Title: Mechanisms and Techniques for QoS-Aware, Coordinated Service Compositions

Authors: UoC, TUW, UniHH, UPM, USTUTT, UniDuE, FBK, UCBL-UPD

Editor: Dragan Ivanović (UPM)

Reviewers: Annapaola Marconi (FBK)
            Harald Psaier (TUW)
            Dimka Krastoyanova (USTUTT)

Identifier: CD-JRA-2.2.6

Type: Contractual Deliverable

Version: 1.0

Date: 15 February 2012

Status: Final

Class: External

Management Summary

This deliverable presents the contributions to technical foundations of self-configuring, adaptive, QoS-aware service compositions. The contributions deal with monitoring and analysis of service compositions and choreographies and their QoS characteristics in the scope of cross-organisational business processes. The role of QoS characteristics of service compositions are considered in the scope of Quality Assurance and thus foster integration with other workpackages.
Members of the S-Cube consortium:

University of Duisburg-Essen (Coordinator) Germany
Tilburg University Netherlands
City University London U.K.
Consiglio Nazionale delle Ricerche Italy
Center for Scientific and Technological Research Italy
The French National Institute for Research in Computer Science and Control France
Lero - The Irish Software Engineering Research Centre Ireland
Politecnico di Milano Italy
MTA SZTAKI – Computer and Automation Research Institute Hungary
Vienna University of Technology Austria
Université Claude Bernard Lyon France
University of Crete Greece
Universidad Politécnica de Madrid Spain
University of Stuttgart Germany
University of Hamburg Germany
Vrije Universiteit Amsterdam Netherlands

Published S-Cube documents

All public S-Cube deliverables are available from the S-Cube Web Portal at the following URL:

http://www.s-cube-network.eu/results/deliverables/
The S-Cube Deliverable Series

Vision and Objectives of S-Cube

The Software Services and Systems Network (S-Cube) will establish a unified, multidisciplinary, vibrant research community which will enable Europe to lead the software-services revolution, helping shape the software-service based Internet which is the backbone of our future interactive society.

By integrating diverse research communities, S-Cube intends to achieve world-wide scientific excellence in a field that is critical for European competitiveness. S-Cube will accomplish its aims by meeting the following objectives:

Re-aligning, re-shaping and integrating research agendas of key European players from diverse research areas and by synthesizing and integrating diversified knowledge, thereby establishing a long-lasting foundation for steering research and for achieving innovation at the highest level.

Inaugurating a Europe-wide common program of education and training for researchers and industry thereby creating a common culture that will have a profound impact on the future of the field.

Establishing a pro-active mobility plan to enable cross-fertilisation and thereby fostering the integration of research communities and the establishment of a common software services research culture.

Establishing trust relationships with industry via European Technology Platforms (specifically NESSI) to achieve a catalytic effect in shaping European research, strengthening industrial competitiveness and addressing main societal challenges.

Defining a broader research vision and perspective that will shape the software-service based Internet of the future and will accelerate economic growth and improve the living conditions of European citizens.

S-Cube will produce an integrated research community of international reputation and acclaim that will help define the future shape of the field of software services which is of critical for European competitiveness. S-Cube will provide service engineering methodologies which facilitate the development, deployment and adjustment of sophisticated hybrid service-based systems that cannot be addressed with today’s limited software engineering approaches. S-Cube will further introduce an advanced training program for researchers and practitioners. Finally, S-Cube intends to bring strategic added value to European industry by using industry best-practice models and by implementing research results into pilot business cases and prototype systems.

S-Cube materials are available from URL: http://www.s-cube-network.eu/
This page intentionally left blank.
Contents

1 Deliverable Overview 8
  1.1 Introduction ................................................. 8
  1.2 Deliverable Structure ..................................... 8
  1.3 Overview of the Contributions ............................ 9
  1.4 Key Research Challenges and Results ..................... 11
  1.5 Relationship to Other Work Packages ........................ 12

2 Mechanisms and Techniques for QoS-Aware, Coordinated Service Compositions 14
  2.1 Awareness-Based Realizability Analysis Of Choreographies With Exception Handling Constructs ................................................. 14
     2.1.1 Background ................................................. 14
     2.1.2 Problem Statement ....................................... 15
     2.1.3 Contribution Relevance ................................... 15
     2.1.4 Contribution Summary .................................... 15
     2.1.5 Contribution Evaluation .................................. 15
     2.1.6 Relation to the Research Framework ..................... 15
     2.1.7 Conclusions ............................................... 15
  2.2 Automatic Attribute Inference In Complex Orchestrations Based On Sharing Analysis 16
     2.2.1 Background ................................................. 16
     2.2.2 Problem Statement ....................................... 16
     2.2.3 Contribution Relevance ................................... 16
     2.2.4 Contribution Summary .................................... 17
     2.2.5 Contribution Evaluation .................................. 17
     2.2.6 Relation to the Research Framework ..................... 17
     2.2.7 Conclusions ............................................... 17
  2.3 Towards Deriving Specifications for Composite Web Services 18
     2.3.1 Background ................................................. 18
     2.3.2 Problem Statement ....................................... 18
     2.3.3 Contribution Relevance ................................... 18
     2.3.4 Contribution Summary .................................... 19
     2.3.5 Contribution Evaluation .................................. 19
     2.3.6 Relation to the Research Framework ..................... 19
     2.3.7 Conclusions ............................................... 19
  2.4 Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitutions 20
     2.4.1 Background ................................................. 20
     2.4.2 Problem Statement ....................................... 20
     2.4.3 Contribution Relevance ................................... 20
     2.4.4 Contribution Summary .................................... 21
     2.4.5 Contribution Evaluation .................................. 21
2.4.6 Relation to the Research Framework ........................................... 21
2.4.7 Conclusions ................................................................. 21

2.5 Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Adaptation ................................................................. 22
2.5.1 Background ................................................................. 22
2.5.2 Problem Statement ......................................................... 22
2.5.3 Contribution Relevance .................................................... 22
2.5.4 Contribution Summary ..................................................... 22
2.5.5 Contribution Evaluation ................................................... 23
2.5.6 Relation to the Research Framework ....................................... 23
2.5.7 Conclusions ................................................................. 23

2.6 A Context-Aware Framework for Business Processes Evolution ............ 24
2.6.1 Background ................................................................. 24
2.6.2 Problem Statement ......................................................... 24
2.6.3 Contribution Relevance .................................................... 25
2.6.4 Contribution Summary ..................................................... 25
2.6.5 Contribution Evaluation ................................................... 25
2.6.6 Relation to the Research Framework ....................................... 25
2.6.7 Conclusions ................................................................. 25

2.7 Dynamic Composition of Pervasive Process Fragments ......................... 26
2.7.1 Background ................................................................. 26
2.7.2 Problem Statement ......................................................... 26
2.7.3 Contribution Relevance .................................................... 26
2.7.4 Contribution Summary ..................................................... 27
2.7.5 Contribution Evaluation ................................................... 27
2.7.6 Relation to the Research Framework ....................................... 27
2.7.7 Conclusions ................................................................. 28

2.8 Designing Future-Context-Aware Dynamic Applications with Structured Context Prediction ................................................................. 28
2.8.1 Background ................................................................. 28
2.8.2 Problem Statement ......................................................... 28
2.8.3 Contribution Relevance .................................................... 29
2.8.4 Contribution Summary ..................................................... 29
2.8.5 Contribution Evaluation ................................................... 29
2.8.6 Relation to the Research Framework ....................................... 29
2.8.7 Conclusions ................................................................. 30

2.9 An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering ................................................................. 30
2.9.1 Abstract .......................................................... 30
2.9.2 Background ................................................................. 30
2.9.3 Problem Statement ......................................................... 30
2.9.4 Contribution Relevance .................................................... 31
2.9.5 Contribution Summary ..................................................... 31
2.9.6 Contribution Evaluation ................................................... 31
2.9.7 Relation to the Research Framework ....................................... 31
2.9.8 Conclusions ................................................................. 31

2.10 Tweetflows - Flexible Workflows with Twitter .................................. 32
2.10.1 Background ................................................................. 32
2.10.2 Problem Statement ......................................................... 32
2.10.3 Contribution Relevance .................................................... 32
2.10.4 Contribution Summary .................................................. 32
2.10.5 Contribution Evaluation ................................................ 33
2.10.6 Relation to the Research Framework ............................... 33
2.10.7 Conclusions .............................................................. 33

2.11 A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules .................................................. 33
2.11.1 Background .............................................................. 34
2.11.2 Problem Statement ...................................................... 34
2.11.3 Contribution Relevance ............................................... 34
2.11.4 Contribution Summary ............................................... 34
2.11.5 Contribution Evaluation .............................................. 34
2.11.6 Relation to the Research Framework ............................... 35
2.11.7 Conclusions .............................................................. 35

3 Conclusions ........................................................................ 36
3.1 Summary .......................................................................... 36
3.2 An Outline of Future Work ................................................. 37

Bibliography ........................................................................ 38
A Attached Papers .................................................................. 42
A.1 Awareness-Based Realizability Analysis Of Choreographies With Exception Handling Constructs .................................................. 43
A.2 Automatic Attribute Inference In Complex Orchestrations Based On Sharing Analysis .................................................. 116
A.3 Towards Deriving Specifications for Composite Web Services .............................................................. 124
A.4 Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitutions .................................................. 134
A.5 Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Adaptation .................................................. 149
A.6 A Context-Aware Framework for Business Process Evolution .............................................................. 165
A.7 captevo: Context-aware Adaptation and Evolution of Business Processes .................................................. 174
A.8 Dynamic Composition of Pervasive Process Fragments .............................................................. 176
A.9 Designing Future-Context-Aware Dynamic Applications with Structured Context Prediction .................................................. 184
A.10 An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering .................................................. 204
A.11 Tweetflows - Flexible Workflows with Twitter .................................................. 219
A.12 A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules .................................................. 226
Chapter 1

Deliverable Overview

1.1 Introduction

The goal of the S-Cube work package WP-JRA-2.2 is to establish the foundation for Quality-of-Service (QoS) aware adaptable service compositions. Such compositions are able to adapt themselves in response to changes in the QoS characteristics of the invoked services and systems within which they execute and with which they communicate. QoS characteristics of a services are specified by different Service-Oriented Architecture (SOA) functional layers, from the infrastructure, to individual “atomic” services, properties of other services, to high-level business processes. Therefore, service compositions and their adaptation are influenced by a combination of those factors. The constraints to which the actual QoS of an executing service instance are expected to conform are usually defined using Service Level Agreements (SLA). The work included in this work package thus relies on inputs, capabilities, and QoS requirements from both the Business Process Management and the Service Infrastructure SOA functional layers, and uses them throughout service composition lifecycle, including modeling, verification, monitoring, and adaptation.

The goal of this deliverable is to present mechanisms and techniques for QoS-aware, coordinated service compositions. That includes the technical foundations of self-configuring, adaptive, QoS-aware service compositions, and the work on monitoring and analysis of service compositions and choreographies and their QoS characteristics in the scope of cross-organisational business processes. The work investigates how such monitoring results can be used for prediction of fluctuations in QoS characteristics and the potential selection of adaptation strategies and mechanisms for the overall Service-Based Application (SBA).

Being the final deliverable in the S-Cube WP-2.2 series, its important objective is integration of the results with the other work packages within S-Cube. Especially, the role of QoS characteristics of service compositions will be investigated in the context of Quality assurance and thus foster integration with other WPs, most notably WP-JRA-1.3, End-to-End Quality Provision and SLA Conformance.

1.2 Deliverable Structure

This deliverable is based on scientific publications developed within the framework of S-Cube. This first introductory chapter gives an overview of the contributed publications, describes how they relate to the goal of this deliverable, positions the contributions in the S-Cube Integrated Research Framework (IRF), and puts them in relationship with other S-Cube work packages.

Chapter 2 on page 14 gives a detailed description of each contribution included in the deliverable. Besides the basic information on the contribution title, the participating S-Cube partners, keywords related to the content, and the state of submission/publication, each description provides background, problem statement, contribution relevance to the deliverable, its summary, evaluation and conclusions. The actual
papers are attached in the appendix at the end of the deliverable.

Finally, Chapter 3 on page 36 gives a brief summary of the findings contained in the contributions, and provides an outline of future work. It is followed by the bibliography.

1.3 Overview of the Contributions

This deliverable includes the following 11 contributions:

- **Awareness-Based Realizability Analysis Of Choreographies With Exception Handling Constructs**
- **Automatic Attribute Inference In Complex Orchestrations Based On Sharing Analysis**
- **Towards Deriving Specifications for Composite Web Services**
- **Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitutions**
- **Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Adaptation**
- **A Context-Aware Framework for Business Processes Evolution**
- **Dynamic Composition of Pervasive Process Fragments**
- **Designing Future-Context-Aware Dynamic Applications with Structured Context Prediction**
- **An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering**
- **Tweetflows - Flexible Workflows with Twitter**
- **A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules**

The contributions presented in this deliverable present a number of techniques and approaches applicable to QoS-aware, coordinated service compositions, from different perspectives and taking into account various relevant aspects of the Service-Oriented Architecture (SOA). They are motivated by scenarios and goals of service providers, users, designers, and developers that appear in different stages of SBA life-cycle. The contributions extend the work that has been published and presented in the earlier deliverables in the S-Cube JRA-2.2 series on fragmentation [7], coordinated service compositions [45], and derivation of QoS specifications for services and service compositions [12]. Relationship of the contributions presented in this deliverable is discussed in section 1.5 Relationship to Other Work Packages on page 12. Detailed descriptions of the contributions can be found in Chapter 2 on page 14, and the actual papers are attached in the Appendix (starting at page 42). In the remainder of this section, we give a brief overview of the key highlights.

Using and Developing Formal Methods

Several contributions rely on the existing formal methods and models (or the development of new ones) for analyzing/verifying properties of service compositions, and/or synthetizing service compositions from specifications. **Awareness-Based Realizability Analysis Of Choreographies With Exception Handling Constructs** (p. 14) addresses the issue of detecting defects in realization of service choreography specifications (built from process algebra formulas) and providing precise diagnostics that can help service designers and developers to detect statically (at design-time) when their choreography specification cannot be realized, and to facilitate the changes and evolution of these specifications towards attaining realizability.
Automatic Attribute Inference In Complex Orchestrations Based On Sharing Analysis (p. 16) uses the notion of data sharing to model behavior of individual activities, complex constructs in a service orchestration (branches, loops, sub-workflows) with respect to their input and output data. It employs static analysis to infer which (user-defined) attributes of input data to a service orchestration are (potentially) shared by individual activities and intermediate/resulting data items in the orchestration. That information can be used for, e.g., splitting the orchestration into fragments for distributed enactment on the basis of information flow.

Towards Deriving Specifications for Composite Web Services (p. 18) is concerned with deriving specifications for composite services from the known specifications of the component services (pre- and post-conditions expressed as logic formulas), and with deciding which is the minimum part of these derived specifications that needs to be exposed to the user of the composition, so that, e.g., it can be used for constructing higher-level compositions.

Adaptation Based on Quality Prediction

Detecting ahead of time possible, probable, and/or imminent failure of service composition quality requirements (whether on the level of operational metrics, or related to higher-level business goals) during composition execution is the topic of two contributions presented in this deliverable.

Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitutions (p. 20) leverages machine learning techniques to identify, at a number of checkpoints in service composition execution, whether the composition is likely to violate its SLA requirements, which is usually expressed as a constraint on some QoS metrics that can be observed from monitoring. When such a likely violation is detected, the approach uses the techniques from Aspect-Oriented Programming (AOP) to substitute the current orchestration fragments with the new ones that have the chance of preventing the predicted SLA violation by changing the behavior of the composition.

Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Adaptation (p. 22) also relies on machine learning techniques, but with a different focus. Instead of focusing on low-level QoS metrics, here the authors look at the Key Performance Indicators (KPIs) of the business processes and try to infer influential factors on the process level that may lead to KPI violation on the business level. Several models (metrics, adaptation actions, constraints and preferences) are used to select the appropriate adaptation strategy based on adaptation requirements.

Context-Aware Service Compositions

Importance of context awareness for development of pervasive, dynamic and flexible service-based applications is the motivating factor behind several contributions presented in this deliverable. Dynamic Composition of Pervasive Process Fragments (p. 26) addresses the problem of dynamically creating service composition by putting together process fragments depending (among other things) on the context in which the composition needs to execute. That is done by using the globally available declarative specifications that define when a fragment can be used (relative to the context and other relevant requirements), and locally available imperative specifications that define its behavior.

Another interesting question is to predict context in which a service composition can find itself in some stage of execution. Designing Future-Context-Aware Dynamic Applications with Structured Context Prediction (p. 28) addresses this problem by using a learning model at design time (which can be also updated at run-time), from which a service-based application retrieves prediction during execution.

A Context-Aware Framework for Business Processes Evolution (p. 24) covers several phases of the adaptation life-cycle: execution, analysis and evolution. In the execution phase, the framework manages the execution and adaptation of the system, and logs the relevant information for future use. In the analysis phase, the framework evaluates the quality of the observed KPIs, determines the need for evolution of a given process model, and identifies the contextual evolution problem that needs to be
solved. Finally, in the evolution phase, the framework computes the new process model that behaves better in terms of KPI and/or avoids violation of context-induced constraints.

**Other Novel And Innovative Approaches**

*An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering* (p. 30) aims at extending the existing management approaches to involve an environmental dimension. It proposes an architecture and a corresponding methodology that can help organizations define, measure, analyze and adapt service compositions with the green computing goal in mind.

*Tweetflows - Flexible Workflows with Twitter* (p. 32) introduces a human-mediated service-oriented layer on top of the publish-subscribe paradigm of the Twitter micro-blogging social network, to allow solicitation, provisioning, monitoring, and to facilitate adaptation of human- and computer-provided services in light-weight, mobile and dynamic project team environments.

Finally, *A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules* (p. 33) uses fuzzy logic rules to deduce the amount of penalty that need to be compensated in case of an SLA violation, taking into the account the assessment of the degree (i.e., the amount) of violation and the characteristics of the service consumers.

### 1.4 Key Research Challenges and Results

The S-Cube Integrated Research Framework (IRF) defines the following research challenges for the workpackage JRA 2.2:

**Formal Models and Languages for QoS-Aware Service Compositions**: This challenge deals with formal models and Languages for QoS-aware service compositions. The challenge is substantiated by the facts, that firstly, there are no formal models for service compositions available that take into account the QoS and behavioral characteristics of these compositions and secondly, that the formal models are extremely important to guarantee that the final result of a composition services possesses the required characteristics.

