interface using specific message encodings and transport protocols (e.g., SOAP/HTTP, JMS) (iii) the Web service *endpoint* (a.k.a. `port` in WSDL 1.1), which provides a binding at a specific network address.

The SOAP-Messaging-Framework [W3C07] defines a message format and processing rules for messages in WS interactions. A SOAP message consists of a SOAP envelope, which contains an arbitrary set of SOAP headers and a SOAP body. The actual transport of a SOAP message can be performed over different transport protocols such as HTTP, SMTP, and JMS. A SOAP binding [W3C07] defines how the SOAP message is to be serialized and transported over a concrete protocol.

WS-Addressing [W3C06] is a specification which deals with (i) referencing of Web service endpoints via *Endpoint References (EPR)* and (ii) defines the *message addressing properties* which should be part of messages exchanged between WS endpoints. An EPR is issued by the service provider and contains information which is needed to address a WS endpoint. It includes (i) the endpoint *address* as an Internationalized Resource Identifier (IRI), (ii) an unbounded set of *reference parameters* which are domain-specific properties required to address the endpoint, and (iii) an unbounded set of *metadata* elements describing the endpoint.

In this thesis, Web services are used as basis for the implementation of services and processes. The presented specifications are the basic specifications of the WS stack. Additional relevant specifications related to service orchestration (cf. Section 2.1.4), service choreography (cf. Section 2.1.5), and service

management (cf. Section 2.2.2) are presented in the corresponding sections in more detail.

2.1.4 Orchestration of Web Services

A service orchestration implements a new service by composing a set of given services. Thereby, a logically central component interacts with the orchestrated services according to an orchestration model. Services which are orchestrated can be atomic services, i.e., services which do not use any other services, or again service orchestrations.

While service orchestrations could be implemented in 3GL programming languages such as Java, typically higher level languages are used as they support the idea of two-level programming [LR97]. Thereby, the functionality specified in 3GL languages is orchestrated using a workflow language, which often has a graphical representation. The so created orchestration model is deployed on a workflow engine for execution. In the context of Web services, the Web Service Business Process Execution Language 2.0 (WS-BPEL, BPEL for short) [OAS07] is the standard workflow language for implementing Web service orchestrations.

For implementing orchestration logic, BPEL provides a set of activity types. *Basic activities* allow interaction with external Web services (e.g., `invoke` for a Web service invocation, `receive` or `pick` for receiving messages from external Web services), data handling (e.g., `assign` for copying data between process variables), and some other activities (e.g., `wait` for pausing the execution for a certain time, or `throw` for signaling internal faults). *Structured activities* are used for implementing the control flow of the process. BPEL provides, among others, `sequence` for sequencing activities, `flow` for the parallel execution of activities, `if` for implementing alternative parts of execution, and `while` for implementing loops. Alternatively to the block-structured modeling style, BPEL allows also graph-based modeling inside of the `flow` activity by specifying `link` elements between activities thus imposing an execution ordering. BPEL also supports more advanced features via `scope` such as fault handling, compensation handling, and event handling. In BPEL, data flow is implicit. Activities can access process variables, which can be defined either globally for

the whole process or for specific (nested) scopes (which restrict their visibility).

BPEL supports both stateless and stateful interactions. A stateless interaction is given if the BPEL process invokes a Web service synchronously using an `invoke` activity or if it is invoked by the client and uses the *receive-reply* pattern. A stateful interaction (a.k.a. conversation) is given if the process invokes a partner Web service asynchronously using an `invoke`, continues its execution, and then later receives the response in a `receive` activity. In that case, the incoming message from the service has to be correlated to the corresponding process instance which initiated the interaction. Rather than using WS-Addressing, which would perform correlation based on message headers, BPEL performs this correlation based on specific properties defined in the message payload. Therefore, the BPEL-specific mechanism of a *correlation set* is used.

The Web services which are orchestrated in a BPEL process are defined in WSDL 1.1. As the BPEL process is itself exposed as a Web service, it also has a WSDL description. For each conversational relationship between the BPEL process and a Web service one specifies a `partnerLinkType`, which references the corresponding WSDL `portType`. For a `partnerLinkType` one or more `partnerLink` elements can be defined. When defining a BPEL interaction activity, one then specifies the `partnerLink` which should be used and the WSDL `operation` to invoke or to be invoked by the partner. These definitions use only the abstract interface definition of the WSDL description. The concrete binding information and the EPR of the endpoints is provided later at deployment time and is out of scope of the process model. BPEL also allows to bind a `partnerLink` dynamically by providing mechanisms to receive EPRs over messages and assign them to the `partnerLink` while the process is running.

BPEL process models can be defined as abstract or executable. An *executable* process model can be deployed and executed in a BPEL engine. After process deployment, the BPEL process is exposed as a Web service to service consumers. When a service consumer invokes a Web service operation which maps to an instantiating `receive` or `pick` activity, a new process instance is created and starts its execution. During execution, the process engine typically publishes

events, which signal state changes of the process instance and its activities (cf. Section 2.2.4), which can be used for process monitoring.

In *abstract* process models, certain BPEL constructs may be hidden, either explicitly through the inclusion of *opaque* language extensions or implicitly through omission. BPEL defines a *Common Base*, which defines basic rules for abstract processes, e.g., which elements may be defined as opaque. Additional profiles refine the Common Base for specific use cases and give a well-defined semantics to the abstract process. This includes defining when an executable process is a valid *executable completion* of an abstract process. BPEL defines two such abstract process profiles: a profile for observable behavior and a profile for templates. This set however can be extended, as in the case of BPEL4Chor (cf. Section 2.1.5). The *Abstract Process Profile for Observable Behavior* can be used for the definition of business process contracts in cross-organizational interactions. It allows a service provider to specify (in addition to a WSDL interface) its behavior, a so called public process, in the context of Web services exchanges. The profile allows hiding private processing logic which is not to be exposed to partners. Obviously, a valid executable completion of such an abstract process must not add additional interaction activities with the partner as this would break the contract. However, activities for interacting with other partners can be added.

In this thesis, BPEL processes play a prominent role. They are used for the definition of executable processes and as basis for monitoring. The prototype implementation is based on the open-source Apache ODE BPEL engine.

2.1.5 Service Choreographies

As opposed to the service orchestration model, which focuses on the *local view* of one participant (service) and its interactions with other participants, a *service choreography* specifies the *global point of view* of interactions between multiple participants [Pel03]. A choreography model is not executable itself, but rather the execution is performed in a distributed manner whereby each participant implements a service (orchestration) which behaves as specified in the choreography model. Participants thereby can belong to the same but also

different organizations.

The typical usages of choreography modeling are: (i) agreement on how business partners should interact with each other; (ii) standardization of typical partner interactions in a certain domain (e.g., RosettaNet Partner Interface Processes (PIPs) [Ros] and Supply Chain Operations Reference Model (SCOR) [Sup05]); (iii) verification and conformance checking of existing interacting service orchestrations, e.g., checking whether two partners can interact with their existing service implementations; (iv) analysis and optimization of partner collaborations. The first two usages are top-down approaches, where participants of a collaboration first create a choreography model to agree beforehand on the interactions they should support and then later implement their services (or adapts existing services) so that they conform to the choreography model. The latter two usages are bottom-up approaches which start with existing service implementations and create the choreography model for analysis purposes.

There are two different paradigms for modeling choreographies: the *interconnected interface (behavior) model* and the *interaction model* [DKLW09]. The interaction model paradigm specifies the interactions of all participants in *one* model. The basic activities, which model the interactions (e.g., request-response and one-way) between participants, are combined using structured activities, which specify the control flow and the data flow from a global point of view. Popular choreography languages that support this paradigm are WS-CDL [W3C05], Let's Dance [ZBDtH06] and BPMN 2.0 choreography [OMG11]. The interaction model focuses only on interaction activities and does not support modeling of silent activities (a.k.a. opaque activities), i.e., other activities which are part of the public processes of participants. Another drawback of the interaction model is that it is possible to define interactions that are not *realizable* by the choreography participants (e.g. [FBS05]).

The interconnected interface paradigm defines the choreography by specifying the connections between the behavioral interfaces of each participant. Each participant specifies its behavioral interface as in the case of a service orchestration, however focusing only on its public process and hiding private process logic. The choreography is thus not defined in one global model, but

spread across several participant behavior models. The choreography model connects the participant behavior models by specifying message-based interactions between them. Popular choreography languages which support this paradigm are BPEL4Chor [DKLW07] and BPMN 2.0 collaboration [OMG11]. The advantage of this paradigm is that it is easier to derive the implementation of each participant when starting with a choreography model, and vice versa. Also this paradigm is not concerned with the problem of realizability. However, the interfaces of participants may be incompatible, which could result in deadlocks during execution [MMGF06].

For this thesis, choreography models are important as they are used as basis for monitoring of processes across participants. The thesis focuses on interconnected interface models as they allow modeling of public processes which contain more than just interaction activities. The concrete language used is BPEL4Chor, which is presented in the following in more detail.

BPEL4Chor. BPEL4Chor [DKLW07, DKLW09] implements an interconnected interface model and is based on BPEL. It uses the Abstract Process Profile for Observable Behavior of BPEL [OAS07] as basis for modeling of participant interfaces and adds a topology description, which connects the interfaces of the participants. In addition, BPEL4Chor decouples the choreography logic from the implementation details based on WSDL. A BPEL4Chor choreography model consists thus of three different artifact types: (i) participant behavior description (PBD), (ii) a participant topology, and (iii) a participant grounding.

A PBD specifies the abstract process of a participant type. The abstract process is defined based on the *Abstract Process Profile for Participant Behavior Descriptions*, which is based on the Abstract Process Profile for Observable Behavior of BPEL and inherits all of its constraints. In addition it specifies that interaction activities have to contain an identifier (so that they can be easily referenced from the topology) and forbids the usage of `partnerLink`, `portType`, and `operation` as these elements are WSDL-specific.

The *participant topology* defines the structural aspects of a choreography by connecting the PBDs. It specifies participant types, participant references and participant sets, and message links. A *participant type* refers to a PBD and thus

defines the behavior of possibly several participants in the choreography. There are three possible relationships between participant types and participants in a choreography instance: (i) there is only one participant for a participant type; (ii) there are several participants for a participant type and they are known at design time; (iii) the number of participants is not known until runtime. A *participant reference* defines one concrete participant as an instance of a participant type, while *participant sets* are used for supporting several participants per participant type. Finally, *message links* connect two PBDs by defining the sending participant, the sending activity in the sender PBD, the receiving participant (or participant set), the receiving activity in the receiver PBD, and the message name. If needed, a message link also specifies participant references which are to be passed in the message to the receiver for enabling link passing mobility.

The *participant grounding* defines the technical configuration of the choreography based on WSDL. Therefore, each message link from the topology is grounded to a `portType` and `operation`. In addition participant references are grounded to typed WSDL properties.

2.2 Process Monitoring

The term *monitoring* typically refers to the process of collecting relevant data on monitored resources in order to evaluate properties of interest and report results of that evaluation in a timely manner. Timeliness can range from milliseconds (e.g., measuring the duration of a service invocation) to several days or even months (e.g., measuring the cash-to-cash cycle time in a supply-chain or the return on investment).

The boundaries between monitoring, and analysis and prediction approaches (based on monitoring) are often not clear-cut. In this thesis, the following distinction is used. *Monitoring* is used to evaluate functional and non-functional properties of the system in order to check whether the system meets predefined (functional or non-functional) requirements; this is typically done by collecting raw data and processing it using arithmetic and aggregation operators to evaluate higher level properties. *Analysis* uses monitoring results in order to

learn and *explain* why they happened; in particular if requirements have not been met (e.g., incorrect behavior or duration too long) the goal is to find out how to optimize the system in order to meet the requirements in future; even if requirements are met one often wants to discover patterns or models which help to better understand how the system works. *Prediction* uses monitoring results in order to predict the property values in future.

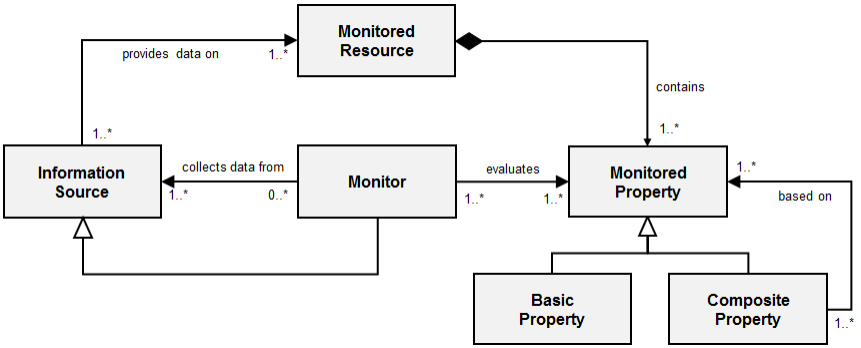

Figure 2.1: Conceptual Monitoring Metamodel

Figure 2.1 depicts the main entities involved in monitoring. Monitoring is performed by *monitors*, which collect data from various information sources. *Information sources* in the SOA and BPM domain are for example process engines, ERP systems, legacy systems, human task managers, operational databases, and the IT infrastructure. Monitors, which collect data from these systems, can provide this data or an evaluated property based on this data again recursively to other monitors. In that case monitors act as information sources themselves. Information sources provide data on *monitored resources* (e.g., the current execution state of a process instance). Monitored resources are for example a process instance, a service endpoint, IT infrastructure, and human resources involved in process execution.

Monitors evaluate *monitored properties* based on data obtained from information sources. Simple properties correspond to basic data (e.g., state of a process activity instance), composite properties (such as process duration) are defined as a function over several data items from one or more information

sources. Monitored properties can be defined for diverse functional and non-functional aspects: (i) some non-functional aspects are duration, quality, cost, security, compliance, and usability; (ii) some functional aspects are correctness of the execution, behavioral properties, assertions, invariants, or the state of the application instance.

Data collection from an information source can be *active* whereby the monitor *pulls* the data from a monitoring interface of the information source or from the information source itself (a.k.a. probing), or *passive* whereby the monitor typically subscribes to events which are *pushed* to the monitor by the instrumented information source or another monitor. The monitor can be integrated with the information source (intrusive) or decoupled communicating, e.g., over message queues (non-intrusive).

There are several *monitoring usages* in the BPM and SOA domain: (i) *process tracking*, i.e., observing the execution state of running process instances; in case of failures (exception handling), manual termination of process instances; providing tracking information to customers (for example, in the case of shipment tracking); (ii) *process controlling* (e.g., activity based costing), which evaluates financial KPIs periodically (lagging indicators) and BAM, which denotes evaluation of operational KPIs in near real time. BAM includes typically notification of stakeholders in case of target violations and presentation of results in business dashboards. (iii) *system monitoring* (e.g., resource utilization of the infrastructure) (iv) *SLA monitoring*, i.e., monitoring of agreed objectives considering the performance and availability of the processes and infrastructure, and detection of SLA violations (v) monitoring for audit purposes; (vi) run-time correctness analysis, which checks whether the process execution conforms to the specified process model.

The following sections present monitoring technologies and approaches which are particularly relevant to the monitoring approach of the thesis, as presented in Chapter 3. Event processing techniques (cf. Section 2.2.1) are used for evaluating composite monitored properties, Web Services Distributed Management (cf. Section 2.2.2) is used as basis for distributed monitoring. Business Activity Monitoring (cf. Section 2.2.3), business process monitoring (cf. Section 2.2.4), and cross-organizational monitoring approaches (cf.

Section 2.2.5) are presented and compared to the approach of the thesis.

2.2.1 Event Processing

Event processing is a technique which enables the push-based monitoring paradigm and near real time evaluation of monitored properties. The following term definitions are based on [LS08, Luc02]. An *event* can be anything that happens (e.g., a purchase order has been received). In order to enable its processing, an event is represented by an *event message* (in the following the term event is used interchangeably and it is explicitly distinguished between event and event message only if needed), which contains a set of *event attributes* (a.k.a. event properties). An event contains at least a timestamp attribute which denotes the creation time of the event message. Concrete event messages are based on *event definitions*, which define the schema of event messages.

One can distinguish between several types of events [LS08]: (i) *simple events* are events which are not created based on other events; (ii) *complex events* are abstractions of other events called its members; (iii) *derived events* are complex events which are created based on one or more events; (iv) *composite events* are derived, complex events which have been created based on a set of member events (which can be any of the types) using a set of constructors such as conjunction, disjunction, sequence etc. The composite event includes the base events from which it is derived. Derived and composite events are sometimes called aggregate events.

Event processing performs operations on events. *Event Stream Processing (ESP)* deals with event processing based on event streams. An *event stream* consists of a linearly ordered sequence of events, typically ordered by time. A stream thereby can contain events of different event types. A *window* is a view on an event stream defining a subsequence of it based on time or length constraints. For example, one can define a time window for events gathered each day or define a window for last 1000 events gathered. Event stream processing queries existing streams, performs operations on events in these streams, creating new events which are put into a new event stream. ESP originates from active databases and data streams management [LS08].

As opposed to ESP, *Complex Event Processing (CEP)* deals with processing based on *event clouds*, which consist of a set of events that are only partially ordered. In CEP, one specifies *event patterns*. An event pattern matches when an event or multiple events occur that match a pattern expression. A pattern expression consists of event types and filter expression on these event types and pattern operators which combine those events logically and temporally. Event patterns are the basis for rules, also called event pattern triggered reactive rules, which define the actions to be performed if an event pattern is detected.

Event processing is performed by event processing agents, which support an Event Processing Language (EPL) that can support either ESP or CEP or both. An EPA can act as an *event producer*, *event consumer*, or even take both roles. In an *event processing network*, a set of event processing agents is connected via *event channels*. Event channels can be point-to-point based on queues or publish-subscribe channels where multiple event consumers subscribe to topics. An event channel can carry different types of events. The runtime deployment of an event processing network can be distributed across multiple networks, software components, and even organizations.

ESPER. Event processing is typically provided by designated event processing frameworks. The framework used in this thesis is ESPER [Esp]. It is an open source, Java-based framework which provides an EPL for event stream processing and event pattern matching, support for different types of event representations (based on JavaBeans, name value pairs, XML events, etc.), Java-based API for querying results, Input-/Output Adapter, e.g., JMS, and relational database access via SQL.

ESPER EPL supports both ESP and CEP. Event Stream processing is enabled by an SQL-like language with `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses. Thereby, event streams replace tables as the source of data, events replace rows as the basic unit of data and event properties replace table columns. An example query could be specified as follows: `INSERT INTO CustomerAmountDay SELECT customer, sum(amount) FROM OrderReceived.win:time_batch(1day) GROUP BY customer`. Here, one specifies that at the end of a time interval of one day (tumbling

time window), one collects `OrderReceived` events, groups them by customer, and creates a new event stream which for each customer carries an event with the overall ordered amount for that day.

In addition to ESP, ESPER also supports CEP via event pattern matching. Event patterns match when an event or multiple events occur that match the pattern expression. A pattern expression consists of pattern atoms and pattern operators. Pattern atoms are either events or filter expressions over events or so called observers for time-based events (e.g., timer:interval(10 seconds)). Pattern operators combine atoms logically (`and, or, not`) or temporally (followed-by (`->`)). An example query could be specified as follows: `SELECT a.orderId,(b.time-a.time) as duration FROM PATTERN[ EVERY a=OrderReceived ->b=OrderShipped(orderId=a.orderId)]`. Thereby, the duration between order receipt and order shipment is calculated by correlating events of those two types based on the orderId. The pattern is evaluated for *every* order received.

In this thesis, event processing is used as the monitoring technique for evaluation of composite monitored properties. Thereby, simple events obtained from the process execution infrastructure are aggregated using an EPL. ESPER is used in the implementation of the prototype.

2.2.2 Web Services Distributed Management

Web Services Distributed Management (WSDM) [OAS06b] is an OASIS standard that deals with management aspects in the Web services environment. *Management* thereby includes (i) (passive) monitoring, i.e., read access to properties of interest from a manageable resource, (ii) active control, i.e., write access to properties of a manageable resource. It consists of two specifications: Management Using Web Services (MUWS) and Management of Web Services (MOWS).

MUWS [OAS06d] defines how *manageable resources* can be managed using Web services. Manageable resources thereby do not have to be Web services themselves (e.g., a printer could be such a resource) but they just have to expose a WSDL-based *manageability endpoint*, which provides management function-

ality to *manageability consumers*. Therefore, the manageability consumer first obtains an EPR of the manageability endpoint and then communicates with it as specified in the corresponding WSDL document of the manageability endpoint.

A manageability endpoint provides a set of *manageability capabilities* that allow getting information and modifying the functional and non-functional aspects of the manageable resource (e.g., getting information on the state of the resource, setting configuration parameters, etc.). A manageability capability is associated with a set of properties, operations, events, and metadata. *Properties* are described using XML schema and are exposed to consumers in a resource properties document as defined in the Web Services Resource Framework (WSRF) [OAS06e]. In addition to the *operations* specified in WSRF, also specific operations can be provided and have to be accordingly defined in WSDL. *Events* are defined as a combination of a topic QName (as defined in WS-Topics [OAS06f]) and the event message content which is based on a WSDM event format. WS-BaseNotification [OAS06a] is used for implementing the notification mechanism. Finally, *metadata* can be included specifying, for example, the valid values of properties or whether properties are mutable or modifiable.

MUWS has predefined a set of *manageability capabilities*. The *Identity* capability allows the identification of a resource by providing access to a *ResourceId* property and is the only mandatory capability for a WSDM manageable resource. The *State* capability exposes the state of a resource based on a domain-specific state model. The *Operational Status* capability provides information on the availability of a resource. The *Metrics* capability allows a resource to expose a set of metrics. The definition of a metric contains two types of metadata. The *value modifiers* are needed to correctly interpret the metric value. For example, one can define when the metric value was *last updated*, when its value was *reset*, and the *time window* of measurements. The *definitional metadata* specifies how the metric value is obtained. Thereby, one can specify the *change type* of a metric such as *counter* or *gauge*, its *time scope* (e.g., interval or point in time), its *gathering time* (e.g., on change, periodic, on demand), and the *calculation interval*, i.e., the frequency of metric value updates.

In addition to manageability capabilities, which are always offered by the

manageability endpoint of a single resource, MUWS defines several *management-related capabilities*, which can be offered by any Web Service endpoint. The *Relationships* capability enables managing of relationships between resources. The *Advertisement* capability provides notifications in case of creation or destruction of manageable resources. This kind of notifications cannot be provided by the manageability endpoint as it does not exist before the corresponding resource is created.

MOWS [OAS06c] is based on MUWS and deals specifically with Web service endpoints as manageable resources. Therefore, it provides a set of manageability capabilities several of which are already defined in MUWS (e.g., Identity, Operational Status). Some of them are extended, e.g., Metrics and Operation Metrics, which both specify specific metrics applicable for Web service endpoints and operations, respectively. In addition, several new capabilities are defined. The *manageability references* capability provides a WS operation which allows to obtain the endpoint reference to the manageability endpoint for a Web service. The *(Operation) Operational State* and *Request Processing State* capabilities provide information on the state of the endpoint or operation (e.g., busy, idle, stopped, crashed) and request (e.g., received, processing, completed, failed).

The monitoring framework developed in this thesis uses MUWS as basis for exposing processes of choreography participants as manageable resources. Therefore, new capabilities have been developed which enable evaluation and access to basic and composite properties of BPEL processes in BPEL4Chor choreographies.

2.2.3 Business Activity Monitoring

Business Activity Monitoring (BAM) is a term coined by Gartner [McC02] and denotes near real-time monitoring of business processes. Those business processes are often spread between different systems such as process engines, ERP systems, operational databases, and legacy systems. These systems generate business events. Business events have business meaning and denote business situations in business processes, e.g., “Order Received”, “Shipment Delivered”.

Business events are composed and correlated to find trends of execution (e.g., bottlenecks, recurring conditions that lead to exceptions and failures), calculate KPIs and display them in dashboards, provide an overview of the overall state of the running business processes and proactively alert business managers for corrective actions if KPI targets are not met, business rules are violated, or exceptions occur. The monitored information is displayed in customizable dashboards. Monitored properties are typically KPIs (business performance metrics), but can also be based on security and compliance properties.

The development of a BAM solution involves instrumentation of applications (if not yet the case) which results in event models exposed by those applications. Based on monitoring requirements, one specifies a monitor model. It contains definitions of monitored properties based on the event models (e.g., by using CEP statements). It can also include definitions of how the results should be presented in dashboard views and in which cases notifications should be sent. The monitor model is deployed to a BAM tool.

BAM is to be distinguished from system monitoring (a.k.a. technical monitoring), which provides information about the QoS properties of IT resources (e.g., availability and performance). This thesis focuses on BAM rather than system monitoring. Section 7.1 on future work discusses possible extensions towards including also system monitoring into the approach.

IBM Business Monitor. As one BAM product in the market, IBM's Business Monitor is presented in the following as an example for the capabilities of a commercial BAM solution and is compared to the approach of the thesis. The IBM Business Monitor, currently in the version 8.5 [IBM15], is a BAM product which supports end to end monitoring of processes, i.e., it supports the creation of monitor solutions which are based on events coming from different types of products and systems.

The development of monitor models for the IBM Business Monitor is performed in the Integration Designer bottom-up based on events obtained from different systems. Many products are already instrumented to provide events in the expected format, such as the Business Process Manager (BPMN engine), the Process Server (BPEL engine), and the Human Task Manager. If a system does

not provide application data as events in the expected format, then that application data can be mediated into events using products such as IBM Adapters or the IBM Integration Bus.

Based on *inbound events*, the monitor model is created by combining predefined language elements such as *metrics, keys, stopwatches, counters, triggers, and outbound events*. On top of these definitions, one can specify KPIs (metrics with target values), dimensional models (specifying how the monitor model should be stored in a warehouse schema), and visual models (dashboards).

At deployment time, the monitor model is deployed to the Business Monitor server, which can be seen as an execution engine for monitor models. It processes events, stored results in the underlying monitor database, and triggers alerts if needed. The monitoring results as stored in the monitor database are queried by the dashboard components. The monitoring results can also be accessed using a REST API.

The monitoring approach of the thesis and the IBM approach are similar in that they both use events as basis and use a language to construct monitor models bottom up. The language used by IBM is a custom higher-level monitoring language while the thesis uses a standard CEP framework for the evaluation of composite properties. The thesis approach is specifically designed for monitoring of service choreographies and orchestrations, and focuses on the creation of customizable Web service-based monitoring interfaces between choreography participants, which is not supported by the IBM tool. While the IBM approach is more general, in that it can use arbitrary events from any system, the approach of the thesis focuses so far on monitoring of choreography-based events. It could however be extended by additional capabilities for supporting arbitrary events. Finally, the approach of thesis goes beyond the capabilities of a BAM tool by enabling KPI dependency analysis, automated prediction, and adaptation of business processes.

2.2.4 Monitoring of Business Processes

There are many approaches that deal with monitoring of service orchestrations, in particular BPEL processes. They differ in the following aspects: (i) the used

event model, (ii) monitoring languages and types of monitored properties, and (iii) monitoring mechanisms for accessing monitoring information. In the following, for each of those aspects the main approaches are presented and it is discussed how the monitoring approach of this thesis is related to them.

BPEL Event Model. The basis for BPEL monitoring is an *event model* which exposes the *state changes* of BPEL entities at process runtime. For example, a process instance is first *instantiated*, is then *running* for a certain period of time, and is finally *completed*, *faulted*, or *terminated*. The transitions between these states are signaled using events. The states and state transitions of a BPEL entity define the event model of that entity. For runtime monitoring purposes, one can define event models for *instances* of the following BPEL entities: process, activity, link, variable, partner link, and correlation set. In addition to those runtime entities, also the process model itself has an event model, signaling its deployment and undeployment.

The BPEL specification does not define a standard BPEL event model. Thus, every BPEL engine implementation supports a slightly different event model. The event models of different engine implementations are similar because to some extent the event model is derived from the operational semantics of BPEL. However, they differ in their granularity (number of events), event names, event formats, and event contents. [Ste08] presents and compares event models of several BPEL engines, including the commercial IBM WebSphere Process Server, and the open-source Apache ODE. The event models differ also because some of the events are added to support additional features of BPEL engines which go beyond the BPEL specification. One such feature is, for example, active control of process execution, such as suspending and resuming of process execution or skipping of activities. Events are typically stored in an audit trail and are published to a messaging infrastructure for passive monitoring.

In this thesis, the WS-BPEL 2.0 Event Model as defined in [KHK+11] is used. It supports both passive monitoring and active control from external applications which can trigger some of the state transitions by sending messages to the BPEL engine. The overall event model consists of event models for the process model and instances of the following entities: process, activity, scope activity, invoke

activity, loop activity, link, variable, partner link, and correlation set.

The WS-BPEL 2.0 Event Model specifies also the information which the events should contain in order to be able to assign them to the corresponding BPEL entity. For example, a process model is identified by the QName of the process model and its version number. A process instance is identified by the process model identifier and in addition by a globally unique instance ID typically assigned by the BPEL engine. The other types of entities (instances of activities, links, variables, partner links, and correlation sets) are identified by an XPath expression which identifies their definition in the process model relative to the root (e.g., `/process/sequence[1]/receive[1]`), the process instance ID, and the instance ID of the (innermost) scope where the element is nested in. In case of an activity instance, an additional instance ID for that activity is needed. The two latter IDs are needed to support special cases when parallel forEach activities and event handlers are used (cf. [KHK+11] for more details).

This thesis uses the WS-BPEL 2.0 Event Model as basis for monitoring of BPEL4Chor choreographies. In particular, blocking events are used to stop the process execution and perform prediction and runtime adaptation.

Monitoring Languages and Monitored Properties In the context of BPEL monitoring, different types of monitoring languages are used to specify monitored properties.

The DYNAMO framework [BG05] deals with monitoring of BPEL processes focusing on runtime validation of partner behavior. The goal is to detect partner services which deliver unexpected results concerning functional and non-functional expectations. Therefore, monitoring directives are specified as rules using the Web Service Constraint Language (WS-CoL), which is based on WS-Policy. WS-CoL specifies the rules as assertions over runtime data gathered from the BPEL process using Aspect-Oriented Programming (AOP) techniques. The ASTRO framework [BTPT06] supports run-time checking of assumptions under which the partner services are supposed to participate in the BPEL process and the conditions that the process is expected to satisfy. The used Run-Time Monitor specification Language (RTML) based on temporal logic supports specifying boolean, statistic, and time-related properties on instance

level but also across process instances. Both approaches, DYNAMO and ASTRO, can also be combined [BGPT09].

As already described in Section 2.2.3, the IBM Business Monitor [IBM15] supports also BPEL monitoring. The Process Server exposes an event model, which is then used to specify KPIs in the monitor model. Those properties are specified using a domain-specific language, relying on XPath for querying and extracting data from events and predefined elements such as counters, metrics, stopwatches, and KPIs. Similarly, [WSL09] presents a domain-specific XML-based language for defining process performance metrics (PPMs) for BPEL processes. The language does not refer to events, but contains higher-level functions (e.g., duration, cost, state, count) which reference activities in the BPEL process model thus implicitly defining which events are needed. After creating the PPM model, in the deployment phase a monitor model for a specific BPEL engine is generated. [MGA09] also presents a model-driven approach. Process performance indicators (PPIs) are specified for a BPMN model representing a Computation-Independent Model (CIM). That PPI model is then transformed subsequently to a Platform-Independent Model (PIM), Platform-Specific Model (PSM), and finally platform-specific code which is an instrumented BPEL process. [BEMP07] uses a query language directly on the process model (based on XPath) and provides a corresponding simple visual interface which enables users to specify monitoring tasks in an intuitive manner (query by example). Queries are translated to BPEL processes that run on the same process engine as the monitored processes. [FJMM12] extends BPMN enabling the specification of KPIs graphically. BPMN elements for defining duration, frequency, state occurrence, aggregated measures, among others, are introduced. They can be used in combination to define complex KPIs for a BPMN process. The approach however deals only with modeling aspects and does not show how such a model could be transformed to an executable monitor model.

[WML08] presents an approach to monitoring of KPIs of semantically annotated business processes. Thereby, the language specified in [WSL09] has been extended to include semantic annotations of business processes when defining KPIs. At execution time, reasoning technology is used for calculation of

KPIs. [PLW+08] presents another approach for monitoring of semantic business processes. Thereby, a core ontology for business process process analysis has been extended by an events ontology and metrics ontology, and is used for the specification of monitor models.

There are several approaches which use a general purpose event processing language as basis for calculation of monitored properties. In [WLR+11], monitored properties are specified based on an event processing language provided by the event processing framework ESPER (cf. Section 2.2.1). The monitor model is specified in an XML file defining which events are needed from the BPEL engine. In the second part, these events are used in EPL statements to calculate process metrics. [KK15] describes a model-driven approach to event-based process monitoring. The monitoring objectives are specified in a monitor model using a language called ProGoalML. The monitor model is then transformed to monitoring probes for gathering the events, CEP rules, an SQL schema for the data warehouse, and a visualization schema for dashboards. [MZD13] presents another model-driven approach focusing on compliance monitoring in business processes. A domain-specific language allows modeling of compliance rules, which are then transformed to EPL statements and used for monitoring. [MRD10] presents another approach which uses CEP for monitoring of service composition infrastructures. It intercepts the SOAP invocations of the BPEL interaction activities using AOP and generates events corresponding these invocations. These events are then processed using an existing CEP engine, which has been integrated into the overall VieDAME service middleware framework.

When monitoring processes, the events stemming solely from the process layer might not be enough for the evaluation of certain properties but events from different layers might be needed and correlated. In [WLR+11], EPL statements are also used to correlate events emitted by the process engine with events emitted by an IT-level monitor (e.g., monitoring the availability and load of the machine the process engine is deployed on). [BG13] presents a domain-specific language for multi-level service monitoring, the Multi-layer Collection and Constraint Language (mlCCL). It allows to define how to collect, aggregate, and analyze the data in a system consisting of several layers.

The ECoWare framework concretely supports systems based on the Service Component Architecture (SCA).

To summarize, the approaches use either a special domain-specific language, a general purpose language (such as an EPL), or a combination of those by using a model-driven approach. A domain-specific language makes it easier to specify the monitored properties for the domain user, while the general purpose language such as an EPL is very expressive and the corresponding middleware is designed for scalability and performance. In this thesis, an XML format is used for the definition of monitor models, which combines measurement directives (capabilities) for specifying needed basic events in choreographies and uses an existing EPL as basis for defining composite properties based on those events. The difference to the previously mentioned approaches is the focus on choreographies and the support for defining custom monitoring interfaces between choreography participants.

Monitoring Mechanisms. This aspect deals with mechanisms for (i) obtaining the monitoring information from the process engine and (ii) processing that information to evaluate monitored properties.

As described above, the BPEL engine provides an event model. These events can be published to an *audit trail* and to a messaging infrastructure (queues, topics) for monitoring purposes [LR00]. This is typically done as part of an (ACID) transaction, as losing of events in most use cases cannot be afforded. As the publishing of events is done in a transactional way, publishing of all possible events can have a relatively high performance impact on the process execution, in particular in short running processes [LR00]. Also, often one is not interested in getting events for all entities of the process model and all possible events as defined in the event model. Thus, engines typically enable configuring which events are to be published. The granularity varies between engine implementations. Apache ODE allows, for example, to specify in the deployment descriptor which event types are to be published per scope [Apac]. It allows also to deploy an event filter which runs in the same process as the engine and can filter events more flexibly. It can rely on event stream semantics and it can be used for event processing, e.g., augmenting events with

information, creating new events, etc. based on an EPL. In [WLR+11], an event filter has been developed which can be configured to emit only needed events. This configuration is done during deployment of the overall monitor model.

[KKL07a] describes the Pluggable Framework, which implements the WS-BPEL Event Model 2.0 [KHK+11]. Thereby, a *generic controller* is integrated with the BPEL engine and publishes events as they occur to a topic. An arbitrary set of *custom controllers* can subscribe to that topic and process the events. In addition, it allows custom controllers to register for specific blocking events, which stop process instance execution. When a blocking event occurs during process execution it is forwarded to the corresponding custom controller, which then later has to send an unblocking event.

In addition to events which the engine pushes to monitors, the engine can also expose a management interface which allows querying (pulling) monitoring information on demand. Typically, it provides getting information on the deployed process models, running instances, and some statistics, such as number of running or finished instances. This information is typically obtained from the model database and instance database [LR00]. In [vLLM+08] a management framework for BPEL is presented. Process models and process instances are exposed as resources and a resource oriented management API allows to access those resources and their properties in a standardized manner. The approach has been realized based on WSRF and WS-Notification enabling the clients to subscribe to property changes of resources and to explicitly pull the information on demand.

Another possibility for getting monitoring information for running processes is to include monitoring activities into the process model itself. In [RSS06] a BPEL process model is extended with auditing activities in order to publish state changes by invoking operations on the monitoring tool. [MGA09] uses a similar approach whereby the instrumented BPEL process is generated in a model-driven manner.

The monitoring approach of the thesis uses an approach which is similar to the management framework for BPEL as described in [vLLM+08] in that the processes in the choreography are mapped to resources and resource properties. The difference is that it focuses on choreographies and the resource interfaces

are configurable using a monitor model and capabilities, i.e., the resource interfaces are specifically designed for a particular choreography model. Also the definition of composite properties as basis for the evaluation of KPIs is supported. The prototype of the thesis uses the Pluggable Framework [KKL07a] as basis for gathering the events from the process engine.

2.2.5 Cross-Organizational Process Monitoring

Cross-organizational process monitoring approaches deal with scenarios, mechanisms and techniques that go beyond monitoring of single business processes implemented as service orchestrations.

While not explicitly focusing on cross-organizational monitoring, [vLLM+08] presents an approach where BPEL processes are exposed as a set of resources to clients, as already mentioned in the previous section. The approach explains in detail how process models and process instances including activities, variables, and other BPEL entities are mapped to resources. Clients can thus access monitoring information (such as the state of a running process instance) on running processes at the service provider in a standardized way by using the well-known WSRF framework. In a similar way, the approach described in [ZSW+10] represents a process management system as a manageable resource. It uses its own process metamodel derived from XPDL and exposes it as managed resources using the Web Services Distributed Management (WSDM) set of standards. In addition, it supports the definition of processing rules based on CEP statements. The monitoring approach of the thesis is different in that it focuses on choreographies allowing to define monitor models for choreographies which result in custom monitoring interfaces each participant has to provide. The monitor model acts as a monitoring contract shared between several participants in a service choreography.

[KSK07] presents an approach to monitoring of BPEL processes which are deployed on several BPEL engines, with possibly different types of event models. A common audit format is introduced which supports processing and correlating events across different BPEL engines. [LKS+10] deals with end-to-end monitoring and correlation in service-based applications. The concept of a

business composite is introduced which groups a set of service components implemented possibly in different business processes. For monitoring such a business composite end-to-end, information invariants are used to correlate service instances (and corresponding events) to instances of business composites. IBM WebSphere tooling is used to evaluate the approach. Both approaches deal with general correlation issues across processes but do not cover the definition of monitor models, composite properties, and monitoring interfaces in choreographies. [SVDS12] also deals with supporting monitoring of processes running in several process engines across organizational boundaries. The developed event model consists of all the events that can be created by the different process engines. The events are propagated to a client dashboard using a CEP engine. The approach however does not deal with correlation issues between the processes and the creation of monitoring interfaces based on monitor models.

Service choreography models define how participants should interact in terms of message exchanges. At runtime, monitoring can be used for validating whether participant interactions actually comply to the choreography model. [vRR09] describes a monitoring infrastructure needed as a basis for conformance checking in choreographies. The approach assumes that choreographies have been modeled in WS-CDL and describes message correlation and logging mechanisms, and their realization as part of a service bus. The conformance checking itself is not dealt with in the approach. [KEvL+11] presents BPELgold, which is a new choreography language based on BPEL supporting modeling of interaction choreography models. The paper shows how a choreography-aware service bus can be used to ensure that executed message exchanges comply with a predefined choreography modeled in BPELgold. Different types of exception handling mechanisms are discussed in case of protocol violations, e.g., dropping of the wrong message, notifying the sender of the violation, triggering of the default exception handling or a predefined exception handling, and stopping of the choreography. The approach of the thesis does not deal with conformance checking but assumes that participants behave according to the agreed choreography model. [BFPG12] presents another approach to conformance checking by deriving event queries from a choreography model.

The choreography model is transformed to event queries at design time. At runtime, message interactions result in events, which are then evaluated by a CEP engine based on the generated event queries checking for violations of the expected behavior.

In the context of virtual enterprises [Pau09], monitoring of processes within business networks has typically focused only on monitoring in the network formation phase, which determines what can be monitored during process execution. However when business networks and their processes evolve [DCS09], the resulting monitoring contracts also have to change accordingly. In that context, [CVG12] discusses mechanisms for preserving the monitorability of processes for different types of business network evolution situations. Taking into account dependencies between already established contracts, the goal is to update the monitoring infrastructure in order to satisfy the new requirements that arise after network evolution. [WDL+08] presents an approach to cross-organizational process monitoring in service networks. Thereby, KPIs specified in the service network layer are mapped to events in the choreography layer which each participant has to provide to the other participants in the network for calculating the KPIs. The approach of the thesis focuses on the monitoring of processes in the choreography and orchestration layer and does not deal with monitoring in service networks. It could however be used to support service network monitoring as motivated in [WDL+08].

[WKK+10] uses the choreography language BPEL4Chor as a basis for defining monitoring contracts between participants. A monitoring contract defines (i) resource events each participant has to provide for its own public process and (ii) composite events for calculating metrics and rules. Resource events are specified based on state models of BPEL entities using the WS-BPEL 2.0 Event Model [KHK+11], while composite events are defined using CEP statements over resource events and other composite events. After deployment of the monitoring contract on the monitoring infrastructures of the participants, the participants exchange monitoring events as specified when the process instances are executed as defined in the choreography. [BFPG12] is a similar approach to monitoring of service choreographies by exchanging events between participants. Therefore, an External Flow Monitor (EFM) is implemented within each

participating organization and according to the predefined choreography model, each EFM monitors all incoming and outgoing messages of its organization and automatically exchanges events with a predefined subset of other participants. The participants are hierarchically classified thus determining how the events are to be distributed in contrast to the approach presented in [WKK+10] where each event exchange has to be explicitly modeled in the monitoring contract. The approach stays on the conceptual level and does not deal with the technical realization of the monitoring infrastructure.

To summarize, in this thesis, the monitoring approach builds on the concept of monitoring contracts in choreographies as presented in [WKK+10]. It extends that work by using the concepts of manageable resources in a similar way as presented in [vLLM+08] and [ZSW+10] thus using Web service standards for establishing monitoring interfaces between participants in a choreography.

2.3 Process Performance Analysis and Optimization

Process performance management deals with ensuring that business processes achieve performance targets. That includes monitoring of business process performance which has been presented in the previous section, but also the following phases of analyzing and optimizing process performance. These two latter phases and approaches which cover all three phases in an integrated manner are the topic of this section.

Process performance management is part of the broader area of business performance management, which has the scope on the whole organization. One of the most popular methodologies for business performance management on the strategy level is the Balanced Scorecard (BSC) [KN97]. It provides a specific methodology for aligning organizations with business strategy by using a balanced set of metrics across four perspectives: financial, internal business processes, customer, and learning and growth. The four perspectives are presented in a *scorecard*, which is a visual display mechanism that charts progress towards achieving strategic objectives by comparing performance against targets and thresholds. Scorecards are often supported by Business Intelligence (BI) tools. For each perspective one defines (i) strategic objectives

(e.g., increase of customer satisfaction), (ii) measures (e.g., complaint rate, reshipment rate, percentage of purchase orders completed in full and on time), (iii) targets (e.g., complaint rate should be below 5%), and (iv) initiatives (e.g., increase process quality, improve deadline adherence). Measures and targets combined are better known as KPIs, which help assessing the achievement of important objectives. Process-related KPIs are typically evaluated in terms of the three dimensions: time, cost, and quality [SS06]. However, also other dimensions can be used such as flexibility, customer satisfaction, and sustainability [NLS11]. For all of these dimensions one can specify a set of key metrics, which are typically domain-specific. The SCOR framework [Sup05], for example, defines a set of metrics relevant in the supply-chain domain.

While scorecards are used on the strategy layer and are not part of the approach of the thesis, they could be used as input to the process performance management. Thereby, all KPIs specified in the scorecard which can be measured based on executable processes would serve as a starting point for the creation of monitor models.

In the next section, related research approaches in the context of process performance analysis are presented and compared to the KPI dependency analysis approach as presented in Chapter 4. Section 2.3.2 presents runtime process adaptation approaches. In particular self-adaptation approaches are presented and compared to the approach of the thesis as presented in Chapter 5.

2.3.1 Process Performance Analysis

Process performance analysis deals with concepts, methods, and techniques which help analyzing the monitored business processes with the goal of optimizing the process performance. Approaches which deal with analysis of design-time process models only are not considered in the following, i.e., the focus is on approaches which analyze monitored data of process instance executions.

This type of analysis has been traditionally supported by BI tools. They typically use a data warehouse as basis and enable Online Analytical Processing (OLAP) analysis, interactive reporting, and data mining techniques. BI tools

have traditionally however not been integrated with process engines. Thus, several research approaches deal with combining process execution with BI concepts. In the following, after introducing data warehousing and data mining concepts, the relevant research approaches are presented and compared with the KPI dependency analysis presented in Chapter 4.

Data Warehousing. A data warehouse is a database which integrates information from multiple operational systems and makes it available for querying and analysis used for decision support [Inm02].

One can distinguish between a data warehouse, which collects data enterprise-wide, and a data mart, which focuses on one particular subject or department. Both are typically based on a multidimensional data model realized based on a star schema. The multidimensional model is based on the concept of a data cube, which consists of a large set of facts and a number of dimensions. OLAP operations such as slice-and-dice and drill-down can be implemented efficiently using the data cube structure often based on a star schema.

[CBC+06] presents a model-driven approach to BAM which uses a data mart as basis for a dashboard. This approach is used by the IBM Business Monitor [IBM15]. A metric model is specified based on events from different systems. The metrics model is used for configuring the runtime monitoring infrastructure, which evaluates the metrics in near real time and stores them in an operational data store. Frequently, the data is extracted from the operational store and loaded into a data mart, which is queried by a BAM dashboard. The data schema of the data mart is specific to the monitor model and is generated at design-time of the model before its deployment. Mostly, measured KPIs are mapped to fact tables, and business data and time are used as dimensions.

[CCDS07] presents a warehouse design for business process data. It does not focus on a specific workflow system, but tries to support a generic schema which can be used for different types of processes and implementations, and in particular support also non-automated processes. Therefore, an abstract process is modeled. The warehouse schema combines generic fact tables and process-specific fact tables. Generic fact tables are created for tasks and process instances, while specific business data types (e.g., invoice related data) are

stored in their own tables.

[KWL01] describes a design of a performance-related data warehouse for processes. Thereby, the facts model the KPIs (e.g., duration, customer satisfaction, turnover) and contain several measures (e.g., expectation, perception, performance gap in the case of customer satisfaction). The main dimensions associated to these facts are organization, customer, process, and time. For each dimension a hierarchy is built (e.g., for the process dimension: business unit hierarchy, business unit, process, activity). [LM04] shows how the latter approach can be used to implement a corporate performance measurement system which integrates business process information into a traditional warehouse.

[zM01] discusses how to design a process-oriented data warehouse that integrates workflow audit trail data with business object information. The issue arises as workflow internal data often only uses IDs of some business data stored outside of the workflow system in DBs (customer data with all its attributes). In that case they should be integrated into the warehouse. [RSN15] introduces a framework which provides an integrated view on process data generated by workflow systems and operational data stored in separate databases. This integrated view is based on a specialized federation layer and is reflected in a set of operators which are used for posing analysis queries to the integrated view.

The KPI dependency analysis approach developed in this thesis focuses on data mining techniques rather than data warehousing. Obviously, both approaches could be combined by storing monitoring data in a data warehouse schema thus enabling standard reporting and dashboard functionality, in addition to data mining based analysis as developed in this thesis.

Data Mining. Data mining deals with the discovery of patterns from large amounts of data, whereby the data is typically stored in databases or data warehouses [WF05]. It is an interdisciplinary field using techniques from areas such as data warehousing, machine learning, statistics, pattern recognition, and data visualization. The architecture of a data mining system typically includes three layers: the data store, the data mining engine, and a graphical user interface.

Data mining tools find patterns in the data that might take days or weeks for users to discover on their own, if at all. Data mining functionalities include [WF05] mining of association rules, correlation analysis, classification, prediction, clustering, time-series analysis, graph mining, and text mining, among others.

In particular interesting for the work in this thesis is *classification*, which is a form of supervised learning. Thereby, based on historical data a classification model is learned, which explains how the categorical labels of a set of historical observations depend on a set of explanatory attributes. Such a model can also be used for predicting the categorical labels of future observations. More details on classification learning and the concrete learning techniques, the decision trees, is given in Section 4.2.1 when the KPI dependency tree learning is presented. In contrast to classification, *regression* focuses on the prediction of continuous values (rather than categorical ones).

Most closely related to the KPI dependency analysis presented in the thesis is iBOM, a platform for business operation management developed by Hewlett Packard [CCSD05], as it supports both process monitoring, and analysis and prediction based on data mining. In [CCDS04] the authors give an overview and a classification of which data mining techniques are suitable for which analysis and prediction techniques. Thereby, also decision trees are mentioned as one supported technique, which is what is the focus of the analysis approach in this thesis. The platform presented allows users to define and monitor business metrics, perform intelligent analysis on them to understand causes of undesired metric values, and predict future values. The KPI dependency analysis approach of the thesis, as firstly presented in [WLR+09, WLR+11], focuses on the analysis of process-related KPIs using decision trees. Compared to the iBOM approach, the approach is different in that it focuses on BPEL-based processes and choreographies, explains in detail how KPIs are modeled and how explanatory metrics for these processes can be generated in an automated manner based on the process models. It focuses only on decision trees, but provides detailed experimental results. Also, iBOM does not deal with automated adaptation based on the learned decision trees.

[dLvdAD14] presents a framework for analyzing classification questions

based on event logs using decision trees. It is based on the ProM framework, which provides process mining support, in particular process discovery when there is no explicit process model a priori [vdAWM04]. The framework uses an event log as input. The user specifies an analysis use case by defining a classification problem, i.e., a class variable and a set of explanatory variables, based on event characteristics such as activity, case, resource, and timestamp. Based on such an analysis use case the event log is manipulated accordingly and fed into a decision tree algorithm, which generates a decision tree thus providing a classification model answering the question. The approach can be seen as more general than the KPI dependency analysis, as it can specify arbitrary classification problems based on the event log. The approach of the thesis presents a specific solution for KPI dependency analysis and integrates it tightly with monitoring and adaptation.

2.3.2 Self-Adaptive Processes

In this section, approaches which deal with runtime adaptation of processes are presented. The focus is thereby on self-adaptive processes, i.e., approaches where the adaptation decision is done in an automated manner at process runtime.

Runtime adaptation changes the behavior of the process instance as defined at design time and deployment time. The adapted subjects in the context of service orchestrations include (i) the control flow and data flow of the process instance itself, and (ii) the partner services, i.e., the concrete bound service at deployment time is exchanged for another service. The adaptation is enabled by adaptation mechanisms.

In the following, firstly, relevant approaches considering adaptation mechanisms are presented. Then, self-adaptation approaches are described whereby one can distinguish between reactive and proactive approaches.

Process Adaptation Mechanisms. [KLN+06] describes an approach for parameterized BPEL processes enabling dynamic binding of services using different strategies. Thereby, the invocation of partner Web services in a process can be parameterized at design time using four strategies, namely static, prompt,

query, and fromVariable. Thereby, the query strategy allows to specify functional and non-functional requirements on the service. The strategy query and fromVariable are executed automatically at runtime thus realizing dynamic binding of services. [KL09] presents BPELnAspects, an approach to runtime BPEL process adaptation based on AOP. Thereby, aspects which are weaved into the processes are arbitrary Web service operations which can be executed before, after, or instead of activities in processes. They also can overwrite the values of transition conditions and variables. The syntax is based on WS-Policy.

As already mentioned in the context of monitoring, the WS-BPEL 2.0 Event Model supports runtime adaptation via so called *blocking events* [KHK+11]. Blocking events are events which block an activity execution until they are unblocked by another event (potential parallel threads of the process instance are not blocked). This allows suspending the instance execution for a while and adapting the process execution, e.g., by skipping an activity, changing variable values, compensating a scope, and service substitution via writing another EPR to a partnerLink. In order to realize the blocking, some states and corresponding events had to be added. For example, when an activity becomes runnable its state changes from `Inactive` to `Ready` and a blocking event `Activity_Ready` is fired. Now one can unblock the activity by sending the event `Start_Activity`, or for example by sending the event `Complete_Activity` thus skipping the activity execution. If no blocking would be needed, the activity could right away change to the state `Executing` and start execution, and the state `Ready` would not be needed. The event model is supported by the *Pluggable Framework* [KKL07a]. Thereby, a *generic controller* is integrated with the BPEL engine and publishes events as they occur to a topic. In addition, it allows *custom controllers* to register for specific blocking events. When a blocking event occurs during process execution it is forwarded to the corresponding custom controller who then later has to send an unblocking event.

In this thesis, the Pluggable Framework is used as basis for monitoring and adaptation. The adaptation approach, as presented in Chapter 5, does not develop new adaptation mechanisms. It rather deals with how to select and combine *existing* adaptation mechanisms in order to optimize running process

instances in respect to KPI targets.

Reactive Adaptation. When it comes to runtime adaptation one can distinguish between *reactive* and *proactive* approaches [Met11]. Reactive approaches trigger adaptation after a certain undesirable event takes place, e.g., a failure during service invocation. Proactive approaches try to *predict* that a failure or undesirable situation, e.g., considering QoS constraints, will happen, and thus adapt in order to prevent that situation. The approach of the thesis is a proactive approach.

The reactive approaches mostly deal with self-healing after a service invocation failed or a QoS constraint was violated. [EM08] presents a policy-based framework for self-adaptable processes which allows to recover from functional faults during service invocation. The policy language is an extension of WS-Policy and allows specifying discovery and selection of services to be used and how to recover from potential faults using service rebinding at process runtime. [BGNS10] presents an approach to self-supervising BPEL processes which enables reacting to failures during process execution. Supervision consists of monitoring and recovery. Monitoring directives and recovery strategies are defined using policy-based languages based on WS-Policy. Recovery includes several strategies such as ignoring the fault, halting the process instance execution, retry, and rebind, among others. [ZPG10] presents a declarative framework for self-healing service compositions where an event calculus is used for specifying the functional and non-functional constraints. At runtime, monitoring is used for detecting violations by evaluating the event repository which contains the monitored message exchanges. The recovery mechanisms include reinstantiation and replanning of the composition.

As presented in [ACM+07], the Processes with Adaptive Web Services framework enables defining candidate services for invocation activities at design time. Thereby, also QoS constraints can be specified locally for each activity, and globally for the whole BPEL process model. At runtime, the process optimizer component selects candidate services in order to satisfy those QoS constraints. In addition, a self-healing module enables retries or substitutions in case of failures during invocation. [CDPEV08] presents another approach to QoS-aware

rebinding of services in a service composition. It uses genetic algorithms as basis for service selection and performs the rebinding at process runtime when monitoring shows that the actual QoS values deviate from initial estimates or when a service is not available.

Proactive Adaptation. Proactive adaptation includes as a first step the prediction of problems, after which one can analyze the reasons and decide on adaptation actions to execute.

[AZ12] presents an approach to proactive adaptation that uses service replacement when it predicts situations that may lead to unavailable services or services with a too high response time. Prediction is based on function approximation and failure spatial correlation techniques. The approach uses a composition template as basis and selects a set of candidate services to be used in the composition and their replacements. The rebinding supports the replacement of more than one service at once.

[HKMP08] presents a framework that uses online testing to trigger proactive adaptation in service-based applications. Test objects are partner services invoked by service compositions. If an online test fails, the framework triggers adaptation to avoid undesirable events. [SMF+11] presents a framework which combines monitoring, online testing, and quality prediction to enable proactive adaptation. The selection of services to be tested using online tests is performed based on usage frequency.

[dGAD14] presents QoS-based proactive adaptation for service compositions based on fuzzy logic. The adaptation model uses two fuzzy inference systems that evaluate the QoS values of service compositions, based on historical and freshly monitored data. The QoS properties considered are response time, cost, energy consumption, and availability.

There are several approaches which use data mining techniques to learn models based on history data, which are then used for prediction purposes and subsequent adaptation.

[ZLLC08] presents an integrated monitoring and prediction approach which uses machine learning techniques for prediction. It supports not only instance level prediction of metric values but also time series based prediction across

process instances. It however does not deal with adaptation.

[LWR+09] deals with prediction of SLA violations in service compositions. The prediction model for a numerical metric is learned based on historical data using Artificial Neural Networks (ANN). At process runtime, at specific checkpoints in the process instance the prediction is performed by inserting the monitored data into the ANN. [LMRD10] extends this work by using adaptation in order to prevent SLA violations. After an SLA violation is predicted at a checkpoint, the composition adaptor selects the available adaptation actions at the checkpoint, and repeats the prediction for each possible combination of those. Finally, the best fitting combination according to prediction result and adaptation strategy (minimal or safe) is selected and enacted. Data manipulation and service substitution as adaptation actions are supported. [LWK+10a] extends that work by supporting also process fragment substitution as adaptation action. Finally, [LHD13] in addition considers the costs of SLA violations and adaptations to prevent them when selecting the adaptation actions to enact.

The approach of the thesis is based on the approach presented in [WZK+12, KWK+09] and is similar to the previously described approach in that it uses an integrated monitoring, prediction, and adaptation framework, and the prediction is based on data mining. Major differences are as follows. Firstly, the focus is on KPIs, which have categorical values and thus the usage of decision trees as basis for prediction and adaptation. Secondly, the adaptation model is different in that it allows defining (i) several KPIs, for which the prediction and adaptation is to be done at the same time and (ii) a preferences and constraints model which allows specifying weights and constraints on KPIs and other metrics, which guide the selection and ranking of the adaptation strategies based on multiple attribute decision making techniques. Thirdly, the adaptation requirements and strategies are directly extracted from the decision trees rather than enumerating all possible combinations of adaptation actions and repeating the prediction with them. The approach of the thesis does not take into account the cost of adaptations explicitly. However, to a certain extent the cost could be taken into account by modeling a cost-related KPI with an appropriate weight and specifying in the impact model of the adaptation actions how they affect that KPI.

2.4 Summary and Conclusions

This chapter has presented background information and related research approaches needed for understanding the approach and the contributions of the thesis.

Firstly, an overview of the BPM, SOA, and Web services domains has been given. Several Web service specifications which are used as basis of the approach have been presented, in particular service orchestration with BPEL and service choreography modeling with BPEL4Chor.

The next section has described concepts, technologies, and related work in the context of process monitoring. This section is needed for understanding the monitoring approach of the thesis as presented in Chapter 3. Standards such as WSDM and technologies such as event processing have been introduced. Related research approaches in the context of process monitoring and cross-organizational process monitoring have been described.

In the last section, the process performance analysis and optimization topic is presented, which is related to Chapters 4 and 5 of the thesis. Related work in the context of process performance analysis and self-adaptive processes have been described and compared to the approach of the thesis.

CHAPTER 3

PROCESS MONITORING IN SERVICE CHOREOGRAPHIES

As motivated in the introduction (cf. Section 1.2), monitoring and evaluation of KPIs based on single service orchestrations is often not sufficient, as processes can be distributed across service orchestrations run by potentially different organizations in service choreographies.

This chapter presents a solution to that problem by describing an approach to monitoring of business processes in service choreographies. Runtime entities in service choreographies are exposed as manageable resources with corresponding properties. The approach is based on existing WS-* standards and extends them where needed. Concretely, the monitoring infrastructure is based on WSDM and BPEL4Chor is used as the service choreography language. Custom properties used for the definition of KPIs are specified based on event processing.

The presented monitoring approach is the basis of the overall framework, as it enables the evaluation of KPIs in choreographies, which are the focus of the analysis and adaptation phases as described in the following two chapters.

The chapter is structured as follows. Section 3.1 explains the motivation of the approach in more detail. Section 3.2 gives an overview of the monitoring framework and presents the monitoring lifecycle. In Section 3.3, the monitoring approach is presented in detail by describing the metamodel and its usage in the monitoring lifecycle phases. Finally, Section 3.4 concludes the chapter by summarizing the contributions.

3.1 Motivation and Objectives

In service-based applications, business processes are implemented based on services. In the scenario as shown in Figure 1.1 the services of the customer, the reseller, and the shipper interact with each other and there are even more services involved not shown in the choreography model. A service in a choreography can be implemented as a service orchestration (i.e., using an explicit orchestration model such as a BPEL process) or be a service implemented using an arbitrary programming language and described just in terms of a WSDL-based service interface. In the latter case, the implementation of the service is not explicitly modeled as in the orchestration case.

When it comes to monitoring, then services which are just described in terms of a service interface consisting of a set of operations enable monitoring of properties of the invocation of these operations, such as availability of the endpoint, duration of the operation invocation, and the outputs of the operation. Obtaining monitoring information on the implementation of the service is not possible without custom programming. If a service is implemented as a service orchestration running in a process engine then in addition monitoring of implementation aspects can be achieved. As discussed in Section 2.2.4, process engines typically offer event models, which enable detailed tracking of the execution of process instances. For example, if the reseller process is implemented as a service orchestration, then events on the start and completion of its activities and variable values can be obtained. Using monitoring languages one can then use those events for calculation of process performance properties such as the process duration.

In some cases, it is important to obtain monitoring events from more than

one service in order to calculate process properties. Consider, for example, the metric order fulfillment time, which could be measured in the scenario (cf. Figure 1.1) from the start of the activity `Receive PO` in the reseller process until the `While Loop` completes in the customer process. If the reseller wants to calculate that property, he needs to obtain in some way the event from the customer process when the last shipment arrived, as there is no explicit message interaction between those processes in the choreography. Thus, one needs to (i) ensure that both the reseller and the customer service expose the corresponding monitoring information and then (ii) correlate that monitoring information to calculate the process property.

Even if there exist orchestration models which expose the needed monitoring information, there should be a possibility to provide only a subset of that information for privacy reasons. This is because in cross-organizational settings, a service provider is normally willing to provide only monitoring information on its public processes, but not private processes. For example, the reseller process in the choreography model shown in Figure 1.1 does not expose how it interacts with its warehouse and payment services.

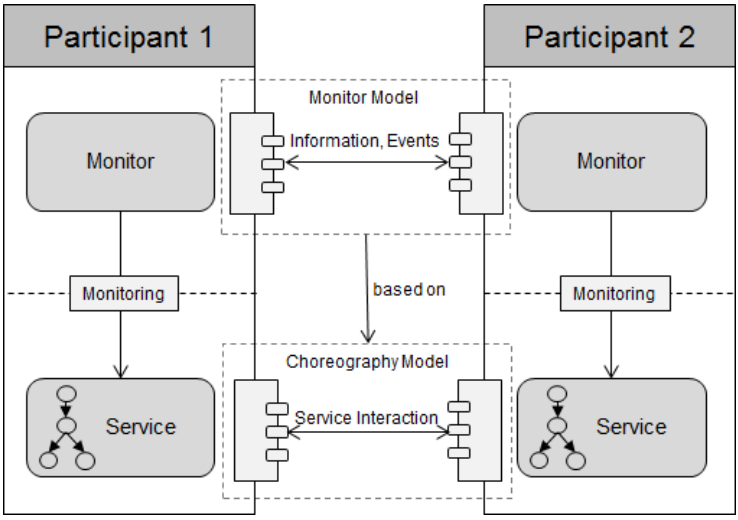

Figure 3.1: Monitoring Objective

In order to support these scenarios, the approach uses choreography models not only for agreeing on interactions but also as basis for monitoring, as sketched in Figure 3.1. Furthermore, it is assumed that an interconnected interface choreography model is created and is used as basis for interaction and monitoring. As discussed in Section 2.1.5 such a model specifies the public processes (abstract processes) of the participants and their connections via message exchanges. It is an agreement of the participants on how they want to interact and in addition can include internal activities of public processes, which are typically defined as opaque but give some information on what kind of business logic is performed in the process. Thus, in addition to interaction activities, also those public activities can be monitored. This is the reason why, in our context, an interconnected interface choreography model is more suitable than an interaction model, which just models the interactions but not the opaque activities.

As shown in Figure 3.1, services of different participants interact over service interfaces as described in the choreography model. Similarly, participants can interact over their monitoring interfaces to exchange monitoring information as described in a monitor model which is based on the choreography model. Therefore, each participant monitors its process and provides a monitoring interface. The monitoring interface provides monitoring information in terms of operations and events to other participants.

3.2 Choreography Monitoring Overview

As motivated in the previous section, the goal is to support monitoring of processes based on choreography models. The overall idea is to expose runtime entities (e.g., process activity instance, process variable instance) and their properties (e.g., activity instance state, variable instance value, process instance execution duration) in the choreography as manageable resources.

As shown in Figure 3.2, for each manageable resource definition, a manageability endpoint is exposed which supports accessing manageable resources of the manageable resource definition. A manageable resource is accessed over the manageability endpoint using an EPR and provides a set of capabilities which

expose properties of that resource over operations and events to manageability consumers.

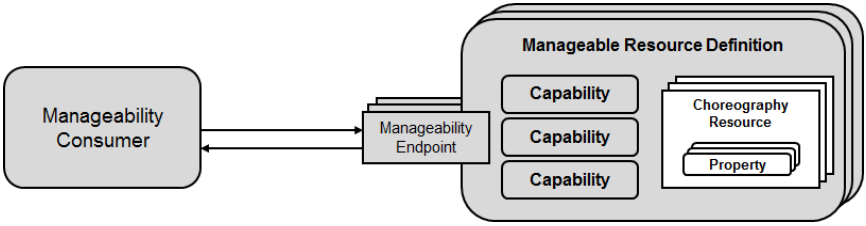

Figure 3.2: Monitoring Approach

As already explained in Section 2.2.2, an implementation technology for manageable resources in the WS-* platform is WSDM. It provides MUWS for management using Web services and MOWS as an extension for management of WSDL Web service endpoints. Similarly, here an extension for service choreographies is provided. That involves supporting a set of new resource types and capabilities. An example resource type in a choreography is an activity instance. A corresponding capability could then expose its state model to consumers, i.e., allowing consumers to query or even modify the current state of the activity instance over operations or notifying subscribed consumers on state changes.