**Monitoring of Quality Characteristics of Service Orchestrations and Service Choreographies**: In the context of QoS-aware service compositions, the focus of this challenge lies on monitoring of quality characteristics of service orchestrations and service choreographies. As service compositions implement business processes and at the same time run on IT infrastructure, their quality characteristics are influenced by both process-level and infrastructure-level metrics. A holistic monitoring approach for quality characteristics of service compositions involves monitoring of service orchestrations in terms of both process-level and infrastructure level factors and in addition monitoring of quality characteristics across participants in service choreographies.

**Analysis and Prediction of Quality Characteristics of Service Compositions**: When monitoring of quality characteristics of service compositions reveals that KPIs do not meet their target values, users are interested in finding out the causes and the most influential factors in order to be able to adapt the composition to prevent those violations in a future. Analysis and prediction mechanisms for quality characteristics will be devised, which are integrated with the monitoring mechanisms and provide input to the adaptation framework on which quality characteristics to adapt.

**QoS-Aware Adaptation of Service Compositions**: Adaptations of Service Compositions driven by changes in the environment and in particular by changes in QoS characteristics still remains a major challenge in service-based applications. Mechanisms for enabling such adaptations will be developed, and the major drivers for adaptation will be defined. The influence of the BPM and SI layers of SBAs on the adaptation of SC must be taken into account to ensure consistency of the adaptation steps.

The relationship between the contributions and the above challenges are illustrated in the Table 1.1.
1.5 Relationship to Other Work Packages

As described above, the motivation for many approaches is closely associated with quality assurance, and the subject of the deliverable is very closely related to the work package WP-JRA-1.3: Quality Definition, Negotiation, and Assurance. The contribution titled *A Context-Aware Framework for Business Processes Evolution* (FBK) is also reported in the deliverable D-JRA-1.3.6 in that workpackage.
<table>
<thead>
<tr>
<th>Contribution</th>
<th>Relationship of the Contributions to the Research Challenges in WP-JRA-2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Awareness-Based Realizability Analysis Of Choreographies With Exception Handling Constructs (USTUTT)</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Automatic Attribute Inference In Complex Orchestrations Based On Sharing Analysis (UPM)</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Towards Deriving Specifications for Composite Web Services (UoC, UPM)</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitutions (TUW, USTUTT)</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Adaptation (USTUTT, FBK)</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>A Context-Aware Framework for Business Processes Evolution (FBK)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic Composition of Pervasive Process Fragments (FBK)</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>Designing Future-Context-Aware Dynamic Applications with Structured Context Prediction (UniHH)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering (USTUTT)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Tweetflows - Flexible Workflows with Twitter (TUW)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules (UCBL, UPD, PolMi)</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Relationship of the contributions to the IRF challenges for WP-JRA-2.2.
Chapter 2

Mechanisms and Techniques for QoS-Aware, Coordinated Service Compositions

2.1 Awareness-Based Realizability Analysis Of Choreographies With Exception Handling Constructs

Publication reference: [27] attached on page 43.

Partners: USTUTT


2.1.1 Background

Service choreographies are a means of composing (web) services by specifying their roles, i.e. expected messaging behaviors, and the interactions among the roles. In the state of the art there are several paradigms for specifying service choreographies:

**Interaction** choreographies specify the message exchanges occurring among the participants in a choreography from a global perspective, e.g. using a workflow-like notation where activities represent message exchanges. Since the behaviour of the choreography is specified from a global perspective, the roles of the participants are specified *implicitly*. Interaction choreographies can be modeled with languages like Let’s Dance [52], WS-CDL [16] and BPMN 2.0 Choreography diagrams [34].

**Interconnection** choreographies specify *explicitly* the roles played by the participants, and “wire” them by specifying how the messages generated by one role are consumed by another. Interconnection choreographies can be modeled, for example, using BPEL4Chor [8] or BPMN 2.0 Collaboration diagrams [34].

**Artifact-centric** choreographies (see e.g. [26]) focus on how the lifecycles of business objects are manipulated by the participants through message exchanges.

The global perspective adopted by interaction choreography modeling gives raise to the issue of realizability. An interaction choreography is realizable if it is possible “…to automatically extract from the choreography the behavioral skeletons of the participants so that the concrete implementations, built on the basis of these skeletons, are guaranteed to satisfy the choreography specification.” [17]
2.1.2 Problem Statement

Realizability is a fundamental property of a choreography model [28], as it guarantees that the choreography can be actually realized faithfully by its participants. While the state of the art comprises a large body of theoretical works on realizability (for example [17, 44]) and methods for verifying it (see e.g. [5, 25, 39]), there is still only limited understanding of how to alleviate the realizability defects of an interaction choreography by modifying it so to make it realizable.

The first step towards a method for alleviating realizability defects of a choreography requires a method that provides meaningful diagnostic information of its realizability defects. In particular, since the unrealizability of a choreography is due to the lack of synchronization among its participants (see e.g. [44, 39]), it is vital for correcting realizability defect to have diagnostic information that pin-points: (1) in which point of the choreography is more synchronization required, and (2) why the participants that should be synchronized, are in fact not.

2.1.3 Contribution Relevance

The realizability analysis proposed in this work provides precise diagnostic information of realizability defects on interaction choreographies modeled with ChorTex, a process algebra-based choreography language based on Chor [51] (which, in turn, is related to WS-CDL).

2.1.4 Contribution Summary

In this work we propose a novel method for the realizability analysis of interaction choreographies modeled with ChorTex. The method is based on the concept of awareness, i.e. using static analysis for plotting the participants’ “perception” on the global state of a choreography enactment. By transforming a ChorTex choreography to a Control Flow Graph (CFG), and annotating the CFG’s nodes with the awareness of its participants using a fixed-point algorithm, the realizability analysis provides very precise information on which participants have perception of which states of the global enactment. We verify the realizability of the choreography by verifying awareness constraints that, if satisfied, guarantee the choreography’s realizability. If the choreography is unrealizable, the unsatisfied awareness constraints given precise and localized information on the synchronization shortcomings of one or more of its participants. Moreover, by traversing the CFG backwards, it is easy to discover at which point in the enactment, the participant has lost the necessary synchronization.

2.1.5 Contribution Evaluation

Formal proofs of correctness and convergence of the proposed method.

2.1.6 Relation to the Research Framework

The realizability analysis presented in this work is meant to support the design and evolution of service choreography by (1) allowing designers to verify the realizability of their choreography specifications (JRA-2.2), and (2) lay the foundation to a method for proposing changes that would solve realizability defects in choreographies, i.e. to steer the adaptation of service choreographies in order to make them realizable (JRA-1.2).

2.1.7 Conclusions

Using the diagnostic information that our realizability method provides, we intend to investigate how to identify change alternatives that would solve the realizability defects. Finally, we are investigating how to use the concept of monitoring contracts (see e.g. [49]) as an alternative method of alleviating realizability defects in choreographies when adapting the choreography specification is not an option.
2.2 Automatic Attribute Inference In Complex Orchestrations Based On Sharing Analysis


Partners: UPM


2.2.1 Background

Service compositions allow expression of a higher level or cross-organizational computation processes based on the existing component services as the building blocks. They are often given in the form of orchestration and defined using a workflow that specifies activities and the control and data dependencies between them. Complex activities include branches, loops and parallel flows, while simple activities denote single steps in the workflow. Some of the latter are executed locally, while others invoke component services, which, in general, may be provided and managed by different organizations. Reasoning about properties of data that is fed as an input to an orchestration, read / written and circulated between its activities, and returned as the result, can be useful in several phases of service lifecycle, such as composition design, adaptation through fragmentation and distributed enactment, and refactoring.

2.2.2 Problem Statement

The contribution addresses the problem of automated inference of attributes (user-defined properties) of data and activities inside a service orchestration that may include complex control and data structures and dependencies. The attributes are chosen by the user (designer) to reflect the relevant aspects from the application domain, such as privacy levels, data source or information content. Starting from the known user-defined matrix of attributes of the inputs to the orchestration and its structure, the goal is to deduce the attributes of the intermediate data items, data read by individual activities, as well as of the end results. That is a generally undecidable problem in presence of complex control structures, such as loops, and therefore, without restricting the class of admissible orchestrations to acyclic cases, the inference can be allowed to produce approximate, but safe (conservative) results, where potential attributes (i.e., those that cannot be decisively ruled out) are included.

2.2.3 Contribution Relevance

In the previous work [14] (presented as a contribution to the previous deliverable [12]), the authors presented the approach in which an input lattice of abstract data properties, such as data access / privacy levels, can be fed to a Horn clause representation [24] of a workflow to produce, based on the notion of data sharing, an output sharing lattice that can be used to identify workflow fragments by grouping together activities that share the same position in that lattice. In the current contribution, the authors present a sketch of a complete automated round-trip solution where a designer uses simple (Boolean) characterization of the inputs, which is internally transformed into an underlying input lattice structure, and is, at the end of the analysis, presented with a Boolean matrix that assigns the user-chosen input attributes to the workflow activities, intermediate data items and the resulting message. The approach works with orchestrations that have complex, real-life control structures which may include loops, branches and parallel flows.
2.2.4 Contribution Summary

The proposed approach uses the notions from Formal Concept Analysis (FCA) to transform a tabular object-attribute specification of the workflow inputs into a context lattice [11]. At the same time, the workflow definition is mechanically transformed to the form of a logic program (a series of Horn clauses) [24], which are then subjected to sharing analysis [31]. The Horn clause translation models both the data dependencies between activities, and control dependencies, such as branches, loops and parallel flows. Data items and activities are represented with logic variables. The sharing analysis operates in an abstract (conservatively approximate, but decidable) sharing and freeness which uses the input concept lattice to set up the initial (abstract) sharing relationship between the logic variables that represent the workflow inputs. The result of sharing analysis, the resulting sharing lattice, is then used as a concept lattice that is processed to recover characterization of the intermediate data items and activities in terms of the user-defined attributes. The final result is presented to the user as a Boolean object-attribute table.

2.2.5 Contribution Evaluation

The applicability of the proposed approach was illustrated on the case of distributed enactment of a medical workflow. The workflow used the inputs that identify a patient, as well as the databases of medical records and previously prescribed medications. In this setting, the user-defined attributes described the information content of the inputs, while the workflow (which involved parallel splits and joins, branches and loops, as well as sub-workflow components) was aimed at distributed enactment between the central health organization and its external partners (such as examiners, pharmacists and clerical service providers). The proposed approach was used to infer the information content of the intermediate data items in the workflow, as well as the content of information “seen” by each individual activity. The process was divided into fragments that were either executed centrally, or sent to a partner, based on data disclosure policies and the inferred information content. Other applications, such as robust top-down development and specification refinement were also considered.

2.2.6 Relation to the Research Framework

This contribution is directly related to the WP-JRA-2.2 research challenge Analysis and Prediction of Quality Characteristics of Service Compositions, where the quality aspect considered falls into the class of “data quality,” such as privacy and security. Its application to fragmentation (as a form of adaptation) is also related to the research challenge QoS Aware Adaptation of Service Compositions.

2.2.7 Conclusions

The contribution shows how an FCA-based characterization of input data to a workflow can be enriched to include intermediate data items and internal activities. These are annotated with attributes which are inferred from emergent properties of the workflow which stem from the workflow structure and relationships between input data. That task can be automated by translating (a) an initial FCA into a lattice from which sharing conditions are derived and (b) the workflow structure into a logic program. Then, (a) and (b) are subjected to a sharing analysis, and the results are mapped back to a resulting lattice and that to a resulting context, whose information can be used as a starting point for a number of other tasks. We have illustrated this methodology with a worked example.

As future work, the authors plan to address the development of automatic translations from common business process specification languages (BPEL, XPDL, YAWL, etc.) into logic programs amenable to sharing analysis in order to further test and refine the techniques proposed herein. Besides, they plan to explore other applications of the concept of sharing to services, aiming not only at (local) data sharing between activities, but also looking towards the representation of stateful service conversations and quality aspects of services.
2.3 Towards Deriving Specifications for Composite Web Services


Partners: UoC, UPM

Status: Conference, Submitted

2.3.1 Background

Service composition enables service-based systems to be built using accepted engineering principles such as service reusability and composability with the aim to provide value-added services that achieve functionality otherwise unattainable by atomic services. In order to fully achieve these goals, composite services should be available to consumers in the same way as atomic services are, abstracting away complex details of the way participating services are orchestrated to achieve the required functionality. This allows service consumers to invoke services regardless of the way they are implemented. This can be accomplished by providing formal specifications of composite services which reveal to the end user the minimum information required to understand the functionality offered, often by describing the inputs, outputs, preconditions and effects (collectively known as IOPEs) of the composite service. The composite service specifications should be based on existing specifications and descriptions of the services taking part in the compositions.

2.3.2 Problem Statement

While existing service description frameworks attempt to describe service compositions using a variety of composition models ranging from orchestrations to choreographies to Finite State Machines, no attempt (to the best of our knowledge) has been made to handle the problem of automatically producing specifications for a composite service, based on the specifications of the participating services. The same is true for automated Web service composition approaches: while each of them offers a way of automatically or semi-automatically producing the composition schema, the control flow and data flow of the composite service, none attempt to derive a complete specification IOPEs that should be provided to the service consumer.

Apart from the convenience they may provide to service consumers, specifications can be an invaluable tool for service providers. Similarly to the case of programming specifications, service specifications could be used as a basis to construct a service based on a set of requirements agreed upon by the parties involved, or to check that some existing specification meets a set of requirements. Specifications also play a major role in verification techniques as well as in the evaluation of the results of service adaptation or service evolution.

Composite specifications also offer great assistance when one attempts to deduce whether a set of services can actually be composed in a meaningful way. During the process of creating the composite specifications, inconsistencies may be detected between preconditions and/or postconditions of the participating services, rendering that particular set of services not composable. Such problems can be prevented before the composite service is delivered to the end user, by replacing the service or services that cause the inconsistencies.

2.3.3 Contribution Relevance

Our work aims to provide a thorough and efficient process of automatically deriving composite specifications based on the specifications of the participating services by attempting to deduce the minimum subset of these specifications that needs to be exposed to the service consumer. In order to do this, the composite specification is derived using structural induction, examining the composition schema using a
bottom-up approach. The proposed process keeps complexity at a minimum (as opposed to trivially including the complete specifications of participating services in the composite service specification) while at the same time retaining the complete set of knowledge that should be provided to the end user.

2.3.4 Contribution Summary

The composite specification should explicitly state all conditions that must be true before the execution of the whole composite service, as well as all conditions that are true after a successful execution. While we have preconditions and postconditions for each participating service, there is no obvious way of deciding which part of them will be included in the composite specification. Exposing the conjunction of the preconditions and postconditions of the participating services ignores the way the services are orchestrated and therefore the composed specification may be cumbersome, if not incorrect.

We propose a derivation process that is based on structural induction and attempts to construct the composite specification using a bottom-up approach. In order to achieve this, we formulate the derivation for all fundamental control constructs, namely sequential composition, AND-Split/AND-Join, OR-Split/OR-Join and XOR-Split/XOR-Join. By deriving preconditions and postconditions for these constructs, we can derive specifications for any composite service that includes such constructs, by considering the composition schema. The derivation process begins by examining the construct deeper in the schema and gradually moves its way upwards, till the whole composition schema is considered.

We also handle loop specification by deriving preconditions and postconditions based on the loop invariant. Generating loop invariants is an active and open research area and we provide precise details on what is necessary from approaches in this area for our derivation process to be valid. Finally, the case of handling asynchronous execution is addressed by employing the static single assignment form (SSA) in order to make sure that preconditions are evaluated in the context of the request and not the response.

2.3.5 Contribution Evaluation

The derivation process is applied to the E-Government case study of S-Cube [32] involving the request of paid-for, government-issued documents with or without certification by citizens. We explore a service evolution scenario, where a composite service implementing part of the document issue process is evolved to a new orchestration covering the complete process, satisfying a revised set of requirements. In order to check the conformance of the evolved service with the new specification, a specification for the new orchestration is derived using the method described in the paper. The resulting specification offers the complete set of preconditions and postconditions that formally describes what the composite service is supposed to provide and under what circumstances.

2.3.6 Relation to the Research Framework

This contribution is closely related to the research challenge for formal models and languages for QoS-aware service compositions, as it provides an automatic process of deriving formal specifications based on first-order logic that describe composite services. Moreover, there is a relation to challenges related to service adaptation, such as QoS Aware adaptation of service compositions, since the derivation process can be used to re-specify a composite service, after its composition schema and possibly some of its participating services have been adapted.

2.3.7 Conclusions

We proposed an approach for inferring composite service specifications given the specifications of the services participating in the composition (in the form of sets of preconditions and postconditions) and the composition schema. The approach attempts to construct the specification by using structural induction based on derivation rules defined for all fundamental control constructs, including loops as well
as supporting asynchronous execution. The resulting specification can be used to formally describe the composite service in terms of its preconditions and postconditions without requiring any knowledge of the internals of the composition, allowing for an actual "black box" view of the whole process. Such a formal description can be then employed to check whether a composition satisfies the requirements set by the requester, especially in cases of service evolution and service adaptation.

The nature of the proposed approach facilitates a possible implementation: structural induction lends itself to be written as a recursive algorithm. Hence, it would be straightforward to create an automated process that takes a set of service specifications and a composition schema and produces the specification for the composite service of the schema.

Future work includes exploring specification simplification, which becomes important as composite services become more complex. We plan to look into the work of Douglas Smith [43] on simplifying precondition formulas. An implementation is also in the works, in order to evaluate the approach in terms of effectiveness with respect to time. Finally, the work is planned to be integrated in a general service specification framework that will handle issues raised by the frame problem and related ones.

2.4 Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitutions


Partners: TUW, USTUTT

Status: Conference, Accepted

2.4.1 Background

In a service-oriented architecture, service providers realize business processes through a service composition which orchestrates services running on a service infrastructure. They offer a service to service consumers agreeing on its QoS characteristics in a service level agreement (SLA).

2.4.2 Problem Statement

For a service provider it is desirable to be able to predict whether an SLA will be violated. In addition, after an SLA violation is predicted, the goal is to proactively adapt the composition in order to prevent the predicted violation. Thereby, it is often not sufficient to substitute services in the composition, but it is often needed to change the process logic in terms of composition fragments.