In order to support the development of monitoring applications based on the resource types and capabilities, a monitoring metamodel has been defined. It specifies how monitor models are created. A monitor model uses provided capabilities to specify for a concrete choreography model which resources should be exposed as manageable resources, which properties should be provided and how they can be accessed.

3.2.1 Monitoring Method

In the following, a high-level overview of the monitoring method is given. Details are then specified later in the respective sections of the chapter. The steps from modeling to monitoring are depicted in Figure 3.3.

Choreography Modeling. The prerequisite of the monitoring approach is a

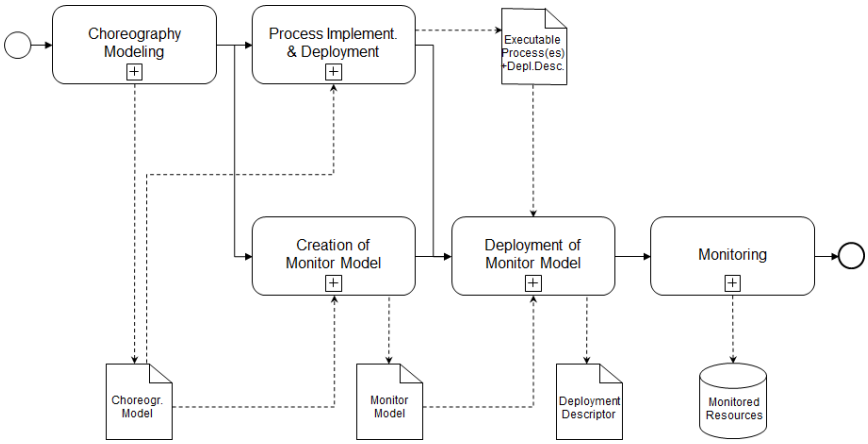

Figure 3.3: Overview of the Monitoring Process

choreography model. The monitoring approach assumes the existence of a choreography description based on the interconnected interface model. The choreography model consists of a set of interconnected abstract processes. Concretely, BPEL4Chor is used as choreography language (cf. Section 2.1.5).

The choreography model can be created either in a top-down or a bottom-up fashion. When using a top-down approach, one first creates the choreography model and then in later phases refines its abstract processes to executable processes (e.g., by refining the abstract BPEL processes to executable BPEL processes [DD04]). The top-down approach is typically used when (i) designing the process landscape from scratch or (ii) as an agreement between partners on how to interact in a cross-organizational scenario. The latter scenario is in particular important if choreography models are standardized in certain domains (e.g., RosettaNet PiPs [Ros]). Figure 3.3 depicts the top-down approach.

The bottom-up approach starts with already existing implemented processes and derives the choreography description bottom-up based on those processes. In both approaches, when designing the choreography model, it is important to include all activities which should later be monitorable.

Process Implementation and Deployment. The processes are implemented

as specified in the choreography. This can be done by refining the abstract processes to executable processes (e.g., by refining the abstract BPEL processes to executable BPEL processes [DD04]), when using the top-down modeling approach. But in general any implementation technology could be used, not necessarily a service orchestration language. The executable processes are deployed. Process deployment information is needed as input to monitor model deployment.

Creation of a Monitor Model. After the creation of the choreography model, a monitor model can be created. It defines the monitored resources and their properties which are to be monitored in a specific choreography and the monitoring mechanisms which allow accessing that information. The monitor model uses a set of monitoring capabilities provided by the monitoring infrastructure.

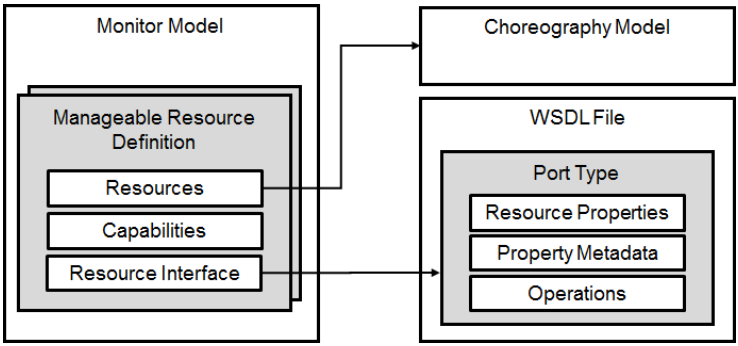

Figure 3.4: Monitor Model

The monitor model is created for each choreography model separately based on the monitoring goals. One possible goal is to allow monitoring consumers to track the execution of the choreography; another goal is the evaluation of KPIs for assessing the process performance. A standardized monitor model which simply monitors everything in the choreography that is available based on the monitoring capabilities is not realizable for several reasons: (i) process properties (used for definition of KPIs) are often process-specific and cannot be standardized (e.g., the calculation of order fulfillment time in the scenario is not simply the duration of the reseller process instance but the duration

between two concrete activities in the specific choreography model, namely the start of `Receive PO` in the reseller process model and the completion of `While Loop` in the customer process model); (ii) monitoring of all possible events of process execution can have a serious impact on process performance; (iii) some of the available monitoring information might not be provided due to privacy constraints.

For these reasons, the user creates a *specific* monitor model for a choreography by defining which resources and corresponding properties of a choreography should be monitored to achieve a certain monitoring goal. As shown in Figure 3.4, the monitor model is defined in terms of a set of *manageable resource definitions*. Each such manageable resource definition is specified based on a *resource type* (e.g., Activity Instance) for a set of *resources* of a specific choreography (e.g., all activity instances of the reseller process). By selecting particular *capabilities*, it is then specified which *resource properties* should be exposed for these resources via *operations* and *events*. A corresponding resource interface, specified as a WSDL `portType`, is created for each manageable resource definition. The `portType` defines resource properties, corresponding metadata information, and WSDL operations. The contents specified in the `portType` are provided by the functionality of the chosen capabilities, or are a subset of that functionality, i.e., not every resource property or operation a capability supports has also to be used in the resource interface. The resource interfaces provided as WS interfaces as defined in the monitor model are created in addition to the WS interfaces of the participants as specified in the choreography model (as also sketched in Figure 3.2).

Monitored properties range from basic properties such as simple state changes, which are domain-independent (i.e., can be used for any choreography model) and are already predefined for the available resource types to custom properties, which are defined based on other properties (e.g., metrics such as average duration between two activities). Those custom properties are defined specifically for a choreography model (e.g., the metric order fulfillment time).

If a monitor model is created between participants in a cross-organizational scenario, it is assumed that participants agree on the monitor model similarly as they agree on the choreography model for interactions. The monitor model

can thereby contain manageable resource definitions provided by different participants in the choreography, e.g., one such definition can be defined for resources in the reseller process and be offered by the reseller, another one can be defined for resources in the shipper process and be offered by shipper. Such a monitor model can in this case be seen as a *monitoring contract* between these participants.

On top of the monitor model, one can add other models such as KPI definitions, SLAs, and dashboards in order to create a monitoring solution used for achieving a particular monitoring goal. In the overall approach, the monitor model is also used as basis for KPI analysis and adaptation purposes.

Deployment of the Monitor Model. Deployment of the monitor model assumes that the corresponding processes of the choreography have been implemented and deployed first. In the deployment phase, the monitor model is deployed to a monitoring infrastructure.

Deployment involves creating a deployment descriptor adding additional information to the monitor model such as concrete endpoint addresses of the monitoring interfaces and capability implementations and their configuration. For example, if an abstract BPEL process has been implemented as an executable BPEL process, then one has to specify that process and where it has been deployed in the deployment descriptor. Also a mapping between abstract process elements and executable process elements might be necessary (cf. Section 6.1.1).

In case the monitor model contains definitions of several participants from different organizations, then each participant deploys its corresponding manageable resource endpoints of the monitor model.

As a result of the deployment, a set of manageability endpoints is deployed and can be used for monitoring. As deployment is specific to an implementation technology, it is described in more detail in the chapter on implementation and evaluation (cf. Section 6.1.1)

Monitoring. In the monitoring phase, the resources and their properties are monitored. The endpoints can be used for pulling the information via operations or subscribing to events.

3.3 Monitoring Metamodel

This section defines the monitoring metamodel which is used for the creation of monitor models.

3.3.1 Overview

In the following, a short overview of the main concepts is given. In the next sections these concepts are then described in detail.

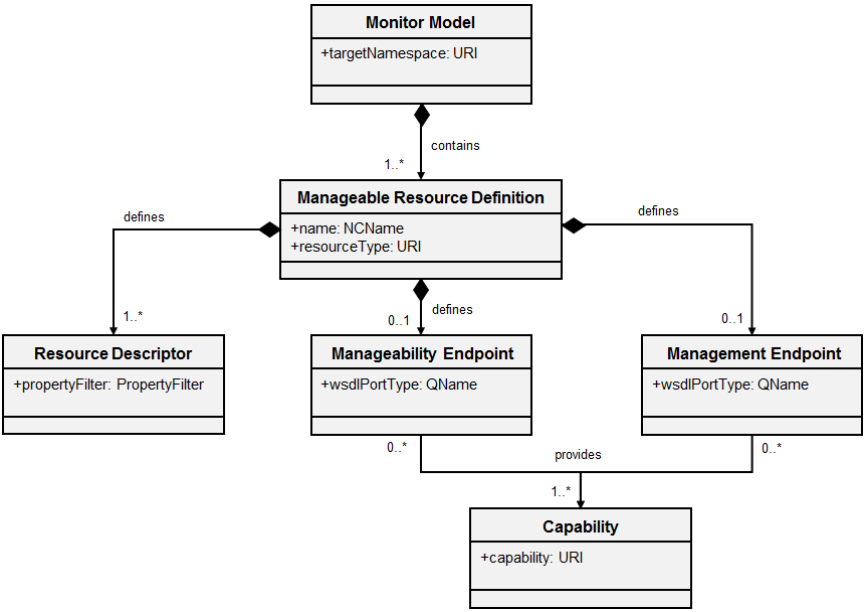

Figure 3.5: Monitoring Metamodel

Figure 3.5 shows the main concepts of the monitoring metamodel in a UML class diagram. A *monitor model* defines a non-empty set of *manageable resource definitions*. A manageable resource definition is defined for a *resource type*. There is a set of predefined resource types (e.g., Activity Instance), but it is also possible to define custom resource types.

The *resource descriptor* specifies the concrete resources (of the resource

type) which should be exposed as manageable resources (e.g., which concrete activities of a process should be manageable). The resource descriptor is specified in terms of a *property filter* by specifying concrete values on the subset of properties of the resource type.

A manageable resource definition can define two types of endpoints. The *manageability endpoint* is provided by each manageable resource (as specified in the resource descriptor) while a *management endpoint* is used for all manageable resources together.

For each endpoint an interface is defined via a WSDL `portType`. The resource interface includes a subset of resource properties, metadata information on the properties, operations, and topics as supported by the capabilities. Each endpoint definition specifies the capabilities it offers. For a resource type there is a set of predefined capabilities which can be used when specifying the monitor model. From this set a subset is chosen for a manageable resource definition.

XML Serialization of the Monitoring Metamodel. A monitoring metamodel is serialized as follows.

Listing 3.1: Monitoring Metamodel Pseudo XML Schema

```
<monitorModel targetNamespace="URI" name="NCName"
xmlns="http://www.iaas.uni-stuttgart.de/m4c/schemas/monitorModel">
 <manageableResourceDefinition name="NCName" resourceType="URI">
  <resourceDescriptor>...</resourceDescriptor> +
  <manageabilityEndpoint resourceInterfacePortType="QName">
   <capability uri="URI">...</capability> +
  </manageabilityEndpoint> *
  <managementEndpoint resourceInterfacePortType="QName">
   <capability uri="URI">...</capability> +
  </managementEndpoint> *
 </manageableResourceDefinition> +
</monitorModel>
```

Example. An example monitor model is shown in Listing 3.2. It defines one `manageableResourceDefinition` element for managing resources of the

resource type `ActivityInstance`. The `resourceDescriptor` element defines which concrete activity instances are to be managed; this is done by defining a `propertyFilter` on the resource identifying properties. In this case, all activity instances of the reseller process of the purchase order choreography are to be managed. For the monitoring of shipper activities one would have created another manageable resource definition, as the corresponding endpoints have to be deployable independently of each other and are implemented by different service providers, i.e., the reseller and the shipper, respectively. There can however be cases where several process models in a choreography are implemented by the same service provider; in that case one manageable resource definition could include activities of several process models.

Listing 3.2: Monitor Model Example

```
<monitorModel name="POChoreographyMonitorModel">
 <manageableResourceDefinition name="ResellerActivities"
     resourceType="http://www.iaas.uni-stuttgart.de/m4c/
     resourceTypes/ActivityInstance">
  <resourceDescriptor>
   <propertyFilter>
    <m4c:topology>po:POChoreography</m4c:topology>
    <m4c:process>reseller:ResellerProcess</m4c:process>
   </propertyFilter>
  </resourceDescriptor>
  <manageabilityEndpoint resourceInterfacePortType="m4cint:
      ActivityPortType">
   <capability uri="http://www.iaas.uni-stuttgart.de/m4c/
       capabilities/ActivityStateCapability" />
  </manageabilityEndpoint>
  <managementEndpoint resourceInterfacePortType="m4cint:
      ActivityStateEventPortType">
   <capability uri="http://www.iaas.uni-stuttgart.de/m4c/
       capabilities/ActivityStateEventCapability" />
  </managementEndpoint>
 </manageableResourceDefinition>
</monitorModel>
```

The `manageabilityEndpoint` element specifies which capabilities should be provided by the manageable resources. The endpoint interface is specified by referencing a WSDL `portType` and listing a subset of capabilities provided by the resource type. In the example, the `ActivityStateCapability` has been used. The `ActivityStateCapability` exposes a `state` resource property and corresponding operations and topics for monitoring the activity state. In the corresponding WSDL file, the `ProcessActivityPortType` defines that resource property and includes the provided operations.

The `managementEndpoint` element provides the capabilities for management functionality used independently of a specific activity instance. In this case, the `ActivityStateEventCapability` is provided and the corresponding resource interface is specified. Based on this definition at runtime a management endpoint is deployed which publishes activity state change events for all activity instances resulting from the definition in the resource descriptor.

3.3.2 Resource Types in BPEL4Chor Choreographies

In the following, resource types in choreography monitoring are defined, i.e., it is specified how the runtime choreography entities are mapped to resources and resource properties and corresponding capabilities. Thereby, BPEL4Chor is used as choreography language, however, in principle, any other choreography language supporting the interconnected interface model could be used.

Process Execution in BPEL4Chor. BPEL4Chor choreographies [DKLW07] are used as monitored subjects. A BPEL4Chor participant topology (topology, for short) defines how a set of participants interact with each other. Each participant is defined in terms of a participant behavior description, which is an abstract BPEL process model. The abstract BPEL process model has the semantics of an observable behavior profile of BPEL with the exception that partner links are not allowed. The interactions between participant behavior descriptions are modeled in terms of message links.

A choreography topology results at runtime in a set of choreography executions. A choreography execution represents one particular execution of the participant processes and their interactions as defined in the topology, i.e., it

consists of a set of process instances whereby each participant instance corresponds to a participant behavior description and is executed by a participant. In the scenario, a choreography execution represents one particular purchase order and spans process instances of the customer, the reseller, and the shipper. A choreography execution can result in several process instances of the same participant, e.g., the shipper process can be instantiated several times per purchase order if the order is split into several shipments.

A process instance is executed according to the operational semantics of BPEL. The execution is exposed using an event model which provides events on instance entities and their state changes as they are executed. The WS-BPEL Event Model 2.0 [KHK+11] (cf. Section 2.2.4) defines state models and corresponding events for the following *instance* entities: Process, Scope, Activity, Invoke Activity, Loop, Link, Variable, Message Variable, Partner Link, and Correlation Set. The events which are fired to state transitions in the event models, carry identifiers of the instance entities they refer to.

To summarize, there is a need to create resource types for (i) BPEL process instance entities including its child entities, and (ii) deal with choreography executions (which span several process instances).

Resource Types. Based on the BPEL event model 2.0, the following resource types have been defined: *Process Instance*, *Activity Instance*, *Link Instance*, and *Variable Instance*. Thereby, all activity types, i.e., scopes, invoke activities, loops, and all other activities, have been combined into the resource type Activity Instance. Also all variable types, i.e., variable, message variable, and correlation set, have been combined into the resource type Variable Instance. Partner link is not included as it is not used in BPEL4Chor. The state models of the corresponding entities are mapped to a resource property *state* (cf. Section 3.3.4). Thereby, e.g., a resource of the type Activity Instance representing a loop activity has a different state model than a resource of the same type representing an invoke activity. In addition, the Variable Instance resource type contains the property *value* for holding the variable value. The events carry a set of properties which are used for identification of the corresponding instance entity. Those properties are also mapped to resource properties and are used

for resource identification purposes (cf. Section 3.3.3).

In addition to process models, which result in process instances at runtime, a BPEL4Chor choreography also defines a topology and participants. A topology can be seen as resulting at runtime in a set of choreography executions. For the monitoring of choreography executions, the resource type *Choreography Interaction* has been defined. A Choreography Interaction groups a set of interacting process instances (as specified in the topology) by specifying how a choreography interaction is identified based on process data exchanged between the process instances. As that process data is different for each choreography model, Choreography Interaction is a custom resource type, which is used specifically for a concrete choreography model based on monitoring objectives, e.g. for calculating domain-specific KPIs across process models. In the purchase order processing scenario, one could define several choreography interactions: an interaction between the customer and the reseller, an interaction between the reseller and a shipper, or also an interaction consisting of all those interactions together as part of a specific purchase order. Choreography interactions thus can be defined on different granularity levels (depending on monitoring objectives) and are specified implicitly by defining how a choreography interaction is identified based on process data exchanged between the process instances.

Considering resource identification in choreographies, two more items of information have to be specified, namely (i) by which participant a process instance has been executed and (ii) in which topology a process instance has been executed. Resource identification and definition of choreography interactions is discussed in the following subsections in more detail.

Finally, an additional resource type *Custom* has been added. It is needed, in some cases, when defining composite properties across resources of existing resource types (cf. Section 3.3.5), e.g., across choreography interactions (e.g., average order fulfillment time) or even choreography models.

Example. Figure 3.6 shows an example on how resources are created during choreography execution. The choreography consists of three process models (customer, reseller, and shipper). At runtime, the figure depicts one particular choreography execution (i.e., one particular purchase order). It results in four

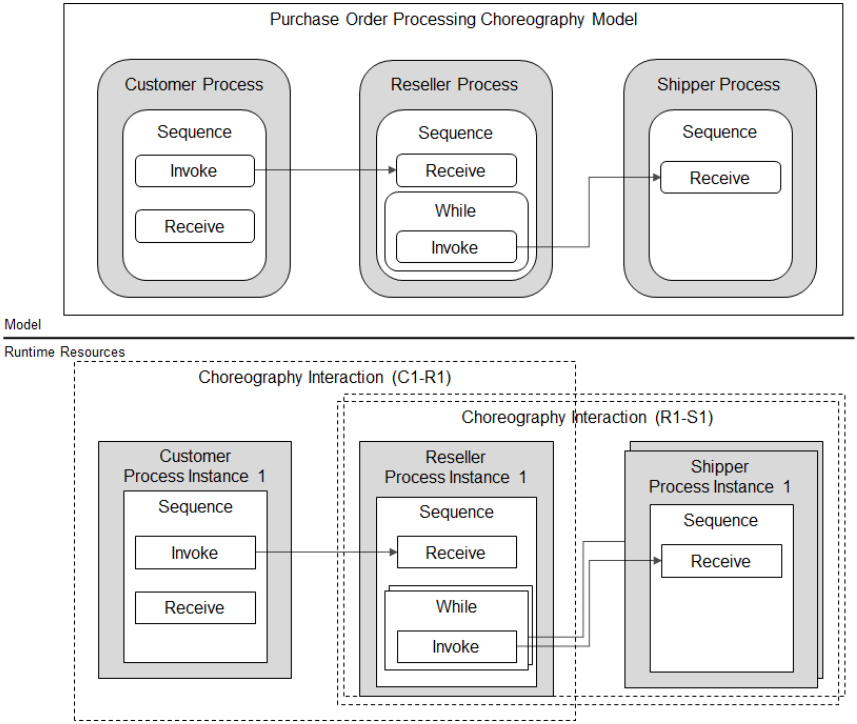

Figure 3.6: Resource Creation Example

resources of the resource type Process Instance. For the customer process model, four resources have been created, a process instance resource, and three activity instance resources. For the reseller process model, which contains a while activity, at runtime two corresponding while instance resources have been created, as the loop is executed two times. In each of the loop iterations a process instance of the shipper process model is instantiated, thus resulting in two process instance resources for the shipper process model.

In addition, the Figure shows three resources of the resource type Choreography Interaction, one is created for the interaction between the customer and the reseller, the other two are created for the interaction between the reseller and the shipper for each shipment. The first choreography interaction could

be used for the correlation of the customer and the reseller process instance based on the shared purchase order identifier and the calculation of the order fulfillment time. The other two choreography interactions could be used for the correlation of the reseller and the shipper process instance based on the shared shipment identifier and the calculation of the shipment time.

3.3.3 Resource Identification

Each resource needs one or more properties which allow identifying the resource. In our context, resource identifiers are used for two purposes. Firstly, they are needed for accessing resources over their manageability endpoints and are therefore defined as part of the endpoint reference of the manageability endpoint. In case of SOAP based access, the resource identifier is mapped to reference parameters. In case of resource-related events, the event contains the resource identifier as part of its event properties. Secondly, resource identifiers are needed for storing resources in a monitor database acting thereby as primary keys.

There are alternative ways to define resource identifiers. Resource identifiers can be (i) purely technical identifiers or (ii) domain-specific identifiers (consisting potentially of a set of properties). Also several identifiers can be used at the same time. For example, a technical identifier can be used in the database, while a domain-specific identifier is defined as part of the EPR. In our context, for example, for a specific activity instance one could generate a Universally Unique Identifier (UUID) as a technical identifier, or use a set of properties which identify the resource in respect to its model element (e.g., process QName) and the instance elements the resource is part of. In the former case, the identifier does not reflect the *semantics* of the resource (i.e., that the resource is an activity instance which is part of a certain process model) while in the latter case the resource is identified in respect to its process model and instance element. In the following, the approach focuses on the latter case, as there is a need for domain-specific identifiers for event correlation purposes, but note that in practice both approaches could be combined.

Based on the resource identification properties, the goal is to be able to deduce

which particular entity instance of a choreography execution the resource represents. That includes (i) the model element in the choreography the resource represents (e.g., in case of an activity instance to which particular activity, in which process and choreography model it belongs) and (ii) the instance element it represents, as a model element can result in several instances at runtime.

Resource Identification Properties	Choreography Interaction	Process Instance	Activity Instance	Link Instance	Variable Instance
Topology QName	X	X	X	X	X
Participant (Set)		X	X	X	X
Participant EPR		X	X	X	X
Process QName		X	X	X	X
Process ID		X	X	X	X
Scope XPath			X	X	X
Scope ID			X	X	X
Activity XPath			X		
Activity ID			X		
Element XPath				X	X

Table 3.1: Predefined Resource Identification Properties

Table 3.1 shows the resource identification properties for each predefined resource type. Each resource is identified by a set of properties identifying its model representation and a set of properties identifying the concrete instance. For example, the model representation of an activity instance is identified using the topology QName, the participant reference name (or alternatively participant set name), process model QName, and the XPath expression identifying the activity definition in the BPEL XML document. The instance of an activity is identified using the choreography interaction ID (domain-specific, thus not shown in the table; discussed further below), the participant EPR (if participant is part of a participant set), process (instance) ID, the scope (instance) ID of the innermost scope, and an activity (instance) ID. The two latter IDs are needed because there can exist several instances of the same activity instance per

scope execution. The instance IDs are technical IDs generated by the process implementation which executes the instance (i.e., typically the process engine). Thereby, it should be made sure that the process ID is globally unique, while other IDs can then be unique only within the process instance. One should also note that not each of the model element identifying properties is necessary for each resource type. For example, the process QName is not needed for identifying an activity instance as the process ID could be used for finding out the process QName by accessing the corresponding process instance resource or reading the properties of a corresponding process instance event, which contains the process QName property. We have decided to include all identification properties of the corresponding model element for practical reasons. Thus, each resource provides complete information on its model element and does not require correlation with other resources, which can be cumbersome.

The IDs corresponding to process entities have already been described in the BPEL event model 2.0 (cf. [KHK+11] for more details). That set has been extended to support choreographies. Firstly, the topology QName specifies in which choreography a resource has been executed. This property is in particular needed if processes with the same QName can be used in different choreographies. Secondly, a participant name and participant set name are needed as a process model can be used by several participants or participant sets. In case of participant sets, in addition a participant EPR is needed to specify the concrete participant which has executed a particular process instance in that participant set.

Thirdly, there is a need for identifying a particular choreography interaction for being able to correlate process instances which interact with each other. If each process instance is identified via a technical process ID, then obviously correlation between those process instances is not possible without further information, because a process instance does not know the technical process instance IDs of other process instances it interacts with in a choreography. A correlation in that case can be done based on information which is exchanged between the process instances over message interactions.

Thereby, properties have to be determined which are either part of the message payload or transported via the message header. For example, in the

scenario, the purchase order ID is exchanged between the reseller process and the customer process, and the shipment ID is exchanged between the reseller process and the shipment process. These IDs are thus known in both process instances and can be used to correlate those two process instances, e.g., for calculating the duration between activities in these process instances. This duration property is then assigned to a manageable resource definition with the resource type Choreography Interaction which has the purchase order ID as identifier. A concrete example is described in more detail in Section 3.3.5.

If no unique identifier is part of the message payload, then the correlation (between two interacting process instances) can only be done based on technical identifiers on protocol level. In synchronous invocations (BPEL invoke with input and output) the correlation between the sent and replied message is done on protocol level, e.g., SOAP/HTTP and not based on message payload (which does not necessarily contain needed identifiers). For SOAP-based communication this technical identifier can be transported in the SOAP header. Obviously, the corresponding middleware, e.g., BPEL engine and service bus [Ley05], would have to be adapted to include and read the identifier during message exchanges, and also write it into corresponding published events. The approach of the thesis does not deal with technical identifiers for choreography interactions, but assumes that the correlation can be done based on message payload.

3.3.4 Capabilities

Each manageable resource exposes a number of capabilities. The resource type thereby determines which capabilities are meaningful and are supported. A capability is the atomic unit of functional definition for a resource type and is identified by an URI. It defines a set of properties, operations, events and metadata items which are used for managing a certain aspect of the resource. Those elements are specified in accompanying files containing the XML schema definitions, WSDL definitions, and metadata descriptors. An endpoint interface of a manageable resource is defined by combining a set of capabilities.

In the following, it is discussed which existing capabilities are most useful in the context of the thesis and which new capabilities have been defined.

Table 3.2 shows the new capabilities and the corresponding resource types. For each capability the properties and operations are given. The capabilities also provide topics which are not shown in the table.

Coarsely, the capabilities can be grouped as follows: (i) capabilities dealing with the identification and description of the resource, (ii) capabilities for managing the state of resources and the value of variables, (iii) capabilities for evaluation of custom properties.

Identification and Description of Resources. Each resource type has the mandatory WSDM capability *Identity*, which exposes a URI-based `ResourceId` property giving a unique identifier to the manageable resource. In our approach, a UUID for each resource is generated. In addition to Identity, other useful WSDM capabilities which can be reused are *Manageability Characteristics* and *Correlatable Properties*.

Capability	**Resource Type**	**Properties**	**Operations**
Process State	Process Instance	State, LastStateTransition	triggerStateTransition
Process State Event	Process Instance		
Activity State	Activity Instance	State, LastStateTransition	triggerStateTransition
Activity State Event	Activity Instance		
Link State	Link Instance	State LastStateTransition	triggerStateTransition
Link State Event	Link Instance		
Variable State	Variable Instance	State LastStateTransition	triggerStateTransition
Variable State Event	Variable Instance		
Variable Value	Variable Instance	Value	setValue
Custom Property	*All Resource Types*	*Custom Property*	
Event Composition	*All Resource Types*		
Blocking Event	*All Resource Types*		registerForBlockingEvents unregisterForBlockingEvents

Table 3.2: Capabilities for Choreography Monitoring

Resource State. Each entity type defined in the BPEL event model 2.0 has a corresponding state model [KHK+11]. A state model consists of a set of states and state transitions. A state transition is signaled using events. Thereby, outgoing events are always fired by the process engine, while incoming events can stem from external applications who wish wo actively influence state transitions and thus the process execution. Based on the capabilities and state models of the BPEL event model, the goal is thus to (i) expose the state model using resource properties, and (ii) enable manageability consumers to cause state transitions actively when supported by the event model.

For each of the resource types, a corresponding capability for managing the state has been created (cf. Table 3.2, e.g., *Activity State*). The capability exposes one resource property *State* for representing the current state and one property *LastStateTransition* representing the last state transition. The state models are adopted from the BPEL event model. Both properties can be accessed by using WSRF operations and topics.

The BPEL event model 2.0 supports triggering of state transitions by external applications via incoming events. For example, the process instance state model defines that in the state Ready the process execution can be blocked waiting for incoming events. The incoming event Start_Activity triggers a state transition leading to the state Executing while Skip_Activity leads to the state Completing, thus skipping the activity execution. In both cases, the outgoing event Activity_Executing is fired.

For supporting this feature, a WSDL operation *triggerStateTransition* is provided. It receives the name of the incoming event as parameter and sends it to the engine. In the above example, one would invoke that operation with *Start_Activity* or *Skip_Activity*, respectively. The manageability consumer first has to register for blocking events. Therefore, an operation *registerForBlockingEvents* is provided, which is supported by the capability *Blocking Event*, which is a management capability assigned to the management endpoint. The manageability consumer uses that operation to subscribe for corresponding blocking events, which block the process execution. One should note that alternatively standard WSRF operations could have been used for enabling the triggering of state transitions, such as *setResourceProperties*. In that case, the

implementation of those standard operations would have to make sure that only the allowed properties may be changed (e.g., the state). As in our case there are only few properties which may be changed (state and variable value) and the change has side effects (a change of the state unblocks and resumes the blocked process instance), we have taken the design decision not to use the standard WSRF operations, but offer special operations whereby it is more obvious how to use the operation and what the operation does.

For accessing resources over their manageability endpoints and querying their state (either over operations or by subscribing to events), the manageability consumer first has to know that the corresponding resource exists and have the EPR of the resource endpoint. In our case, the resources are dynamically created at runtime as the choreography is executed. The consumer has several possibilities for *discovery* of resources: (i) advertisement of resource creation via events, (ii) a resource registry, and (iii) navigation via relationships. All of those approaches could be used in our case. In the prototype implementation, the first approach has been used. For each resource type, a management-related capability (e.g., *Activity State Event*) has been defined, which allows subscribing for all events of the corresponding manageable resource definition. This capability is assigned to the management endpoint. Thus, on the receipt of the first state event of a resource, the consumer knows that the resource has been created and uses the resource identification properties (which are part of the event) for accessing the manageable resource. For example, the consumer can use the Activity State Event capability for subscribing to all activity state change events of a certain process. After receiving the first event of an activity instance, he can use the Activity State capability for retrieving the current value of the state property or trigger state transitions.

Other Capabilities. In addition to the state capabilities several other capabilities have been created. The capability *Variable Value* manages the variable value of the corresponding resource of the resource type Variable Instance. The *Variable State Event* capability provides events which contain the new value of the variable when it has been changed. Finally, the capabilities *Event Composition* and *Custom Property* allow evaluating custom properties based on events;

these two capabilities are discussed in detail in the next section.

3.3.5 Custom Properties

So far, properties have been evaluated based on the events resulting from the process execution based on the BPEL event model. This in particular includes state changes of activities and values of process variables. Those *simple events* published based on the BPEL event model can be used as basis for evaluating *custom properties*. A custom property is, for example, the duration between two activities. It can be calculated based on the timestamps of two simple events which are fired on particular state transitions of the two activities.

In order to evaluate custom properties, simple events can be recursively correlated and aggregated to *composite events*. Event correlation and composition is a well-known topic in the area of CEP, and there are different languages available for the specification of composite events [Luc02]. In this thesis, the event processing language of ESPER (cf. Section 2.2.1) is used. ESPER is the CEP implementation used in the prototype. However, alternatively, any other language could be used instead.

Composite events are specified based on simple events provided by the BPEL event model (or other composite events) using an event processing language. The language provides an extensive set of functions (arithmetic, aggregation, relational) for calculating a new property value. The value of the custom property is extracted from the corresponding event property.

The definition of composite events is supported by the capability Event Composition, which is assigned to the management endpoint of the manageable resource definition. A custom property is then specified by using the capability Custom Property, which is assigned to the manageability endpoint. It specifies the event (simple event or composite event) and defines then how an event property value is to be mapped to the custom property value.

Definition Process for Custom Properties. In contrast to a predefined property, a custom property definition has to specify how the property value is evaluated. This includes in particular (i) specifying the consumed event (which contains the custom property as event property) by referencing the corre-

sponding manageable resource endpoint and topic and (ii) defining the event property which contains the custom property value. If the existing events are not sufficient, then first a new composite event has to be specified using event composition.

The steps for defining a custom property are as follows:

1. *Identification and definition of source event(s):* The first step towards creation of a new custom property, is to define the underlying *source events* which are needed as basis and the corresponding manageable resource definitions. This can be either only one existing event from which a property is to be extracted or several events which have to be composed first using event composition. In both cases, one has to make sure that the corresponding manageable resource definitions are part of the monitor model. For example, for the calculation of the *duration* between two activities, for the corresponding activities a manageable resource definition has to be defined which contains the capability Activity State Event.

2. *Specification of the corresponding manageable resource definition:* The new custom property has to be defined as part of a manageable resource definition. If the property cannot be assigned to one of the existing definitions, one has to define a new one for the custom property. If the predefined resource types cannot be used, then a custom resource type is defined. For example, the calculation of properties assigned to a choreography interaction needs the definition of a new manageable resource definition representing that choreography interaction. To the created manageable resource definition one then adds the capabilities as specified in the next two steps.

3. *Specification of the Event Composition:* If the existing events are not sufficient, one uses the capability Event Composition to specify a new event based on existing events. This is done by consuming events specified in the first step and then defining an EPL statement for creating a new event. Finally, one defines the target topic to which the event is published.

The Event Composition capability is added to the management endpoint of a manageable resource definition.

4. *Specification of the Custom Property:* Finally, one specifies the custom property. This includes adding the Custom Property capability and defining how the property value is extracted from an event property of the consumed event. One then specifies a corresponding resource property by defining its name and type in XML-Schema and defines metadata for that property. Metadata includes in particular defining the *unit* (e.g., seconds, hours). This includes adding the composite property to the corresponding resource interface (WSDL file).

Example. Figure 3.7 sketches a monitor model which defines the custom property `OrderFulfillmentTime`. The custom property is calculated by subtracting the timestamps of two events, namely the receipt of a new purchase order in the reseller process and the receipt of the order at the end of the customer process. As these two activities are part of different process instances, the two events cannot be correlated based on their own event properties. It is assumed that the purchase order ID is exchanged between the two processes via messages and is held in process variables in both processes. As basis for the calculation, thus four events are needed: two activity state change events of the corresponding activities for calculating the time difference, and two variable state events of the two processes which both hold the purchase order id for correlation.

Therefore, the corresponding four manageable resource definitions are created (cf. Figure 3.7). Each of the definitions includes the capability for publishing activity state changes and variable state changes, respectively. A new manageable resource definition `CustomerResellerInteractionDefinition` is created for the custom property. It specifies a new composite event by using the `EventCompositionCapability`. The event is specified by composing the four events mentioned before. Based on this event definition, the custom property is defined using the `CustomPropertyCapability` which extracts the corresponding property value from the event. Finally, as the created manageable resource definition has the resource type Choreography Interaction,

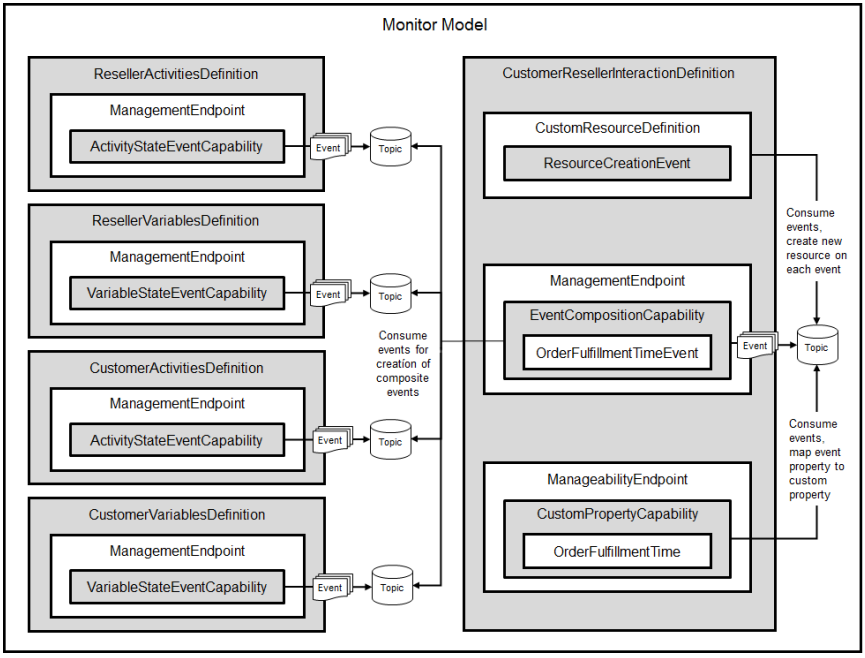

Figure 3.7: Custom Property Definition Example

which is a special case of a custom resource type, one has to define how the resources of this type are created and identified (`CustomResourceDefinition` element). More details are described in the rest of the section.

Composite Events and Custom Properties. A composite event is defined by using the capability *Event Composition*. When a composite event is to be specified for a manageable resource definition, the capability is inserted into the management endpoint definition and configured by defining a set of *composite event* definitions. Each such composite event definition includes the following three elements:

- *Consumed events:* Firstly, one has to specify the names of the manageable resource endpoints and corresponding topics from which the events should be aggregated. Required events can be simple events or composite

events. Each event is a `ManagementEvent` and carries the identification properties of the corresponding resource.

- *Event composition statement:* This is an EPL statement which correlates and aggregates the consumed events to a new event. The new event contains at least the resource identifying properties (of the corresponding resource type) and the calculated custom property.

- *Target topic:* Finally, one has to specify a topic to which the new event is to be published.

On deployment of such a Composite Event definition, the monitoring framework subscribes at the topics of the consumed event definitions. The event composition statement is registered in the CEP engine. It should be noted, that at runtime, for such a composite event definition, not a single composite event, but an *event stream* of composite events is created, one per resource of the corresponding resource type. In the example, the result is a stream of `OrderFulfillmentTimeEvents`, whereby each event is associated with one particular purchase order.

A custom property is defined by using the *Custom Property* capability. When a custom property is to be specified for a manageable resource definition, the capability is inserted into the manageability endpoint definition and configured by defining a set of *property* definitions. Each such property definition includes the following elements:

- *Name:* The name of the property is specified as a QName. The property is defined in XML schema and is defined as resource property in the WSDL interface of the manageability endpoint.

- *Consumed event:* The consumed event is the event which contains an event property which is used for the definition of the custom property. The consumed event can be a basic event or a composite event.

- *Event property:* This element specifies using XPath which event property is to be used as basis for the custom property.

Listing 3.3: Event Composition Example

```
<capability uri="http://.../EventCompositionCapability">
 <compositeEvent targetTopic="poChoreo:OrderFulfillmentTime">
  <consumedEvent manageableResourceDefinition="ResellerActivities"
   topic="ActivityEventCapability" alias="ResellerActivityEvent"/>
  <consumedEvent manageableResourceDefinition="ResellerVariables"
   topic="VariableEventCapability" alias="ResellerVariableEvent"/>
  <consumedEvent manageableResourceDefinition="CustomerActivities"
   topic="ActivityEventCapability" alias="CustomerActivityEvent"/>
  <consumedEvent manageableResourceDefinition="CustomerVariables"
   topic="VariableEventCapability" alias="CustomerVariableEvent"/>
  <eventCompositionStatement>
   <![CDATA[
  SELECT abs(d.timestamp - a.timestamp) AS orderFulfillmentTime,
    c.value.purchaseOrder.poId AS poId, c.topology AS topology
  FROM PATTERN
   [EVERY
       a=ManagementEvent(name="ResellerActivityEvent",
       activityXPath='/process/sequence[1]/receive[1]',
       state="Ready")
    -> b=ManagementEvent(name="ResellerVariableEvent"
       elementXPath='/process/variables[1]/variable[1]',
       processId=a.processId)
    -> c=ManagementEvent(name="CustomerVariableEvent"
       elementXPath='/process/variables[1]/variable[2]',
       value.purchaseOrder.poId=b.value.po.id)
    -> d=ManagementEvent(name="CustomerActivityEvent",
       activityXPath='/process/sequence[1]/while[1]',
       state="Complete",
       processId=c.processId)]
   ]]>
  </eventCompositionStatement>
 </compositeEvent>
</capability>
```

Example. An exemplary definition of a composite event is shown in Listing 3.3 (namespaces left out for readability reasons). First, the four `consumedEvent` elements are defined by referencing the corresponding manageable resource endpoints and corresponding topics. Note that these definitions result in four *event streams*, whereby each event is part of one particular process instance. When defining the composite event, one has to correlate the corresponding events from the different event streams.

Then, one defines an `eventCompositionStatement` which calculates the order fulfillment time as the time difference of the corresponding event timestamps and stores the value in the event property `orderFulfillmentTime`. In addition, the event contains the resource identification properties of the corresponding choreography interaction, i.e., the `topology` QName and the `poId`. Those three event properties are mapped to XML elements in the resulting XML event. The composite event is defined by correlating four events. The first two events are correlated based on the `processId` of the reseller process instance. The second and third event are correlated based on the `poId`, and finally the third and fourth event are correlated based on the `processId` of the customer process instance. Thus, the first and fourth event are also correlated and can now be used for calculating the property value by subtracting their `timestamps`. The resulting event is published to the corresponding `targetTopic`. One should note that writing such event composition statements is rather cumbersome and requires knowledge of the concrete event processing language (ESPER EPL in our case). Also referencing the BPEL elements via XPath (e.g., setting the correct `activityXPath`) is error-prone when done by hand. In practice, a GUI-based tool could be developed which supports the user in creating such statements.

Listing 3.4 shows how a corresponding custom property is defined. It is defined as part of a `CustomPropertyCapability`. The `consumedEvent` definition references the previously defined composite event. The attribute `eventProperty` specifies in an XPath expression how the property value is to be extracted from the event.

Listing 3.4: Custom Property Example

```
<capability uri="http://.../CustomPropertyCapability">
 <property name="poChoreo:orderFulfillmentTime" >
  <consumedEvent
   manageableResourceDefinition="mM:CustomerResellerInteraction"
   topic="poChoreo:OrderFulfillmentTime" />
  <eventProperty>/orderFulfillmentTime</eventProperty>
 </property>
</capability>
```

Custom Resource Types. Custom properties can be defined for any resource type. For example, activity instance duration is defined for the resource type Activity Instance, while process instance duration is defined for the resource type Process Instance. In addition to the resource types derived from the choreography model, one can define properties which cannot be assigned to any of those resource types. For example, this is the case when they are related to a group of several process instances; the average process instance duration cannot be assigned to a particular process instance.

Therefore, a new resource type *Custom* can be used to construct arbitrary resource types implicitly by (i) defining how the resources are created, and (ii) specifying the resource identifying properties. Therefore, in the definition of a manageable resource endpoint the element `customResourceDefinition` is provided. It consists of two elements. The element `resourceCreationEvent` specifies the event stream which triggers the resource creation. It is defined by referencing an existing manageable resource definition and a topic. The second element `resourceIdentificationProperties` specifies the identification properties for the resources of the resource type. These properties have to be part of the resource creation event.

Example. Listing 3.5 shows the definition of manageable resource definition with a resource type `ChoreographyInteraction`. A `Choreography Interaction` is a custom resource type, as for a choreography interaction the resource identification properties are domain-specific. In this case, the `OrderFulfillmentTime` event is defined to be the resource creation event

(cf. Listing 3.3). Furthermore, two resource identification properties are defined, `topology` specifying the corresponding BPEL4Chor topology, and `poId` specifying the instance of the choreography interaction.

Listing 3.5: Custom Resource Type Example

```
<manageableResourceDefinition name="CustomerResellerInteraction"
    resourceType="http://www.iaas.uni-stuttgart.de/m4c/
    resourceTypes/ChoreographyInteraction">
 <customResourceDefinition>
  <resourceCreationEvent
    manageableResourceDefinition="mM:CustomerResellerInteraction"
    topic="poChoreo:OrderFulfillmentTime" />
  <resourceIdentificationProperties>
   <property name="m4c:topology" eventProperty="/topology" />
   <property name="reseller:poId" eventProperty="/poId" />
  </resourceIdentificationProperties>
 </customResourceDefinition>
 ...
</manageableResourceDefinition>
```

At runtime, on receipt of such an event, the resource is created and can be accessed using its manageability interface, e.g., by accessing the value of the `OrderFulfillmentTime` property over corresponding WSRF operations.

3.4 Summary and Conclusions

This chapter has presented an approach to monitoring of business processes based on service choreography models. The approach is based on a monitoring metamodel which enables specifying a set of manageable resource definitions for runtime monitoring of choreographies. Therefore, a set of resource types and a set of corresponding capabilities have been defined which expose functionality for accessing resource properties. In particular, it has been shown how custom properties can be calculated across processes in a choreography based on event processing. The approach is concretely based on WSDM and uses the BPEL event model 2.0 and BPEL4Chor models as basis.

The presented monitoring approach serves as a basis for the following analysis phase presented in the next chapter, which uses the monitored custom properties for assessing the process performance in terms of KPIs and analyzing the influential factors of KPIs.

CHAPTER 4

ANALYZING THE INFLUENTIAL FACTORS OF BUSINESS PROCESS PERFORMANCE

The monitoring framework presented in the previous chapter enables monitoring of process properties in service choreographies. For evaluating the process performance, the next step is to define KPIs based on those properties which enable evaluating the process performance towards business goals. If monitoring shows unsatisfactory KPI values, then the goal is to understand the dependencies of these KPIs on a set of influential factors. That knowledge can then be used to adapt the process in order to improve the performance in future.

This chapter presents an analysis framework which enables analyzing the process performance in terms of KPIs and influential factors. More concretely, classification learning based on decision trees is used to analyze the influential factors of KPIs of monitored process instances. The analysis result is a KPI dependency tree, which explains how a KPI depends on a set of influential

factors.

The KPI dependency tree can also be used for predicting KPI values for future instances. Thus, the analysis framework is not only used for analyzing and explaining the KPI performance of historical instances but can also be used as basis for prediction and adaptation as presented in the following chapter.

The chapter is structured as follows. Section 4.1 explains the motivation of the approach in more detail. Section 4.2 gives an overview of the approach by explaining the analysis process and giving needed background on classification learning. Section 4.3 shows how KPIs and potential influential factors are modeled. Section 4.4 then explains, how based on the analysis model decision tree learning can be used to analyze the influential factors of KPIs. Finally, Section 4.5 summarizes the contributions.

4.1 Motivation and Objectives

The previous chapter has described an approach which enables monitoring of properties of business processes in service choreographies. As a next step, the goal is to use monitoring as a basis for evaluating the performance of business processes. When measuring the performance of business processes, one first defines business goals and based on those goals specifies a set of KPIs which measure whether those goals are reached in a certain time period. A KPI is based on an arbitrary monitored property (the KPI property) and in addition specifies a *categorization function* which enables the interpretation of the property values in correspondence to the business goals. The categorization function maps value ranges of the KPI property to a set of categories (a.k.a. KPI classes). For instance in the purchase order processing scenario typical KPI properties are order fulfillment time (process duration from order receipt until shipment arrives at the customer) and order delivery in full and in time [Sup05]. For such a KPI property definition, one could define a categorization function as follows: order fulfillment time < 3 days is "good", < 5 days is "medium", and otherwise "bad". One should note that when referring to "performance" in the context of KPIs, one does not only refer to properties which reflect the time dimension (e.g., process duration), but also other dimensions such as quality,

cost, customer satisfaction, and flexibility can be used. While order fulfillment time is used in most of the examples in this thesis, the approach does not make any assumptions on the semantics of the underlying property and thus supports KPI definition based on arbitrary properties.

After defining the KPIs, they have to be measured based on executed process instances. If after a while the monitoring shows an unsatisfying result in terms of reached KPI categories, then the goal is to find out *why* the performance goals have not been reached. In our scenario, understanding the reasons why certain orders are delivered on time and others are not, is often not trivial, as the KPI depends on the combination of several influential factors such as ordered product types and amounts (input data of the process), duration and availability of the internal services, duration and reliability of external services, and many more. For example, standard shipment duration could take from one to five days or in untypical cases even longer, supplier delivery time might depend on certain product types and amounts and their availability in stock. These deviations in service behavior lead to different outcomes of process instances considering KPI categories.

If there are many thousand process instances, it becomes difficult to understand which of the potential influential factors lead to different KPI values for different process instances. Data warehousing and OLAP (cf. Section 2.3) could then be used to analyze the potential problems and answer business questions. The typical approach would be to create a data mart by defining the KPI as a fact and select a set of dimensions which reflect the potential influential factors. The user could then manually pose analysis questions as queries in order to find out the influential factors. However, in that case a user has to *manually* perform the analysis and search for patterns in the data, i.e., suspect the influential factors and then perform queries, which can be very time-consuming.

In order to increase the automation in the analysis of influential factors, data mining techniques can be used. The analysis problem is mapped to a classification problem and machine learning, in particular decision trees, are used to learn and explain the factors which lead to different KPI categories. The so created decision trees are called KPI dependency trees. KPI dependency trees (i) visualize and explain the influential factors and (ii) can be used for

prediction of future KPI values.

4.2 Solution Overview and Method

This section first gives background information on classification learning, which is the basis of the approach and sketches the approach by describing how classification learning maps to the KPI analysis problem and then describes the method.

4.2.1 Classification Learning and KPI Dependency Analysis

In classification learning, the input is a dataset consisting of a set of *instances* (a.k.a., observations, examples). Each instance is described in terms of a set of *explanatory attributes* and one categorical *target attribute*. An explanatory attribute may be categorical or numerical. Unlike in regression, the target attribute is always categorical and takes a finite number of values (a.k.a., *classes*).

In our context, the KPI property is the categorical target attribute. The possible attribute values consist of a set of *KPI classes* (e.g., green, yellow, red). The explanatory attributes consist of a set of lower-level (monitored) process properties which potentially influence the KPI class. The properties are therefore representing *potential influential factors*. The analyzed dataset consists of a set of monitored process instances. For each instance of the dataset the KPI class and the values of the explanatory attributes can be determined from monitored data.

Based on such a dataset as input, the goal of classification learning is to create a *classification model* which identifies recurring relationships among the explanatory variables which describe the instances belonging to the same class. In our case, the classification model thus explains how the KPI classes depend on the lower-level process properties.

Classification Learning Phases. In classification learning, a subset of the instances in the dataset, the training set, is used for *training* a classification model, i.e., for deriving the functional relationship between the target variable

and the explanatory variables. The remaining instances of the available dataset, the test set, are used later to evaluate the accuracy of the created classification model.

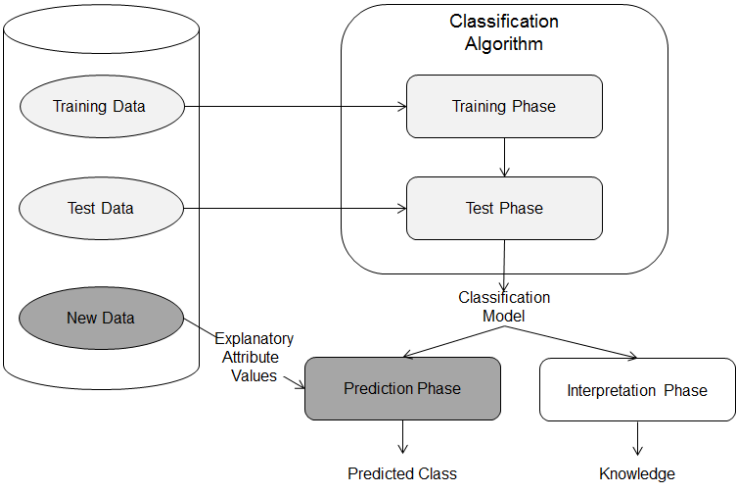

Figure 4.1: Classification Learning Phases

The development of a classification model consists therefore of two main phases, as shown in Figure 4.1:

Training phase During the training phase, the classification algorithm is applied to the instances belonging to the training set, which is a subset of the overall dataset. The classification model is created based on the training set.

Test phase In the test phase, the classification model generated during the training phase is used to classify the instances of the test set, for which the target class value is already *known*. To assess the accuracy of the classification model, the explanatory attribute values of each instance in the test set are used as input to the classification model and the predicted class (the output of of the model) is then compared with the known class of the test set instance. The accuracy metric is then given

by the percentage of correctly classified instances from the test set. It allows assessing the quality of the model considering its application for interpretation and prediction. In our approach, a set of monitored instances is provided as input to an existing decision tree algorithm, which then splits these instances into training data and test data and performs the training phase and test phase automatically.

Interpretation and Prediction. The so created classification model can then be used for *interpretation* and *prediction*. Interpretation means that the classification model should provide experts in the analyzed application domain with some new non-trivial knowledge about the analyzed data, which the expert would not have obtained by simply looking at the dataset. Classification models which are particularly well-suited for interpretation typically express and visualize the identified regular patterns in the data in a way that they can be easily understood by experts in the application domain. In our case, the goal of the classification model is to *explain* to the expert how a KPI depends on a set of lower-level process properties. In that way, the expert can try to adapt the process in order to improve the performance.

While interpretation explains the classes of past instances, the purpose of prediction is to anticipate the class of instances in the future. Typically, thereby the classification model is given the values of explanatory variables as input and then provides the predicted class of the instance as output. In our approach, prediction will be used in the next chapter for predicting the KPI class of a running process instance.

Decision Trees. There are different types of classification models and corresponding algorithms available. Some of the better known techniques are decision trees, classification rules, and support vector machines [WF05]. These models have different characteristics when it comes to suitability for interpretation or prediction, support for handling numerical or categorical data, prediction precision, and performance. For the KPI analysis problem, in the approach of this thesis *decision trees* have been chosen as the classification model.

Decision trees have several advantages when used for our purposes. They are easy to depict graphically and are simple to understand and interpret. Non-experts are able to understand decision tree models after a brief explanation. They are thus suitable for both interpretation and prediction, in contrast, for example, to black box models such as artificial neural networks, which are used for prediction only.

Decision trees are able to handle both numerical and categorical data, which is needed for representing different types of influential factors. Other techniques are usually specialized in analyzing datasets that have only one type of variable. For example, relation rules can be used only with nominal variables while neural networks can be used only with numerical variables.

Another big advantage of decision tree algorithms, especially so in our context, is their non-parametric nature. They need only a very limited set of parameters (in the simplest case none) as input, and can therefore be expected to provide useful results from the first run, without the need for extensive experiments with different parameter sets. This is why the approach is suitable for business analysts, who are generally no experts in data mining. Of course, expert users can still customize the algorithms if they want to, which may lead to better results in some cases. The usage of different parameters is discussed in the evaluation section (cf. Section 6.2). Apart from parameters, decision tree learning also requires little data preparation. Other techniques often require data normalization, the creation of dummy variables, and the removal of blank values.

Decision trees are a standard technique for supervised learning (i.e., concepts are learned based on historical data where the classification of the instances is known). Decision tree learning uses a "divide and conquer" approach to learning of concepts. Thereby, a tree of decision nodes is iteratively constructed, each decision node consisting of a test on an explanatory attribute, such as whether a given numerical attribute is smaller or greater than a given threshold. Leaf nodes in the tree represent a classification to a category, i.e., contain a value of the target attribute.

Figure 4.2 shows a decision tree example. It contains five decision nodes specifying tests on four explanatory attributes (A1-A4). The leaf nodes contain

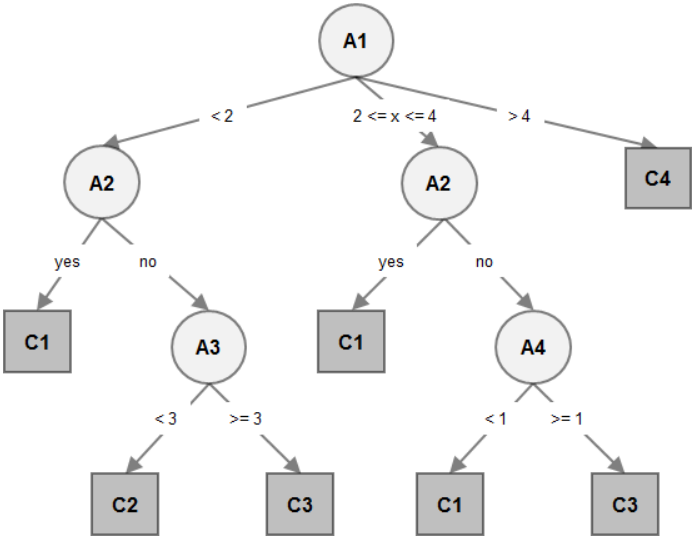

Figure 4.2: Decision Tree Example

the target attribute classes (C1-C4). The decision tree shows, for example, that if A1 < 2 and A2 $=$ yes, then the target attribute class is C1. That rule has been deduced automatically by the decision tree algorithm based on historical data which was used for creating the training set. Each path from the root to the leaf of the tree can be seen as a rule which shows how a target attribute class *depends* on the combination of explanatory attributes with certain value ranges. In addition to the interpretation aspect, the tree can be used for predicting the outcome of future instances. This is done by starting at the root of the tree and following the path according to the values of explanatory attributes of the future instance. The path ends at a leaf, which is the predicted target attribute class.

There exist many well-researched algorithms to construct decision trees from data, such as the C4.5 [Joh93] or the alternate decision tree, ADTree [FM99]. Two different existing algorithms have been used to evaluate the approach. The experiment results are discussed in Section 6.2.

4.2.2 Overview of the KPI Dependency Analysis Process

In the following, an overview of the KPI dependency analysis process is given. It uses the development process for monitoring as basis (cf. Section 3.2.1) and extends it by adding the steps needed for analysis. The steps of the process are depicted in Figure 4.3.

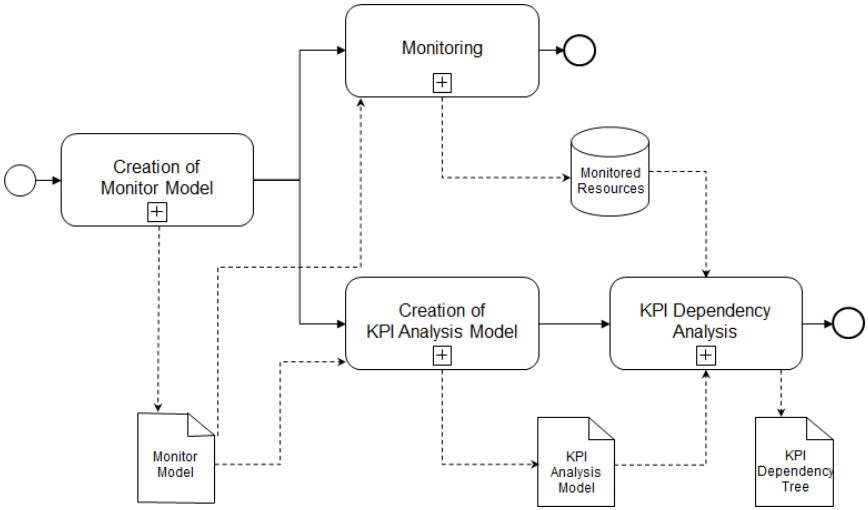

Figure 4.3: Overview of the Analysis Process

Creation of a Monitor Model. The first step of the process is to create a monitor model. The prerequisite to this step is the creation of the corresponding choreography model (or orchestration model) for which the performance is to be analyzed (cf. Section 3.2.1). The monitor model defines all properties which should be monitored and which are needed for later analysis purposes. Those properties include (i) the KPI properties and (ii) all the potential influential factors. Potential influential factors are those monitored properties which are *suspected* to influence the KPI values. Typically, one would start with the creation of KPI property definitions and in a second step model the potential influential factor properties. As the definition of a potentially big number of influential factor properties can be rather tedious and time-consuming, a set

of templates for the most typical ones have been predefined, which make the definition easier, as discussed in Section 4.3.4.

Deployment and Monitoring. The monitor model is deployed as described in Section 3.2.1. At process runtime, the processes are continuously monitored evaluating KPI properties and influential factor properties as defined in the monitor model. The monitored properties (as part of corresponding monitored resources) are stored in a monitor database and are from now on available for KPI dependency analysis.

Creation of a KPI Analysis Model. The basis for KPI dependency analysis is the creation of a KPI analysis model. The analysis model is only used as input to the analysis phase and thus does not have to be created necessarily before monitoring starts. The analysis model contains the following types of information:

- *Key Performance Indicators:* A KPI is defined by selecting a resource property from the monitor model (the KPI property) and defining a categorization function which maps KPI property values to a set of KPI classes based on business goals. The analysis model can define several KPIs.
- *Influential factors:* A set of potential influential factors is defined by referencing a subset of the available resource properties in the monitor model.
- *Analysis tasks:* An analysis task is defined by selecting one concrete KPI and a subset of influential factors (default value: all available influential factors) which should be used as explanatory attributes for this particular KPI. In addition, the dataset is specified by defining how many instances should be analyzed (e.g., last 1000 instances). Optionally, the user can select the algorithm used for analysis and adjust the algorithm parameters. If not selected by the user, default values are used.

KPI Dependency Analysis. The KPI dependency analysis can be triggered manually by the user (on demand) or automatically by the system (e.g., after

a certain number of instances has been executed). The prerequisite of the KPI dependency analysis is that the instances have been monitored and are available in the monitor database. The dependency analysis is performed for each analysis task specified in the analysis model and consists of the following steps, which are performed automatically:

- *Data preparation:* The goal of this step is to prepare the dataset used as input to decision tree learning. The input to this step is (i) an analysis task as defined in the analysis model and (ii) the values of monitored properties stored in the monitor database. The first step of the data preparation is determining the historical instances which should be used. This selection has been specified as part of the analysis task. For each instance the corresponding monitored property values of the KPI property and the influential factors are then retrieved from the Monitor DB. The KPI class is evaluated and assigned to each instance.
- *Learning of the KPI dependency tree:* The created dataset is provided as input to decision tree learning. A decision tree is learned using a standard decision tree algorithm like C4.5 or ADTree.
- *Displaying and storing the KPI dependency tree:* As a result of the dependency analysis, the tree is displayed to the user (if the learning was triggered by the user) or stored for later use.

At the end of the KPI dependency tree learning phase, the user evaluates the result. In some cases, the generated tree might not be satisfactory to the user. The typical cases are discussed in the evaluation section (cf. Section 6.2). In that case, the user can adjust analysis task settings and repeat the analysis.

4.3 Modeling for KPI Dependency Analysis

This section defines the metamodel which is used for the creation of KPI dependency analysis models (analysis models, for short). An analysis model references elements from one or more monitor models as specified in Section 3.3.

An analysis model defines KPIs, influential factors and analysis tasks. An analysis task groups a KPI and a set of potential influential factors. Each analysis task later results in one KPI dependency tree after decision tree learning.

4.3.1 Overview

The main elements of the KPI dependency analysis metamodel are shown in Figure 4.4. The analysis model defines a non-empty set of KPIs, a non-empty set of influential factors, and a non-empty set of analysis tasks. A *KPI* defines the KPI property by selecting a resource property of a manageable resource definition specified in a monitor model (cf. Section 3.3). A KPI definition specifies a set of *KPI classes* which are the possible categorical values of the KPI. The categorization function is specified in terms of *predicates* which specify how the KPI classes are calculated based on the underlying KPI property values. An *influential factor* is specified in the same way as the KPI property by selecting a resource property from a monitor model. An *analysis task* combines a KPI and a list of influential factors. It further specifies the number of instances which should be used as basis for analysis and optionally the algorithm settings.

XML Serialization of the KPI Analysis Metamodel. An analysis metamodel is serialized as follows:

Listing 4.1: KPI Analysis Metamodel Pseudo XML Schema

```
<kpiAnalysisModel targetNamespace="URI" name="NCName"
xmlns="http://www.iaas.uni-stuttgart.de/m4c/schemas/analysisModel">
 <kpi name="NCName">
  <propertySelector manageableResourceDefinition="QName"
                    property="QName"/>
  <kpiClass name="String">
   <predicate type="LOWER|GREATER|..." > String </predicate> +
  </kpiClass> +
 </kpi> +
 <influentialFactor name="NCName">
  <propertySelector manageableResourceDefinition="QName"
                    property="QName"/>
```

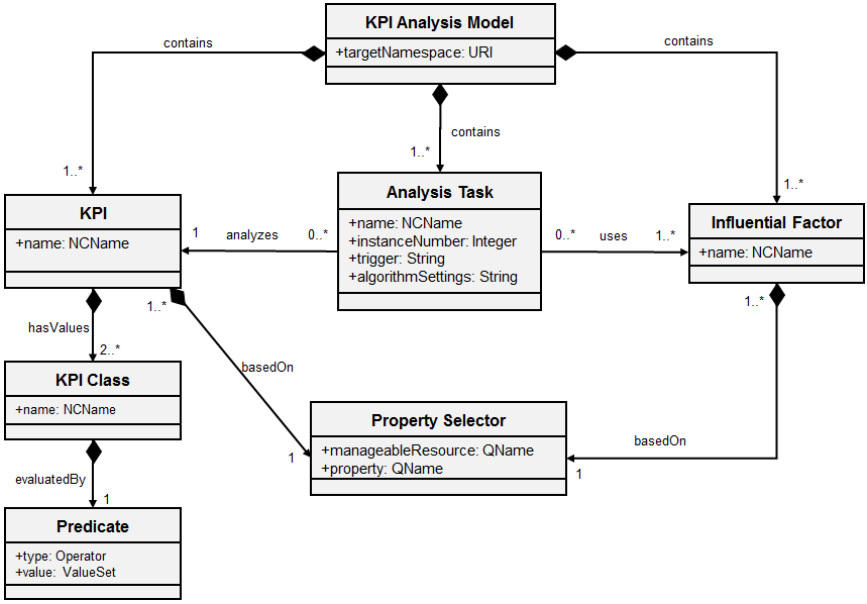

Figure 4.4: KPI Dependency Analysis Metamodel

```
 </influentialFactor> +
14 <analysisTask name="NCName" instanceNumber="Integer"
              trigger="String" algorithmSettings="String">
16  <kpi name="String" />
  <influentialFactor name="String" /> +
18 </analysisTask> +
</kpiAnalysisModel>
```

4.3.2 Key Performance Indicators

Key Performance Indicators (KPIs) are used to assess the process performance in respect to business goals. A KPI is defined based on a resource property (KPI property) and maps value ranges of that property to a set of categorical values (KPI classes). KPI classes thus allow evaluating how good the KPI property value conforms to business goals.