2.4.3 Contribution Relevance

The presented approach is based on and extends several previous S-Cube papers. [50] has shown how machine learning techniques can be used for KPI and SLA dependency analysis, i.e. finding out the influential factors of SLA violations. Here, we use a similar learning approach focusing on prediction. [22] has introduced the concept of check points which we reuse in the approach. [20] has dealt with prevention of SLA violations by integrating monitoring, prediction and adaptation in a similar fashion, however focusing just on service substitution as the adaptation mechanism. We extend that approach by enabling the substitution of arbitrary process logic in terms of composition fragments.
2.4.4 Contribution Summary

At design time, first a set of checkpoints is defined in the service composition where the prediction and potential adaptation should take place. Then, a set of fragment alternatives is modeled by the user and assigned to the checkpoints. A fragment is an (alternative) implementation of a part of business logic in the target composition. It is a standalone and typically “incomplete” composition part focusing on a certain part of business logic and is linked into the original composition via join points. The linking is performed at design time by the user. Therefore, one specifies at which points in the original composition the control flow changes from the target composition to the fragment and vice versa. The join points in the fragment are defined via virtual activities denoting the START, END, or TRANSPARENT parts of the fragments. In addition to the functional part, a fragment definition also contains an impact model which defines how this fragment affects composition performance metrics. This allows later to select appropriate fragments which will likely prevent the predicted violation. The impacts can be derived from SLAs or from history measurements. Finally, one can also specify dependencies between fragments, e.g. one fragment requiring another one.

At process runtime, for each checkpoint a prediction model is learned based on history data using machine learning techniques. Whenever a composition instance passes the checkpoint, the monitored data of that instance is used as input for the prediction model. If an SLA violation is predicted, the adaptation is triggered. A set of adaptation alternatives is identified based on the impact models by repeating the prediction for each potential adaptation alternative (all possible fragment combinations are enumerated) as if adaptation has taken place. From the alternative adaptation alternatives, the alternative whose predicted SLA value is closest to the target value is weaved into the target composition.

2.4.5 Contribution Evaluation

The approach has been implemented based on Windows Workflow Foundation technology and has been experimentally evaluated. Experiments show that dynamic weaving introduces only a very small overhead (in [45:80] ms) which could be relevant only for very short running processes.

2.4.6 Relation to the Research Framework

This contribution aims to provide techniques and a development process to design QoS-aware adaptable service compositions which is one of the main tasks of JRA-2.2. It covers the research challenges “Analysis and Prediction of Quality Characteristics of Service Compositions”, “Comprehensive and integrated adaptation and monitoring principles, techniques, and methodologies”, “QoS Aware Adaptation of Service Compositions”, and “Proactive Adaptation and Predictive Monitoring”. It therefore connects the domain layer Service Composition & Coordination with the cross-cutting concerns Engineering and Design, Quality Definition, Negotiation and Assurance, and Adaptation and Monitoring. Regarding the S-Cube reference life-cycle, the approach covers the life-cycle states Requirements Engineering and Design and Construction by explaining how fragments and their linking are to be modeled at design-time, and then focuses on the runtime states from Operation and Management to Adaptation Enactment.

2.4.7 Conclusions

The presented approach deals with runtime adaptation of service compositions for preventing SLA violations. It is based on previous S-Cube work on monitoring and runtime prediction of SLA violations. The main contribution is a new adaptation technique which substitutes composition fragments dynamically at runtime. At design time, composition fragments are modeled separately and are explicitly linked into the original composition using virtual activities. At runtime, at predefined checkpoints they are then weaved into the original composition if an SLA violation is predicted. Future work includes taking into account several service level objectives at the same time and the cost of adaptation.
2.5 Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Adaptation

Publication reference: [48] attached on page 149.

Partners: USTUTT, FBK

Status: Journal, Submitted

2.5.1 Background

Service-based applications (SBAs) realize business processes through a service composition which orchestrates services running on a service infrastructure. An important aspect when managing such applications, is to ensure that they meet certain business goals, typically expressed via a set of Key Performance Indicators (KPIs). KPIs specify target values on the time, cost, and quality dimensions of business processes.

2.5.2 Problem Statement

KPIs are typically continuously monitored at process runtime using business activity monitoring techniques. If monitoring shows bad results, i.e. KPI targets are often violated, then one needs to analyze the influential factors which lead to those violations. In complex business processes, the dependencies of KPIs on lower-level metrics are neither explicit nor easy to reveal. In addition to finding out the reasons for KPI violations, it is desirable to be able to predict whether a KPI target will be violated. This would allow to proactively adapt the process in order to prevent a predicted violation. Thereby, one should take into account that typically several competing KPIs can be specified (e.g., quality vs. cost), which has to be addressed during prediction and adaptation. The overall goal is to create an approach which integrates the monitoring, analysis, prediction and adaptation phases to achieve self-adaptable KPI-aware service compositions.

2.5.3 Contribution Relevance

The presented approach is based on and related to several previous S-Cube papers. [50] has shown how decision tree learning can be used for KPI dependency analysis, i.e. finding out the influential factors of KPI target violations. Here, we use the same learning approach with the focus on using the KPI dependency trees for prediction and adaptation purposes. In [18], we have already discussed on a higher-level how adaptation requirements can be extracted from dependency trees and how to derive adaptation strategies. That work has been substantially extended, implemented and evaluated in the current approach. [22] has introduced the concept of check points which we reuse in our approach. [20] and [21] (presented in Section 2.4) deal with prevention of SLA violations by integrating monitoring, prediction and adaptation in a similar fashion. The prediction thereby focuses on numerical metrics using machine learning techniques such as artificial neural networks. In our approach, we predict nominal-valued metrics using decision trees which is a white-box model explaining explicitly dependencies between the KPIs and lower-level metrics and thus allowing extraction of adaptation requirements. We also support prediction and adaptation in respect to several competing KPIs.

2.5.4 Contribution Summary

At design time, several models are created which are then used as input to the runtime monitoring and adaptation phases. The metrics model contains the KPI definitions and the lower-level metrics which are needed for dependency analysis and prediction. The adaptation actions model contains the available
adaptation actions (AAs) which can be used to adapt the process, e.g. service substitution, change of a process variable value, skipping of an activity etc. In addition, an AA specifies its impact on a set of metrics which is later needed for deciding whether this AA actually helps to prevent the KPI violation. The check point model defines the check points in the process where the prediction and potential adaptation should take place. Finally, the constraints and preferences model defines (i) constraints on metrics and KPIs which have to be satisfied when performing adaptation, (ii) preferences in terms of weights on the KPIs and metrics which allow selecting an adaptation strategy in case of several alternatives.

At runtime, the metrics model is used in the monitoring phase to calculate KPIs and metric values based on runtime events. After a certain number of process instances has been executed, for each check point and each KPI of that check point, a KPI dependency tree is learned. In the prediction phase, the process instance is halted at a check point and the KPI dependency trees are used to predict the KPI classes for that process instance by inserting the available metric values at the check point into the trees. The prediction result is a decision tree (derived from the learned KPI dependency tree) which shows the predicted KPI classes in relation to the metrics affected by available adaptation actions. Form the derived instance trees we extract adaptation requirements (conjunctions of metric value predicates) which specify how we have to improve certain metrics. In the subsequent step, we then select and combine available adaptation actions which satisfy the adaptation requirements, thus creating adaptation strategies. One adaptation strategy is finally selected based on the specified preferences model using simple additive weighting as part of multiple attribute decision making.

### 2.5.5 Contribution Evaluation

The approach has been implemented and experimentally evaluated based on a purchase order processing scenario. The experiments show that the prediction and adaptation time together are below a second thus making it only a performance factor for short running processes. More importantly, we have shown that the number of KPI violations has been considerably decreased with the use of our framework. The prevention effectiveness mainly depends on (i) check point positioning, i.e., the later the check point, the higher the prediction accuracy, however the lower the number of available adaptation actions, (ii) conformance of actual metric effects of chosen adaptation actions to the modeled (estimated, guaranteed) metric effects as specified in the adaptation actions model; (iii) preferences model, i.e. provided weights on the different KPIs and metrics.

### 2.5.6 Relation to the Research Framework

This contribution aims to provide techniques and a development process to design QoS-aware adaptable service compositions which is one of the main tasks of JRA-2.2. It covers the research challenges “Analysis and Prediction of Quality Characteristics of Service Compositions”, “Comprehensive and integrated adaptation and monitoring principles, techniques, and methodologies”, “QoS Aware Adaptation of Service Compositions”, and “Proactive Adaptation and Predictive Monitoring”. It therefore connects the domain layer Service Composition & Coordination with the cross-cutting concerns Engineering and Design, Quality Definition, Negotiation and Assurance, and Adaptation and Monitoring. Regarding the S-Cube reference life-cycle, the approach covers the life-cycle states Requirements Engineering and Design and Construction by explaining the different models which have to be created at design-time, and then focuses on the runtime states from Operation and Management to Adaptation Enactment.

### 2.5.7 Conclusions

The presented approach integrates monitoring, KPI dependency analysis, prediction and proactive adaptation to realize QoS-aware adaptable service compositions. The monitoring and adaptation are thereby currently focused on the service composition layer. Future work consists of extending the approach
towards cross-layer monitoring and adaptation, e.g. taking into account the impact of the service infrastructure QoS metrics on the KPIs and performing adaptations on service infrastructure level accordingly.

2.6 A Context-Aware Framework for Business Processes Evolution


Partners: FBK

Status: Workshop Paper presented at EVL-BP 2011 and Demo paper accepted at ICSOC 2011

2.6.1 Background

Run-time adaptability is a key feature of dynamic business environments, where the processes need to be constantly refined and restructured to deal with exceptional situations and changing requirements. The execution of such a system results in a set of adapted process variants instantiated on the same process model but dynamically restructured to handle specific contexts. Process evolution exploits the information on process variants to identify the best performing recurring adaptations and adopt them as general solutions in the process model. However, process variants are strictly related to specific execution contexts and cannot be adopted as general solutions. We propose a framework supporting context-aware evolution of business processes based on process instance execution and adaptation history. Instead of looking for recurring adaptations, we propose to look for recurring adaptation needs (i.e., process instances with the same context constraint violation and system configuration). Based on the analysis of adapted instances, we automatically construct and rank corrective evolution variants which can handle the problematic context. At the same time, we try to identify preventive evolution variants by constructing process variants which can prevent the adaptation need. We demonstrate the benefits of our approach using a car logistics scenario.

2.6.2 Problem Statement

Adaptation needs of dynamic business processes may be triggered by specific cases to be handled, unexpected situations depending on environmental conditions, or changing requirements.

This need for continuous adaptation results in a system characterized by a huge set of process executions that, although instantiated on the same process model, strongly differ in terms of process structure. In such a dynamic environment, the process models cannot remain unchanged; the short-term adaptations applied to process instances should be used to derive long-term changes in the process models. Providing support for process model evolution is becoming one of the main requirements for managing the lifecycle of dynamic processes. In particular, the set of adapted process instances together with the information concerning their execution should be used as training cases for evolution mechanisms in order to progressively improve process models that are then used to instantiate future process instances.

Most existing approaches addressing this problem (e.g., [38, 23]) derive model-level changes by analyzing frequently occurring changes at the instance-level. In other words, if an instance-level change/adaptation occurs more frequently then a predefined threshold, the change will be propagated at the model-level. These evolution approaches present two major drawbacks. First, an instance-level adaptation variant is not good "in general", it is good for a specific context/situation, and thus cannot simply be propagated to the process model without taking into account the adaptation need it was devised for. Moreover, plugging-in adaptation variants in the original process model is not always a good solution, since it may result in embedding fault-handling activities rather than trying to solve the problem that required runtime adaptation.
2.6.3 Contribution Relevance

To overcome the limitations, presented in the previous section, we present a context-aware evolution framework that, instead of searching for recurring process changes, searches for recurring adaptation needs. We describe the modeling artifacts of our evolution framework, as well as their lifecycle. Finally, to illustrate the role of these artifacts, we use a car logistics scenario.

2.6.4 Contribution Summary

The framework that we propose is able to cover the following phases of an evolution life-cycle, namely (1) execution phase, (2) analysis phase, and (3) evolution phase. During the first phase, the evolution framework proposed, is responsible for managing the execution and adaptation of the system and for logging all the information that may be useful to the other phases (e.g., execution traces, adaptation needs, adaptation variants, execution performances). During the analysis phase, the framework controls and evaluates the quality of execution of the processes with respect to the KPIs, decides the need for evolution for a certain process model, and, on the basis of the execution history, identifies the contextual evolution problem in terms of recurring system configuration that required adaptation. Finally, in the evolution phase, the framework uses the information obtained from the analysis phase to compute process model variants that either embed the best performing (with respect to KPIs) adaptation variants or prevent the violation of the context constraint. These evolution variants are then presented to the process designer, who decides whether they should be adopted for future executions. The process designer obtains the evolved process model using a set of supporting tools.

2.6.5 Contribution Evaluation

The evolution framework has been implemented as a part of the CAptEvo prototype [4], a tool of the ASTRO project 1, that integrates sophisticated techniques for managing the execution, adaptation, and evolution of context-aware business processes. Finally, to demonstrate the CAptEvo framework in action, we use a real-world scenario from the domain of logistics.

2.6.6 Relation to the Research Framework

This contribution spans over several challenges of different workpackages. In particular, it is closely related to the JRA-2.2 research challenge of Analysis and Prediction of Quality Characteristics of Service composition and QoS Aware Adaptation of Service Compositions, since it provides techniques to automatically improve the process composition on the basis of QoS performances. The approach also contributes to the Comprehensive and integrated adaptation and monitoring principles, techniques, and methodologies challenge of JRA-1.2, and to Run-time Quality Assurance Techniques and Quality Prediction Techniques to Support Proactive Adaptation challenges of JRA-1.3. Regarding the S-Cube lifecycle, the proposed framework covers all the phases related to monitoring, adaptation and enactment, namely Operation & Management, Identify Adaptation Need, Identify Adaptation Strategy, and Enact Adaptation.

2.6.7 Conclusions

We have presented a framework for evolving process models based on a history of process instance executions and adaptations. Our approach is context-driven. If the need to evolve the process model is detected, we analyze the relevant adapted process instances and look for recurring adaptation needs (i.e., the same constraint violation and system configuration). This allows us to construct and rank evolution variants which can handle the problematic context (corrective evolution). It also allows us

---

1www.astroproject.org
to construct evolution variants which can prevent the adaptation need (preventive evolution). In our future work, we will develop concrete solutions for corrective and preventive evolution. For corrective evolution we plan to develop techniques to automatically transform instance-level adaptation variants into evolution variants using the built-in adaptation tools. For preventive evolution we will apply AI planning techniques to re-plan the process model in order to avoid the critical configurations. We will implement and evaluate our solutions on realistic scenarios, such as the car logistic scenario introduced in this work.

2.7 Dynamic Composition of Pervasive Process Fragments


Partners: FBK

Status: Conference paper (accepted) and journal paper (on-going)

2.7.1 Background

Applying the SOA paradigm to pervasive systems (Internet of Things) is a hot research topic that introduces several interesting challenges for the Service Community. Pervasive systems change their behavior, reconfigure their structure and evolve over time reacting to changes in the operating conditions, so to always meet users’ expectations. This is fundamental since those systems live in distributed and mobile devices, such as mobile phones, PDAs, laptops etc., thus their environment may change frequently. Also, user goals and needs may change dynamically, and systems should adapt their functionalities accordingly, without intervention from technicians. In order to achieve the required degree of flexibility and dynamicity a SOA based solution must cope with the fact that services must be context-aware, self-configurable and adaptive.

2.7.2 Problem Statement

The ability to model partial and incomplete knowledge about the processes and to dynamically compose them into complete executable processes is a key feature of pervasive applications, where the actual process to be executed depend on the specific context and on the available services. Process fragments [10] represent a tool for modeling incomplete and local process knowledge. The knowledge is incomplete since the modeler is allowed to leave gaps in the process specification. Further, it is local, since the availability and usability of the fragments is determined by context (e.g., location, time, situation, people). For example, the process fragment execution may be bound to a certain location. We propose a solution for automatically compose process fragments according to predefined complex flow goals and taking into account fragments availability and context properties.

2.7.3 Contribution Relevance

Several approaches have been proposed to dynamically compose services [47, 37, 19], however, they cannot be applied to pervasive domains since they do not deal with the context aspect. Moreover, all these approaches represent the process goal either through an imperative or a declarative language. Imperative process models focus on the way to achieve the goal, and assume that the environment is stable. Declarative process models specify the goal through constraints which approximate the desired behavior. Such descriptions are suitable for frequently changing environments, but cannot be executed completely automatically. We use both process description techniques to achieve a greater flexibility with respect to the environment. The declarative part corresponds to goals, and is globally available. The imperative, concrete part corresponds to pervasive process fragments, and is available only locally.
2.7.4 Contribution Summary

We propose an approach for composing pervasive process fragments according to complex goals. Our approach is based on previous results from Web service composition [37], and in particular on the work of [2], which addressed the problem of Web service composition with complex composition requirements. The problem of fragment composition is conceptually different from Web service composition, since the components are not orchestrated, but integrated into a complete flow model. Due to this difference, important issues in Web service composition, such as the need for Web services to communicate via message exchange, do not appear in fragment composition. On the other hand, pervasive fragment composition introduces new problems. For example, the fragments can overlap, or can have gaps in the specification, and the composition task must take into account context properties. To deal with these differences, we have significantly modified and extended the approach in [2]. Our approach is based on a three-layer representation, where the first layer captures the specific context knowledge required for the composition task, the second layer describes the abstract processes in terms of the goals it should achieve, and the third layer is the concrete, context-dependent part of the process definition, represented using fragments. We present a complete model that allows to relate context properties, goals, and fragment models, and a mechanism for composing fragment by encoding object diagrams, goals and fragments into an AI planning problem.