The KPI property is defined based on a resource property of a specific manageable resource definition in the monitor model. The resource type of the manageable resource endpoint is typically a process instance but in general any resource type could be used as basis (e.g., a variable instance, an activity instance, or a custom resource type).

The KPI value is defined in terms of KPI classes, which are categorical values. Thereby, at least two KPI classes have to be defined, e.g., "KPI target fulfilled" and "KPI target violated". There can, however, be also more than two classes, such as the traffic light function (green, yellow, red). The categorization function defines how the values of the underlying KPI property map to KPI classes. The function is specified implicitly via predicates over KPI property values. Each KPI class is assigned a predicate which compares the KPI property value with another value or a set of values using a set of operators. The following set of operators have been predefined: EQUAL, NOT_EQUAL, LOWER, GREATER, LOWER_EQUAL, GREATER_EQUAL, IN, BETWEEN. The KPI value is the KPI class whose predicate evaluates to true for the KPI property value.

Listing 4.2: KPI Example

```
<kpi name="OrderFulfillmentTime">
  <propertySelector
    manageableResourceDefinition="mM:CustomerResellerInteraction"
    property="poChoreo:orderFulfillmentTime" />
  <kpiClass name="green">
    <predicate type="LOWER">4</predicate>
  </kpiClass>
  <kpiClass name="yellow">
    <predicate type="BETWEEN">4;7</predicate>
  </kpiClass>
  <kpiClass name="red">
    <predicate type="GREATER">7</predicate>
  </kpiClass>
</kpi>
```

Example. In the following example, the KPI `OrderFulfillmentTime` is de-

fined for the KPI property `orderFulfillmentTime`, which has been defined for the manageable resource definition `CustomerResellerInteraction` in the monitor model. The KPI classes green, yellow, and red are defined. The categorization function maps the values of the underlying KPI property to the KPI classes: < 4 days → green, >= 4 days and <= 7 days → yellow, otherwise red. The function is specified via predicates assigned to each KPI class. At analysis time, for each monitored resource (as specified in the analysis task), the KPI property value is retrieved, the predicates are evaluated and the one which returns true provides the KPI class.

4.3.3 Influential Factors

In addition to the KPI, which is the target attribute in classification learning, the potential influential factors, which represent the explanatory attributes have to be defined. The goal is, in general, to define a big set of properties, and then let the decision tree algorithm find out which of those are the influential ones. An expert user later still has the possibility to restrict the set as part of the analysis task definition.

An influential factor is defined based on a resource property of a specific manageable resource definition in the monitor model. The constraint is thereby that is has to be a manageable resource endpoint definition of one of the KPIs defined in the analysis model. This is because at analysis time, both the KPI class and the values of the corresponding influential factors are assigned to the resources (instances) of the corresponding manageable resource definition.

The definition of potential influential factors can be done semi-automatically, i.e., a subset can be generated automatically based on the process model as described in the following section 4.3.4. The other subset consists typically of domain-specific properties and has to be defined manually. As a result, the influential factor list in the analysis model contains all potential influential factors.

Example. The following example shows an influential factor definition based on the property `supplierDeliveryTime`.

Listing 4.3: Influential Factor Example

```
<influentialFactor name="SupplierDeliveryTime">
 <propertySelector
     manageableResourceDefinition="mM:CustomerResellerInteraction"
     property="poChoreo:supplierDeliveryTime" />
</influentialFactor>
```

4.3.4 Generating Influential Factor Properties

The core of the KPI dependency analysis approach is the availability of a meaningful and complete set of potential influential factors. Thereby, in the first step it is important to have a rather big set of properties and let the decision tree algorithm find out the relevant ones. In further analysis steps, a user can still restrict the set and perform analysis tasks on specific analysis factors. The evaluation section describes in which cases that might make sense (cf. Section 6.2).

The manual definition of influential factor properties is rather cumbersome and time-consuming. In our case, each such property is typically a custom property which has to be defined based on an event processing statement (cf. Section 3.3.5). Many of those properties can be defined in an automated manner based on the process model definition. Some domain-specific properties can then still be added by the user manually.

The automated approach analyzes the process models, in our case BPEL process models, and defines meaningful resource properties for the different elements of the process. The following rules to generate properties for a KPI defined for the process instance resource type are supported:

- For the process instances of the given BPEL process model, one generates (i) a property representing the end state of the process instance (exited, completed, or faulted) and (ii) a property representing the execution time of the process instance.
- For every BPEL `invoke` activity which is not part of a loop activity (i.e., which is executed 0 or 1 times in every process instance), one generates a

property representing the execution time of the activity instance (i.e., the time between the first and last state change event related to that activity instance).

- For every BPEL `invoke` activity which is part of one or more (nested) loop activities (i.e., which is potentially executed several times in a process instance), one generates a property representing the average execution time of the activity in the process instance. Additionally, one generates properties representing the number of times the activity has been executed in this process instance, and the minimum and maximum execution time of the activity instances in the process instance.

- For every loop activity (sequential `forEach` activity, the `repeatUntil` activity, or the `while` activity), one generates a property representing the number of iterations in the particular process instance.

- For every branching activity, one generates a property representing the branch that has been executed.

- For every link, one generates a property stating whether the link was activated or not.

- For every asynchronous interaction, one generates a property representing the callback time of every asynchronous activity in the BPEL process, i.e., the execution time between an invoke activity and the corresponding receive activity.

Properties for short-running activities such as `assign` activities are not generated, as the focus is on longer running processes and their duration can mostly be neglected in comparison to service invocations.

Which of these rules should be used, can be configured. For each of the property types a template has been created which contains the EPL statement with placeholders (e.g., `${activityXPath}`) which are then replaced with concrete values during generation. The properties are generated as part of the monitor model, and are thus monitored. It should be noted that the generated

properties provide only a partial list of meaningful properties. The user (a domain expert) still has to define additional domain-specific properties. These are in particular related to variable values. The properties are specified by selecting specific parts of those variables (e.g., number of ordered items, types of ordered products in case of a purchase order variable).

4.3.5 Analysis Tasks

Analysis tasks define the KPI dependency analysis problems to solve. An analysis task specifies a classification problem by using a KPI as the target attribute, a set of potential influential factors as explanatory attributes, and defining the instances (the dataset) for which the KPI should be evaluated. Each analysis task results later in a KPI dependency tree.

The prerequisite for the definition of an analysis task is that a set of KPIs and influential factors have been defined as described in the previous sections. When defining an analysis task, the user first chooses the KPI to analyze. In the second step, he then selects a set of potential influential factors he wants to use as explanatory attributes. The restriction thereby is that the selected influential factors must be based on the same manageable resource definition as the selected KPI. In the default case, all influential factors which have been defined in the analysis model will be chosen. If the user wants to analyze the dependencies on specific properties, he will choose a specific subset of the influential factors. For example, the user might want to analyze the KPI dependency on input data to the process. In that case, he will restrict the subset to properties representing attributes of the input variable. In another case, he might be interested in analyzing only the dependencies on external service invocations. For the same KPI, several analysis tasks with different sets of influential factors can be specified, depending on the analysis goal.

After defining the KPI and the influential factors, one has to specify which instances (resources) should be used for analysis. In our approach, one simply specifies the number of instances whereby the instances are selected according to their completion date (youngest instances first). In addition, one can specify a trigger when the analysis process should be started. This is needed if learning

is to be done automatically in the background as in our adaptation approach, presented in the next chapter. The trigger is specified as a cron expression as supported by the Quartz Scheduler [Ter]. Finally, one can adjust the decision tree algorithm and its parameters which is needed only in those cases, when the default settings do not lead to satisfactory results (see discussion in Section 6.2.1).

Example. In the following example, an analysis task for the previously defined KPI `OrderFulfillmentTime` is defined. It uses (only) four potential influential factors and defines that the last 10000 instances should be analyzed. The learning is triggered on every Sunday at 1:00 am (one uses a cron trigger expression used by the Quartz Scheduler: seconds, minutes, hours, day of month, month, day of week). The J48 decision tree algorithm [HFH+09] is to be used.

Listing 4.4: Analysis Task Example

```
<analysisTask name="OrderFulfillmentTime" instanceNumber="10000"
              trigger="0 0 1 ? * SUN" algorithmSettings="-J48">
 <kpi name="OrderFulfillmentTime" />
 <influentialFactor name="SupplierDeliveryTime" />
 <influentialFactor name="ShipmentDeliveryTime" />
 <influentialFactor name="OrderInStock" />
 <influentialFactor name="ItemQuantity" />
</analysisTask>
```

4.4 KPI Dependency Analysis

This section describes how the KPI dependency trees are created based on the analysis model definitions and how they are to be interpreted.

4.4.1 Learning of KPI Dependency Trees

KPI dependency analysis is performed for each analysis task defined in the analysis model. It can be triggered automatically (as specified in the analysis

task via the trigger definition) or by the user on demand. The next steps are then performed automatically. A dataset is constructed based on the settings in the analysis model and the monitoring results and fed into a decision tree algorithm. Existing well-known decision tree algorithms are used. As a result, a KPI dependency tree is created, which can be used for interpretation and prediction.

For each analysis task the dataset is created as follows. First, the set of instances to be analyzed are determined. Therefore, for the specified manageable resource definition a set of resources (instances) is obtained from the monitor database. Then for each instance, the values of configured influential factor properties and the KPI property are also obtained from the monitor database. The KPI class is determined by evaluating the specified predicate using the categorization function and the KPI class is assigned to the instance. As a result, one obtains the data set as a table consisting of a set of instances as rows and influential factor properties and KPI class as columns (as shown in Figure 4.6). This dataset is used as input to the algorithm.

In our approach, the popular J48 algorithm has been used to generate KPI dependency trees [HFH+09]. The metamodel of the generated tree is shown in

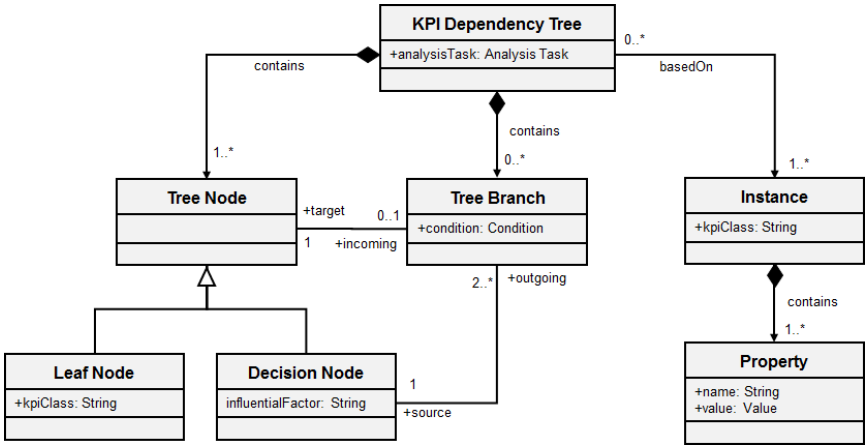

Figure 4.5: KPI Dependency Tree Metamodel

Figure 4.5. A *KPI dependency tree* consists of a (possibly empty) set of non-leaf *decision nodes* representing influential factors and a non-empty set of *leaf nodes* representing KPI classes. Thereby, a particular influential factor or KPI class can be present in the tree zero to several times. An outgoing *branch* of a tree node defines a *condition* on the influential factor property of that node. The property values on outgoing branches of a node are disjoint. The tree has exactly one root node (with no incoming branches). Each leaf node contains as information the KPI class and in addition the *number of instances* which satisfy the path of this leaf to the root. Thus, by following the path from the root to a leaf node, one learns which property values lead to a particular KPI class, and for how many instances that was the case.

Example. The KPI dependency analysis steps are depicted in Figure 4.6. In the KPI analysis model an analysis task has been defined for the KPI order fulfillment time with three KPI classes. Furthermore, a set of potential influential factors has been defined. Those properties have been monitored for a set of instances and stored in the Monitor DB. Based on these two inputs a dataset is created, shown in the Figure 4.6 as a table. Each row of the table contains the property values (representing potential influential factors) of one particular instance, whereby the first column specifies the KPI class, i.e., the result of the application of the categorization function on the KPI property value of that instance.

Based on this dataset, a decision tree is automatically learned. It shows on which combinations of influential factors and their value ranges the KPI classes are reached. For example, one can see that if the ordered products were not in stock and supplier delivery time was higher than 7,5 days for an instance, that instance has lead to a red KPI class. That was the case for 55 instances in the dataset. If, however, the supplier delivery time was lower or equal 3 days and the shipment delivery time was below 2,2 days then a green KPI class was reached. The tree shows thus how the KPI class depends on the values of the influential factors. This information can be used by an analyst to think about possible adaptations of the process.

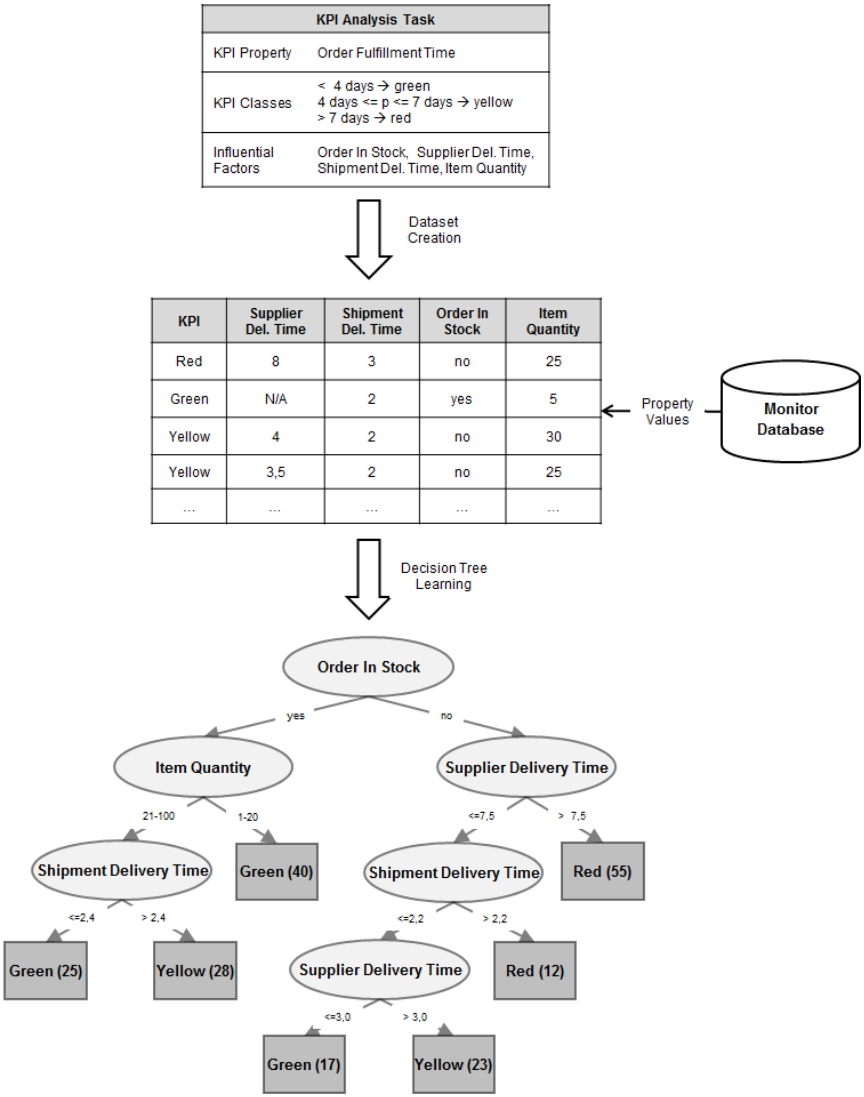

Figure 4.6: KPI Dependency Tree Learning Example

4.5 Summary and Conclusions

This chapter has presented an approach to the analysis of process performance in terms of KPIs and their influential factors. A KPI analysis metamodel has been defined which enables the definition of KPIs and a set of potential influential factors based on properties of manageable resources provided by the monitoring framework. Based on the KPI analysis model, established decision tree learning algorithms are used to automatically generate a KPI dependency tree, which explains how the influential factors affect the KPI values. The prototypical implementation and an experimental evaluation of the approach are described in Chapter 6.

The generated KPI dependency trees are not only useful for explaining the KPI dependencies to the user, but can also serve as prediction models for future process executions. In the next chapter, KPI dependency trees are used for predicting the KPI classes of running process instances. The result of the prediction is then used as basis for runtime adaptation of process instances.

CHAPTER 5

RUNTIME ADAPTATION BASED ON KPI DEPENDENCY ANALYSIS

The previous chapter has described an approach for the analysis of KPIs of business processes based on KPI dependency trees. The focus was thereby on providing the user an explanation of KPI values based on monitored historical process instances. After understanding the influential factors of KPIs, the next step is to adapt the process accordingly in order to improve the performance in future.

In this chapter, the monitoring and analysis framework presented in the last two chapters is extended by enabling the *proactive adaptation* of process instances with the goal to improve the KPI performance. Therefore, running process instances are halted at predefined checkpoints where based on KPI dependency trees the KPI classes are predicted. Based on the prediction result, the adaptation requirements are identified and a set of adaptation strategies is derived and ranked. Finally, the process instance is proactively adapted by enacting the selected adaptation strategy with the goal to improve the process performance. The adaptation framework does not explicitly deal with the

adaptation of choreographies, but supports choreography adaptation through the adaptation of the underlying process instances running in a choreography. The adapted process instances, however, do not have to necessarily run as part of choreographies.

The chapter is organized as follows. Section 5.1 explains the motivation of the approach. Section 5.2 gives an overview of the approach by explaining the overall process. Section 5.3 explains the models created in the modeling phase. Section 5.4 explains how KPI dependency trees can be used to predict KPI values at process runtime and describes how based on prediction results adaptation strategies can be derived in order to adapt the running process instance. Finally, Section 5.5 summarizes the contributions of this chapter.

5.1 Motivation and Objectives

The KPI dependency analysis generates decision trees which explain how KPI classes depend on a set of influential factors. The tree paths which lead to unsatisfactory KPI classes are in particular interesting as they show which influential factors and combinations of those might need to be improved. There are several aspects to consider when thinking about improving the process based on this particular type of analysis.

Depending on the concrete process, some of the influential factors might be improved easily, while others might be more difficult or impossible to improve or change. For example, consider the influential factors shipment delivery time and supplier delivery time used in the scenario. For an organization, it might be perfectly possible to define or redefine the SLAs with the shipper and the supplier or use alternative services from the providers or choose completely different service providers. On the other hand, changing influential factors representing ordered product types or contextual conditions such as dependencies on weather might be more difficult or impossible to accomplish. So, there are influential factors which can be improved in our context and others which are well-suited for explaining the KPI value, but which are not improvable.

When improving the influential factors, one can either change and redeploy the process model or adapt only particular instances. The latter choice is in

particular interesting in our case, as the KPI dependency tree can be used to *predict* the KPI class of an instance and thus enable proactive adaptation. As already discussed in Section 4.2, decision trees can be used as prediction models. Thereby, explanatory attribute values are given as input and the tree provides the target value as output. The idea is thus to halt process instance execution at certain points, provide the monitored values collected until that point as influential factors to the tree, and obtain the predicted KPI class as output. In that case, one has more information on whether and how to adapt the running process instance. After potential adaptation, the instance execution is resumed.

The runtime adaptation approach assumes that the user has thought beforehand about possible adaptations and that there exist certain adaptation mechanisms from which the user can choose. One such mechanism, which is used in the approach, is service substitution. Therefore, it is assumed that there is a set of candidate services for a set of service types used in the process. E.g, there might be several alternative shippers which offer different SLAs via shipment options (e.g., standard, premium, overnight express). Each of those options can be modeled as a candidate service with different quality of service characteristics (such as shipment delivery time, shipment cost, reliability). The goal is then to select one of the alternatives at runtime based on KPI dependency analysis.

When adapting the process in order to improve certain influential factors, one has to take into account that those adaptation actions might negatively influence other influential factors and other KPIs. For example, choosing a faster shipment delivery will often imply higher cost. Thus, while improving the duration-based KPI, one would deteriorate the cost-based KPI. In order to deal with this issue, a constraints and preferences model is used which allows to specify hard constraints and preferences in terms of weights on KPIs and influential factors, and thus allow taking those into account during adaptation.

To summarize, the goal is to perform runtime adaptation of processes based on KPI dependency analysis, in order to proactively improve the KPI performance and take into account specified constraints and preferences.

5.2 Solution Overview and Method

In the following, an overview of the overall process is given. It uses the monitoring and KPI dependency analysis as presented in the previous two chapters as basis and adds models and algorithms needed for proactive runtime adaptation. The steps of the process are depicted in Figure 5.1.

Modeling for Adaptation. The first step of the process is to create appropriate models, which are then used at runtime for monitoring, analysis, and adaptation.

The following elements are defined as part of the *adaptation model*:

- *Adaptation Subjects:* An adaptation subject specifies an adaptable entity of the process (e.g., a shipment service used by the process) for which (i) there are several alternatives available (e.g., two or more alternative shipment services) and (ii) for which there is an adaptation mechanism available which allows runtime adaptation. An adaptation subject defines a set of properties (e.g., service duration, service cost) which are called the characterizing properties of the adaptation subject. These properties are used as basis for selecting an adaptation alternative for the adaptation subject, as different alternatives have different effects on the characterizing properties.

- *Adaptation Alternatives:* For each adaptation subject a set of alternatives is specified. When the process is deployed, for each adaptation subject one particular alternative is configured thus creating an initial configuration. During process execution, this runtime configuration can change for each process instance due to assigning other adaptation alternatives to the adaptation subjects. Each alternative specifies how it affects the characterizing properties of the adaptation subject. These effects are used for ranking and selection purposes.

- *Checkpoints:* A checkpoint defines where in the process, the process instance execution should be halted in order to perform KPI prediction and potential adaptation.

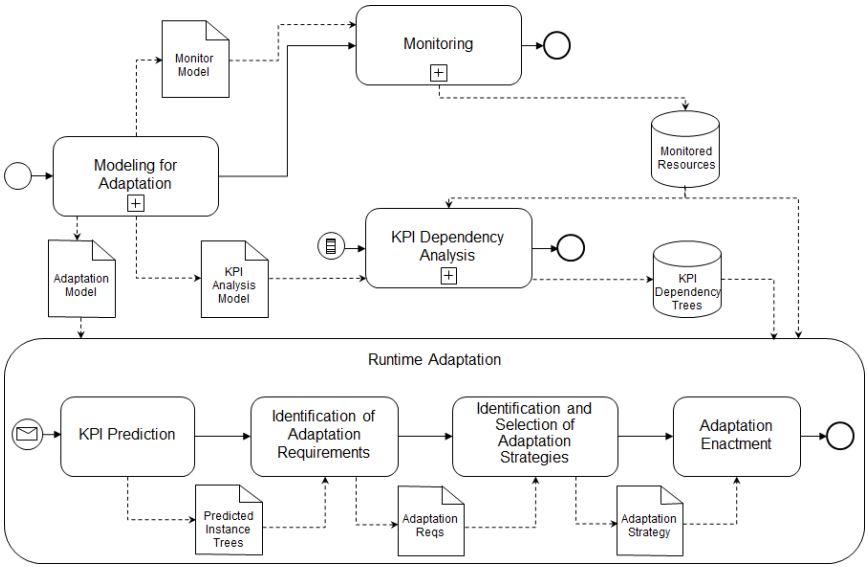

Figure 5.1: Overview of the Adaptation Process

- *Preferences and Constraints:* In addition to the KPI prediction result, preferences and constraints are used for ranking and selecting adaptation strategies at a checkpoint. Both preferences and constraints are specified on KPIs and characterizing properties of adaptation subjects. Constraints are used for removing adaptation alternatives from the set, while preferences are used for ranking purposes.

In addition to the adaptation model, a corresponding monitor model and an analysis model are created. The monitor model contains the manageable resource definitions for the adaptation subjects and corresponding properties needed for calculation of KPIs and influential factors. The analysis model defines the KPIs, influential factors and analysis tasks. Thereby, analysis tasks are specified per KPI and per checkpoint.

Monitoring. In the monitoring phase, all properties specified in the monitor model are monitored. That includes in particular the KPI properties and the

properties of the potential influential factors. As a result, property values for a set of executed process instances are obtained and stored in a monitor database.

Monitoring is also used for triggering of the checkpoints, i.e., for halting the execution of the process instance after the triggering event of a checkpoint has been received. Checkpoints are realized via blocking events.

KPI Dependency Analysis. KPI dependency analysis is performed automatically (without user support) and "offline" (in the background) for each KPI and per checkpoint. That means that learning does not affect process execution and adaptation. It creates KPI dependency trees which are then used for prediction. The trees are learned and later relearned after a certain configurable number of instances (e.g., every 1000 instances). The generated trees are stored in the database and can then be used for prediction.

KPI Prediction. A checkpoint specifies at which point in the process the KPI prediction should take place. When a running process instance reaches a checkpoint, it halts its execution. The property values which have been measured until the checkpoint for that instance are gathered and used as input to the (already created) KPI dependency tree. The prediction result is (in the special case) a predicted KPI class (e.g., green, yellow, or red) or (in the general case) an *instance tree*, i.e., a subtree of the original tree, which shows for a particular running process instance which properties should be improved to reach a specific KPI class and serves thus as basis for adaptation. There can be more than one KPI specified; in that case the KPI prediction is performed for each KPI separately.

Identification of Adaptation Requirements. Adaptation requirements are identified by extracting influential factor properties which should be improved from the instance tree. If several KPIs have been defined, then from each tree a set of requirements is extracted and those requirements are then combined.

Identification and Selection of Adaptation Strategies. Based on the adaptation requirements, a set of alternative adaptation strategies is identified by taking into account available alternatives for the available adaptation subjects. An adaptation strategy thus consists of a set of alternatives which should be

used in the process instance in order to reach a certain KPI class.

The list of alternative adaptation strategies is filtered and ranked based on the *constraints and preferences model*. Preferences are specified as weights on characteristics of alternatives (e.g., cost, duration, reliability), which enables ranking of strategies according to scores.

Adaptation Enactment. The first ranked adaptation strategy is enacted. This is done by enacting the adaptation alternatives of the adaptation strategy. It can happen that those alternatives are already specified in the current runtime configuration of the process instance. In that case, nothing has to be done. The process execution is finally unblocked and continues its execution.

While the adaptation is being performed at runtime for a certain number of instances, monitoring is continued. After a certain number of instances, the effectiveness of the adaptations can be evaluated by checking whether the KPI classes reached have been improved. This might lead to adjustment of the models, e.g., adjustment of KPI targets, (re)moving or adding of checkpoints, and adjustment of the constraints and preferences model.

5.3 Modeling for Adaptation

This section defines the metamodel which is used for the creation of adaptation models. An adaptation model defines (i) *what* can be adapted in terms of *adaptation subjects and alternatives*, (ii) *where* in the process the runtime prediction and potential adaptation should be triggered in terms of *checkpoints*, and (iii) *how* a particular adaptation strategy should be selected in terms of *constraints and preferences*.

5.3.1 Overview

The main elements of the adaptation metamodel are shown in Figure 5.2. An adaptation model specifies a set of *adaptation subjects* thus defining what can be adapted in the process. An adaptation subject is characterized by a set of *characteristics*. A characteristic is specified by referencing an influential factor from an analysis model. For each adaptation subject a set of *alternatives* is

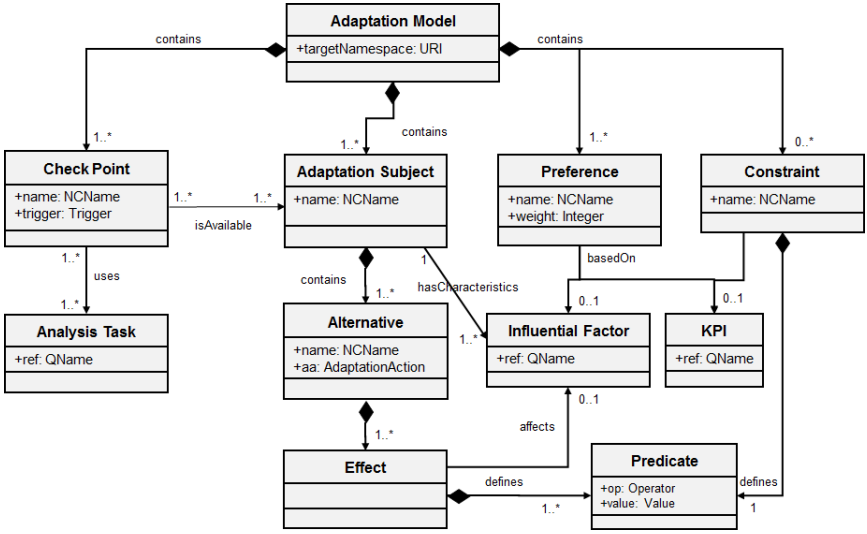

Figure 5.2: Adaptation Metamodel

defined. Each alternative specifies how it affects the characteristics of the adaptation subject in terms of *effects*. Each effect specifies how it affects a characteristic by defining a *predicate*. An alternative specifies an *adaptation action* which defines how the adaptation subject is to be adapted in order to use the corresponding alternative.

An adaptation model defines a set of *checkpoints* where the prediction and potential adaptation should take place. A checkpoint defines a *trigger* which specifies where in the process the checkpoint should be triggered. A checkpoint specifies a set of *available* adaptation subjects at the checkpoint, i.e., those adaptation subjects for which still an alternative can be selected for the running instance after the checkpoint. A checkpoint also specifies the analysis tasks used at the checkpoint for KPI dependency analysis.

Finally, an adaptation model specifies a set of *preferences* and *constraints*. A preference is specified by referencing an influential factor or KPI from the analysis model and assigning a weight to this factor. A constraint is specified as a predicate over an influential factor or KPI.

The adaptation model is based on an already defined analysis model as it references influential factor definitions and KPI definitions from such an analysis model. The analysis model is again dependent on an already existing monitor model.

XML Serialization of the Adaptation Metamodel. An adaptation metamodel is serialized as follows.

Listing 5.1: Adaptation Metamodel Pseudo XML Schema

```
<adaptationModel targetNamespace="URI" name="NCName"
xmlns="http://.../m4c/schemas/adaptationModel">
 <adaptationSubject name="NCName">
  <characteristic name="NCName">QName</characteristic> +
  <alternative name="NCName">
   <effect characteristic="String"><predicate /></effect> +
   <action>...</action>
  </alternative> +
 </adaptationSubject> +
 <constraint property="QName"><predicate /></constraint> *
 <preference property="QName" weight="Integer" /> +
 <checkPoint name="NCName">
  <trigger>...</trigger>
  <adaptationSubject name="String" /> +
  <analysisTask>QName</analysisTask> +
 </checkPoint> +
</adaptationModel>
```

5.3.2 Adaptation Subjects

An adaptation model defines a non-empty set of adaptation subjects. An adaptation subject represents an adaptable entity of a specific process. In a BPEL process, an adaptation subject could be, for example, a particular partner link instance, activity instance or a variable instance. Possible adaptations for these subjects would then be service substitution, skipping of the activity, or changing of the variable value. The definition of adaptation subjects depends on the

available adaptation mechanisms and on the given alternatives for the adaptation subject in a specific process, i.e., the definition of an adaptation subject only makes sense, if there are at least two concrete adaptation alternatives (e.g., two or more alternative services).

An adaptation subject defines a non-empty set of characteristics. Characteristics is a set of properties which are affected by the adaptation subject and which are used for alternative selection purposes. Each alternative of an adaptation subject has to specify how it affects those characteristics. This makes it possible to compare the different alternatives in the selection process. Characteristics have to be defined as resource properties in the monitor model and as influential factors in the analysis model. They are defined in the adaptation model by referencing the corresponding influential factors in the analysis model.

An adaptation subject defines at least two alternatives (the default one at deployment time and an additional one) whereby exactly one alternative can be used for an adaptation subject at a point of time. When the process is deployed, then each adaptation subject has a default alternative assigned.

Each alternative specifies how it affects the characterizing properties of the corresponding adaptation subject. Therefore, an alternative specifies an effect for each characterizing property. An effect is specified as a predicate over property values (e.g., delivery time < 3 days). One possible source of information needed for the definition of effects are past measurements. If no such measurements are available then they have to be estimated by experts or can be derived from SLAs (e.g., in case of service substitutions). Obviously, after a certain period of time, one can compare the specified effects with actual monitored effects and then correspondingly adapt the effect definitions if needed.