2.7.5 Contribution Evaluation

We implemented the approach into a prototype tool. The tool translates object diagrams (XML), process fragments (BPEL extension) and goals into a planning problem. The planning problem is given as input to WSYNTH, one of the tools in the ASTRO toolset (http://www.astroproject.org). The output returned by WSYNTH is the planning controlled composition, which we then translate back to executable BPEL. We evaluated our tool on a logistic scenario (handling and delivering boxes in an airport) entailing a high level of complexity and pervasiveness. We consider two features specific to fragment composition. First, there is a tradeoff between designing fragments with a large number of activities (a higher burden on the designer) and with a small number (longer composition time). Second, the set of available fragments can contain more fragments than actually necessary for composition. We observe that larger fragments lead to a significant speedup. Using fragments of two or more activities can lead to a time increase of 20% in the worst case, and a decrease of up to 95% in the best case. For this domain involving complex composition problems, larger fragments provide a clear benefit in terms of composition time. In the second experiment, we test how the number of available fragments influences the performance. For this purpose, we vary the total number of fragments, while keeping constant the number of fragments actually composed. As expected, the composition time grows exponentially in the number of fragments, due to the hardness of the planning problem. However, this can be solved using existing selection techniques to reduce the number of component fragments.

2.7.6 Relation to the Research Framework

This contribution is closely related to the JRA-2.2 research challenge of Formal Models and Languages for QoS-Aware Service Compositions, since it provides a framework for modeling partial, local, and context-aware process knowledge through fragments. It is also related to the research challenge of QoS Aware Adaptation of Service Compositions, since it provides an automatic way of composing at run-time the set of available, and context/goal-specific fragments to obtain a context-aware executable process. Regarding the S-Cube life-cycle, the proposed modeling framework (domain objects, pervasive fragments, complex goals) can be mapped to the life-cycle phase Requirements Engineering and Design Construction. While the dynamic fragment composition contributes to the Enact Adaptation phase, since compositions can be automatically derived at run-time considering the execution context.
2.7.7 Conclusions

The approach presented in this Section allows to compose pervasive process fragments according to context and goals. We first specify the properties of the context using object diagrams, and use these properties to define process goals and to annotate the fragments. We then encoded object diagrams, goals and fragments into an AI planning problem. The result of composition is a complete adaptable pervasive process which satisfies the goals and corresponds to a synthesis of the pervasive fragments. An on-going work is the complete formalization of the proposed framework and its evaluation on a logistic scenario in the automotive domain.

2.8 Designing Future-Context-Aware Dynamic Applications with Structured Context Prediction


Partners: UniHH

Status: Accepted for publication in Software—Practice & Experience (SPE)

2.8.1 Background

Middleware support for advanced service-based applications includes dealing with heterogeneous systems and dynamic changes of execution environments. If applications are context-aware they are able to detect such changes at runtime and react to them accordingly. Following Dey and Abowd, context is defined as any information characterizing the situation of surrounding and situational entities relevant to an application and its users [9]. In consequence, the ability to obtain, to process, to manage and to provide relevant context information describing the user’s environment and situation has become one important requirement for such systems. In addition to that, the prediction of future context is another important step for enabling devices and applications to also proactively support the user or to enable the desired automatic execution of his tasks even in dynamic environments [30]. However, the design of reusable middleware support for such future-context-aware service-based applications is still challenging – since a supporting prediction system has to be both generic and, at the same time, as efficient and accurate as possible.

2.8.2 Problem Statement

In summary, a general applicability of a context prediction system imposes several principal requirements: First, the system should support a preferably wide range of applications and diversity of exploitable contexts [6] in order to maximize reusability. Furthermore, there are individual differences between users [15] which can also change continuously [41]. Thus, the system has to adapt to the individual user at runtime by learning about the characteristics and regularities which determine the future context [15, 41].

Besides being generic and adaptive, the prediction system should be able to produce customized predictions for different scenarios in a reliable and satisfying way, i.e. as accurately and efficiently as possible. Especially mobile devices often suffer from a lack of resources [40], so that the corresponding requirement for efficiency makes this trade-off even more challenging. Finally, the motivation to support application developers implies a preferably low effort for them.
2.8.3 Contribution Relevance

Many existing approaches aim to support predictions for a particular domain or a specific context criteria. Only few approaches explicitly address the design of a prediction framework to support the development of future-context-aware applications in general. Here, the approaches of Mayrhofer [29], Sigg [41] and Petzold [36] are considered to be major contributions towards generic context prediction. However, the analysis of existing work has shown that there is still no approach which is sufficiently generic and offers high accuracy and efficiency at the same time. Based on the experiences with existing approaches this contribution introduces the approach of Structured Context Prediction (SCP) which addresses open issues and therefore presents an alternative to existing context prediction frameworks.

2.8.4 Contribution Summary

The approach of Structured Context Prediction (SCP) realizes a generic prediction system which can be utilized in order to develop future context-aware applications or enhance existing applications with the ability to make assumptions about future situations. It is based on two fundamental principles derived from the preceding analysis of existing approaches: First, knowledge about the application domain is used as valuable information incorporated by the application developer at design time or at runtime. The second principle is a hybrid application of multiple, exchangeable prediction methods. Thus, methods which are appropriate to ensure accuracy and efficiency of domain-specific predictions can be selected and combined by the application developers respectively. Based on that, so-called prediction models can be created and incorporated at design time of the application or can be distributed at runtime. Additionally, an adaptation to the individual user is achieved by adaptive online-learning as the default learning mechanism.

The knowledge about an application domain is described as a prediction model which specifies the way predictions have to be performed. Basically, it assigns a method to each relevant variable in order to predict its value. As input, the method uses the values of other variables – which are again predicted by their own methods or which are known (e.g. measured by a sensor or distributed at runtime). The main part of a prediction model specifies how the methods and respective variables are connected. This part is called prediction net. A prediction net specifies that the value of a variable at a specific point of time is predicted using the values of other variables at the same point of time or earlier.

From the perspective of an application developer, the general procedure of using the prediction system consists of two parts: The first part is determined by the development of the prediction model at design time. The second part is the retrieval of predictions by the respective application at runtime. Run-time learning is executed automatically so that most aspects of learning are hidden from the application and from the developers.

2.8.5 Contribution Evaluation

The proposed approach has been evaluated in a mobile environment, where the successful execution of service-based business processes depends highly on the availability of required services. In order to access such unsteadily available services and exploit local functionalities, processes can be migrated to other mobile or stationary devices. In such a scenario, it is advantageous to guide the migration of a mobile business process by predicting future service availability.

This scenario has been used in order to evaluate the proposed approach in a case study by creating an corresponding prediction model and by examining realistic context data about the behaviour of a user and its mobile device spanning an interval of seven days.

2.8.6 Relation to the Research Framework

This contribution aims to provide techniques and a development process to design or enhance future-context-aware service-based applications. It therefore connects the domain layer Service Composition...
& Coordination with the cross-cutting concerns Engineering and Design and – in parts – Quality Definition, Negotiation and Assurance. Regarding the S-Cube reference life-cycle, the proposed procedure of developing a future-context-aware application can be mapped to the life-cycle states Requirements Engineering and Design and Construction. The contribution also introduces techniques for the actual context prediction, which are utilized in the phase Operation & Management.

2.8.7 Conclusions

As a further step towards realizing dynamic pervasive environments, the approach of Structured Context Prediction proposes and realizes a context prediction system which is generic and provides potential for high accuracy and efficiency at the same time. In consequence, this approach provides a new combination of characteristics compared to existing approaches. In contrast to previous approaches, the composability of prediction methods and the integration of domain-specific knowledge as proposed here enable support for a wide range of new (service-based) applications and thus allow integration into existing middleware frameworks.

2.9 An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering

Publication reference: [33] attached on page 204.

Partners: USTUTT

Status: International Conference on ICT as Key Technology for the Fight against Global Warming - ICT-GLOW 2011, accepted and published.

2.9.1 Abstract

Sustainability and responsible resource exposure has become a major issue in everyday life. Government, customers, and increasing social responsibility force more and more organizations to positively optimize their environmental impact towards a better, livable planet. In this paper we propose a four-layered architecture and corresponding four-phased methodology to enable organizations to (1) define ecological characteristics, (2) sense and measure these ecological characteristics, (3) identify, localize and visualize their environmental impact, and (4) help them to develop appropriate adaptation strategies in order to optimize their environmental impact without neglecting the organization’s competitiveness.

2.9.2 Background

Organizations are based on their underlying business processes. When using service compositions to implement those business processes, certain Quality of Services need to be considered in order to achieve a successful process execution considering several internal and external constraints and objectives. Key Ecological Indicators are specific Key Performance Indicators and strategically define the organizations objectives with respect to their environmental impact. Thus, organizations need adequate methodologies and techniques to properly design their service compositions based on their environmental requirements.

2.9.3 Problem Statement

The general problem is the identification of environmentally-bad parts of business processes in terms of choosing a “good” service composition that changes the process into a more sustainable one. However,
current approaches do not cover specific environmental optimization objectives to optimize corresponding service compositions and business processes. Organizations need methods and technologies to define, measure, and analyze their business processes with respect to their economic and environmental impact. A solution to this problem is relevant in order to cope with increasing awareness of customers and legislative requirements.

2.9.4 Contribution Relevance

The paper describes an architecture and a corresponding methodology that describes one way how organizations can define, measure, analyze and adapt service compositions. The approach extends existing management approaches by introducing an environmental dimension.

2.9.5 Contribution Summary

The contribution of this paper is twofold: Firstly, we introduce an architecture that includes four layers to serve the different aspects of gBPR: (1) Strategy, (2) Sensing & Monitoring, (3) Analysis & Management, and (4) Adaptation. This architecture covers the proper monitoring, analysis and adaptation of green reengineering approaches and thus helps organizations to identify the relevant aspects for optimizing their environmental impact. Secondly, we introduce a methodology to enable the process stakeholders to reduce the environmental impact utilizing the proposed architecture.

2.9.6 Contribution Evaluation

A prototype for simulation is still ongoing work and not included in the paper. However, we used an exemplary use case in order to describe the feasibility of the approach.

2.9.7 Relation to the Research Framework

The conceptual approach proposed in this work consists of different cross-cutting concerns and is related to the Business Process Management and Service Composition domain layers within the conceptual research framework of S-Cube. As the focus is on how to optimize service compositions that build the business processes of an organization, it covers the “Engineering and Design” and “Adaptation and Monitoring” perspectives. It also covers the “Quality Definition” perspective as KEIs are a special kind of KPIs that define certain quality requirements. Based on the addressed layers and perspectives of the conceptual research framework, the approach covers the following phases of the S-Cube Reference Lifecycle: Requirements Engineering and Design, Construction, Deployment and Provisioning, as well as Operation and Management.

2.9.8 Conclusions

The architecture presented in this paper describes fundamental layers needed to achieve more sustainable organizational environments in the cross-cutting concern of green Business Process Reengineering. We described each layer in detail, identified the roles within an organization responsible for each layer and sketched the main issues of each layer. Moreover, the corresponding methodology presented in this work describes a walk through this architecture. It helps organizations to plan and define their ecological objectives in form of Key Ecological Indicators (KEIs) and to identify and localize the most dissipative parts of their processes based on these KEIs. To realize the Analysis & Management as centerpiece layer of the architecture we used the approach of “process views” that enables a proper visualization of the process model utilizing so-called view transformations. Consequently, in the Adaptation layer organizations can derive adaptation strategies to optimize their collective environmental impact while considering both, their ecological and economic objectives.
The approach bridges the gap of missing interconnection between existing Green IT and Green IS approaches towards a holistic environmental impact analysis and optimization in organizational structures. In our future work we will investigate a classification for KEIs and their application in intra-organizational and cross-partner environments. Within this work we will also address the problem of how to sense and monitor the environmental influence factors on a per task basis. We will further develop different process view patterns that allow organizations the application of process views in a re-usable fashion and will devise algorithms that support the trade-off between KPIs and KEIs. Discusses the key elements of the.

2.10  Tweetflows - Flexible Workflows with Twitter


Partners:  TUW

Status:  Journal Paper submitted to SOCA journal (currently under revision)

2.10.1 Background

Tailored implementations of the SOA stack for mobile devices allow users to consume remote Web services. While as an implementation technology for mobile SOA, adopting the already broadly supported Web services (WS) is justified, studies show that the SOA stack is too complex to be implemented for mobile devices. In fact, the perceived complexity of the SOA stack hinders a wide spread adoption of SOA on mobile devices. Thus, we approach the subject of mobile SOA from a different perspective. First of all, we study the applicability of existing infrastructures like App Stores and mobile Apps for SOA principles. The second key aspect of our work is our focus on using social ties of mobile Service providers (users) for Service coordination purposes. Combined with human provided Services we lay the foundation for mobile SOA that is enriched with social aspects.

2.10.2 Problem Statement

In our work, we address three specific issues of mobile SOA: (1) the mapping of the SOA infrastructure (e.g. Service registry) onto the corresponding counterpart in the mobile domain (e.g., App Store), (2) a lightweight coordination language that satisfies the requirements of mobile Service coordination in a crowd setting and (3) the use of human provided Services to facilitate the integration of humans into mobile SOA.

2.10.3 Contribution Relevance

In our work, we introduced a conceptual mapping of SOA principles to infrastructure that is used in the mobile domain. Based on these principles, we showed how to use a lightweight coordination language (Tweetflows) to coordinate the execution of Services on mobile devices. The discovery and execution of mobile Services can be mediated by the user of the mobile phone. In doing so, we tap into the social network of the user.

2.10.4 Contribution Summary

Tweetflows introduce a new format and semantics for short Twitter text messages on top of its basic publish-subscribe message exchange pattern. In the style similar to the Twitter’s established ‘re-tweet” (RT) primitive, users can use other primitives to announce, request, reply, delegate and query state of a
logical service, which can be human-provided, or automatically processed by a software that reads the messages from a Twitter stream. The mechanism uses hash-tags to identify the service and its attributes, while (due to the text message size limitation) service argument/result passing is done using (short) URLs that refer to them.

Tweetflow messages support the entire service life-cycle, including publishing, consumption, execution, and monitoring. Service requests are tweeted to find the adequate service providers. The originator (i.e., the coordinator) of the service request first posts a Tweet that contains a concatenation of actions that need to be executed sequentially (in a pipeline), where each service passes its result to the next one. The control over the execution is distributed: upon completion of a service in the pipeline, the provider is responsible for tweeting the invocation of the next service. Since Tweetflows are created ad hoc, their structure can be dynamically adjusted. Finally, an automatic binding between Tweetflows and WSDL/SOAP descriptions of human-provided services (HPS) can be generated (semi-)automatically.

2.10.5 Contribution Evaluation

We evaluated our approach by implementing several prototypes for the iOS and Android platform.

2.10.6 Relation to the Research Framework

This contribution is related to the JRA-2.2 research challenge QoS Aware Adaptation of Service Compositions, where the QoS aspect is mostly related to the capability and availability of human participants to perform the required service, but may also reflect the dynamic nature of the crowdsourcing environment in which it runs.

2.10.7 Conclusions

We have presented an approach for the application of SOA principles on mobile Apps. Specifically, we have investigated the mapping of existing infrastructure (Apps, App Store) to SOA concepts like Service, Service registry and Service consumer. After establishing the mapping, we have introduced a lightweight communication schema (Tweetflows) that uses social network structures and provides for the integration of human-provided services. Our next steps will be the extension of the communication schema to support more complex processes and the investigation of the distributed execution of Tweetflows in crowd scenarios. Another area of future work is the implementation of intelligent message forwarding mechanisms for the crowd. If we want to move our approach beyond the personal character of mobile Service, we need to consider the crowd as a whole as a Service provider network. If a message is sent to the crowd, the message needs to be automatically forwarded to other users, without any need for user intervention. To increase the probability of reaching the intended service, we require a directed message forwarding, based on heuristics with Twitter user profile data.

2.11 A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules


Partners: UCBL, UPD, PolMi

Status: Published in at ICSOC 2011, December 2011.
2.11.1 Background

While selecting services, the Quality of Service (QoS) guarantees are commonly defined in Service Level Agreements (SLAs) between provider and consumer of services. Such guarantees are often violated due to various reasons. QoS violation requires a service adaptation and penalties have to be associated when promises are not met. However, there is a lack of research in defining and assessing penalties according to the degree of violation. The need is to apply penalties for QoS violations when partial satisfaction occurs.

2.11.2 Problem Statement

The contribution addresses the problem of applying penalties for QoS violations when partial satisfaction occurs. In fact, QoS guarantees defined in contracts may be violated due to various reasons. This situation needs to be handled through applying adaptation techniques not to bring dissatisfaction. The concept of penalty has been used in SLAs to compensate the conditions under which guarantee terms are not met. Despite some research have been done on the description, negotiation and monitoring of SLAs, however there is not much work on the definition of penalty clauses. studied on WS-Agreement specification to define penalties based on different types of violation. However, penalties are assigned to violation of a single property instead of assigning penalties to violation of overall QoS. Moreover, the approach introduces a method for measuring penalties which is for a predefined number of violations, instead of measuring the extent of violation and assigning penalties accordingly. One main issue is how to determine the appropriate amount of penalties as compensations from providers to satisfy customers. As quality parameters can be satisfied partially, the assessment of penalties can be based on the degree of quality violation. Understanding the violation degree is a prerequisite for assessing penalties.

2.11.3 Contribution Relevance

As quality parameters can be satisfied partially, the assessment of penalties can be based on the degree of quality violation. Understanding the violation degree is a prerequisite for assessing penalties. However, measuring such violation is yet an open research challenge. In addition, the influencing factors in defining penalties need to be identified. A static amount of penalty (manual approaches) does not reflect the extent of violation at runtime. The amount and level of penalties are related to the degree of quality violation provided from the provider side. On the other side, the customers characteristics may also affect the amount of penalties. For example a penalty to satisfy a gold/loyal customer is different with the one for an occasional customer. To the best of our knowledge, there is no formal relation between the assigned penalty and its influencing factors. The authors are describing the relation my means fuzzy logic.

2.11.4 Contribution Summary

The goal of this work is to apply an inference technique using fuzzy logic as a solution to propose a penalty-based approach for compensating conditions in which quality guarantees are not respected. Fuzzy logic is well suited for describing QoS and measuring quality parameters. We demonstrate a penalty inference model with a rule based mechanism applying fuzzy set theory. Measuring an appropriate value for penalties with respect to the amount of violation is the main contribution of the paper.

2.11.5 Contribution Evaluation

We have simulated our approach in a simulator based on fuzzy inference system. Initial membership functions were designed based on the contract in the motivating example and fuzzy rules are defined by the expert of the system. Figure 2c illustrates membership function for QoS violation. Having defined the QoS violation, we measure the extent of penalties taken into account the state of customers and previously applied penalties for the same service. For this, fuzzy rules are defined considering...
all three influencing factors. Some figures depict fuzzy rules for penalty based on QoS violations, customer’s state and service status with respect to previous penalties which are defined by the service-state parameter represented by fuzzy set.

2.11.6 Relation to the Research Framework

This contribution is directly related to the WP-JRA-2.2 research challenge *Analysis and Prediction of Quality Characteristics of Service Compositions*, and is also related to the research challenge *QoS Aware Adaptation of Service Compositions*.

2.11.7 Conclusions

Applying penalties is a complex research issue in service-oriented computing which has not been paid enough attention in the literature. In this work, we elaborated the concept of penalty and propose a mechanism for modelling and measuring penalties. Penalties are modelled using a fuzzy approach and applying fuzzy set theory. The relation between penalties and their influencing factors are defined by fuzzy rules through an inference method. We have demonstrated the proposed penalty model through a motivating example and performed some initial result in measuring penalties.
Chapter 3

Conclusions

3.1 Summary

The focus of this deliverable is to present mechanisms for QoS-aware, coordinated service compositions. The contributions presented in it address that problem from several perspectives, and using different approaches, including:

- Detecting defects in realizability of service choreographies and providing diagnostics at design-time;
- Inferring user-defined attributes of orchestration activities and complex control constructs based on the notion of data sharing and an automated static analysis;
- Deriving logical specifications for service compositions from component specifications;
- Using aspect-based fragment substitution to prevent SLA violations based on machine learning predictions;
- Deriving adaptation strategies based on prediction of KPI violations in service compositions;
- Automatic composition of service fragments based on contextual requirements;
- Predicting future contexts for service compositions based on a learning model;
- Providing a framework for context-aware business process evolution;
- Proposing an architecture and a methodology for a green business process reengineering;
- Using Twitter to organize and execute (human provided) service workflows in mobile, distributed, and dynamic project teams; and
- Inferring the amount of penalties that the providers need to compensate to the service users in cases of SLA violations.

The results presented in this deliverable address the following research challenges in WP-JRA-2.2:

- Formal Models and Languages for QoS-Aware Service Compositions;
- Monitoring of Quality Characteristics of Service Orchestrations and Service Choreographies;
- Analysis and Prediction of Quality Characteristics of Service Compositions; and
- QoS Aware Adaptation of Service Compositions.

Several contributions are related to other work-packages, most notably WP-JRA-1.3 and WP-JRA-1.2, and address their corresponding research challenges.
3.2 An Outline of Future Work

The work presented in this deliverable extends the previous work within this workpackage on algorithms and techniques for splitting and merging service compositions (deliverable CD-JRA-2.2.3), on models, mechanisms and protocols for coordinated service compositions (deliverable CD-JRA-2.2.4), and on derivation of QoS for services and service compositions (deliverable CD-JRA-2.2.5). While this deliverable is the last in the S-Cube series in JRA-2.2, the following directions for the outline of the future work by the partners as well as by the wider academic and industrial community have been identified:
Bibliography


Appendix A

Attached Papers
1 Introduction

Service choreographies are technical contracts which specify the message-based interactions among collaborating parties, called participants. Figure 1 introduces the basic terminology related to service choreographies. The ordering and timing of the message-based interactions that a participant is tasked with performing in a choreography is called role. The participants implement in participant implementations the messaging behaviors required by the roles they are assigned. An enactment is the cumulative execution of the participants implementations and the resulting message exchanges. When their participant implementations collectively perform an enactment, the participants are said to enact the choreography.

There are multiple paradigms for specifying service choreographies, the most prominent of which are interconnection and interaction. In interconnection choreographies, each role is modeled explicitly as a participant skeleton [DKLW07, DKB08], i.e. an orchestration – possibly non-executable – that models the part of a possible internal behavior of the participant limited to the generation and consumption of messages. The participant skeletons are “wired” together by means of the message exchanges. Since each role is specified separately, interconnection choreographies may suffer from deadlocks that lead some of their enactments to get “stuck,” i.e. the enactments reach states in which the participants cannot perform the actions required for the enactments to progress [Loh08].

The interaction paradigm for modeling choreographies foresees the specification of the messaging occurring among the participants from a global perspective, e.g. as a workflow in which the message exchanges among the participants are modeled as activities. Languages that model choreographies using the interaction paradigm are, among others, the Business Process
Model and Notation (BPMN v2.0) [OMG11] (choreography diagrams) and the Web Services Choreography Description Language (WS-CDL) [W3C05]. Since the ordering and timing of the message exchanges are specified from a global perspective, it is surprisingly simple to specify choreographies that cannot be correctly enacted by their participants. Consider for example a choreography that specifies that the participant \( p_1 \) sends a message of type \( m_1 \) to \( p_2 \), and immediately after \( p_3 \) sends a message of type \( m_2 \) to \( p_4 \). The opposite order, namely first the dispatching \( m_2 \) and only then \( m_1 \), is not allowed by the choreography. Assume that the choreography is self-contained, i.e. it does not allow other communication among its participants except what is explicitly modeled as message exchanges, and that the message exchanges are performed in secrecy, i.e. that only senders and recipients have access to the content of the message and know that the message exchange has occurred. Since \( p_3 \) does not partake the message exchange that delivers \( m_1 \), then \( p_3 \) has no means of knowing when it is expected of sending \( m_2 \).

A choreography that is specified so that its participants cannot play their roles due to similar issues is said to be unrealizable.

Realizability is a fundamental property of interaction choreographies [MPR09]. In a sense, an unrealizable interaction choreography fails its purpose: it specifies a distributed messaging behavior that cannot be enacted accurately by its participants. In this work we present a method for the analysis of the realizability of interaction choreographies based on the concept of participant awareness. In a nutshell, the participant awareness is a symbolic representation of what a participant “knows” of the global state of the enactments it partakes. The realizability analysis presented in this work is specified on the basis of ChorTex, a choreography modeling language.
Based on process algebras. We adopt ChorTex instead of an already existing modeling language to investigate the impact in terms of realizability of constructs that interrupt the enactment of others. In particular, ChorTex has exception throwing and handling constructs that are very similar to those available in orchestration languages like the Business Process Execution Language for Web Services (WS-BPEL), and that can be used to realize interrupting, event-based constructs like termination end events in BPMN v2.0. Our finding is that, due to the distributed nature of choreography enactments, modelers must use such constructs with extreme care.

Figure 2 provides an overview of the realizability analysis presented in this work. First, a Control Flow Graph (CFG) – a well-known data-structure often used in compilers theory [Lou97] – is generated from the choreography. Such CFG represents the events that occur in an enactment, e.g. the beginning or completion of an activity, and their ordering. In the second step, the nodes of the CFG are annotated with information on the participant awareness, which is a symbolic representation of which events can each participant observe, and that is calculated using techniques borrowed from the field of control-flow analysis in programming languages. A CFG whose nodes are annotated with the awareness of the participants is called Awareness Model (AWM). Finally, the realizability of the choreography is tested by verifying constraints on the awareness of the participants that, if verified, guarantee that the participants are able to play by their roles as specified.

This work is structured as follows. Section 2 presents ChorTex, the choreography modeling language that we employ in this work. In Section 3 we provide the necessary background on choreography realizability and state the definition of choreography realizability that we verify on ChorTex choreographies. Section 4 shows how to construct CFGs of ChorTex choreographies. Section 5 elaborates the concept of awareness and presents the algorithm to annotate nodes of a CFGs with the participant awareness, thus creating AWMs. Section 6 presents how the realizability constraints are specified and verified on AWMs. Finally, Section 7 concludes the work by discussing the proposed method in light of the related work and our findings on the impact of interrupting constructs in terms of realizability.
2 ChorTex: A Choreography Modeling Language

ChorTex is based on Chor [YZCQ07], a choreography modeling language providing mechanisms for exception handling inspired by WS-BPEL. The syntax and operational semantic of ChorTex are presented in Section 2.1 and Section 2.2, respectively. The details and rationale of the differences between ChorTex over Chor are discussed in Section 2.3.

2.1 Syntax and Overview of ChorTex

The syntax of ChorTex is presented in Figure 3 by means of the Xtext Domain Specific Language (DSL)\(^1\). The syntax of the Xtext DSL is similar to Backus-Naur Form (BNF) grammars, with the addition of the possibility of naming non-terminal symbols. For example, consider the following rule:

\[
\text{BlockActivity}: \{ [ \text{name=ID} ] \text{ activities } + = \text{ Activity} ( ; ; \text{ activities} + = \text{ Activity} ) * \} \]

The example above means that the non-terminal symbol Block has at its two ends open and closed brackets, namely the ‘{’ and ‘}’ literals, which surround one or more productions of the non-terminal symbol Activity, with every two contiguous productions separated by the literal ‘;’. The productions of the non-terminal symbol Activity are grouped under the name of “activities” using the += operator, which concatenates a terminal to a list of terminals. In Figure 3, the symbols +, ? and * represent cardinalities, meaning that the groups preceding them (groups are delimited by parentheses) have to appear at least one, zero or one times, or any number of times, respectively.

A choreography specifies a body which represents the “normal flow” of activities, and exception handlers which specify the activities to be enacted in reaction of exceptions propagating from the body. If the body successfully completes, i.e. no exception is raised during the body’s enactment, the enactment of the choreography successfully completes as well. On the contrary, if the enactment of the body results in an exception of type \(e\) been thrown, the exception handlers of the choreography are matched against the type of the exception that is thrown. The body of an exception handler consists of the activities to be enacted when that exception handler is triggered. In ChorTex there are two types of exception handlers: named and default.

A named exception handler catches exceptions of one single type. Unlike Java and similarly to WS-BPEL, exception types in ChorTex do not have type hierarchy. Instead, the match between the type of the propagating exception and the one declared by the named exception handler is literal, i.e.

---

\(^1\)Xtext (www.eclipse.org/Xtext/) is an Model-Driven Architecture (MDA) framework based on Eclipse for specifying textual DSLs. A guide to the syntax of Xtext’s grammar can be found at: http://www.eclipse.org/Xtext/documentation/
Choreography: 'chor' ([ name=ID ])? (' body=Activity (' nameExceptionHandlers+=NamedExceptionHandler)* (' defaultExceptionHandler=DefaultExceptionHandler)? ')';

NamedExceptionHandler: exceptionType=ID ':' body=BlockActivity;

DefaultExceptionHandler: ' * ' body=BlockActivity;

Activity: BasicActivity | ComplexActivity;

BasicActivity: SkipActivity | MessageExchangeActivity | OpaqueActivity | ThrowActivity;

ComplexActivity: BlockActivity | ChoiceActivity | IterationActivity | ParallelActivity | Choreography;

SkipActivity: 'skip' ([ name=ID ])?;

MessageExchangeActivity: ([ name=ID ])? sender=ID '→' messageType=ID 'to' recipients+=ID (',' recipients+=ID)*;

ThrowActivity: 'throw' ([ name=ID ])? exceptionType=ID;

OpaqueActivity: 'opaque' ([ name=ID ])? (' participants+=ID (',' participants+=ID)+ ');

BlockActivity: ([ name=ID ])? (' activities+=Activity (',' activities+=Activity)* ');

ChoiceActivity: 'choice' ([ name=ID ])? decisionMaker=ID 'either' branches+=BlockActivity (or' branches+=BlockActivity)+;

IterationActivity: 'iteration' ([ name=ID ])? decisionMaker=ID 'do' body=BlockActivity;

ParallelActivity: 'parallel' ([ name=ID ])? 'do' branches+=BlockActivity (and' branches+=BlockActivity )+;

Figure 3: The syntax of ChorTex expressed using the Xtext DSL.
matching string-wise the names of the two exception types. Default exception handlers can handle any type of exception, and intercept any exception propagating from the body that is not otherwise caught by named exception handlers. In other words, named exception handlers have precedence on the default one when determining which will catch an exception. If no exception handler (neither named nor default) for an exception is found, the choreography terminates and propagates the exception to its parent activity (if any is specified). When a choreography is terminated, its body (or the currently running exception handler) is terminated “on cascade.” When an exception propagates outside the root choreography, the entire enactment is terminated. If an exception handler matching the thrown exception is found, its body is enacted. If the body of the exception handler completes, i.e. the exception has been dealt with, the enactment of the choreography completes successfully. However, a body that is terminated because of exceptions propagating from it cannot be “resumed” after that the exception has been handled. Otherwise, if the enactment of body of the exception handler results in another exception been thrown, the choreography terminates propagating this last exception to its parent activity (if the choreography was nested into another), or terminating the enactment if the choreography is the root one.

The actual participants of a choreography, i.e. the entities such as services or individuals, are specified at design time in the choreography. We assume that each participant “knows” all the others, and that their identifiers are sufficient information for dispatching messages to them. Moreover, we assume that each participant is given “a copy” of the choreography, which is used as an artifact in the software development process of the participant implementation.

When the message exchange $p_s \rightarrow m$ to $p_{r_1}, \ldots, p_{r_n}$ is enacted, the participant $p_s$ sends a message of type $m$ to the participants $p_{r_1}, \ldots, p_{r_n}$. The participant that dispatches the message is called \textit{sender}, i.e. $p_s$ in the previous example. The participants that receive the message are called \textit{recipients}. A participant can not act as a sender and recipient in the same message exchange.\footnote{This assumption simplifies the realizability analysis presented in Section 5 without sacrificing the expressiveness of ChorTex. After all, a message sent by a participant to itself is more an \textit{internal action} than a proper message exchange.}

The type of the message that is exchanged over a message exchange is uniquely identified using an identifier like $m$. We assume that a participant can uniquely identify the type of a message by observing the latter’s content. In the scope of Service Oriented Architecture (SOA) technologies, this is a realistic assumption: messages can include meta-data like information on their type, e.g. as Simple Object Access Protocol (SOAP) headers. This assumption allows the recipients of a message to “trace back” which message exchange has been enacted solely on the basis of the message they
have received.

In this work we do not consider synchronous message exchanging, as it is not realistic in the scope of Service-Based Applications (SBAs), i.e. distributed systems realized on the basis of SOA tenets. Instead, the message exchanges among the participants are asynchronous. We assume the asynchronous communications among the participants to have the following capabilities:

**In-order reception:** A participant receives messages in the same order in which they are sent;

**Exactly-once reception:** A message dispatched once by the sender is received exactly once by each of its recipients;

**Always successful:** No messages are lost or recipients are unreachable;

**No message corruption:** The sender and recipients of a message see exactly the same content;

These assumptions may sound strong; however, they are feasible in current state of the art of SOA through the adoption of technologies like WS-ReliableMessaging [FPD+09] (using “exactly-once” and “in-order” policies), and high-availability features of modern messaging services. It is not relevant in the scope of this work how asynchronous messaging is actually realized, e.g. if recipients have queues and which is their size. Due to the asynchronous nature of message exchanges in ChorTex, the recipient of a message exchange will receive and consume the message at some point in the future after its delivery. There is no guarantee about exactly when the message reception and consumption by one recipient will occur. More specifically, there is no guarantee that multiple recipients of a message will consume it at the same time, nor in any particular order (e.g. the participant $p_i$ before or after $p_j$).

The skip activity is the “empty” activity. The enactment of a skip activity involves no actions performed by the participants and it always completes successfully and instantaneously.

The activity **opaque** $(p_1, \ldots, p_n)$ represents an unspecified part of the choreography that involves the participants $p_1, \ldots, p_n$. That is, an opaque activity is a “free form” activity that, when enacted, allows its participants to engage in any amount and ordering of message exchanges. The particular message exchanges to be performed and their order can be specified (1) later in the modeling of the choreography, or (2) at run-time by the participants that partake that opaque activity in the fashion of ad-hoc modeling [AtHEvdA06, WRRM08]. Irrespective of which of the two options is adopted, the participants of the opaque activity agree on its completion. In other words, all participants are assumed know when the enactment of the opaque activity is completed. It is outside the scope of this work to
specify how the participants achieve this. This provision is necessary for the soundness of the operational semantics of ChorTex. The actual mechanisms that the participants employ to achieve this agreement on the completion of the opaque activity is outside the scope of this work. For example, the participants might have an agreed-upon protocol that is enacted in place of the opaque activity.

The enactment of the activity throw e results in an exception of type e being thrown. The exception handling mechanisms in ChorTex are control-flow constructs for specifying the interruption of the enactment of concurrent activities, and the triggering of others as a result. Unlike programming languages like Java or orchestration languages like WS-BPEL, an exception thrown while enacting a ChorTex choreography is not represented by a data-structure. In a nutshell, the throwing of an exception represents a “jump” in the enactment of the choreography and in possible interruption of some of the activities that are currently been enacted.

The block activity \{ A_1,\ldots,A_2 \} denotes the sequential enactment of the activities \( A_1,\ldots,A_2 \). The completion of the first activity triggers the enactment of the second, and so on until all activities have been completed. If the enactment of one activity results in an exception being thrown, the next activities (if any) are not enacted and the exception is propagated.

The construct parallel do \( A_1 \) and \( \ldots \) and \( A_n \) specifies the concurrent enactment of the branches \( A_1,\ldots,A_n \). For simplicity, we assume each branch to be specified as a block; of course, however, a block representing a branch may very well contain just a single activity. A parallel activity completes successfully when all its branches have completed successfully. If the enactment of one of the branches results in an exception been thrown, the other branches that have not yet completed are immediately terminated, the parallel activity is itself terminated, and the exception is propagated to the parallel activity’s parent.

The construct choice p either \( A_1 \) or \( \ldots \) or \( A_n \) models the conditional choice (i.e. “if then else”). The decision about which of the \( A_1 \ldots A_n \) activities, called branches, is enacted is taken internally by the participant \( p \), which is said to be the decision maker. Since the decision is internal, the choice construct does not specify the criteria used by the decision maker for deciding which branch is executed. Is important to notice that there is no “visible” proof of the outcome of the decision maker’s decision. The other participants must understand which branch is enacted by observing what happens after the decision maker has taken the decision, e.g. by observing which message exchanges take place thereafter.

Finally, the iteration activity iteration p do \( A \) specifies the repeated enactment (i.e. the “while-do”) of the activity \( A \), which is said to be the body of the iteration. Similarly to the choice construct, the decision whether to iterate again the activity is taken internally by the decision maker \( p \). If an exception propagates from the body, the iteration activity is terminated
and the exception is propagated to its parent.

2.2 Operational Semantics of ChorTex

Before detailing the operational semantics of ChorTex (Section 2.2.2), we need to lay some groundwork (Section 2.2.1).