In addition to its effects, the alternative defines how the adaptation should concretely be performed. This is done by defining an adaptation action. The definition of an adaptation action depends on the available adaptation mechanism. Three adaptation action types have been predefined, which can be used for adapting a running BPEL process instance after it has been halted at a checkpoint: (i) *WritePartnerLink* allows changing the service EPR (endpoint reference as defined in WS-Addressing) property in a partner link in the BPEL

process thus effectively performing service substitution; (ii) *WriteProcessVariable* allows changing process variable values, which can be used for example for changing the control flow in data-based branching activities (e.g., if-else); (iii) *ChangeActivityState* allows, e.g., skipping of activities. Of course, this set of adaptation action types could be extended to include other types of adaptation such as infrastructural reconfiguration.

Listing 5.2: Adaptation Subject Example

```
<adaptationSubject name="ShipmentService">
 <characteristic name="DeliveryTime">aM:ShipmentDeliveryTime
 </characteristic>
 <characteristic name="Reliability">aM:Shipper-Reliability
 </characteristic>
 <characteristic name="Cost">aM:Shipper-Cost
 </characteristic>
 <alternative name="ShipmentStandard">
  <effect characteristic="DeliveryTime">
   <predicate type="LOWER_EQUAL" value="3" />
  </effect>
  <effect characteristic="Reliability">
   <predicate type="EQUAL" value="0.5" />
  </effect>
  <effect characteristic="Cost">
   <predicate type="EQUAL" value="0.3" />
  </effect>
  <action>
   <WritePartnerLink serviceURI="http://.../StandardShipment" />
  </action>
 </alternative>
  ...
</adaptationSubject>
```

Example. In the scenario, one defines the shipment partner link in the reseller process as an adaptation subject, as it is assumed that there are alternative shipment services with different QoS characteristics available. For the adap-

tation subject, one defines `ShipmentDeliveryTime`, `Shipper-Cost` and `Shipper-Reliability` as characterizing properties. These properties have been defined in the analysis model as influential factors. One alternative `ShipmentStandard` is created, which specifies its effects on the three properties. It also defines the adaptation action `WritePartnerLink` and specifies the `serviceURI` of the service to invoke.

5.3.3 Checkpoints

An adaptation model defines a non-empty set of checkpoints. Checkpoints define where in the process the KPI prediction and potential adaptation should take place. Therefore, a checkpoint definition contains three types of information: (i) the trigger of the checkpoint, (ii) the available adaptation subjects at the checkpoint, and (iii) the analysis tasks for creating checkpoint specific KPI dependency trees.

The checkpoint trigger is defined as a process runtime event typically signaling the start or completion of an activity. The event is typically but not necessarily configured to be blocking, i.e., to stop process instance execution until prediction and potential adaptation are performed. The trigger definition references the corresponding topic of a manageable resource definition from which the events are retrieved.

A checkpoint furthermore defines adaptation subjects which are still available for adaptation at the checkpoint. Obviously, if process execution has already passed an adaptable subject *before* a checkpoint is triggered, then that adaptation subject is no more available at that particular checkpoint. The earlier a checkpoint is defined in the process, the more adaptation subjects are available. At the same time, however, the KPI prediction accuracy is lower, and vice versa.

Definition of Analysis Tasks for Checkpoints. When the checkpoint has been triggered, a prediction at that checkpoint is performed for each KPI. Therefore, corresponding KPI dependency trees (one tree per KPI) have to be already available, i.e., they should already have been learned based on historical instances.

For the creation of these dependency trees, one has to create corresponding

analysis tasks in the analysis model. Thereby, for each checkpoint in general specific analysis tasks have to be defined and thus also each checkpoint will have different KPI dependency trees. This is because checkpoints differ in the *available* influential factor values and adaptation subjects, resulting in different influential factor sets as part of analysis task definitions.

A KPI dependency tree at a checkpoint should in particular show how the KPI depends on the characteristics of the available adaptation subjects. This knowledge can then be used to select the most appropriate alternatives for those adaptation subjects. When defining influential factors for an analysis task of a checkpoint, two groups of influential factors are selected: (i) influential factors whose values have already been monitored and thus are known at the checkpoint; (ii) influential factors representing characteristics of available adaptation subjects at the checkpoint. The values of the latter group of influential factors are possibly not yet known at the checkpoint as the adaptation subject has not yet been executed. However these influential factors are needed for learning how the KPI depends on the characteristics.

The first group of influential factors is defined using the method as presented in Section 4.3.4 with the constraint that the underlying property values have to be already known when the checkpoint is executed. The second group of influential factors is defined by simply selecting the influential factors of all characteristics of all available adaptation subjects of the corresponding checkpoint. In our scenario, consider a checkpoint which is placed right after the warehouse check service. At that point one knows the ordered products, the result of the warehouse check and the execution duration of the process instance until the checkpoint. These properties are specified as influential factors as they are important factors for predicting a (duration-based) KPI value of an instance. These factors belong to the first group of factors. Assuming that at this checkpoint the available adaptation subjects are the supplier service and the shipment service, one would add in addition characteristics of these two adaptation subjects as influential factors, e.g., supplier delivery time, shipment delivery time, supplier reliability, shipment cost etc. These factors belong to the second group. Some of these might be constant values, which are not measured at runtime (e.g., supplier reliability), while others are measured

after the checkpoint for an instance, e.g., supplier delivery time, and is thus unknown during checkpoint execution. Their values are specified as part of the effects of corresponding alternatives, which is used in the adaptation process.

Obviously, the set of available factors of the first group increases in size the later the checkpoint is defined in the process thus increasing prediction accuracy, however at the same time the set of available adaptation subjects decreases, and thus there are fewer adaptation possibilities or it could even be too late for adaptation. Thus, there is a tradeoff between prediction accuracy and adaptation possibilities. In long-running processes where the prediction and adaptation only marginally influence the overall process execution time, one could define and use many different checkpoints in a process model.

Example. As shown in the listing, a checkpoint is defined after the warehouse check in the reseller process. The checkpoint trigger is specified by referencing the corresponding topic of the manageable resource definition in the monitor model. Also the checkpoint is defined to be blocking. The available adaptation subjects are the supplier service and the shipment service as they can still be selected after the checkpoint. Finally, an analysis task in the analysis model is referenced which defines the influential factors for a KPI for this specific checkpoint.

Listing 5.3: Checkpoint Example

```
<checkPoint name="AfterWarehouseCheck">
 <trigger>
  <triggerEvent blocking="true"
   manageableResourceDefinition="mM:CustomerResellerInteraction"
   topic="poChoreo:WarehouseCheckExecuted" />
 </trigger>
 <adaptationSubject name="SupplierService" />
 <adaptationSubject name="ShipmentService" />
 <analysisTask>aM:OrderFulfillmentTimeWarehouseCheck</analysisTask>
</checkPoint>
```

5.3.4 Constraints and Preferences

After KPI prediction, when several alternative adaptation strategies are identified, one needs to make a decision which of those alternatives is to be *selected*. Thereby, the approach addresses two aspects. Firstly, one wants to be able to specify that certain values of KPIs and characteristics should always be avoided. Secondly, as there are typically competing KPIs and properties (e.g., time vs. cost), one wants to be able to specify the preferences considering these properties thus allowing to rank adaptation strategy alternatives. The former aspect is addressed via constraints, the latter via preferences.

Constraints. A constraint is specified for a KPI or a characteristic and defines via a predicate the allowed values of the corresponding property. If during the selection of a strategy a constraint evaluation results in the value false for an adaptation strategy, then that strategy is removed from the set of alternatives. In the scenario, for example, one can use constraints for defining which KPI classes are allowed and which ones should be prevented in any case, e.g., by specifying that the KPI classes of a specific KPI should be green and yellow (and thus red is to be avoided). In the exceptional case that each adaptation strategy violates one or more constraints, constraints are neglected, and the strategy selection is performed solely based on preferences.

Preferences. Preferences are used for ranking of adaptation strategies according to a score represented by a number between 0 and 1. Therefore, Simple Additive Weighting as part of Multiple Attribute Decision Making [HY81] is used.

As part of the adaptation model, the user assigns weights to KPIs and characteristics of all adaptation subjects, whereby the sum of all weights has to be 1 (so that the resulting score is between 0 and 1; see equation 5.2 in Section 5.4.3). In addition, one has to specify for each underlying property of a KPI or characteristic (i) whether a higher value is better (e.g., reliability) or a lower value is better (e.g., cost), and (ii) a mapping of property values to a cardinal scale (if needed). For example, for a KPI with categorical values one could map green to 1, yellow to 0.5, and red to 0. These two pieces of

information are specified in the corresponding metadata descriptor of the property. Section 5.4.3 describes how based on this information a score for each adaptation strategy is calculated enabling ranking of adaptation strategies.

Example. In the following example, a constraint is specified which defines the allowed KPI classes. Furthermore, preferences are specified by giving weights on the KPI (0.3) and the six characteristics of all adaptation subjects in the adaptation model.

Listing 5.4: Constraints and Preferences Example

```
<constraint property="aM:OrderFulfilmentTime">
 <predicate type="IN" value="green;yellow"/>
</constraint>
<preference property="aM:OrderFulfilmentTime" weight="0.3"/>
<preference property="aM:ShipmentDeliveryTime" weight="0.15"/>
<preference property="aM:Shipper-Reliability" weight="0.1"/>
<preference property="aM:Shipper-Cost" weight="0.1"/>
<preference property="aM:SupplierDeliveryTime" weight="0.15"/>
<preference property="aM:Supplier-Reliability" weight="0.1"/>
<preference property="aM:Supplier-Cost" weight="0.1"/>
```

5.4 Runtime Adaptation based on KPI Prediction

Based on the adaptation model, at process runtime, prediction and adaptation are performed. KPI prediction involves monitoring of KPIs, learning of KPI dependency trees, and the actual runtime prediction based on the KPI dependency trees at predefined checkpoints. The adaptation part of the approach involves identification of adaptation requirements based on prediction results, the identification of adaptation strategies, and selection and enactment of one of those strategies.

Figure 5.3 shows the artifacts which are created and used at runtime during the prediction and adaptation process. Based on the monitor model and the analysis model, *instances* and *properties* are monitored and used for the creation of *KPI dependency trees*. When a checkpoint is triggered for a running instance,

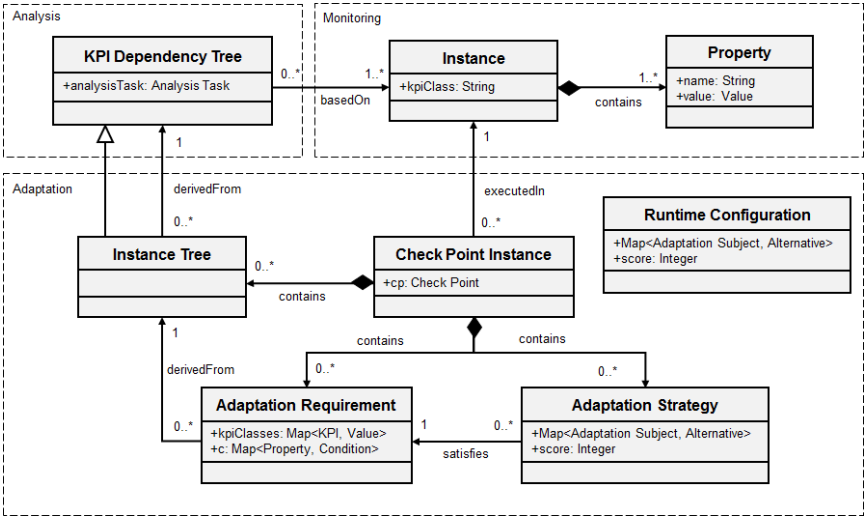

Figure 5.3: Runtime Artifacts Metamodel

a *checkpoint instance* is created. For each KPI, a KPI prediction is performed based on the KPI dependency tree resulting in an *instance tree*. An instance tree is a subtree of the corresponding KPI dependency tree and is derived by removing those nodes and branches which are no more relevant for the particular process instance based on monitored property values gathered so far until the check point. From an instance tree a set of *adaptation requirements* is extracted specifying how the contained influential factors should be improved. For each adaptation requirement a set of *adaptation strategies* is identified by using the available adaptation alternatives at the checkpoint. For each strategy a score is calculated based on the preferences model. Finally, resulting from the selected adaptation strategy, the *runtime configuration* contains the current assignment of an adaptation alternative to an adaptation subject (for all adaptation subjects).

5.4.1 Runtime Prediction of KPIs

Runtime prediction consists of two steps. First, the needed KPI dependency trees have to be learned for each checkpoint based on a set of already executed instances. Then, when a checkpoint is triggered for an instance, based on the trees, for each KPI a prediction is performed. The KPI prediction removes all those nodes and branches which are no more relevant for the running process instance and as a result creates an instance tree (per KPI). The instance tree contains only those influential factors (as nodes) which are still adaptable in the particuar process instance by using the available adaptation alternatives at the check point. The instance tree is then used as input to the next phase, the identification of adaptation requirements.

KPI Dependency Analysis for Adaptation. It has already been shown in the previous chapter how decision trees can be used for explanation purposes, i.e., to explain how KPI classes depend on a set of influential factors. In the context of the adaptation framework, those trees are utilized for prediction.

The adaptation model defines a set of checkpoints. Each checkpoint references a set of analysis tasks, whereby exactly one analysis task is defined for each KPI (cf. Section 5.3.3). For each analysis task, a KPI dependency tree is learned as described in Section 4.4.1. If, for example, two checkpoints and three KPIs are defined for the process, then six KPI dependency trees are created and stored in the monitor database. Learning is performed in parallel to the process execution (in the background) and is started after the configured number of instances has been executed.

Runtime Prediction based on Decision Trees. When the process instance execution reaches a checkpoint as specified in the checkpoint trigger, a new checkpoint instance is created. The process instance execution is halted if the triggering event is defined as blocking. For each defined analysis task of the checkpoint, the corresponding KPI dependency tree is obtained from the database. The prediction is then performed as described in the following for each KPI.

When using KPI dependency trees for prediction, based on the values of

Algorithm 5.1 Prediction based on KPI Dependency Trees

```
1:  T = (N, B) // KPI dependency tree as a set of nodes and branches
2:  IT = ∅ // instance tree
3:  P = {p_1, p_2, ...} // set of known influential factor properties

4:  IT = InstanceTree(rootNode(T)) // creates instance tree for T

5:  function INSTANCETREE(n)
6:      if isLeaf(n) then
7:          return (n, ∅)
8:      end if
9:      if propertyOf(n) ∈ P then
10:         for b ∈ B, sourceNode(b) = n do
11:             if propertyValueOf(n) ∈ values(b) then
12:                 return InstanceTree(targetNode(b))
13:             end if
14:         end for
15:     else
16:         IT_r = (n, ∅) // new subtree with n as root node
17:         for b ∈ B, sourceNode(b) = n do
18:             // create tree IT' for target node
19:             IT' = InstanceTree(targetNode(b))
20:             b' = (n, rootNode(IT')) // connect n with IT'
21:             IT_r = IT_r ∪ (∅, b') ∪ IT'
22:         end for
23:         return IT_r
24:     end if
25: end function
```

influential factors, the tree is traversed from the root to a leaf. In our case, the tree can contain (i) *known* (measured) influential factors and (ii) influential factors representing characteristics of adaptation subjects which are called in the following *adaptable factors*.

The tree is traversed breadth-first (cf. Algorithm 5.1). The recursive function *InstanceTree* starts at the root node of the dependency tree (line 4) and traverses the tree until the leaf nodes are reached (lines 6-8). If the current node

corresponds to a known factor (line 9), one follows the outgoing branch whose predicate is satisfied by the measured factor value (line 11) and replaces the current node with the target node of that branch for which the recursive function is continued (line 12); otherwise, in case of an adaptable factor one keeps the node in the tree (and continues with its children until a leaf node is reached) (lines 16-23).

As a result one gets a subtree of the original one (in the following denoted as instance tree) consisting either of (i) just one leaf representing the prediction of the corresponding KPI class (the special case); (ii) a tree containing one or more nodes of adaptable factors, i.e., properties representing characteristics of the available adaptation subjects. In the latter (general) case, the KPI class is thus predicted in relation to the values of adaptable factors.

Example. Figure 5.4 shows a KPI dependency tree (see also Section 4.4.1, Figure 4.6) generated for a checkpoint defined right after the interaction with

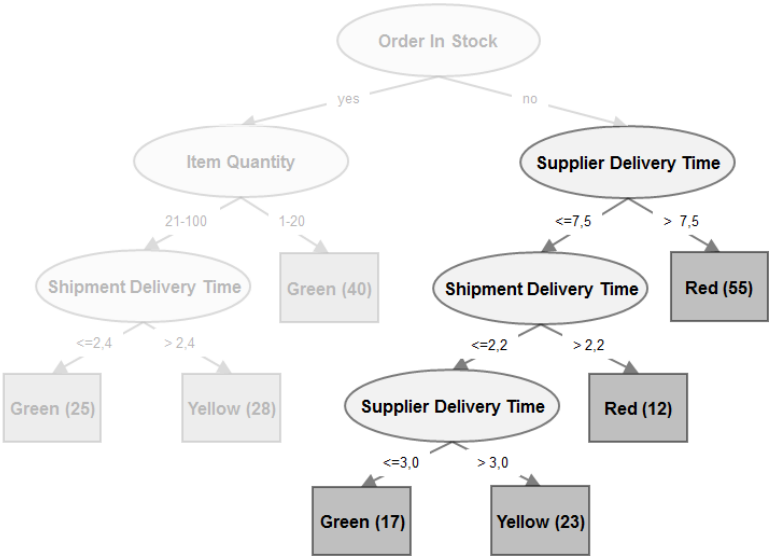

Figure 5.4: Instance Tree for *Order In Stock* = *No*

the warehouse and the corresponding instance tree. The original tree contains two known factors (order in stock, item quantity) and two adaptable factors (supplier delivery time, shipment delivery time), as the supplier and shipper still can be selected after the checkpoint.

If for the particular process instance order in stock = no, then the shown instance tree is created as a result of prediction. Starting at the root of the tree, order in stock is removed as it is a known factor and the left subtree is removed. The right subtree contains only adaptable factors which are not removed.

The resulting tree now consists of two distinct adaptable factors. It shows how the KPI classes depend on the value of supplier delivery time and shipment delivery time. This information is used in the next steps for selecting an adequate supplier service and shipment service.

5.4.2 Identification of Adaptation Requirements

At a checkpoint, after obtaining an instance tree for each KPI, it has to be decided whether adaptation is needed, and if yes, which properties should be improved and how. An instance tree shows how the KPI class of the running instance depends on the characteristics of available adaptation subjects.

If the instance tree contains only one leaf denoting the KPI class, then the predicted KPI class is independent of the adaptable subjects and an adaptation would not lead to another KPI class (for this KPI). If the instance tree contains more than just one leaf (as the one in Figure 5.4), then the non-leaf nodes correspond to influential factor properties which are adaptable by the predefined adaptation alternatives and the tree shows how one should adapt. For example, if in the instance tree in Figure 5.4 a supplier delivery time below 3 and a shipment delivery time below 2,2 are ensured, one will very likely (assuming that the classification model has a high accuracy) reach the KPI class green.

The idea towards adaptation based on the instance tree is thus (i) to extract tree paths and the corresponding conditions (specified on tree branches), and then (ii) select adaptation alternatives which will lead to satisfaction of those conditions. Each path (consisting of a conjunction of conditions) is an alternative *adaptation requirement* for the corresponding KPI. All conditions (of a path)

have to be true in order to satisfy the adaptation requirement. Note that at this point, the paths which lead to bad KPI classes (e.g., red) are also extracted and considered. This is because in case of several KPIs and corresponding settings in the preferences model, such a path could still be relevant.

An adaptation requirement (AR) is specified in terms of (i) the predicted KPI class per specified KPI, and (ii) a conjunction of conditions which should be achieved in order to reach those KPI classes.

Adaptation requirements are identified as shown in Algorithm 5.2. In case more than one KPI has been defined, alternative adaptation requirement sets are extracted from each instance tree separately (line 4, 17-30). Thereby, the recursive function *ExtractRequirements* extracts each root-to-leaf path of the instance tree and creates one AR per path by extracting the conditions from the branches (lines 23-27). Arriving at a leaf of the tree, an adaptation requirement is created by adding the KPI class of the leaf and all conditions collected so far on the corresponding root-to-leaf path (lines 19-21).

The adaptation requirements of the instance trees are combined by iteratively building a Cartesian product between them (using nested for-loops in lines 6-14), i.e., $AR_1 \times AR_2 \times ... \times AR_n$ whereby AR_i denotes the set of adaptation requirements for the i-th instance tree. For each resulting element of the product a new adaptation requirement is created by combining the classes and conditions of the underlying adaptation requirements using the union operator (line 11). As a result, one gets a set of *alternative* adaptation requirements each consisting of a conjunction of conditions over adaptable factors which have to be satisfied to reach the corresponding KPI class(es).

Example. For the example instance tree in Figure 5.4, four adaptation requirements are extracted, one for each tree path. For example, for the tree path leading to the green KPI class, the adaptation requirement (green, Supplier Delivery Time <= 7,5 and Shipment Delivery Time <= 2,2 and Supplier Delivery Time <= 3,0) is created (the function *ExtractRequirements* is called in this case four times until it reaches the leaf where the corresponding adaptation requirement is created (line 20)). The two adaptation requirements with KPI class red are also extracted. While in most cases the corresponding strategies

Algorithm 5.2 Identification of Adaptation Requirements

$IT = \{it_1, it_2, ...\}$ // set of KPI instance trees
$AR = \emptyset$ // set of resulting adaptation requirements $ar_i = (C, CN)$, whereby C contains a KPI class for each KPI, CN is a set of branch conditions

for $it_i \in IT$ **do**
    $AR_i = ExtractRequirements(rootNode(it_i), \emptyset)$
    $AR_{new} = \emptyset$
    for $(C, CN) \in AR_i$ **do**
        if $AR = \emptyset$ **then** // first tree
            $AR_{new} = AR_{new} \cup (C, CN)$
        else// combine with ARs of other trees so far
            for $(C', CN') \in AR$ **do**
                $AR_{new} = AR_{new} \cup (C \cup C', CN \cup CN')$
            end for
        end if
    end for
    $AR = AR_{new}$
end for

function EXTRACTREQUIREMENTS(n, CN)
    $AR_{new} = \emptyset$
    if $isLeaf(n)$ **then**
        $AR_{new} = AR_{new} \cup (kpiClass(n), CN)$
        return AR_{new}
    else
        for $b \in B, sourceNode(b) = n$ **do**
            $CN_{new} = CN \cup condition(b)$
            $b' = targetNode(b)$
            $AR_{new} = AR_{new} \cup ExtractRequirements(b', CN_{new})$
        end for
        return AR_{new}
    end if
end function

will be irrelevant during strategy selection (cf. Section 5.4.3) due to their bad score, they might be relevant in the case that this particular KPI gets a low weight in the preferences model if there is more than one KPI specified.

5.4.3 Identification and Ranking of Adaptation Strategies

After the requirements have been identified, the next step is to identify adaptation strategies which can be used to satisfy the adaptation requirements. An adaptation strategy defines for each adaptation subject (of a checkpoint) an adaptation alternative which (i) satisfies the conditions of one particular adaptation requirement and (ii) satisfies the specified constraints in the adaptation model.

Identification of Adaptation Strategies. Adaptation strategies are identified as shown in Algorithm 5.3. For each adaptation requirement a set of alternative strategies is identified by using the function *DetermineStrategies*. For each adaptation subject (line 10), one searches for qualifying alternatives (of that subject) which do not violate the conditions of the adaptation requirement according to their specified effects (lines 14-23). An effect violates a condition (line 17) if it refers to the same influential factor and the effect predicate does not satisfy the condition. If the underlying characteristic has a cardinal scale, then the *worst value* of the predicate is taken and inserted into the condition. For example, if an effect predicate specifies that Supplier Delivery Time <= 5 and the condition is Supplier Delivery Time < 3, then the effect does not satisfy the condition as 5 is the worst value of the value range expressed by the effect predicate and $5 < 3$ is false. If the underlying characteristic has a nominal scale (i.e., only EQUAL, NOT_EQUAL operators are allowed in the condition and the effect predicate), then each nominal value resulting from the predicate (whereby several nominal values are possible if predicate uses the NOT_EQUAL operator) is checked against the condition. If no qualifying alternative is found for an adaptation subject, then for the corresponding AR no valid adaptation strategy exists (lines 24-26). For already executed adaptation subjects (which can no more be adapted), the chosen alternative is used as the single qualifying alternative of that subject (not shown in the Algorithm 5.3). Each qualifying

Algorithm 5.3 Identification of Adaptation Strategies

$AR = \{ar_1, ar_2, ...\}$ // set of alternative adaptation requirements
$AS = \{as_1, as_2, ..., as_n\}$ // set of adaptation subjects
$A = \{a_{11}, a_{12}, ..., a_{nm}\}$ // set of alternatives where a_{nm} is the m-th alternative of the n-th adaptation subject
$S = \emptyset$ // set of resulting adaptation strategies $s_i = \{(as_1, a_{1a}), ..., (as_n, a_{nb})\}$
for $ar \in AR$ **do**
    $S = S \cup DetermineStrategies(ar)$
end for
function DETERMINESTRATEGIES(ar)
    $S_r = \emptyset$ // resulting strategies
    for $as_i \in AS$ **do**
        $S' = S_r$ // temporary set of partial strategies
        $S_r = \emptyset$
        $QA = \emptyset$ // qualifying alternatives
        for $a_{ij} \in A$ **do**
            for $ef \in effectsOf(a_{ij})$ **do**
                for $cn \in conditionsOf(ar))$ **do**
                    if $violates(ef, cn)$ **then**
                        $continue\ with\ a_{ij+1}$ // alternative does not qualify
                    end if
                end for
            end for
            $QA = QA \cup a_{ij}$ // alternative qualifies
        end for
        if $QA = \emptyset$ **then**
            return $\emptyset$ // ar cannot be satisfied
        end if
        for $a_{ij} \in QA$ **do**
            if $S' = \emptyset$ **then**
                $S_r = S_r \cup \{(as_i, a_{ij})\}$
            else
                for $s \in S'$ **do**
                    $S_r = S_r \cup (s \cup (as_i, a_{ij}))$
                end for
            end if
        end for
    end for
    return S_r
end function

alternative of one adaptation subject is combined with each qualifying alternative of other adaptation subjects thus creating a set of alternative adaptation strategies (lines 27-35). The number of resulting adaptation strategies is given by $|QA_1| \cdot |QA_2| \cdot ... \cdot |QA_n|$ whereby QA_i denotes the set of qualifying alternatives for the i-th adaptation subject. As a result, a set of adaptation strategies for an AR is created, whereby each adaptation strategy contains for each adaptation subject exactly one alternative.

Finally, the resulting set of alternative adaptation strategies is the union of adaptation strategies for each (alternative) AR (lines 4-6).

Filtering of Adaptation Strategies. After the identification, the adaptation strategy set is filtered according to the constraints defined in the adaptation model. If a constraint evaluation evaluates to false for a strategy, then that strategy is removed from the set. The result is set of alternative valid adaptation strategies.

Ranking of Adaptation Strategies. Based on the specified preferences in the adaptation model, for each strategy a score number is calculated. The strategies are then ranked according to the score and the strategy with the highest score is enacted.

The score of an adaptation strategy is calculated based on the preferences model, which assigns weights to the property set $P_w = \{p_1, p_2, ..., p_m\}$ consisting of KPIs and characteristics of the adaptable entities (cf. Section 5.3.4). For each adaptation strategy x and property $y \in P_w$ one can determine the property value v_{xy}. In case of a KPI, v_{xy} is the predicted KPI class (specified in the referenced adaptation requirement). In case of a characteristic, it is the value defined by the predicate of the effect definition of the corresponding alternative (the worst value is taken in case the predicate specifies a value set). For example, if the predicate is specified as duration <= 5 days, then 5 days is used as value. The v_{xy} is always a real number; if the underlying property has not a cardinal scale, a mapping has to be provided (cf. Section 5.3.4).

Before applying the simple additive weighting (SAW) [HY81], one has to normalize these values to make the different properties comparable. The normalized property value nv_{xy} can be calculated by using the division by

maximum value method:

$$nv_{xy} = \begin{cases} \frac{v_{xy}}{max_y(v_{1y}, v_{2y}, ..., v_{my})} & \text{if higher values are better} \\ \\ \frac{{v_{xy}}^{-1}}{max_y({v_{1y}}^{-1}, {v_{2y}}^{-1}, ..., {v_{my}}^{-1})} & \text{if lower values are better} \end{cases} \tag{5.1}$$

The normalized metric values nv_{xy} are in the range between 0 and 1, whereby the value 1 is always given to the best property value among all strategies. One thereby has to distinguish between properties where a higher value is better (e.g., reliability) and properties where a lower value is better (e.g., cost). This has to be specified for each property in its metadata descriptor.

Finally, for each strategy, a score is calculated by summing up the weighted metric values:

$$score_x = \sum_{y=1}^{m} w_y nv_{xy} \tag{5.2}$$

Example. Table 5.1 shows two adaptation requirements extracted from the instance tree (Figure 5.4) and the identified alternative strategies per requirement. Each strategy consists here of a combination of a shipper service and supplier service with different effects. For each strategy a normalized metric values vector is constructed containing the corresponding KPI class (green is mapped here to the value 1.0, while yellow is assigned 0.5), and the duration, cost, and reliability characteristics for the shipper and the supplier, respectively. Based on the weight distribution (0.3, 0.1, 0.2, 0.05, 0.1, 0.2, 0.05) in the preferences model (0.3 being given to the KPI), the score for each strategy is calculated and used for ranking.

5.4.4 Adaptation Enactment

When the process is deployed, all the adaptation subjects have to be assigned a (default) adaptation alternative. This *default runtime configuration* of all

Requirements		Adaptation Strategies			
ID	KPI Class	Strategy	Constr.	Score	Rank
1	green (1.0)	Sh1 (0.9, 0.2, 1.0), Su1 (1.0, 0.3, 0.9)	ok	0.69	2
		Sh1 (0.9, 0.2, 1.0), Su2 (0.9, 0.5, 0.8)	ok	0.71	1
		Sh2 (1.0, 0.1, 0.8), Su1 (1.0, 0.3, 0.9)	nok	0.67	-
		...	...	...	
2	yellow (0.5)	Sh3 (0.7, 0.7, 0.6), Su3 (0.6, 0.9, 0.6)	ok	0.66	3
		Sh3 (0.7, 0.7, 0.6), Su4 (0.7, 0.7, 0.7)	ok	0.64	4
		Sh3 (0.7, 0.7, 0.6), Su2 (0.9, 0.5, 0.8)	ok	0.62	5
		...	...	...	

Table 5.1: Identification and Ranking of Adaptation Strategies

adaptation subjects is then dynamically changed if the selected adaptation strategy contains different adaptation alternatives than the default ones for at least one adaptation subject. Otherwise, no adaptation has to take place, as the default configuration is already the optimal one.

For creating the default configuration before deployment, the specified constraints and preferences can be used. Therefore, one first creates a set of adaptation strategies by simply enumerating all adaptation alternatives for each adaptation subject and then building the Cartesian product over all subjects. The constraints for each strategy are checked and the score is calculated for each strategy based on characteristics only, i.e., without KPIs as these are obviously not known at design time. Thus, the ranking is done based on characteristics of the adaptation alternatives.

After the first deployment, the framework starts monitoring the execution of process instances based on the default configuration. Until the KPI dependency trees are learned, the default configuration can be used for all monitored process instances. This leads to KPI dependency trees which have been learned only based on execution data of the default configuration. They do not reflect the behavior of all other adaptation alternatives, as those have not yet been executed. In our example, the trees would reflect only the behavior of a specific supplier and a specific shipper; the behavior (e.g, supplier delivery time, shipment delivery time) of other alternative suppliers and shippers would not be present in the historical data and thus also not be present in the dependency

trees. Thus, the accuracy of the trees after this first learning would not be optimal in respect to all available adaptation alternatives, in particular if the behavior of those alternatives differs a lot from the default ones. There are two possibilities to deal with this issue: (i) neglect this fact, perform the runtime adaptation for some time (taking into account the lower accuracy) and then relearn the trees; (ii) explicitly use a bootstrapping phase where not only the default configuration is used but many different or all possible configurations of adaptation alternatives. After the bootstrapping phase, learn the trees and then start with the adaptation.

5.5 Summary and Conclusions

This chapter has presented a proactive runtime process adaptation approach based on KPI dependency analysis. Based on a monitor model and a KPI analysis model, an adaptation model is created which defines (i) adaptation subjects and corresponding alternatives, (ii) checkpoints, and (iii) constraints and preferences. Based on these settings, at runtime the adaptation framework adapts the running process instances automatically, trying to improve KPI performance and take into account specified constraints and preferences. This is done by halting the process execution at predefined checkpoints and performing a KPI prediction based on KPI dependency trees. The KPI prediction result is an instance tree, which is used for the extraction of adaptation requirements. Based on the predefined adaptation alternatives, a set of adaptation strategies is identified. Finally, one strategy is selected and enacted based on the predefined constraints and preferences. The prototypical implementation and an experimental evaluation of the approach are described in Chapter 6.

CHAPTER 6

IMPLEMENTATION AND EVALUATION

This chapter presents a prototypical implementation and an experimental evaluation of the approach. In Section 6.1, a Java-based prototype is described which implements the monitoring, prediction, and adaptation concepts as presented in the previous chapters. Section 6.2 then describes experiments performed based on this prototype. Therefore, the purchase processing scenario has been developed and run on the prototype.