2.2.1 Basic Definitions

During the enactment of a choreography, its participants perform actions such as the dispatching of a message or deciding which branch of a choice activity to enact.

**Definition 1** (Actions and Acting Participants). The participants that perform a certain action are its acting participants. Table 1 correlates the various types of actions that are performed in ChorTex choreographies with their acting participants.³

<table>
<thead>
<tr>
<th>Action</th>
<th>Acting Participants</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{mex}]) (p_s \stackrel{m}{\rightarrow} p_r_1,\ldots,p_r_n)</td>
<td>(p_s)</td>
<td>Dispatching of the message (m) by the sender (p_s) to the recipients (p_r_1,\ldots,p_r_n) when enacting the message exchange activity (\text{mex})</td>
</tr>
<tr>
<td>([\text{o}]) (\langle p_1,\ldots,p_n \rangle)</td>
<td>(p_1,\ldots,p_n)</td>
<td>Enactment of the opaque activity (o) by (p_1,\ldots,p_n)</td>
</tr>
<tr>
<td>([\text{c}]) (p \xrightarrow{x} x \in {A_1,\ldots,A_n})</td>
<td>(p)</td>
<td>The decision maker (p) decides which of the branches (A_1,\ldots,A_n) of the choice activity (c) must be enacted</td>
</tr>
<tr>
<td>([\text{i}]) (p \xrightarrow{x} x \in {\top,\bot})</td>
<td>(p)</td>
<td>The decision maker (p) decides whether to iterate the body of the iteration activity (i); (\top) and (\bot) denote “true” and “false”, respectively</td>
</tr>
</tbody>
</table>

Table 1: The types of actions and corresponding acting participants.

Not all possible enactment traces are valid for a certain choreography:

**Definition 2** (Enactment Traces). An enactment trace \(\langle a_1,\ldots,a_n \rangle\) is a sequence of actions \(a_1,\ldots,a_n\) performed collectively by the participants during an enactment of a choreography. The concatenation of enactment traces is performed through the operator \(\circ\), which is defined as follows:

\[
\langle a_1,\ldots,a_m \rangle \circ \langle a'_1,\ldots,a'_n \rangle = \langle a_1,\ldots,a_m,a'_1,\ldots,a'_n \rangle
\]

The concatenation implies a temporal order between two traces that are united together. In particular, all the actions specified by the first trace have been enacted before those of the second trace.

Not all possible enactment traces are valid for a certain choreography:

**Definition 3** (Valid Enactment Traces). An enactment trace \(\sigma\) is valid for a choreography if the participants do not violate that choreography by performing in order the actions specified by \(\sigma\).

³ is a weather symbol representing fog; it seems fitting, since “what happens” during the enactment of an opaque activity is not visible in the enactment trace.

10
The validity of an enactment trace is verified by “simulating” on the choreography the sequence of actions it specifies by applying the operational semantics rules specified in Figure 14 and Figure 15. Notice that, is a choreography specifies iteration activities, there may possibly be infinitely many valid enactment traces.

The initiating actions of a choreography are those actions that, when performed by their acting participants, can “kick-start” enactments of that choreography. (It is important to differentiate between initiating and non-initiating actions because the former are treated differently than the latter in terms of the realizability analysis, see Section 6.1.)

**Definition 4 (Initiating Actions).** An action is initiating if its performing by the acting participants may cause the initiation of an enactment.

Intuitively, the possible initiating actions of a choreography are all those that can appear as first in a valid enactment trace. Notice that a choreography may have more than one initiating action. Consider, for example, the simple choreography shown in Figure 4. Since the activities that specify actions, namely the message exchange activities $mex_1$ and $mex_2$, are located in branches of the parallel activity, $prl$, both $mex_1$ and $mex_2$ can initiate an enactment of the choreography.

```plaintext
chor [chor1] (  
  [b] {  
    parallel [prl] do [b1] {  
      [mex1] p1 → m1 to p2  
    } and [b2] {  
      [mex2] p2 → m2 to p1  
    }  
  }  
)
```

Figure 4: An example of choreography with multiple initiating actions.

This choreography requires both the message exchange activities $mex_1$ and $mex_2$ to be performed in order for an enactment to complete. Assume that at some point in time, first $p_1$ dispatches $m_1$ to $p_2$, and then $p_2$ dispatches $m_2$ to $p_1$. But does the dispatching of $m_2$ by $p_2$ represent the initiation of a new enactment, or just the completion of the one initiated by $p_1$ with the dispatching of $m_1$? To solve this dilemma, we assume that participants employ a session mechanism that allows them correlate message exchanges (but, more generally, action) to enactments. Fundamentally, each enactment is uniquely identified by an identifier generated upon the performing of the initiating action, and that is later included in the meta-data of each message exchanged between the participants. Similar techniques are used for Hypertext Transfer Protocol (HTTP) session...
tracking and in distributed transaction management. In the current SOA landscape, such session mechanism is easily realized by means of, for example, WS-Addressing [BCC+06].

The enactment state is defined in terms of the enactment trace, i.e. the “history” of the enactment up to that moment, the state of each of the activities specified by the choreography (see the activity life-cycles in Figure 5 through Figure 13), and the enactment mode, i.e. whether an exception is been propagated or, on the contrary, the enactment is “proceeding normally”.

**Definition 5** (Enactment States). The state $\chi$ the enactment of a choreography is a tuple:

$$\chi := \left[\delta, \sigma, \mu\right]$$

The symbol $\delta$ denotes the enactment environment, i.e. a key-value map of the states of the activities specified by the choreography. The keys of the enactment environment are the activities specified by the choreography, and the respective values are the activities’ own states (specified later in Section 2.2.2). The fact that the activity $A$ is in the state $s$ is denoted by:

$$\delta[A] = s$$

The symbol $\sigma$ represents the enactment trace. Finally, $\mu$ denotes the enactment mode, i.e. if the enactment is being enacted “normally”, denoted by $\checkmark$, or else an exception $e$ is been propagated, denoted by $\check{\ } e$.

### 2.2.2 Operational Semantics of ChorTex Activities

This section presents the operational semantics of ChorTex activities as both State Diagrams (SDs) of ChorTex elements (Figure 5 through Figure 13), as well as structured operational semantics (Figure 14 and Figure 15). While semantically equivalent, these two different representations have different readability and intended use. On one hand, structured operational semantics has been widely adopted to describe the meaning of process algebras. SDs of the life-cycle of ChorTex elements, on the other hand, are easier to read and provide a perspective on the enactment of a ChorTex choreography focused on the single choreography elements.

Often, approaches to structured operational semantics, the progress of the enactment is tracked by “rewriting” the choreography specification removing the activities that have already been completed. On the contrary, the operational semantics of ChorTex presented in Figure 14 and Figure 15 adopts a “state-based” style, representing the current progress of the enactment as a combination of the states of the single activities. The reason for the adoption of this style is that the resulting operational semantics mirrors closely the life-cycles of the activities that, in our opinion, are far
Figure 5: The enactment life-cycle of a skip activity.

Figure 6: The enactment life-cycle of a message exchange activity.

Figure 7: The enactment life-cycle of an opaque activity.

Figure 8: The enactment life-cycle of a throw activity.

Figure 9: The enactment life-cycle of a block.
Figure 10: The enactment life-cycle of a choice activity.

Figure 11: The enactment life-cycle of an iteration activity.

Figure 12: The enactment life-cycle of a parallel activity.

Figure 13: The enactment life-cycle of a choreography.
more intuitive to the reader familiar with workflows and service composition languages than “mainstream” structured operational semantics.

In Figure 14 and Figure 15, the symbols $p$ denote participant identifiers, $m$ message types and $e$ exception types. The assignment of the state $s$ to the activity $A$ is denoted by:

$$\delta[A\backslash s]$$

The symbols that represent the states of the activities are the same adopted in the activities’ life-cycles depicted in Figure 5 through Figure 13.

A skip activity, the life-cycle of which is shown in Figure 5, is instantaneous (see Section 2.1). This is represented in Rule Skip, which can be read in natural language as follows: “as soon as the activity skip is initiated (i.e. its state is $\epsilon$), it completes successfully (i.e. its state becomes $\delta$)”.

Notice that skip activities can be enacted only when the enactment mode is “normal” (denoted by $\checkmark$). However, since skip activities are instantaneous, when they are initiated can never be interrupted by exception propagation.

The life-cycle of message exchange activities is shown in Figure 6. When its state is initiated and the enactment mode is “normal,” the message exchange activity enters the state Pending, denoted by the hourglass-like symbol $\sqsubseteq$ (Rule Message Exchange Initiation). In “normal” enactment mode, the message exchange activity performs the transition from the Running to the Completed state when the sender dispatches the message to the recipients (Rule Message Exchange Completion). Notice that, since the (Rule Message Exchange Completion) assumes the enactment mode to be “normal”, participants that dispatch messages when the enactment state in “exception propagation” mode ($\forall e$) are violating the choreography. After a message exchange activity enters the state Pending, its completion is not necessarily instantaneous. Instead, the message exchange becomes enactable, meaning that, if the sender dispatches the message to the recipients, the choreography is not violated. In other words, the sender of a message exchange can delay the dispatch of the message indefinitely, thereby “stalling” the enactment of the message exchange activity. There are no restrictions on how long can the sender delay the completion of the message exchange. Since message exchange activities are not instantaneous, they may be terminated while they are in the state Pending. Notice that there is no rule that represents in the specific the termination of a message exchange activity. Instead, the termination of a message exchange activity – i.e. the changing of its state to Terminated, denoted by $\times$ – occurs while executing a rule that processes the termination of the complex activity that is parent to that message exchange activity. In particular, the rules that, when executed, can result in the termination of a nested message exchange activity are those for the termination of choreographies (Rule Choreography Termination 1, Rule Choreography Termination 2 and Rule Choreography Termination 3), blocks (Rule Block Termination), choice ac-
In the reminder, \( s \) denotes: \( \text{skip} \)

\[
\delta[s] = \text{skip}
\]

\[
\delta[\text{Block Next Activity Initiation}] = \begin{cases} \delta[\text{Block Activity Initiation}] & \text{if } n < \text{block activity count} \\ \delta[\text{Block Completion}] & \text{if } n = \text{block activity count} \end{cases}
\]

In the reminder, \( m \) denotes: \( p_1 \rightarrow p_{i+1} \rightarrow \ldots \rightarrow p_n \)

\[
\delta[m] = \begin{cases} \delta[\text{Block Initiation}] & \text{if } n < \text{block initiation count} \\ \delta[\text{Block Exception}] & \text{if } n = \text{block initiation count} \end{cases}
\]

\[
\delta[c] = \begin{cases} \delta[\text{Choice Initiation}] & \text{if } p \text{ picks } A_i \text{ out of } A_1, \ldots, A_n \\ \delta[\text{Choice Completion}] & \text{if } p \text{ enacts } A_i \end{cases}
\]

\[
\delta[e] = \begin{cases} \delta[\text{Choice Initiation}] & \text{if } e \text{ either } A_1 \text{ or } \ldots \text{ or } A_n \\ \delta[\text{Choice Completion}] & \text{if } e \text{ enacts } A_i \end{cases}
\]

\[
\delta[\text{Throw}] = \text{throw } e
\]

\[
\delta[\text{Block Initiation}] = \begin{cases} \delta[\text{Block Activity Initiation}] & \text{if } n < \text{block activity count} \\ \delta[\text{Block Completion}] & \text{if } n = \text{block activity count} \end{cases}
\]

\[
\delta[\text{Choice Initiation}] = \begin{cases} \delta[\text{Choice Initiation}] & \text{if } p \text{ picks } A_i \text{ out of } A_1, \ldots, A_n \\ \delta[\text{Choice Completion}] & \text{if } p \text{ enacts } A_i \end{cases}
\]

\[
\delta[\text{Throw}] = \text{throw } e
\]

\[
\delta[\text{Block Initiation}] = \begin{cases} \delta[\text{Block Activity Initiation}] & \text{if } n < \text{block activity count} \\ \delta[\text{Block Completion}] & \text{if } n = \text{block activity count} \end{cases}
\]

\[
\delta[\text{Choice Initiation}] = \begin{cases} \delta[\text{Choice Initiation}] & \text{if } p \text{ picks } A_i \text{ out of } A_1, \ldots, A_n \\ \delta[\text{Choice Completion}] & \text{if } p \text{ enacts } A_i \end{cases}
\]

\[
\delta[\text{Throw}] = \text{throw } e
\]

Figure 14: The structured operational semantics of ChorTex (Part 1).
In the reminder, $i$ denotes: iteration $p$ do $A$

$$\delta[i] = \epsilon$$

$$[\delta, \sigma, \checkmark] \rightarrow [\delta[i], \sigma, \checkmark]$$

(Iteration Initiation)

$$[\delta, \sigma, \checkmark] \rightarrow [\delta[i] \triangleright A | \forall p \ni \epsilon \rightarrow \top, \checkmark]$$

(Iteration Decision True)

$$[\delta, \sigma, \checkmark] \rightarrow [\delta[i], \sigma \circ \{p \ni \top\} | \checkmark]$$

(Iteration Decision False)

$$[\delta[i] = \epsilon \land \delta[A] = \emptyset, \checkmark]$$

(Iteration Body Completion)

$$[\delta[i] = X \land \delta[A] \neq \emptyset \land \epsilon]$$

(Iteration Termination)

In the reminder, $prl$ denotes: parallel do $A_1$ and \ldots and $A_n$

$$\delta[prl] = \epsilon$$

(Parallel Initiation)

$$[\delta, \sigma, \checkmark] \rightarrow [\delta[prl] \land \delta[A_1 | \forall p \ni \emptyset, \checkmark] \ldots | \forall p \ni A_n, \checkmark, \checkmark]$$

(Parallel Completion)

$$[\delta, \sigma, \checkmark] \rightarrow [\delta[prl] \land \forall i \in [1, n] : \delta[A_i] = \emptyset, \checkmark]$$

(Parallel Completion)

$$[\delta[prl], \epsilon] \rightarrow [\delta[prl] \land \{A_1, \ldots, A_n\}, \sigma, \checkmark]$$

(Parallel Completion)

In the reminder, $chor$ denotes: $\text{chor}\ (A | e_1 : A_1 | \ldots | e_n : A_n | \epsilon : A_\epsilon)$

$$\delta[chor] = \epsilon$$

(Choreography Initiation)

$$[\delta, \sigma, \checkmark] \rightarrow [\delta[chor] \land \delta[A | \forall p \ni \emptyset, \checkmark] \ldots | \forall p \ni A_n, \checkmark, \checkmark]$$

(Choreography Completion)

$$[\delta, \sigma, \checkmark] \rightarrow [\delta[chor] \land \forall i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception 1)

$$[\delta[chor], \epsilon_i] \rightarrow [\delta[chor] \land \{A_1, \ldots, A_n\}, \sigma, \checkmark]$$

(Choreography Exception 2)

$$[\delta[chor], \epsilon_i] \rightarrow [\delta[chor] \land \{A_1, \ldots, A_n\}, \sigma, \checkmark]$$

(Choreography Exception 3)

$$[\delta[chor], \epsilon_i] \rightarrow [\delta[chor] \land \{A_1, \ldots, A_n\}, \sigma, \checkmark]$$

(Choreography Exception 4)

$$[\delta[chor] = \epsilon, \checkmark \land \exists i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception Handling 1)

$$[\delta[chor] = \epsilon, \checkmark \land \exists i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception Handling 2)

$$[\delta[chor] = \epsilon, \checkmark \land \exists i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception Handling 3)

$$[\delta[chor] = \epsilon, \checkmark \land \exists i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception Handling 4)

$$[\delta[chor] = \epsilon, \checkmark \land \exists i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception Handling 5)

$$[\delta[chor] = \epsilon, \checkmark \land \exists i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception Handling 6)

$$[\delta[chor] = \epsilon, \checkmark \land \exists i \in [1, n] : \epsilon_i = \epsilon \land A_i = \emptyset, \checkmark]$$

(Choreography Exception Handling 7)

Figure 15: The structured operational semantics of ChorTex (Part 2).
activities (Rule Choice Termination), iteration activities (Rule Iteration Termination) and parallel activities (Rule Parallel Termination).

An initiated opaque activity changes its state to Running when its participants start enacting it (Rule Opaque Initiation). The participants can begin the enactment of an opaque activity only when the enactment mode is “normal” (Rule Opaque Initiation). When the participants complete an opaque activity, the latter changes its state to Completed (Rule Opaque Completion). The participants can complete the enactment of an opaque activity only when the enactment mode is “normal” (Rule Opaque Completion). No restrictions apply to the amount of time that its takes to the participants to complete an opaque activity. Thus, opaque activities are not instantaneous, and can be terminated. Similarly to the case of message exchange activities, there is no rule specific to the termination of opaque activities. Instead, the termination of an opaque activity – i.e. its state been changed to Terminated, denoted by $\mathcal{X}$ – occurs while executing a rule that processes the termination of the complex activity that is the parent of that opaque activity.

Throw activities, like skip ones, are instantaneous: they complete as soon as they are initiated (Rule Throw). The completion of a throw $e$ activity causes the enactment mode to change from “normal” (denoted by $\checkmark$) to “exception $e$ propagating” (denoted by $\nabla e$).

When a block is initiated, it initiates “on cascade” its first nested activity (Rule Block Initiation). A block can be initiated only when the enactment mode is “normal” (Rule Block Initiation). In a “normal” enactment mode, the completion of one nested activity that is not the last in the block triggers the initiation of the following nested activity (Rule Block Next Activity Initiation). In a “normal” enactment mode, the completion of the last nested activity results in the completion of the block (Rule Block Completion). If the enactment of one of the nested activities of a block results in an exception $e$ propagating, the block is terminated with the state $\nabla e$, i.e. the exception propagates to the block’s parent activity (Rule Block Exception). Finally, when a block is terminated by its parent activity, i.e. its state changes to $\mathcal{X}$, the nested activity currently running is also terminated (Rule Block Termination). The states of the previously completed nested activities are not affected by the termination (Rule Block Termination). Notice that the termination of any activity, blocks included, is fundamentally the result of an exception been thrown and not yet handled. Therefore, the termination of a block can occur only when the enactment mode is $\nabla e$.

Choice activities, once initiated, enter the state Pending (Rule Choice Initiation), which represents the “waiting” for the internal decision performed by the decision maker about which of the branches to enact. When the decision maker selects the branch $A_i$ to enact, the block that constitutes
that branch is initiated (Rule **CHOICE DECISION**) and the choice activity enters the state **RUNNING A**. When the branch selected by the decision maker completes, so does the choice activity (Rule **CHOICE COMPLETION**). If the selected branch results in an exception propagating, the state of the choice activity becomes **EXCEPTION PROPAGATION**, and the exception is propagated to its parent (Rule **CHOICE EXCEPTION**). A choice activity can be terminated by its parent while the decision is pending, or when the selected branch has not yet completed (Rule **CHOICE TERMINATION**).

Similarly to the case of choice activities, the enactment of an iteration activity begins with the transition of its state from **INITIATED** to **PENDING** (Rule **ITERATION INITIATION**), which represents the fact that the decision maker has not yet taken its decision on whether to execute once more the iteration’s body. The iteration activity completes if, while it is in the state **PENDING**, the decision maker decides not to iterate the body (Rule **ITERATION DECISION FALSE**). Otherwise, the iteration activity enters the state **RUNNING** and the body is initiated (Rule **ITERATION DECISION TRUE**). When the body completes, the iteration activity returns to the state **PENDING**, so that its decision maker can evaluate once more whether to iterate the body further (Rule **ITERATION BODY COMPLETION**). If the enactment of the body results in the propagation of an exception, the iteration activity enters the state **EXCEPTION PROPAGATION** and propagates the exception to its parent. If the iteration is terminated by its parent and its body is not yet completed, the body is also terminated “on cascade” (Rule **ITERATION TERMINATION**).

When a parallel activity is started, its state changes from **INITIATED** to **RUNNING** and all its branches are initiated (Rule **PARALLEL INITIATION**). A parallel activity completes when all its branches are completed (Rule **PARALLEL COMPLETION**). If the enactment of one the branches results in an exception propagating, the parallel activity enters the state **EXCEPTION PROPAGATION**, the other branches that have not yet been completed are terminated, and the exception is propagated to the parent of the parallel activity (Rule **PARALLEL EXCEPTION**). If the parallel is terminated by its parent, all the branches that have not yet completed are terminated as well (Rule **PARALLEL TERMINATION**).

When the enactment of a choreography begins, its state transitions from **INITIATED** to **RUNNING**, triggering the initiation of the choreography’s body (Rule **CHOREOGRAPHY INITIATION**). If the body completes, the enactment of the choreography completes (Rule **CHOREOGRAPHY COMPLETION**). On the contrary, if the enactment of the body results in the propagation of an exception of type e, the state of the choreography changes from **RUNNING** to **EXCEPTION RAISED** (Rule **CHOREOGRAPHY EXCEPTION 1**). When the choreography enters the state **EXCEPTION RAISED**, the exception handlers defined by the choreography are matched against the exception type e. If a named exception handler matches the type of the propagating exception,
its body is initiated (Rule CHOREOGRAPHY EXCEPTION 2) and the state of the choreography changes to EXCEPTION HANDLING. If no matching named exception handler is found, but the choreography defines a default exception handler, the latter’s body is initiated (Rule CHOREOGRAPHY EXCEPTION 3) and the state of the choreography changes to EXCEPTION HANDLING. If no matching named exception handler is found and the choreography defines no default exception handler (denoted by $A_\star = \epsilon$ in Rule CHOREOGRAPHY EXCEPTION 4), the choreography is terminated and the exception is propagated to its parent. If the choreography is the root choreography, and thus has no parent, the enactment itself is terminated. If an exception handler matching the type of the propagating exception was found, i.e. the choreography is in the state EXCEPTION HANDLING, and the body of that exception handler completes, then the choreography completes as well (Rule CHOREOGRAPHY EXCEPTION HANDLING 1). Otherwise, if the enactment of the exception handler’s body results in the propagation of an exception of type $e'$, the choreography’s state transitions to EXCEPTION PROPAGATION (denoted by $t e'$), and the exception is propagated to the choreography’s parent (Rule CHOREOGRAPHY EXCEPTION HANDLING 2). Similarly to the case when no matching handler for an exception can be found, if the choreography is the root choreography, the enactment is terminated. The termination of a choreography can occur when its body or that of one of the exception handlers is been enacted (Rule CHOREOGRAPHY TERMINATION 1, Rule CHOREOGRAPHY TERMINATION 2 and Rule CHOREOGRAPHY TERMINATION 3), which is immediately terminated.

2.3 Differences between Chor and ChorTex

At first sight, ChorTex adopts a more natural language-like syntax than its predecessor Chor [YZCQ07]. However, differences between Chor and ChorTex go deeper than just the syntax, and are treated in the reminder.

2.3.1 Opaque versus Internal Activities

Chor provides a construct called basic activity which allows to specify internal activities executed by single participants. (Chor’s basic activities are not to be confused with ChorTex’s: in ChorTex, the term “basic activity” is used to collectively denote activities that do not allow others to be nested in them, i.e. skip, message exchange, opaque and throw activities, see Figure 3.) The execution of Chor’s basic activities is instantaneous, infallible (i.e. it never results in an exception being thrown), and, since it does not generate message exchanges, is not registered in the enactment trace. The rationale of Chor’s basic activities in choreographies is not discussed in [YZCQ07]; presumably, basic activities are included in Chor because of its “twin” orchestration language Role. In fact, during the process of pro-
jection, a basic activity $a$ specified in the choreography is transformed into a Role activity in the role of the participant that executes $a$.

In our opinion, allowing the specification of internal activities in a choreography language focusing on interaction modeling is a questionable design decision. Interaction modeling of choreographies focuses on the global behavior of the choreography, detailing the public actions of the participants. Instead, Chor’s basic activities are internal – i.e. private – to the participants that execute them. Unlike the choice and iteration constructs, whose internal activities (the decisions) affect the enactment, basic activities do not influence the sequencing of activities in the choreography, and are therefore semantically void. Therefore, in ChorTex we substitute Chor’s basic activities with the opaque ones, which drop the assumption of instantaneous execution and must be enacted by multiple participants.

2.3.2 Choreography Nesting vs. Choreography Referencing

In Chor, choreographies are uniquely identified by name and can be referenced from inside other choreographies using the perf construct. When the perf $C_m$ activity is traversed, the choreography identified by $C_m$ is enacted. The completion of $C_m$ leads to the completion of perf $C_m$. Similarly, the termination of $C_m$ because of the propagation of an uncaught exception results in that exception propagating outside perf $C_m$. In ChorTex we replace choreography referencing with choreography nesting. Choreography referencing simplifies the modular definition of choreographies; however, it also complicates considerably the exposition of the realizability analysis presented in Section 5 by requiring the adoption of inter-procedural flow analysis techniques. Our design decision is taken for reasons of understandability of the exposition. Nevertheless, it is straightforward to extend the realizability analysis presented in Section 5 to support choreography referencing in ChorTex by means of the inter-procedural flow analysis techniques available in the state of the art, e.g. [Mye81, Cal88, RHS95, MO04].

2.3.3 Finalization Handlers

An extension to Chor presented in [YZCQ07] allows the specification of finalization behaviors, i.e. activities that are executed irrespective of the completion of the body of a choreography or its termination due to exception propagation. Similarly to the case of choreography referencing, ChorTex does not include such functionality for reasons of understandability of the exposition. In fact, presenting the generation and analysis of CFGs that involve finalization handlers is much more convoluted than without it. Known techniques of CFG analysis such as [SH00] can deal with finalization handling, and can easily be adapted to deal with ChorTex choreographies with finalization handlers.
3 Choreography Realizability

The present section provides the necessary background on choreography realizability (Section 3.1) and specifies which type of realizability for ChorTex choreographies we are going to analyze in the reminder of this work (Section 3.2).

3.1 Aspects of Choreography Realizability

The state of the art is rich with heterogeneous definitions of choreography realizability, see e.g. [AEY03, AEY05, FBS05, KP06, SBFZ07, MFEH07, BFS07, BF08, BH10, HB10]. A very general – and intentionally underspecified – definition of choreography realizability is the following.

**Definition 6** (Generic Definition of Choreography Realizability). A choreography is realizable if it is possible to devise participant implementations for its roles that, when interacting with each other, are behaviorally equivalent to the choreography.

Definition 6 is under-specified because it does not clarify which of the many possible behavioral-equivalence relations is required between the composition of the participant implementations and the choreography. The adopted notion of behavioral equivalence, in turn, depends to some extent on design decisions and assumptions that underpin each choreography modeling language. Naturally, this has lead to a variety of different definitions of choreography realizability in the state of the art which can be classified according to the following three dimensions proposed in [Dec09]:

**Complete vs. subset of the behavior:** How much of the behavior of the choreography can be enacted by the composition of the participants implementations. A definition of choreography realizability requires complete behavior if all and only the behaviors (e.g. traces) that are specified by the choreography can be enacted by the composition of the participant implementations. Such definitions of realizability are often called strong, see e.g. [MFEH07, BF07, RS11]. If not all the choreography’s behaviors must necessarily be enactable by the composition of the participant implementations, the definition of choreography realizability requires a subset of the behavior. Such definitions are usually labeled as “weak”, see e.g. [AEY05, SBFZ07].

**Communication model:** What are the assumptions on functional and non-functional characteristics of the communication channels connecting the participants, e.g. synchronous, asynchronous with queues of fixed size, asynchronous of queues of infinite size, and whether the communication channels preserve the ordering of the messages in the queues.
Equivalence notion: The “strength” of the required equivalence notion, e.g. trace-equivalence, language-equivalence or bi-simulation [BIM95].

Given the dimensions listed above, it is clear that there is no “one-size-fits-all” definition of choreography realizability. Particularly interesting is the relevance of the communication model. To be formally specified, a choreography modeling language must necessarily make assumptions on how the participants communicate with each other. In other words, the communication model is a design decision for choreography modeling languages. This suggests that definitions of choreography realizability are necessarily “tailored” to some extent to specific choreography modeling languages.4 The following section presents the definition of realizability for ChorTex choreographies we investigate in this work.

3.2 Strong Realizability of ChorTex Choreographies

The definition of realizability adopted in this work is based on the notion of conversation.

Definition 7 (Conversations). The conversation $c_{cn}$ performed during an enactment is the enactment trace $\sigma$ restricted to only the actions that describe interactions among participants, i.e. message exchanges and opaque activities (which may involve interactions among the participants).

$$
\text{conv}(\langle \langle d \rangle \rangle \circ \sigma) := \begin{cases} 
\langle d \rangle \circ \text{conv}(\sigma) & \text{if } a = p \xrightarrow{m} p_1, \ldots, p_n \\
\langle d \rangle \circ \text{conv}(\sigma) & \text{if } a = \langle \langle (p_1, \ldots, p_n) \rangle \rangle \\
\langle d \rangle & \text{if } \sigma = \langle d \rangle \\
\text{conv}(\sigma) & \text{otherwise}
\end{cases}
$$

The function conv($\sigma$) defined above specifies how to extract a conversation from an enactment trace $\sigma$.

In this work we adapt to ChorTex choreographies the definition of strong realizability proposed in [KP06]:

Definition 8 (Strong Realizability of ChorTex choreographies). A ChorTex choreography is strongly realizable if exist participant implementations for its roles the composition of which is language-equivalent in terms of conversations to the original choreography.

---

4Actually, this coupling between choreography modeling languages and definitions of choreography realizability helps explaining the large variety of (and the limited comparison among) of the latter in the state of the art.
In other words, the definition of strong realizability requires that it is possible to create participant implementations that, during every possible enactment, preserve the ordering of the interactions between the participants as specified by the choreography.

In terms of the three dimensions of choreography realizability discussed in Section 3.1, the notion of strong realizability we adopt in this work is classified as follows:

**Completeness of the behavior**: Complete.

**Communication model**: Asynchronous with queues of infinite length (see Section 2);

**Equivalence notion**: Language-equivalence of conversations.

The rationale for choosing strong realizability as defined in Definition 8 over others is rooted in the fact that choreographies are often used as technical contracts among the participants (see Section 1). Every conversation intensionally defined by a strongly realizable choreography is enactable by the participants, but no restrictions are put on their internal actions. Also very important is what the strong realizability does not require, and in particular: (1) preservation of the ordering of reception and consumption of messages by the recipients and (2) preservation of the ordering of the internal activities, namely the decisions by the decision makers of choice and interaction activities. Recall that ChorTex assumes asynchronous messaging. One of the implications of the asynchronous messaging model assumed is that there are no guarantees about “when” a recipient will consume a message it has received (see Section 2.1). Therefore, there is no way to enforce a particular order between consumption of messages by recipients and the actions in the conversation. (An exception is when a recipient is required to take an action in response of a message it has consumed; in this case, the ordering is “implicitly” enforced, because due to how the choreography is specified, the recipient will not perform the action until it has consumed the message.) The ordering of internal actions performed by the participants cannot likewise be enforced. For example, nothing prohibits a participant to decide in advance, possibly even before an enactment begins, which decisions it will take. In a sense, this is equivalent to repeatedly throwing a dice before the beginning of a game of “Sorry,” recording in order the outcomes, and then using them one in the order they were thrown whenever required in the game.

The trade-off of strong realizability between what is required and what is not is clearly ideal for technical contracts such as choreographies: the participants can provide implementations that comply with all the obligations (i.e. they are able to perform all the specified behaviors), and no other limitations are posed on how each participant realizes its own role.
4 From Choreographies to Control Flow Graphs

The realizability analysis for ChorTex choreographies presented in this work builds on top of the flow analysis framework (see e.g. [All70, Aho07]). The flow analysis framework is used to study static properties of computer programs, i.e. properties that can be inferred from the structure of the programs at design/compilation-time such as dominator and post-dominator analysis, reaching definitions and data dependences [Aho07]. The flow analysis framework is based on the concept of Control Flow Graph (CFG). The CFG of a program (e.g. in Java) is a directed graph in which each represents one of the program’s instructions, and an edge connecting two nodes represents a control dependency between the respective instructions. Section 4.1 explains how to construct CFGs from ChorTex choreographies, while Section 4.2 correlates the CFGs so constructed and the operational semantics of ChorTex presented in Section 2.2.

4.1 Construct CFGs of ChorTex Choreographies

We adapt to ChorTex choreographies the approach proposed by [SH00] for constructing CFGs of Java programs, thus exploiting the similarities between the exception handling mechanisms of ChorTex and those of general-purpose programming languages. Table 2 and Table 3 show how to construct CFGs of ChorTex choreographies by mapping every choreography activity to a CFG sub-graph. The CFG nodes with the grey background in Table 2 and Table 3 are “place-holders” for the sub-graphs of the respective activities. The compositionality of the mapping rules requires a mechanism for dealing with edges that connect nodes resulting from different activities. This is done by labeling edges with conditions like “to first (A),” which means that the target node of the edge is the one labeled as “first” in the sub-graph resulting from the activity A. Similarly, the condition “from last (A)” means that the source of the labelled edge is the node labelled as “last” in the sub-graph resulting from the activity A. It should be noted that, due to the definition of “first” and “last” in Table 2 and Table 3, each CFG sub-graph has precisely one “first” and one “last” node.

Each node of the CFG generated from a ChorTex choreography represents the firing of exactly one event, e.g. the beginning or completion of an activity, during an enactment of the choreography. Consequently, the CFG edges represent the order in which the firing of events may occur while enacting the choreography. Intuitively, a control-flow edge connecting the two nodes n and n’, respectively representing the firing of the events e and e’, means that e’ can be fired only after that e was fired. A more precise interpretation of the ordering of events is presented later in Section 4.2, as it requires an understanding of how enacting activities relates to the firing of events.
<table>
<thead>
<tr>
<th>ChorTex Activity</th>
<th>Corresponding Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip [s]</td>
<td>first, last skip [a]</td>
</tr>
<tr>
<td>[mex] p₁ (\rightarrow) m to (p₁, \ldots, pₙ)</td>
<td>first, last [mex] p₁ (\rightarrow) m to (p₁, \ldots, pₙ)</td>
</tr>
<tr>
<td>opaque [p] ((p₁, \ldots, pₙ))</td>
<td>first, last opaque [o] (p₁, \ldots, pₙ)</td>
</tr>
<tr>
<td>throw [t] (e)</td>
<td>first, last, propagates (e) throw [t] (e)</td>
</tr>
</tbody>
</table>
| \(b\) \{ \(A₁ : \ldots : Aₙ\) \} | \(\begin{align*}
\text{first} & \quad \text{block } [b] \\
\text{start} & \quad \text{to first } (A₁) \\
\text{if last } (A₁) & \quad \text{does not propagate } e \\
\text{from last } (A₁) & \quad \text{to first } (A₁) \\
\text{if last } (Aₙ) & \quad \text{from last } (Aₙ) \\
\text{to first } (Aₙ) & \quad \text{if last } (Aₙ) \text{ does not propagate } e \\
\text{last} & \quad \text{block } [b] \\
\end{align*}\) |

| choice \[c\] \(p\) either \(A₁\) or \(\ldots\) or \(Aₙ\) | choice \[c\] split \text{to first } (A₁) from last (A₁) choice \[c\] join \(\ldots\) choice \[c\] split \text{to first } (Aₙ) from last (Aₙ) |

Table 2: The mapping from ChorTex activities to Control Flow Graphs (Part 1).
<table>
<thead>
<tr>
<th>ChorTex Activity</th>
<th>Corresponding Control Flow Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>iteration [i] (p ) do (A)</td>
<td><img src="chart1" alt="Control Flow Graph" /></td>
</tr>
<tr>
<td>parallel [prl] do (A_1) and ... and (A_n)</td>
<td><img src="chart2" alt="Control Flow Graph" /></td>
</tr>
</tbody>
</table>

Table 3: The mapping from ChorTex activities to Control Flow Graphs (Part 2).
The CFG nodes block \([b]\) start and block \([b]\) end that are generated from a block activity named \(b\) represent the beginning and completion of the enactment of \(b\), respectively. Similarly, choice \([c]\) \(p\) split and choice \([c]\) \(c\) join represent the beginning and completion of the choice activity \(c\), respectively.\(^5\)

The node iteration \([i]\) \(i\) represents at the same time (1) beginning, (2) completion and (3) the taking of the decision by \(p\) of the iteration activity \(i\).\(^6\) The nodes parallel \([p]\) split and parallel \([p]\) join represent the beginning and completion of the enactment of the parallel activity \(p\).