6.1 Prototypical Implementation

In the following, an overview of the architecture and the main components is given. The details are then provided in the following subsections. The prototype is a Java-based application running in a Tomcat server. It uses several existing open-source frameworks and libraries, in particular an existing BPEL process engine, a library implementing the BPEL 2.0 event model, a CEP engine, a framework supporting WSDM, and data mining tooling. The needed

functionality has been implemented on top of them.

Figure 6.1 gives an overview of the main components. The process execution is implemented based on the Apache ODE BPEL execution engine 1.3.4 [Apac]. In addition to ODE, the Pluggable Framework extension for ODE is used [Ins]. It implements the BPEL 2.0 event model [KHK+11] and allows in particular subscribing to events needed for monitoring, and halting and resuming of process instances needed for the implementation of checkpoints.

The monitoring framework is responsible for obtaining events from the process engine, processing them based on CEP, storing the evaluated resource property values in a Monitor DB and exposing the monitoring functionality over a WSDM-based interface. The events are received from the process engine using the Pluggable Framework. The events are thereby published by a generic controller to queues and topics, which are then received by custom controllers. The Pluggable Framework uses JMS for messaging and Active MQ 5.2 [Apaa] is used as the underlying JMS implementation. The events are then processed as defined in the monitor model by the implemented capabilities. Composite properties are evaluated based on the ESPER complex event processing (CEP) framework 3.2 [Esp]. The resources and corresponding property values are stored in the Monitor DB for later use (e.g., by the analysis component) and are also directly forwarded, if needed, to subscribers using the WSDM interfaces. For implementing WSDM, the Apache Muse 2.2 library [Apab] has been used.

The KPI dependency analysis is based on the WEKA suite 3.5 [HFH+09], which provides data mining tooling and in particular decision tree algorithm implementations. A KPI dependency analyzer component performs data preparation based on the KPI dependency analysis model and the data in the Monitor DB and uses the WEKA Java API to construct the dependency trees. WEKA's functionality for displaying the resulting trees in a GUI has also been integrated.

For the implementation of checkpoints and instance adaptation, the Pluggable Framework has been used as basis. The checkpoints are supported via blocking events, which stop process instance execution until they are explicitly unblocked by a corresponding incoming event coming from the adaptation framework. The adaptation actions are also realized using incoming events whereby the adaptation framework populates the corresponding incoming event (e.g., by

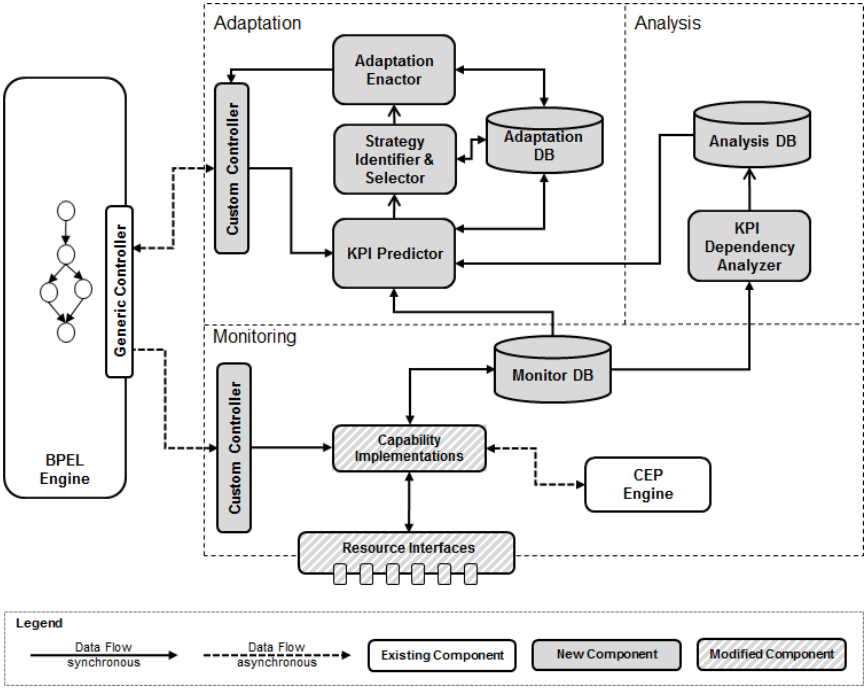

Figure 6.1: Main Components of the Prototype

setting a new partner link value) and sends it to the process engine. The algorithms for identification of adaptation requirements and strategies are implemented in Java.

The Monitor DB, Analysis DB, and Adaptation DB have not been realized based on a database management system, but their contents are simply held in memory in corresponding Java objects, which is sufficient for experimentation purposes.

Based on the prototype, the purchase order process has been implemented, mainly based on the examples that have been used throughout the thesis. The services used by the process are configurable to simulate certain behavior, which allows us to perform experiments and evaluate the approach. The experimental evaluation is presented in Section 6.2 in detail.

6.1.1 Monitoring Framework

The monitoring framework as presented in Chapter 3 uses the WSDM framework as basis and uses it to support monitoring of process-related resource types. When in it comes to implementation, it can be seen as consisting of two parts, namely (i) gathering of the needed monitoring information from the process engine and (ii) exposing that information as manageable resources over corresponding Web service interfaces. The first part is implemented on top of the Pluggable Framework, the second one uses the Apache Muse library as basis, as shown in Figure 6.2.

Usage of the Pluggable Framework. The Pluggable Framework provides a *generic controller*, which runs in the ODE engine and communicates with an arbitrary set of custom controllers using a messaging infrastructure. The generic controller sends outgoing events (as defined in the BPEL 2.0 event model) to a JMS topic, which custom controllers can subscribe to. In addition, they can subscribe to *blocking events* and send *incoming events* to the generic controller

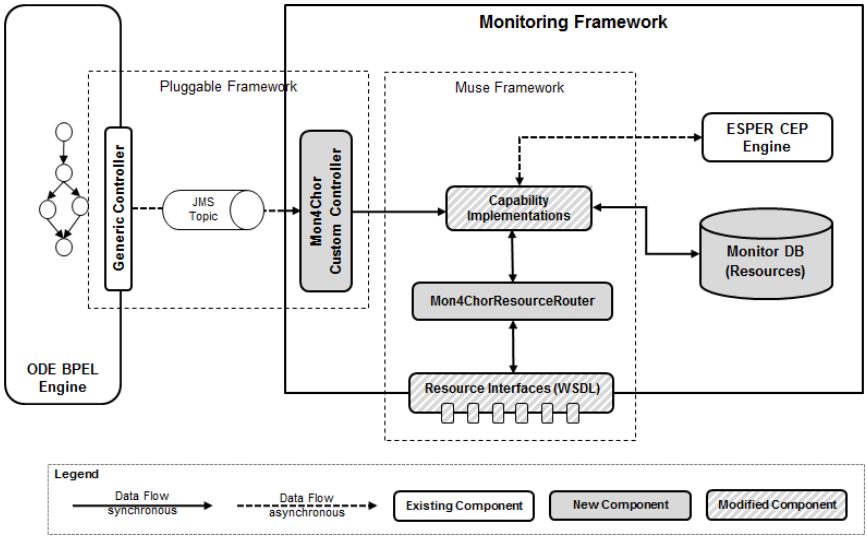

Figure 6.2: Monitoring Framework

for active control of the process instance; this is used for adaptation and is described further below in the context of the adaptation framework.

For supporting the monitoring approach, the `Mon4ChorCustomController` has been implemented. During deployment, it subscribes to the topic of the generic controller. At runtime, it then simply forwards the incoming events to the corresponding capability implementations, e.g., all activity state change related events are forwarded to the `ActivityStateEventCapability` and the `ActivityStateCapability` implementation. It is then in the responsibility of the capability implementation to process the event as specified in the deployed monitor models.

Event Mappings. When using executable BPEL processes as implementation of the abstract processes in a choreography, two additional aspects have to be considered when a monitor model is specified based on choreography models: (i) taking into account the differences between the executable process and the corresponding abstract process considering their event models, and (ii) augmentation of events with resource identifiers related to choreographies.

An abstract BPEL process can define opaque elements and can omit elements which are then added during executable completion. In our context, opaque activities are relevant, as their state model and identifiers can change during executable completion. If an activity is defined as an opaque activity, it is treated as having the resource type Activity Instance and the state model of a standard activity (i.e., no loop or invoke activity). If during executable completion the opaque activity is realized as an invoke activity or a loop activity, i.e., the corresponding state model has changed in the executable process, then it has to be ensured that when monitoring the abstract process, the state model of a standard activity is exposed. Besides defining activities as being opaque, one can omit activities in the abstract process which are then added during executable completion. In that case, it can happen that the identification properties which rely on XPath are no more valid.

Those two aspects are dealt with by defining an event mapping. The source event is the event retrieved from the BPEL engine (based on BPEL event model 2.0), and the target event is the event which (i) corresponds to the defined

manageable resources based on an abstract process, and (ii) contains additional identifiers. Listing 6.1 shows an example event mapping. It is defined for a specific manageable resource definition. In this case the mapping is defined for the activities of the reseller process. It then defines the source event and the corresponding target event which is to be created when the source event is retrieved from the engine. Thereby, the source event properties are replaced by target event properties. In this case, the `activityXPath` is changed and an additional `topology` identifier is added. Also the target event is only published for state changes relevant for opaque activities, i.e., states of invoke activities are mapped to states of opaque activities. Therefore, a mapping between the state model of an invoke activity (and a loop activity, respectively) and the state model of an opaque activity has been predefined. The mapping is performed at runtime by the `EventMappingHandler`, which based on such an event mapping definition gets a source event as input and creates a target event as output.

Listing 6.1: Event Mapping Example

```
<eventMappings>
 <eventMapping
  manageableResourceDefinition="mM:ResellerActivities">
  <sourceEventProperties elementType="InvokeActivity">
   <m4c:process>rs2:PurchaseOrderProcess</m4c:process>
   <m4c:scopeXPath>/process</m4c:scopeXPath>
   <m4c:activityXPath>/process/sequence[0]/invoke[2]
   </m4c:activityXPath>
  </sourceEventProperties>
  <targetEventProperties elementType="OpaqueActivity">
   <m4c:topology>po:POChoreography</m4c:topology>
   <m4c:process>reseller:ResellerProcess</m4c:process>
   <m4c:scopeXPath>/process</m4c:scopeXPath>
   <m4c:activityXPath>/process/sequence[0]/opaqueActivity[1]
   </m4c:activityXPath>
  </targetEventProperties>
 </eventMapping>
```

```
18  ...
</eventMappings>
```

Usage of the Muse Framework. Apache Muse provides a framework for implementing applications which access resources as specified in the WSRF, WS-BaseNotification, and WSDM specifications.

The Muse programming model is as follows. One defines a resource type in terms of a WSDL interface which specifies a set of WSRF resource properties and operations. In a corresponding Muse deployment descriptor one then defines, in addition to the resource router and resource manager implementation, for each resource type the context path and the set of provided capabilities which reflect the functionality of the WSDL interface. Each capability is provided a capability implementation in terms of a Java class. A Muse application is packaged as a WAR file and deployed in a web container, such as the Tomcat server. At runtime, the resource router obtains the SOAP request from the SOAP engine and based on the SOAP headers constructs the EPR of the target resource, obtains the resource to which the request is to be routed from the resource manager based on the EPR, and then delegates the request to the corresponding capability of the resource.

Muse provides a set of standard capability implementations, in particular the ones specified in WSDM, and a standard resource router and resource manager. All of these components can be customized and extended. Muse has been used as follows. New resource router and resource manager implementations have been created, the `Mon4ChorResourceRouter` and the `Mon4ChorResourceManager` respectively, by overriding the existing Muse classes. This was needed because the Muse internal `Resource` representation has not been used. Instead, the resources are stored in the Monitor DB. Internally, a Muse resource type has been mapped to one Muse `Resource`, which then obtains the particular concrete resource based on the EPR from the Monitor DB.

New capabilities as described in Chapter 3 have been implemented. That involved creating the corresponding WSDL definition and a corresponding Java-based implementation for each capability. The capability implementa-

tion realizes the operations, notifications, and property access of the capability interface. A capability implementation can use other capabilities, e.g., for sending of notifications, or internal monitoring mechanisms, such as the `Mon4ChorCustomController` for getting information from the process engine.

For example, the interface of the `ActivityStateCapability` provides a `state` property of an activity instance. A corresponding WSDL definition `ActivityState.wsdl` has been created and a corresponding Java class `ActivityStateCapabilityImpl`, which provides a `getState` method has been implemented. Only one object of that class is instantiated. It acts as a proxy for all resource instances of that capability. If, for example, the service requester invokes a WSRF method (e.g., `getResourceProperty`) on the Activity State interface requesting that property for a specific activity instance referenced in the EPR, the `getState` method is invoked by the Muse framework. The method first obtains the corresponding activity instance from the Monitor DB and then returns the state property value of that resource.

When the Muse application is deployed in the web container, the monitor model is used for configuring the capability implementations. Depending on the capability, this can include establishing filters on which concrete resources are to be monitored (based on the settings in the `resourceDescriptor` element of the manageable resource definition), subscribing to topics of other manageable resource endpoints, or in case of custom properties, registering of CEP statements in the CEP engine.

For implementing notifications, the pub/sub-mechanism of WS-Notification for which Muse already provides several capabilities is used. Thereby, the `NotificationProducer` provides a `subscribe` operation which is used by `NotificationConsumers` to subscribe for a set of topics. They are then notified using the `notify` operation. For instance, when an activity event is received from the generic controller, the `Mon4ChorCustomController` forwards that event to the Java class `ActivityStateEventCapabilityImpl`. The capability implementation first determines based on the monitor model whether that event is relevant at all. In the positive case, a corresponding XML-based WSDM event is created and published to the corresponding topic

of the capability Activity State Event.

The implementation of the Event Composition capability creates composite events based on CEP statements. During initialization, the capability implementation `EventCompositionCapabilityImpl` subscribes to topics for getting events needed for calculation of the corresponding properties and registers the CEP statement in the ESPER engine. At runtime, when an XML-based event is received, it is forwarded to the CEP engine. The CEP engine sends the resulting event back to the capability implementation which forwards it to a corresponding target topic. The property value is extracted from the event by `CustomPropertyCapabilityImpl` and is updated in the corresponding resource in the Monitor DB. It is available for later pull requests and sent to the outgoing topic of the Custom Property capability notifying subscribed consumers.

6.1.2 Analysis and Adaptation Framework

The KPI dependency tree learning is based on the WEKA suite 3.5, which provides well-known decision tree algorithms as a Java library. The concrete algorithms and settings which have been used are described in the experimentation section.

The `KPIAnalyzer` gets as input a `KPIAnalysisModel`. For each analysis task, first the resources to be analyzed are determined in the Monitor DB. For each resource, the needed property values are obtained and the data set as requested by WEKA is built, i.e., the corresponding data objects are populated. The learned tree is returned describing the tree in the DOT format. For manipulating the tree structure for prediction as part of the adaptation framework, a `TreeParser` parses the DOT text and creates a Java-based representation of the tree, a `DecisionTree` object consisting of `TreeNode` and `TreeBranch` objects. It also stores a set of quality metrics provided by the WEKA learning algorithm, e.g., the accuracy of the learned tree. The Java-based representation of the tree is then later used during prediction and extraction of adaptation requirements. If a graphical display of the tree is needed for analysis purposes, the tree can be shown in a Java Swing-based user interface.

The runtime adaptation part as described in Chapter 5 consists of the implementation of checkpoints, runtime prediction based on KPI dependency trees, identification and ranking of adaptation strategies, and adaptation enactment. The components of the adaptation framework are depicted in Figure 6.3.

The prerequisite for adaptation to take place, is that already a set of process instances has been monitored and corresponding KPI trees have been learned. Therefore, a `DependencyAnalysisScheduler` regularly checks in the Monitor DB whether the configured set of process instances has been executed and then triggers the learning process. The `DecisionTree` objects for analysis are stored in the Analysis DB.

The realization of checkpoints is based on the Pluggable Framework and its feature of blocking events. A blocking event halts the process instance execution, until another event (incoming event) is sent to the engine. For supporting checkpoints, a new custom controller, the `CheckPointCustomController`, has been implemented. It is configured via the adaptation model (cf. Section 5.3.3). Based on the checkpoint trigger definition, it registers at the generic controller and creates a temporary incoming queue for receiving needed blocking events from the generic controller.

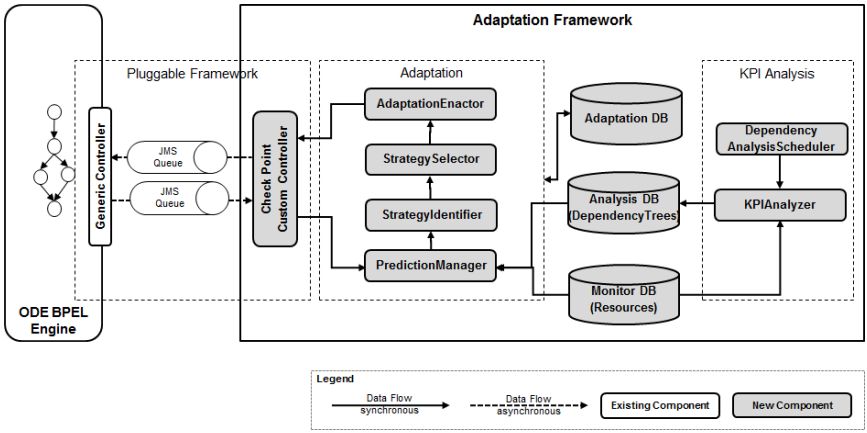

Figure 6.3: Adaptation Framework

During process execution, the CheckPointCustomController receives blocking events and checks whether they trigger a checkpoint. When a checkpoint is triggered, the performAdaptation operation is invoked in the class AdaptationProcess. The operation orchestrates all steps of the adaptation process as follows. A new CheckPointInstance object is created representing an instance of the corresponding checkpoint definition. The corresponding resource (typically the current process instance) is determined and retrieved from the Monitor DB and assigned to the CheckPointInstance.

For each KPI, the PredictionManager retrieves the KPIDependencyTree objects of the checkpoint from the Analysis DB and the known properties for the instances are retrieved from the Monitor DB. The prediction is performed and a new InstanceTree object is created representing the instance tree. The resulting instance trees are saved in the CheckPointInstance object and can be used in the next phase for identification of adaptation requirements.

The RequirementsIdentifier extracts a set of adaptation requirements from the trees. The StrategyIdentifier identifies strategies, which are then ranked by the StrategySelector based on the defined preferences and constraints in the adaptation model. The created AdaptationRequirement and AdaptationStrategy objects are appropriately connected and stored in the Adaptation DB for later analysis purposes (cf. runtime artifacts metamodel in Section 5.4).

The selected adaptation strategy is sent to the AdaptationEnactor, which updates the RuntimeConfiguration with the current binding of adaptation subjects (if needed). The adaptation actions are supported by the Pluggable Framework via incoming events, i.e., as a reply to a blocking event, an incoming event is sent to the engine, which changes the state or value of an activity or variable and unblocks the process execution. Supported adaptation action types are service substitution by changing the endpoint address in a partner link, change of process variable values, and skipping of activity states.

6.2 Experimental Evaluation

The KPI dependency analysis and the runtime adaptation approach have been experimentally evaluated based on the purchase order processing scenario.

The purchase order processing scenario has been implemented based on the prototype. The process of the reseller has been realized as a BPEL process which interacts with six Web services. These Web services are provided in the scenario by two different suppliers, the shipper, the warehouse, the bank and the customer. The customer service thereby triggers the reseller process by sending an order to it. The reseller process interacts with the other five services to process the order.

The Web services have been implemented in Java as mockup services and simulate certain influential factors. For example, the execution time, the availability and the outputs of a service operation can be configured to vary based on certain configurable probabilities and dependent on input data. Additionally, a simple Java client simulating the customer role in the choreography has been created, which triggers the process instances of the reseller. The so created testbed allows to configure the behavior of services in the choreography.

For experimentation, all components of the prototype have been installed on a single laptop computer. This ensures that external influential factors such as network latency cannot influence the experimentation results.

6.2.1 Experimental Evaluation of the KPI Dependency Analysis

In the first step, the KPI dependency analysis has been evaluated without runtime prediction and adaptation focusing solely on evaluating its capability to *explain* the influential factors. The evaluation results presented in the following have already been described in [WLR+09].

Order fulfillment time is defined as the KPI to be analyzed, measured as the overall duration of the reseller process, and create a set of around 30 potential influential factors. The corresponding property definitions are generated as described in Section 4.3.4. In addition, several domain-specific properties such as *product types*, *number of ordered products*, *customer type*, and *order in stock*

are defined manually.

The experimentation procedure is as follows. First, a setting is created by configuring the mockup services to simulate a certain behavior, so that the influential factors which are expected to be shown by the analysis are known beforehand. Then, the execution of a set of process instances is triggered using the test client. As the process instances are executed, both the result of the KPI property and the values of the influential factors are monitored and saved in the Monitor DB. After the execution of all process instances has finished, the dependency analysis is performed. The analysis result is then evaluated in respect to the previously configured expected influential factors.

The configuration simulates the following influential factors: (i) certain product types are configured to not be available in the warehouse with higher probability than others; in that case these products have to be ordered from suppliers, which has a major impact on process duration (ii) the supplier delivery time of supplier 1 is on average higher than expected; (iii) the average shipment delivery time can vary strongly and is relatively high in relation to the overall process duration. The KPI has two classes (green and red). Based on the settings, the expectation is that the dependency analysis shows that the KPI mainly depends on product type, supplier 1 delivery time, and shipment delivery time. Other influential factors such as response times of services also influence the KPI value, but in rather marginal way, and they are expected not to be shown.

Figure 6.4 shows the generated dependency tree after the execution of 390 process instances. The tree was generated using the J48 decision tree algorithm from the WEKA tool suite. It shows that all process instances in which shipment delivery time was greater than 96 seconds (the time unit shown in the tree is milliseconds) were red. It also shows that this was the case for 59 instances (out of 390). In the other case, the outcome further depended on the order in stock property, which denotes whether the orders could be delivered from stock or had to be ordered by the suppliers. All instances where order in stock was true, reached a green KPI value. Otherwise, again the outcome depended on shipment delivery time, supplier 1 delivery time, and response time supplier 1.

When comparing the influential factors in the generated tree with the ex-

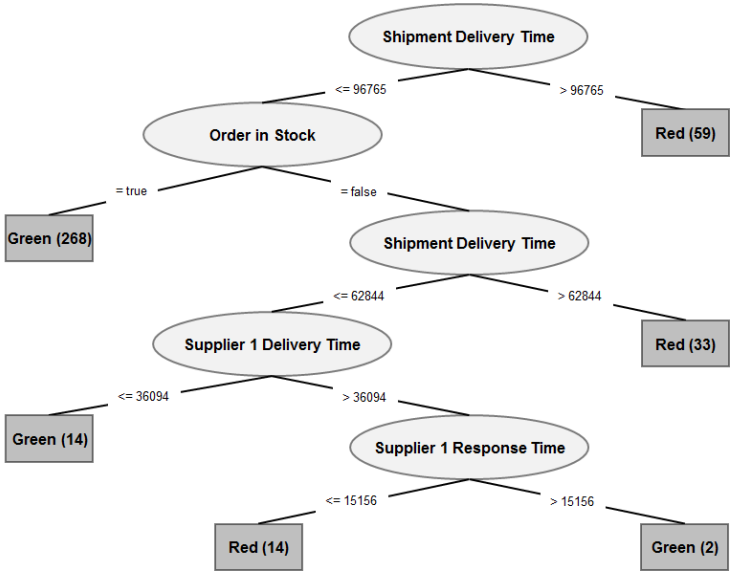

Figure 6.4: Generated Dependency Tree for *Order Fulfillment Time*

pected influential factors that have been configured, one can see that two of the three expected factors are shown, namely the shipment delivery time and the supplier 1 delivery time. The third factor that was expected, the product type, is however missing. Instead, order in stock has been chosen by the decision tree algorithm. This is because the unavailability of the product type directly influences order in stock. Both properties are correlated and influence the KPI value in the same way, and thus only one of them is shown in the tree. This result can be seen as unsatisfactory, as it does not show the root cause, namely a specific product type in this case. Note also that an additional factor has been shown in the tree, which has not been expected, the supplier 1 response time. Displaying of undesirable factors is discussed further below.

In order to deal with a factor shown in the tree "hiding" other factors, because it is coincidentally correlated with those factors, the user who performs the analysis can follow two approaches: (i) he can simply *remove* the factor (in this case order in stock) from the potential influential factor set and repeat the

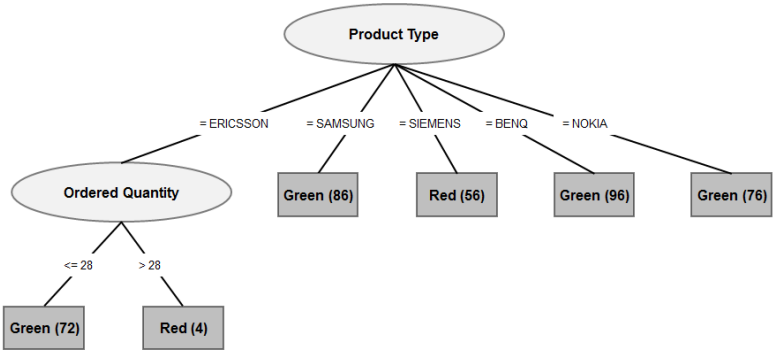

Figure 6.5: Generated Tree for *Order in Stock*

analysis; obviously in that case other factors will be chosen by the algorithm to classify the instances instead; (ii) he can *drill-down* and use the factor as target variable and repeat the analysis. In the second case, the order in stock property would be chosen as KPI property (with the KPI class green in case of order in stock=true) and another tree would be generated which explains when ordered products are not in stock. Such a tree is shown in Figure 6.5. This tree now clearly explains how the availability of products in stock depends on product type and ordered quantity.

Several more experiments have been performed. Table 6.1 summarizes the results.

Different Algorithms. The experiments were performed using two different algorithms from the WEKA suite, J48 and ADTree. Both algorithms were used for generating trees based on different numbers of instances (100, 400, 1000). The experiment results show that for the same number of instances, the ADTree algorithm produced bigger trees than J48 (third column: number of leaves and nodes). However, at the same time, it also reached a higher accuracy (last column: correctly classified instances). The column "Distinct Factors" shows how many distinct influential factors are displayed in the tree. Concerning both the number of distinct factors and also the concrete chosen factors, both algorithms have shown similar results.

Instances	Algorithm	Leaves/ Nodes	Distinct Factors	Correctly Classified
100	J48	4/7	4	95,0 %
100	ADTree	11/16	4	98,0 %
400	J48	6/11	4	97,8 %
400	ADTree	17/26	5	99,0 %
1000	J48	11/18	6	98,8 %
1000	J48 -R	6/11	4	97,9 %
1000	J48 -U	13/22	9	99,2 %
1000	ADTree	19/28	6	99,4 %

Table 6.1: Experiment Results: KPI Dependency Analysis

Tree Size. The experiments have further shown that the generated trees are getting bigger with the number of instances. For example, for 100 instances J48 generated a tree with 7 nodes, for 1000 instances however a tree with 18 nodes. The accuracy of the bigger tree improved thereby only by 1%. Bigger trees are less readable and contain often several factors of only marginal influence. For reducing the tree sizes, experiments were performed with several algorithm parameters. The table shows the results of using unpruned trees (maximizing accuracy; -U) and reduced error pruning for J48 (-R). Thereby, reduced error pruning (J48 -R) was shown to be effective for reducing the tree size while not losing too much of accuracy. For example, for 1000 instances reduced error pruning reduced the size of the tree by half, sacrificing only 1% of accuracy. In general, however, the usage of parameters has lead to only marginal changes in the experiments. In case of too many undesirable (marginal) factors shown in the tree, an option is to simply remove those factors from the analyzed factor set and repeat the analysis. This assumes that the user has a certain domain knowledge being able to decide which factors are marginal.

Learning Performance. The performance of the learning of a tree is about 30 seconds for 1000 instances (on a standard laptop computer). As learning can

be done in the background, it does not affect the instance execution.

Accuracy of the Learned Classification Model. The quality of the trained tree as a classification model can be assessed in terms of its *accuracy*, which is the percentage of correctly classified instances from a test set. This metric is provided by the decision tree algorithm after validation of the model (cross-validation) [WF05]. As already discussed, ADTree generates trees with slightly higher accuracy than J48 resulting in bigger trees. For explanation purposes, it is more preferable to have readable trees than to maximize accuracy. When the tree is used automatically for prediction, then a higher accuracy would be preferable instead.

Conclusions. The experiments have demonstrated that the generated dependency trees show the expected influential factors in a satisfactory manner and produce suitable results with default settings without requiring the user to adjust settings manually. Two issues have been identified which may arise and require the user to adjust the analysis settings in order to improve the analysis result: (i) the tree might "hide" an influential factor, if it affects the KPI in the same way as some other factor; in that case removal or drill-down might help; (ii) the tree grows with the number of analyzed instances making it less readable for explanation purposes; in that case one can try to use reduced error pruning (reducing accuracy) or remove marginal factors from the potential influential factor set manually. Both cases require the user to have domain-knowledge in order to either suspect the possible dependencies or the marginal factors, respectively.

6.2.2 Experimental Evaluation of Prediction and Adaptation

In the second step, runtime prediction and adaptation are evaluated. The evaluation results presented in the following have already been described in [WZK+12].

An adaptation model with order fulfillment time as KPI and two checkpoints is defined. The first one is triggered after completion of the *Check Stock* activity thus allowing still to select both the supplier and shipper. The second one

is placed right before the shipment thus allowing to still select the concrete shipment service.

As service substitution is supported as adaptation action, overall 30 service candidates for the shipper and supplier services with different QoS characteristics are created. The QoS characteristics are time, cost, and reliability. While cost and reliability are defined as constant values, execution time can vary for each execution of the service, in particular because the services are configured to implement some random behavior to some extent. The adaptation model defines how these services affect the influential factors at each checkpoint. A configuration for these services is created to simulate the behavior of the specified QoS characteristics, however with small deviations dependent on certain influential factors (e.g., duration is made dependent on factors such as product types and amounts, and random behavior).

Before the self-adaptation can take place, in a bootstrapping phase, a set of process instances has to be executed first in order to create dependency trees for each checkpoint. In this phase, for each process instance the concrete supplier and shipper services are randomly selected in order to ensure that historical data used for learning contains QoS data on each of these services and on most of their combinations. The execution of 500 process instances is triggered using a test client. After process execution, for each ckeck point a dependency tree is learned using the J48 algorithm. For the first checkpoint the decision tree has an accuracy of 88.2%, for the second one the accuracy is 94.7%.

Two different constraints and preferences models are created, one preferring lower cost, the other lower duration. The preferences are specified in respect to the KPI, and the three QoS characteristics (time, cost, reliability). Table 6.2 shows the weights specified for the two preferences models (second column). The KPI is specified to have three KPI classes (green, yellow, red). In addition, a constraint is specified that the KPI class red is to be prevented.

The experiment is now performed as follows. One triggers the execution again using a test client. For each of the two constraints and preferences models three experimental runs are performed, with 200 instances per run; the first run is performed with the default configuration (optimal according to

the preferences model) without using the adaptation framework; the second run performs the adaptation at the first checkpoint only, the third run at the second checkpoint only.

	Prediction and Adaptation (200 instances per run)				
Check Point	Weights KPI/time/cost/rel.	No Need pred./meas.	Too Late pred./meas.	Adapt. Need pred./success	KPI Evaluation
None	0.2/0.1/0.5/0.2	N/A	N/A	N/A	110/26/64
	0.2/0.5/0.1/0.2	N/A	N/A	N/A	148/31/21
Wareh.	0.2/0.1/0.5/0.2	102/63	0/0	98/88	119/32/49
	0.2/0.5/0.1/0.2	108/105	0/0	92/90	183/12/5
Shipm.	0.2/0.1/0.5/0.2	85/85	6/6	109/92	157/20/23
	0.2/0.5/0.1/0.2	105/103	5/5	90/88	180/11/9

Table 6.2: Experiment Results: Prediction and Adaptation

Table 6.2 shows the experiment results. Each row of the table depicts one particular run of 200 process instances. The columns "No Need", "Too Late", and "Adaptation Need" show what is predicted at a checkpoint (first value) and whether the prediction has been correct (second value) when measured at the end of the process instance ("measured"). This is done for the three cases "No Need" (predicted KPI class is green or yellow), "Too Late" (predicted KPI class is red), and "Adaptation Need" (instance tree has more than one leaf). For example, the value of "102/63" in the "No Need" column means that for 102 process instances the KPI class green or yellow was predicted at the checkpoint, however at the end only 63 instances reached those predicted classes, the other ones in this case led to the KPI class red. The last column "KPI Evaluation" depicts the overall result for the KPI classes (green, yellow, red) of the 200 instances in each run.

Duration and Cost of Adaptation. The prediction and adaptation time together have been measured to be below a second. Thus, only in case of very short running processes, this should be taken into account. In case of usage of adaptation mechanisms which take longer to be enacted, the adaptation time and potentially other factors reflecting the cost of adaptation could also be modeled as influential factors and be given a weight in the preferences model.

Then, they would be taken into account during selection of an adaptation strategy.

Adaptation Effectiveness. The results of the KPI evaluation (column "KPI Evaluation") show that the KPI performance has been considerably improved by using the adaptation framework (runs 2 and 3 using adaptation at checkpoints outperform the first one without checkpoints). For example, for the first preference model the number of KPI violations (KPI class = red) has been reduced from 64 to 49 and 23, respectively.

The effectiveness of preventing KPI violations obviously depends strongly on the preferences model. In case of preference on service execution time rather than cost, faster services are selected and the overall process duration is decreased. For example, at the warehouse checkpoint in case of preference on time only 5 KPI violations rather than 49 have been reached (last number in column "KPI Evaluation"). The reason that there are still violations, is that the services do not behave exactly as specified in the impact model. They have been configured to deviate in a certain range.

The experiments show further that the prediction accuracy is the higher the later the checkpoint is executed. For example, at the shipment checkpoint when no adaptation was needed (column "No Need") the prediction accuracy was 100% (85/85) for preference on cost and 98% for preference on service execution time. However, the later the adaptation takes place, the higher the risk that it is too late to adapt, which was the case for several instances at the shipment checkpoint (column "Too Late"). Of course, for even better effectiveness, one could predict and adapt at both checkpoints for each process instance.

6.3 Summary and Conclusions

This chapter has presented a prototypical implementation and an experimental evaluation of the approach. The prototype is based on the Apache ODE BPEL engine and uses the Pluggable Framework and the Muse framework for implementing the monitoring framework. The dependency analysis uses

decision tree algorithms from the WEKA suite to implement the dependency tree analysis. Finally, the adaptation framework algorithms are implemented in Java. Based on this prototype, the purchase order scenario has been implemented. Thereby, the reseller process has been implemented as a BPEL process, while other choreography participants have been implemented as Java mockup services with configurable behavior needed for experiments.

The experimental evaluation of the analysis approach has shown that the generated dependency trees explain the influential factors in a satisfactory manner out-of-the-box. It has been sketched how the user can perform drill-down analysis and how the size of trees can be decreased, if necessary. The evaluation of the adaptation approach has shown that KPI performance is improved and that effectiveness in particular depends on the placement of checkpoints, the weights set in the preferences model, and the conformance of effects specified in the adaptation model to the actual measured values.

CHAPTER 7

CONCLUSIONS AND OUTLOOK

In this thesis an integrated approach to monitoring, analysis, and adaptation of processes based on KPIs was presented. In the following the main contributions of the thesis are summarized and possible extensions for future work are discussed.

In Chapter 1, the application area and the motivation of the thesis were introduced. This thesis focused on business processes implemented as service orchestrations running in service choreographies. One important aspect of such processes is management of their performance in terms of KPIs. Management thereby includes three aspects. Firstly, KPIs have to be monitored in service choreographies. Secondly, if monitoring shows that KPI performance is not satisfactory, there is a need for analyzing and understanding the influential factors. Thirdly, after analyzing how the KPIs depend on a set of influential factors, the process is to be proactively adapted in an automated fashion in order to improve its performance. The goal of the thesis was thus to provide an automated approach to monitoring, analyzing, and adapting processes in service choreographies with the goal to improve their KPI performance.

After motivating the work, in Chapter 2, related BPM and SOA concepts and technologies were presented. These include orchestrations and choreographies

and corresponding languages used in the thesis, namely BPEL and BPEL4Chor, and Web services technologies such as WSDM. In the second part of the chapter related research approaches were discussed and compared to the approach of the thesis.

As the first contribution of the thesis, Chapter 3 presented a monitoring approach for monitoring of processes in service choreographies. The presented monitoring metamodel allows defining monitor models in terms of a set of manageable resources and their properties. Resources and their properties can be accessed over WSDL-based resource endpoints. The supported resource types are derived from the runtime instance types of a choreography. A monitor model is created by defining a set of resource endpoints each assigned to a resource type and specifying a set of concrete resource instances which should be exposed, e.g., which concrete activities of the purchase order process should be monitored. In addition to basic properties such as activity state or variable value, a resource can expose custom properties, which are calculated based on events using an event processing language. These custom properties are needed to evaluate the performance characteristics of the processes and are used for the evaluation of KPIs. The monitoring approach is the basis of the overall approach as it allows to monitor KPIs of the processes in choreographies.

The second contribution, presented in Chapter 4, builds on the monitoring framework and extends it in order to support the analysis of process performance in terms of KPIs and their dependencies on influential factors. The KPI dependency analysis is mapped to a classification problem and decision tree algorithms are used for classification learning. A KPI analysis model is defined on top of a monitor model specifying a set of KPIs, potential influential factors and analysis tasks. Within the classification learning, a KPI represents the target attribute and is defined based on a resource property specified in the monitor model and a set of KPI classes; influential factors are the explanatory attributes. For each defined analysis task, a KPI dependency tree is learned. It explains to the user how the KPI depends on the selected potential influential factors. The created KPI dependency trees can also be used for prediction, which is the basis for runtime adaptation.

The third contribution extends the monitoring and analysis framework in

order to enable proactive runtime adaptation of processes with the goal to improve the KPIs. Chapter 5 described an adaptation metamodel which enables specifying a set of adaptation subjects with corresponding characteristics and adaptation alternatives, a set of checkpoints, and a set of constraints and preferences. Each adaptation alternative specifies how it affects the characteristics of the corresponding adaptation subject. This information is then used at runtime to pause the process execution at specified checkpoints, perform a KPI prediction based on KPI dependency trees, and select the most appropriate alternatives for each adaptation subject according to the specified constraints and preferences. The algorithm to select the adaptation alternatives uses dependency trees as input which are learned for each checkpoint and each KPI.

Chapter 6 presented the prototypical realization and scenario-based evaluation of the approach. In order to demonstrate the realizability of the proposed concepts, a prototype has been developed. It is based on an existing BPEL engine, a CEP framework, a WSDM framework, and a data mining framework. Based on those components the monitoring, analysis and adaptation framework has been implemented. The purchase order scenario has been implemented based on BPEL and has been used for experiment-based evaluation of the approach.

7.1 Outlook

The approach presented in the thesis can be extended in several ways in future work.

The monitoring approach of the thesis supports monitoring in choreographies across participants of different organizations. It assumes that these participants have agreed on the choreography model and then have derived the WSDM-based monitoring interfaces which each participant has to provide. This approach could be extended towards SLAs supporting the creation of monitoring agreements containing SLAs between participants in choreographies. Using WS-Agreement terminology [Ope07], a service provider could provide an agreement template defining service level objectives based on resource properties specified in its WSDM-based monitoring interface. He could also define several

agreement templates which offer different levels of monitoring and management functionality as a service. Such an SLA specified for a choreography could not only model the monitoring interface the service provider provides, but also the monitoring interface he requests from the service requester, e.g., to calculate composite properties across processes of different participants which are the basis of service level objectives.

The analysis approach of the thesis deals with analyzing how a KPI depends on a set of influential factors. Firstly, one should note that the approach is not constrained to be used only for the typical KPI dimensions, i.e., time, quality, and cost. But the approach could also be used to analyze indicators from other areas such as flexibility, sustainability, and compliance. Secondly, one important aspect of the approach is the modeling of potential influential factors used for analysis. Manual modeling is cumbersome and time-consuming, thus several rules have been presented how influential factors can be generated based on the process model. This approach could be refined and extended for supporting the generation of an appropriate set of potential influential factors for different types of KPIs and different analysis questions. For example, the set of potential influential factors is different for a KPI specified only for one process model (e.g., process duration) and a KPI specified across several process models (e.g., overall choreography duration), and even other granularities are possible. Also, it should be possible to pose different analysis questions, such as analysis of process duration in respect to process input data only or analysis in respect to service infrastructure metrics. In both cases, different sets of potential influential factors should be used.

The adaptation approach relies on existing adaptation mechanisms. In addition to already supported mechanisms, more sophisticated adaptation mechanisms could be integrated such as splitting of processes or process fragment substitution (e.g., [KKL07b, LWK+10b]). Another challenge is how to express the effects of the corresponding adaptation actions on influential factors. In the thesis, predicates with concrete values have been specified on influential factor property values. In some cases such a predicate might be unknown, so one could specify effects which simply state that an adaptation action improves or deteriorates a certain property. Accordingly, the selection algorithms would

have to be adapted.

Service-based applications can be divided into several logical layers, a choreography layer, an orchestration layer, a service layer, and a service infrastructure layer. In all of those layers, there are different types of resources and properties which can be monitored, analyzed, and adapted. Also, properties in one layer often influence properties in other layers. For example, the unavailability of the service infrastructure can have severe impact on the overall process duration in the process layer. This thesis focused on the service choreography layer and the service orchestration layer. The approach could be extended to take into account also the service layer and the service infrastructure layer. One possible approach in that direction has been presented in [GKMW11]. This requires integrating additional monitoring and adaptation mechanisms. For example, in the service infrastructure layer system monitoring and management mechanisms could be used. If business processes run in the cloud, then cloud infrastructure management can be integrated. In those cases, additional influential factors can be monitored and additional adaptation actions are available (e.g., provisioning of new resources in the cloud). In addition to simply adding those monitoring and adaptation mechanisms, one has to correlate information between the layers.

On top of single applications one can also look at service networks. [WDL+08] has discussed that KPIs specified in the service network layer result in events in the choreography and orchestration layer. One challenge would be in providing methods to derive monitor contracts in service networks in a model-driven manner. If a service network is mapped to a service choreography between participants which is again refined to executable service orchestrations, then in the same way KPIs and SLAs between participants specified in the service network layer could be mapped to monitoring contracts in the service choreography layer and further refined to monitor models for service orchestrations.

Acknowledgements

This thesis would not have been possible without the support of several people who I want to thank.

First and foremost I want to thank my supervisor Frank Leymann for giving me the opportunity to work in his research group, for his advice, patience, and continuous support. I also want to say a big thank you to Dimka Karastoyanova for her constant support, many valuable discussions and suggestions, encouragement, and guidance during the entire process.

I want to thank my colleagues at the IAAS for many helpful discussions on technical topics but also for the great times we spent together outside the institute. I especially want to thank my office mates Zhilei Ma and Alexander Nowak for the pleasant time and support in all the tasks we had to fulfill. Special thanks go to Alexander Nowak and Steve Strauch for reviewing the thesis and giving me valuable comments. I also thank Oliver Kopp who patiently answered all my questions on choreography topics. Thanks also go to various other colleagues at IAAS, most notably Olha Danylevych, Tammo van Lessen, Daniel Martin, Joerg Nitzsche, Daniel Schleicher, David Schumm, Sebastian Wagner, and Daniel Wutke, with whom I worked in EU projects.

During my work on EU projects I collaborated with many people of other research institutions who gave me valuable insight and ideas. Special thanks go to Philipp Leitner from the Technical University of Vienna for our collaboration

on machine learning topics and Raman Kazhamiakin from the FBK Trento for our collaboration on adaptation topics.

Last but not least I want to thank my family for their continuous motivation and support.

BIBLIOGRAPHY

[ACM+07] Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Barbara Pernici, and Pierluigi Plebani. PAWS: A Framework for Executing Adaptive Web-Service Processes. *IEEE Software*, 24(6):39–46, 2007.

[Apaa] Apache Software Foundation. Apache ActiveMQ 5.2. http://activemq.apache.org/.

[Apab] Apache Software Foundation. Apache Muse 2.2.0. http://attic.apache.org/projects/muse.html.

[Apac] Apache Software Foundation. Apache ODE 1.3.4. http://ode.apache.org/.

[AZ12] Rafael R. Aschoff and Andrea Zisman. Proactive Adaptation of Service Composition. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '12)*, pages 1–10. IEEE Computer Society, 2012.

[BEMP07] Catriel Beeri, Anat Eyal, Tova Milo, and Alon Pilberg. Monitoring Business Processes With Queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 603–614. VLDB Endowment, 2007.

[BFPG12] Aymen Baouab, Walid Fdhila, Olivier Perrin, and Claude Godart. Towards Decentralized Monitoring of Supply Chains. In

Proceedings of the 19th International Conference on Web Services (ICWS '12), pages 600–607. IEEE Computer Society, 2012.

[BG05] Luciano Baresi and Sam Guinea. Towards Dynamic Monitoring of WS-BPEL Processes. In *Proceedings of the 3rd International Conference of Service-Oriented Computing (ICSOC '05)*, pages 269–282. Springer, 2005.

[BG13] Luciano Baresi and Sam Guinea. Event-Based Multi-level Service Monitoring. In *Proceedings of the 20th International Conference on Web Services (ICWS '13)*, pages 83–90. IEEE Computer Society, 2013.

[BGNS10] Luciano Baresi, Sam Guinea, Olivier Nano, and George Spanoudakis. Comprehensive Monitoring of BPEL Processes. *IEEE Internet Computing*, 14(3):50–57, 2010.

[BGPT09] Luciano Baresi, Sam Guinea, Marco Pistore, and Michele Trainotti. Dynamo + Astro: An Integrated Approach for BPEL Monitoring. In *Proceedings of the 16th International Conference on Web Services (ICWS '09)*, pages 230–237. IEEE Computer Society, 2009.

[BTPT06] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *Proceedings of the 13th International Conference on Web Services (ICWS '06)*, pages 63–71. IEEE Computer Society, 2006.

[CBC+06] Pawan Chowdhary, Kumar Bhaskaran, Nathan Caswell, Henry Chang, Tian Chao, Shyh-Kwei Chen, Michael Dikun, Hui Lei, Jun-Jang Jeng, Shubir Kapoor, Christian Lang, George Mihaila, Ioana Stanoi, and Liangzhao Zeng. Model driven development for business performance management. *IBM System Journal, Vol. 45, No. 3*, 2006.

[CCDS04] Malú Castellanos, Fabio Casati, Umeshwar Dayal, and Ming-Chien Shan. A Comprehensive and Automated Approach to Intelligent Business Processes Execution Analysis. *Distributed and Parallel Databases*, 16(3):239–273, 2004.

[CCDS07] Fabio Casati, Malu Castellanos, Umeshwar Dayal, and Norman Salazar. A Generic Solution for Warehousing Business Process Data. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1128–1137. VLDB Endowment, 2007.

[CCSD05] Malú Castellanos, Fabio Casati, Ming-Chien Shan, and Umeshwar Dayal. iBOM: A Platform for Intelligent Business Operation Management. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 1084–1095. IEEE Computer Society, 2005.

[CDPEV08] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. A Framework for QoS-aware Binding and Re-binding of Composite Web Services. *Journal of Systems and Software*, 81(10):1754–1769, 2008.

[Cha04] David A. Chappell. *Enterprise Service Bus. Theory in Practice.* O'Reilly Media, 2004.

[CVG12] Marco Comuzzi, Jochem Vonk, and Paul Grefen. Measures and Mechanisms for Process Monitoring in Evolving Business Networks. *Data and Knowledge Engineering*, 71(1):1–28, 2012.

[DCS09] Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Amoeba: A Methodology for Modeling and Evolving Cross-organizational Business Processes. *ACM Transactions on Software Engineering and Methodology*, 19:6:1–6:45, 2009.

[DD04] Remco Dijkman and Marlon Dumas. Service-oriented Design: A Multi-viewpoint Approach. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.

[dGAD14] Silvana de Gyvés Avila and Karim Djemame. Proactive Adaptation in Service Composition using a Fuzzy Logic Based Optimization Mechanism. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER '14)*, pages 257–267. SciTePress, 2014.

[DKLW07] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of the 14th International Conference on Web Services (ICWS '07)*, pages 296–303. IEEE Computer Society, 2007.

[DKLW09] Gero Decker, Oliver Kopp, Frank Leymann, and Mathias Weske. Interacting Services: From Specification to Execution. *Data & Knowledge Engineering*, 68(10):946 – 972, 2009.

[dLvdAD14] Massimiliano de Leoni, Wil van der Aalst, and Marcus Dees. A General Framework for Correlating Business Process Characteristics. In *Proceedings of the 12th International Conference on Business Process Management (BPM '14)*, pages 250–266. Springer, 2014.

[EM08] Abdelkarim Erradi and Piyush Maheshwari. Dynamic Binding Framework for Adaptive Web Services. In *Proceedings of the 3rd International Conference on Internet and Web Applications and Services (ICIW '08)*, pages 162–167. IEEE Computer Society, 2008.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[Esp] EsperTech. ESPER 3.2. http://www.espertech.com/esper/.

[FBS05] Xiang Fu, Tevfik Bultan, and Jianwen Su. Realizability of Conversation Protocols with Message Contents. *International Journal of Web Services Research*, 2(4):68–93, 2005.

[FJMM12] Jan-Philipp Friedenstab, Christian Janiesch, Martin Matzner, and Oliver Muller. Extending BPMN for Business Activity Monitoring.

In *Proceedings of the 45th Hawaii International Conference on System Sciences (HICSS '12)*, pages 4158–4167. IEEE Computer Society, 2012.

[FM99] Yoav Freund and Llew Mason. The Alternating Decision Tree Learning Algorithm. In *Proceedings of the 16th International Conference on Machine Learning (ICML '99)*, pages 124–133. Morgan Kaufmann Publishers Inc., 1999.

[GKMW11] Sam Guinea, Gabor Kecskemeti, Annapaola Marconi, and Branimir Wetzstein. Multi-layered Monitoring and Adaptation. In *Proceedings of the 9th International Conference on Service Oriented Computing (ICSOC '11)*, pages 359–373. Springer, 2011.

[HFH+09] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11:10–18, 2009.

[HKMP08] Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A Framework for Proactive Self-adaptation of Service-Based Applications Based on Online Testing. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet (ServiceWave '08)*, pages 122–133. Springer, 2008.

[HY81] Ching-Lai Hwang and Kwangsun Yoon. *Multiple Attribute Decision Making: Methods and Applications*. Springer, 1981.

[IBM15] IBM Knowledge Center. IBM Business Monitor 8.5.5 documentation, 2015. http://www-01.ibm.com/support/knowledgecenter/SS7NQD_8.5.5.

[Inm02] William H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, Inc., 3rd edition, 2002.

[Ins] Institute of Architecture of Application Systems (IAAS). Pluggable Framework Extension for ODE (ODE-PGF). http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/.

[Joh93] John R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan-Kaufmann, 1993.

[KEvL+11] Oliver Kopp, Lasse Engler, Tammo van Lessen, Frank Leymann, and Jörg Nitzsche. Interaction Choreography Models in BPEL: Choreographies on the Enterprise Service Bus. In *Proceedings of the 2nd International Conference on Subject-Orientation as Enabler for the Next Generation of BPM Tools and Methods (S-BPM ONE '10)*, pages 36–53. Springer, 2011.

[KHK+11] Oliver Kopp, Sebastian Henke, Dimka Karastoyanova, Rania Khalaf, Frank Leymann, Mirko Sonntag, Thomas Steinmetz, Tobias Unger, and Branimir Wetzstein. An Event Model for WS-BPEL 2.0. Technical Report 2011/07, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2011.

[KK15] Falko Koetter and Monika Kochanowski. A Model-driven Approach for Event-based Business Process Monitoring. *Information Systems and e-Business Management*, 13(1):5–36, 2015.

[KKL07a] Rania Khalaf, Dimka Karastoyanova, and Frank Leymann. Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. In *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA '07)*, pages 376–387. Springer, 2007.

[KKL07b] Rania Khalaf, Oliver Kopp, and Frank Leymann. Maintaining Data Dependencies Across BPEL Process Fragments. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC '07)*, pages 207–219. Springer, 2007.

[KL03] Rania Khalaf and Frank Leymann. On Web Services Aggregation. In *Proceedings of the 4th International Workshop on Technologies for E-Services (TES '03)*, pages 1–13. Springer, 2003.

[KL09] Dimka Karastoyanova and Frank Leymann. BPEL'n'Aspects: Adapting Service Orchestration Logic. In *Proceedings of the 7th International Conference on Web Services (ICWS '09)*, pages 222–229. IEEE Computer Society, 2009.

[KLN+06] Dimka Karastoyanova, Frank Leymann, Joerg Nitzsche, Branimir Wetzstein, and Daniel Wutke. Parameterized BPEL Processes: Concepts and Implementation. In *Proceedings of the International Conference Business Process Management (BPM '06)*, pages 471–476. Springer, 2006.

[KN97] Robert S. Kaplan and David P. Norton. *Balanced Scorecard – Strategien erfolgreich umsetzen*. Schäffer-Poeschel Stuttgart, 1997.

[KSK07] Shinji Kikuchi, Hisashi Shimamura, and Yoshihiro Kanna. Monitoring Method of Cross-Sites' Processes Executed by Multiple WS-BPEL Processors. In *Proceedings of the 9th IEEE International Conference on nterprise Computing, E-Commerce, and E-Services (CEC/EEE '07)*, pages 55–64. IEEE Computer Society, 2007.

[KWK+09] Raman Kazhamiakin, Branimir Wetzstein, Dimka Karastoyanova, Marco Pistore, and Frank Leymann. Adaptation of Service-based Applications Based on Process Quality Factor Analysis. In *Proceedings of the 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+ '09)*, pages 395–404. Springer, 2009.

[KWL01] Peter Kueng, Thomas Wettstein, and Beate List. A Holistic Process Performance Analysis through a Performance Data Warehouse. In *Proceedings of the American Conference on Information Systems (AMCIS '01)*, pages 349–356, 2001.

[Ley05] Frank Leymann. The (Service) Bus: Services Penetrate Everyday Life. In *Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC '05)*, pages 12–20. Springer, 2005.

[LHD13] Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Cost-Based Optimization of Service Compositions. *IEEE Transactions on Services Computing*, 6(2):239–251, 2013.

[LKS+10] Geetika T. Lakshmanan, Paul Keyser, Aleksander Slominski, Francisco Curbera, and Rania Khalaf. A Business Centric End-to-End Monitoring Approach for Service Composites. In *Proceedings of the 7th International Conference on Services Computing (SCC '10)*, pages 409–416. IEEE Computer Society, 2010.

[LM04] Beate List and Karl Machaczek. Towards a Corporate Performance Measurement System. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC '04)*, pages 1344–1350. ACM, 2004.

[LMRD10] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, Prediction and Prevention of SLA Violations in Composite Services. In *Proceedings of the 17th International Conference on Web Services (ICWS '10)*, pages 369–376. IEEE Computer Society, 2010.

[LR97] Frank Leymann and Dieter Roller. Workflow-based Applications. *IBM Systems Journal*, 36:102–123, 1997.

[LR00] Frank Leymann and Dieter Roller. *Production Workflow – Concepts and Techniques*. Prentice Hall, 2000.

[LS08] David Luckham and Roy Schulte. Event Processing Glossary - Version 1.1, 2008. http://www.complexevents.com/2008/08/31/event-processing-glossary-version-11.

[Luc02] David Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, 2002.

[LWK+10a] Philipp Leitner, Branimir Wetzstein, Dimka Karastoyanova, Waldemar Hummer, Schahram Dustdar, and Frank Leymann. Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution. In *Proceedings of the 8th International Conference on Service-Oriented Computing (ICSOC '10)*, pages 365–380. Springer, 2010.

[LWK+10b] Philipp Leitner, Branimir Wetzstein, Dimka Karastoyanova, Waldemar Hummer, Schahram Dustdar, and Frank Leymann. Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution. In *Proceedings of the 8th International Conference on Service-Oriented Computing (ICSOC '10)*, pages 365–380. Springer, 2010.

[LWR+09] Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, and Frank Leymann. Runtime Prediction of Service Level Agreement Violations for Composite Services. In *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing (NFPSLAM-SOC '09)*, pages 176–186. Springer, 2009.

[McC02] David McCoy. Business Activity Monitoring: Calm before the Storm. Technical Report LE-15-9727, Gartner, 2002.

[Met11] Andreas Metzger. Towards Accurate Failure Prediction for the Proactive Adaptation of Service-oriented Systems. In *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems (ASAS '11)*, pages 18–23. ACM, 2011.

[MGA09] Christof Momm, Michael Gebhart, and Sebastian Abeck. A Model-Driven Approach for Monitoring Business Performance in Web Service Compositions. In *Proceedings of the 4th International*

Conference on Internet and Web Applications and Services (ICIW '09), pages 343–350. IEEE Computer Society, 2009.

[MMGF06] Axel Martens, Simon Moser, Achim Gerhardt, and Karoline Funk. Analyzing Compatibility of BPEL Processes. In *Proceedings of the Advanced International Conference on Internet and Web Applications and Services (AICT-ICIW '06)*. IEEE Computer Society, 2006.

[MRD10] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Event Driven Monitoring for Service Composition Infrastructures. In *Proceedings of the 11th International Conference on Web Information Systems Engineering (WISE '10)*, pages 38–51. Springer, 2010.

[MZD13] Emmanuel Mulo, Uwe Zdun, and Schahram Dustdar. Domain-specific Language for Event-based Compliance Monitoring in Process-driven SOAs. *Service Oriented Computing and Applications*, 7(1):59–73, 2013.

[NLS11] Alexander Nowak, Frank Leymann, and David Schumm. The Differences and Commonalities between Green and Conventional Business Process Management. In *Proceedings of the International Conference on Cloud and Green Computing (CGC '11)*, pages 569–576. IEEE Computer Society, 2011.

[OAS06a] OASIS. *Web Services Base Notification 1.3 (WS-BaseNotification) – OASIS Standard*, 2006.

[OAS06b] OASIS. *Web Services Distribued Management V1.1 (WSDM) – OASIS Standard*, 2006.

[OAS06c] OASIS. *Web Services Distributed Management: Management of Web Services 1.1 (MOWS) – OASIS Standard*, 2006.

[OAS06d] OASIS. *Web Services Distributed Management: Management Using Web Services 1.1 (MUWS) – OASIS Standard*, 2006.

[OAS06e] OASIS. *Web Services Resource Properties 1.2 (WS-ResourceProperties) – OASIS Standard*, 2006.

[OAS06f] OASIS. *Web Services Topics 1.3 (WS-Topics) – OASIS Standard*, 2006.

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0 (WS–BPEL) – OASIS Standard*, 2007.

[OMG11] OMG. Business Process Model and Notation (BPMN), Specification, Version 2.0, 2011.

[Ope07] Open Grid Forum. *Web Services Agreement Specification (WS-Agreement)*, 2007.

[Pap08] Michael P. Papazoglou. *Web Services - Principles and Technology*. Prentice Hall, 2008.

[Pau09] Paul Grefen and Rik Eshuis and Nikolay Mehandjiev and Giorgos Kouvas and Georg Weichhart. Internet-Based Support for Process-Oriented Instant Virtual Enterprises. *Internet Computing, IEEE*, 13(6):65–73, 2009.

[Pel03] Chris Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, 2003.

[PLW+08] Carlos Pedrinaci, Dave Lambert, Branimir Wetzstein, Tammo van Lessen, Luchesar Cekov, and Marin Dimitrov. SENTINEL: A Semantic Business Process Monitoring Tool. In *Proceedings of the First International Workshop on Ontology-supported Business Intelligence (OBI '08)*, pages 1–12. ACM, 2008.

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38–45, 2007.

[Ros] RosettaNet. RosettaNet Partner Interface Processes (PIPs). http://www.rosettanet.org/.

[RSN15] Sylvia Radeschütz, Holger Schwarz, and Florian Niedermann. Business Impact Analysis – a Framework for a Comprehensive Analysis and Optimization of Business Processes. *Computer Science - Research and Development*, 30(1):69–86, 2015.

[RSS06] Heinz Roth, Josef Schiefer, and Alexander Schatten. Probing and Monitoring of WSBPEL Processes with Web Services. In *Proceedings of the 8th International Conference on E-Commerce Technology (CEC-EEE '06)*, 2006.

[SMF+11] Osama Sammodi, Andreas Metzger, Xavier Franch, Marc Oriol, Jordi Marco, and Klaus Pohl. Usage-Based Online Testing for Proactive Adaptation of Service-Based Applications. In *Proceedings of the 35th Annual International Computer Software and Applications Conference (COMPSAC '11)*, pages 582–587. IEEE Computer Society, 2011.

[SS06] Hermann J. Schmelzer and Wolfgang Sesselmann. *Geschäftsprozessmanagement in der Praxis*. Hanser Verlag München, 2006.

[STA05] August-Wilhelm Scheer, Oliver Thomas, and Otmar Adam. *Process Aware Information Systems: Bridging People and Software Through Process Technology*, chapter Process Modeling Using Event-Driven Process Chains. Wiley-Interscience, 2005.

[Ste08] Thomas Steinmetz. Ein Event-Modell für WS-BPEL 2.0 und dessen Realisierung in Apache ODE. Diploma Thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2008.

[Sup05] Supply Chain Council. Supply Chain Operations Reference Model Version 7.0, 2005.

[SVDS12] Thomas Schlegel, Krešimir Vidačković, Sebastian Dusch, and Ronny Seiger. Management of Interactive Business Processes

in Decentralized Service Infrastructures Through Event Processing. *Journal of King Saud University - Computer and Information Sciences*, 24(2):137–144, 2012.

[Ter] Terracotta. Quartz Scheduler 1.8.6. http://www.quartz-scheduler.org.

[vdAWM04] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[vLLM+08] Tammo van Lessen, Frank Leymann, Ralph Mietzner, Jörg Nitzsche, and Daniel Schleicher. A Management Framework for WS-BPEL. In *Proceedings of the 6th IEEE European Conference on Web Services (ECOWS '08)*, pages 187–196. IEEE Computer Society, 2008.

[vRR09] Michael von Riegen and Norbert Ritter. Reliable Monitoring for Runtime Validation of Choreographies. In *The 4th International Conference on Internet and Web Applications and Services (ICIW '09)*, pages 310–315. IEEE Computer Society, 2009.

[W3C01] W3C. *Web Services Description Language (WSDL) 1.1, W3C Note*, 2001.

[W3C05] W3C. *Web Services Choreography Description Language Version 1.0, W3C Candidate Recommendation*, 2005.

[W3C06] W3C. *Web Services Addressing 1.0 - Core – W3C Recommendation*, 2006.

[W3C07] W3C. *SOAP Version 1.2 Part 1: Messaging Framework – W3C Recommendation*, 2007.

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture:*

SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall, 2005.

[WDL+08] Branimir Wetzstein, Olha Danylevych, Frank Leymann, Marina Bitsaki, Christos Nikolaou, Willem-Jan van den Heuvel, and Mike Papazoglou. Towards Monitoring of Key Performance Indicators Across Partners in Service Networks. In *Proceedings of the Workshop on Service Monitoring, Adaptation and Beyond (MONA+ '08)*, pages 7–18. ICB, 2008.

[Wes07] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2. edition, 2005.

[WKK+10] Branimir Wetzstein, Dimka Karastoyanova, Oliver Kopp, Frank Leymann, and Daniel Zwink. Cross-Organizational Process Monitoring based on Service Choreographies. In *Proceedings of the 25th Annual ACM Symposium on Applied Computing (SAC '10)*, pages 2485–2490. ACM, 2010.

[WLR+09] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Schahram Dustdar, and Frank Leymann. Monitoring and Analyzing Influential Factors of Business Process Performance. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference (EDOC '09)*, pages 141–150. IEEE Computer Society, 2009.

[WLR+11] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Schahram Dustdar, and Frank Leymann. Identifying Influential Factors of Business Process Performance Using Dependency Analysis. *Enterprise Information Systems*, 5(1):79–98, 2011.

[WML08] Branimir Wetzstein, Zhilei Ma, and Frank Leymann. Towards Measuring Key Performance Indicators of Semantic Business Processes. In *Proceedings of 11th International Conference on Business Information Systems (BIS '08)*, pages 227–238. Springer, 2008.

[WSL09] Branimir Wetzstein, Steve Strauch, and Frank Leymann. Measuring Performance Metrics of WS-BPEL Service Compositions. In *The 5th International Conference on Networking and Services (ICNS '09)*, pages 49–56. IEEE Computer Society, 2009.

[WZK+12] Branimir Wetzstein, Asli Zengin, Raman Kazhamiakin, Annapaola Marconi, Marco Pistore, Dimka Karastoyanova, and Frank Leymann. Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Runtime Adaptation. *Journal of Systems Integration*, 3(1):3–18, 2012.

[ZBDtH06] Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In *Proceedings of the On the Move to Meaningful Internet Systems Conferences (OTM '06)*, pages 145–162. Springer, 2006.

[ZLLC08] Liangzhao Zeng, Christoph Lingenfelder, Hui Lei, and Henry Chang. Event-Driven Quality of Service Prediction. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC '08)*, pages 147–161. Springer, 2008.

[zM01] Michael zur Muehlen. Process-driven Management Information Systems - Combining Data Warehouses and Workflow Technology. In *Proceedings of the 4th International Conference on Electronic Commerce Research (ICECR '01)*, pages 550–566. IFIP, 2001.

[ZPG10] Ehtesham Zahoor, Olivier Perrin, and Claude Godart. DISC: A Declarative Framework for Self-Healing Web Services Composition. In *Proceedings of the 17th International Conference on Web Services (ICWS '10)*, pages 25–33. IEEE Computer Society, 2010.

[ZSW+10] Sonja Zaplata, Daniel Straßenburg, Benjamin Wunderlich, Dirk Bade, Kristof Hamann, and Winfried Lamersdorf. Ad-hoc Management Capabilities for Distributed Business Processes. In *Proceedings of the 3rd International Conference on Business Process and Services Computing (BPSC '10)*, pages 139–152. GI-Edition, Lecture Notes in Informatics, 2010.

The URLs have been checked for validity on March 28, 2016.

List of Figures

List of Tables

List of Listings