The enactment states of a choreography \(ch\) are the beginning (represented by the node chor \([ch]\) start), completion (chor \([ch]\) end), invocation of the default exception handler (chor \([ch]\) eh \(*\)) or named exception handler for the exception type \(e\) (chor \([ch]\) eh \(e\)), and the propagation outside the choreography of an exception of type \(e\) (chor \([ch]\) err \(e\)).

Dealing with exceptions requires special care. Nodes that represents the throwing or propagation of an exception \(e\) to parent activities are labeled “propagates \(e\)”. Consider for example the block \(\{ A_1; A_2 \}\). If \(A_1\) is a throw \(e\) activity, \(A_2\) will never be enacted; therefore the corresponding subgraph in the CFG must not have a control-flow edge connecting \(A_1\) with \(A_2\).\(^7\) For the sake of understandability, the exception control-flow edges, i.e. the control-flow edges that represent the propagation of exceptions, are depicted dashed in Table 2 and Table 3. It should be noted that only the rule for creating CFG sub-graphs of choreographies specify exception control-flow edges. The reason is that the “wiring” of exception propagating and handling is done only at the level of choreographies.

Figure 16 introduces the running example of this work, namely a choreography that specifies sub-choreographies and has a non-trivial propagation of exceptions. The exception of type \(e_1\) that can be thrown during the enactment of chor\(3\) is caught by the named exception handler for \(e_1\) specified in chor\(1\), which in turn throws an exception of type \(e_2\), which terminated the choreography. The exception propagation is evident in Figure 17, which shows the CFG resulting from the choreography in Figure 16. It is interesting to notice that there are some nodes, namely chor \([chor_1]\) eh \(*\), skip \([s_2]\), chor \([chor_2]\) eh \(*\) and skip \([s_1]\), that are not reachable – in

\(^5\)It would be possible to map choice activities without resorting to join nodes. For example, we could connect the last nodes of the sub-graphs generated by the branches with the first node of the activity after the choice. However, having join nodes greatly simplifies the specification of the mapping rules.

\(^6\)It would be possible to represent separately these events with distinct nodes, but it would not provide any concrete advantage in terms of control-flow analysis.

\(^7\)In ChorTex it is indeed possible to specify a block \(\{ \text{throw } e; \text{skip } \}\). The skip, in this case, is “dead-code,” which is usually considered something to be avoided, but not a defect in itself. However, such cases may be excluded by dedicated well-formedness rules, e.g. that only the last activity in a block can be a throw activity.
chor [chor1] ( 
  [chor1body] {
    chor [chor2] ( 
      [chor2body] {
        parallel [prl1] do
          [prl1branch1] {
            [mex1] p_2 \rightarrow m_1 \to p_1 
          }
        and
          [prl1branch2] {
            [mex2] p_3 \rightarrow m_2 \to p_1, p_2 
          }
        );
        [mex3] p_3 \rightarrow m_3 \to p_1, p_2 
      }
    }
  }
);
chor [chor3] ( 
  [chor3body] {
    choice [choice1] p_1 
    either
      [choice1either] {
        throw [throw1] e_1 
      }
    or
      [choice1or] {
        [mex4] p_1 \rightarrow m_4 \to p_2, p_3 
      }
    }
  | e1: 
    throw [throw2] e_2 
  | *:
    skip [skip2] 
) 

Figure 16: The running example.
Figure 17: Control Flow Graph of the choreography shown in Figure 16.
terms of “pure” graph traversing – from the start node of the CFG, i.e. the “first” node of the sub-graph generated from the root choreography. In Figure 17, the nodes not reachable from the start node are painted with reduced opacity (i.e. they look “more transparent” than others), as well as the control-flow edges that originate from and/or target them. Specifically, block [choice1|either] end is not reachable because the last activity of its block is a propagate $e_1$ node, namely a throw node, and thus there is no edge connecting the throw and block [choice1|either] end nodes. The other unreachable nodes, instead, represent exception handlers that are never triggered.

4.2 Reconciling Actions and Events in ChorTex Enactments

On the basis of method to construct CFGs of ChorTex choreographies presented in Section 4.1, this section (1) provides a formal interpretation of the ordering of events as specified by the edges in the CFGs and (2) relate the events with the actions performed by participants that constitute an enactment trace (see Section 2.2). The content of this section is instrumental to the definition of the concept of awareness and the constraints for verifying the strong realizability of ChorTex choreographies that is presented later in Section 5 and Section 6. Figure 18 summarizes the terminology and relationships between the terms that are introduced in this section.

**Definition 9** (Preceding and Succeeding Events). An event $e$, represented in the CFG by a node $n_e$, precedes an event $e'$, represented in the CFG by a node $n_{e'}$, if there is a control-flow edge with source $n_e$ and target $n_{e'}$ in the CFG. If $e$ precedes $e'$, $e$ is said to be a predecessor of $e'$. Conversely, if $e$ precedes $e'$, then $e'$ succeeds $e$ or, equivalently, $e'$ is a successor of $e$.

The precedence of events expressed by the edges in CFGs is a temporal correlation. If an event $e$ precedes $e'$ in the CFG, then in an enactment the firing of $e$ might cause the firing of $e'$. Precedence, however, is not causality: the firing on an event does not necessarily cause the firing of its successors. For example, the event that represents the taking of the decision in a choice activity is predecessor to all the events that represent the beginning of the enactment of one of the choice’s branches. However, every time a decision is taken, only one of those start events of the branches may occur. (Actually, it could even be the case that none will occur, if the choice activity is terminated because of the propagation of an exception thrown by a concurrently enacted activity.)

We distinguish between two types of events: participant-activated and reactive.

**Definition 10** (Participant-Activated Events and Participant-Activated Nodes). The firing of a participant-activated event represents the performing
Figure 18: Terminology related to enactments, actions and events.
of an action by some of the participants. The types of participant-activated events are the following:

- Dispatching of a message;
- Enactment of an opaque activity by the participants involved in it;
- Decision of which branch of a choice activity to enact;
- Decision of whether to iterate the body of an iteration activity.

A node representing the firing of a participant-activated event is said to be a participant-activated node.

Straightforwardly, each type of participant-activated events corresponds to one of the types of actions specified in Definition 1. Consistently with the fact that the reception of a message by a recipient is not considered an action (see Definition 1), it is not represented as an event either.

**Definition 11 (Reactive Events and Reacting Nodes).** Reactive events are fired as a consequence of the firing of participant-activated events. The types of reactive events are all those not explicitly denoted as participant-activated in Definition 10. A node that represents the firing of a reactive event is said to be a reactive node.

To exemplify reactive and participant activate events, consider the CFG shown in Figure 19 obtained from a straightforward choreography.

The enactment of the choreography represented by the CFG in Figure 19 begins when the participant $p$ dispatches the message $m_1$ to the participant $p'$. The dispatching of the message $m_1$, and therefore the firing of the participant-activated event $n_{m1}$, triggers the firing of the reactive events $n_{c1}$, $n_{b1}$ and $n_{b2}$ (fired in this order). The events $n_{c1}$ and $n_{b2}$ are “retroactively” fired because the dispatching of $m_1$ is an initiating action (see Definition 4). Since the enactment does not exist before $m_1$ is dispatched, the firing of $n_{c1}$ and $n_{b1}$ is implied to “fill the gap” in the traversal of the CFG from the start node to $n_{m1}$. After the firing of $n_{b2}$, the enactment “waits” for the dispatching of $m_2$ by $p'$. The dispatching of the message $m_2$ triggers the reactive events $n_{b3}$, $n_{b4}$ and $n_{c2}$, the latter representing the end of the enactment.
Definition 12 (Choreography Violations). An action \( a \) performed by some participants in the enactment state \( \chi \) is said to violate the choreography if there is no rule of the operational semantics of ChorTex that is applicable to \( \chi \) and that results in \( a \) been appended to the enactment trace.

For example, considering the CFG in Figure 19, if \( p' \) were to send \( m_2 \) to \( p \) before receiving \( m_1 \), that would violate the choreography. In other words, a violation of a choreography consists in the performing an action in an enactment state that, according to how the choreography is specified, does not allow it. In order to avoid violations of the choreography, acting participants have to abide restrictions in terms of which enactment states allow their actions can be performed.

Definition 13 (Enactable Actions). An action \( a \) is enactable in the enactment state \( \chi \) if and only if \( a \) performed by its acting participants in \( \chi \) does not violate the choreography.

Building on the notion of enactable activity, it can be specified when a participant-activated event is fireable.

Definition 14 (Fireable Events). The participant-activated event \( e_a \) associated with the action \( a \) is fireable in the enactment state \( \chi \) if \( a \) is enactable in \( \chi \).

5 Awareness Models of ChorTex Choreographies

This section discusses how to create an AWM by annotating the CFG of a ChorTex choreography (created as explained in Section 4.1) with information on the participant awareness. The current section is structured as follows. Section 5.1 discusses the concept of participant awareness, while Section 5.2 explains how the participant awareness is calculated.

5.1 Participant Awareness in Choreographies

As introduced in Section 2.2, a ChorTex choreography is fundamentally the intensional specification of sequences of actions such as message exchanges and internal decisions, that the participants are tasked to perform during enactments. The CFG of a ChorTex choreography represents the ordering of the firing of events representing e.g. the beginning and completion of an activity, and are fired during an enactment as a result of the actions performed by the participants (see Section 4.2).

When enacting a choreography, the participants are subject to the phenomenon called blindness [DS08, DDC09], i.e. they might be able to observe only some of the actions that are performed. Because of the blindness, it might be the case that some participants have not enough information on
how the enactment proceeds to perform their roles as mandated by the choreography. For example, it may happen that during one enactment a participant does not know that its performing of a certain action is necessary for the progress of the enactment, or that performing a certain action violates the choreography. Put it in terms of events, a participant might be unable to observe sufficient events to play its role as specified, and this is symptomatic of the unrealizability of the choreography that is being enacted (see Section 3). The goal of this section is to provide symbolic means for describing which events a participant can and cannot observe during the enactments of a choreography, which is the basis for the verification of the strong realizability of a choreography presented in Section 6.

**Definition 15 (Observable Actions).** The participant $p$ can observe the performing of an action $a$ (equivalently, $a$ is observable by $p$), if and only if:

- $p$ is an acting participant of $a$ (see Definition 1), or
- $a$ is a message exchange, and $p$ is one of its recipients.

In any other case, the participant $p$ cannot observe the performing of the action $a$.

In other words, a participant is able to observe only those actions it performs as their acting participant, such as the dispatching of a message, or that have observable consequences for the participant, i.e. the reception of a message addressed to it. As a result, the participants have local perceptions on the enactment traces.

**Definition 16 (Local Perceptions).** The local perception of the participant $p$ of an enactment trace $\sigma$ is the trace obtained by purging $\sigma$ of the actions not observable by $p$ (see Definition 15).

\[
\pi\left(p, \{a\} \circ \sigma\right) := \begin{cases} 
\{a\} \circ \pi\left(p, \sigma\right) & \text{if } (a = [\text{mex} \downarrow p_{s} \Rightarrow] \rightarrow \downarrow p_{s_{1}}, \ldots, p_{s_{n}}) \land (p = p_{s} \lor p \in \{p_{r_{1}}, \ldots, p_{r_{n}}\}) \\
\{a\} \circ \pi\left(p, \sigma\right) & \text{if } (a = [\text{o} \downarrow (p_{1}, \ldots, p_{n})] \Rightarrow p \in \{p_{1}, \ldots, p_{n}\}) \\
\{a\} \circ \pi\left(p, \sigma\right) & \text{if } (a = [\text{i} \downarrow x] \Rightarrow x) \\
\pi\left(p, \sigma\right) & \text{otherwise}
\end{cases}
\]

The above defined function $\pi\left(p, \sigma\right)$ specifies how to project the local perception of the participant $p$ from the enactment trace $\sigma$.

It should be noted that the local perception of a participant on an enactment trace does not necessarily represent the order in which the participant has observed the actions. For example, in the running example shown in Figure 16, due to the delay between the dispatching of a message and its...
processing by the receivers, it might be that the participant \( p_1 \) processes the message \( m_2 \) before \( m_3 \), even thought they have been dispatched by the respective senders in the opposite order.

Due to their local perception, participants have a different knowledge on which actions have been performed up to that point in an enactment, and therefore which events have been fired so far. The knowledge of a participant with respect of the firing of an event is modeled by the concept of \textit{participant awareness}. We distinguish between the participant awareness of a participant \textit{before} and \textit{after} the firing of an enacting event \( e \):

**Participant awareness of \( p \) before \( e \)** represents the capability of \( p \) of observing that \( e \) becomes fireable (see Definition 14).

**Participant awareness of \( p \) after \( e \)** represents the capability of \( p \) of observing that \( e \) has been fired.

The differentiation of the participant awareness between before and after the firing of an event \( e \) allows to describe situations in which a participant \( p \) knows that \( e \) may be fired in the current enactment state, causing the transition to another enactment state, but the actual firing of \( e \) is not observable by \( p \). This is extremely important for enacting the choreography without violations. For example, it may be the case that the action \( a \) is enactable in the current enactment state \( \chi \), but not in the enactment state \( \chi' \) that results from the firing of an event \( e \) which was fireable in \( \chi \). The participants rely solely on their local perceptions to decide when to perform the actions that they \textit{believe} are enactable. Therefore \( p \) may unknowingly violate the choreography by performing \( a \) in the enactment state \( \chi' \) because it did know that the enactment state had transitioned from \( \chi \) to \( \chi' \) due to the firing of \( e \).

Table 4 presents the four possible \textit{participant awareness states}, i.e. the “extents” of participant awareness that a participant may have about the becoming fireable and the firing of an event \( e \).

**Immediate awareness**, denoted by “ia,” means that the participant knows \textit{as soon as it happens} that the event becomes fireable or is fired (awareness before and after the event, respectively). For example, the sender of a message exchange is immediately aware that the message has been dispatched. Similarly, the decision maker of a certain decision is immediately aware when that decision has been taken.

**Eventual awareness**, denoted by “ea,” is a “relaxed” form of awareness. The participant does not necessarily know that the event has become fireable or has been fired the moment it happens, but will be able to observe it “sooner or later”, i.e. \textit{eventually} as understood in Linear Temporal Logics (LTL) [BK08]. For example, due to the asynchronous nature of message exchanges in ChorTex, the recipient of a message will eventually know (at the
<table>
<thead>
<tr>
<th>Awareness state</th>
<th>Annotation</th>
<th>Applied to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediately aware</td>
<td>$p_{ia}$</td>
<td>$e$ becoming fireable</td>
<td>$p$ can observe that $e$ is fireable as soon as $e$ becomes fireable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$e$ being fired</td>
<td>$p$ can observe the firing of $e$ as soon as $e$ is fired</td>
</tr>
<tr>
<td>Eventually aware</td>
<td>$p_{ea}$</td>
<td>$e$ becoming fireable</td>
<td>$p$ is able to observe that $e$ becomes fireable immediately when or at some point after $e$ has become fireable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$e$ being fired</td>
<td>$p$ is able to observe the firing of $e$ immediately when or at some point after $e$ has been fired</td>
</tr>
<tr>
<td>Unaware</td>
<td>$p_{ua}$</td>
<td>$e$ becoming fireable</td>
<td>$p$ cannot observe that $e$ becomes fireable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$e$ being fired</td>
<td>$p$ cannot observe the firing of $e$</td>
</tr>
<tr>
<td>Not involved</td>
<td>$p_{ni}$</td>
<td>$e$ becoming fireable</td>
<td>$p$ is not yet involved in the enactment when $e$ becomes fireable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$e$ being fired</td>
<td>$p$ is not yet involved in the enactment when $e$ is fired</td>
</tr>
</tbody>
</table>

Table 4: The possible participant awareness states of the participant $p$ with respect to an event $e$. 
moment of the reception) that the message has been dispatched. Straightforwardly, immediate awareness implies eventual awareness.

An unaware, denoted by “ua,” participant cannot observe at all an event becoming fireable or being fired. For example, a participant $p$ that is neither sender nor recipient of a message $m$ is unaware of participant-activated event that represents the dispatching of that message. Of course, due to some later interaction with the other participants, $p$ might be able to “imply” that $m$ has been dispatched. This is the case, for example, when the dispatching of $m$ must have necessarily been occurred before another message $m'$ is dispatched, and $p$ can observe the dispatching of $m'$. However, this type of implication is not of interest to the realizability analysis proposed in this work, and has no effect on the definition of the participant awareness states.

Finally, a participant is not involved, denoted by “ni,” with respect an event becoming fireable or been fired if, when that happens, the participant has not yet taken part in the enactment. (Notice that the participant might later become involved in the enactment, in which case the participant will be immediately-, eventually- or unaware of some other events and might imply that the events he is not-involved of have been fired; however, as explained before, this type of implication is not relevant to the ends of participant awareness, which is based on observation.) Non-involvement implies unawareness. In fact, if a participant has not yet partaken an enactment, it cannot be aware of any of the enactment events that have taken place thus far.

5.2 Annotating Participant Awareness States

This section shows how to create AWMs by annotating CFGs generated from ChorTex choreographies (see Section 4) with the participant awareness states introduced in the previous section. Since the AWM is simply a CFG with additional annotations, the nodes and control-flow edges of an AWM are exactly the same of the original CFG.

The annotations on the participant awareness states associated with the nodes in the AWMs are called node awareness states.

**Definition 17 (Node Awareness States).** In the AWM of a choreography specifying the participants $p_1,\ldots,p_m$, the node awareness state of a node $n$ that represents the firing of the event $e$ is defined as the following couple:

$$\left( AW_{\text{fireable}}(n), AW_{\text{fired}}(n) \right)$$

$AW_{\text{fireable}}(n)$ and $AW_{\text{fired}}(n)$ are sets comprising $m$ items, each of them representing the participant awareness state of one participant with respect to the event $e$ becoming fireable and $e$ being fired, respectively. The
AW\_fireable \( (n) \) and AW\_fired \( (n) \) of an arbitrary node \( n \) are both formally defined as follows:

\[
\left\{ P_i^\alpha : \forall i \in [1, m] \exists! \alpha \in \{ia, ea, ua, ni\} \right\}
\]

That is, they contain exactly one participant awareness state associated to each of the participants specified by the choreography.

AW\_fireable \( (n, p) \) and AW\_fired \( (n, p) \) denote the participant awareness state of the participant \( p \) in AW\_fireable \( (n) \) and AW\_fired \( (n) \), respectively.

5.2.1 The Awareness Annotation Algorithm

Figure 20 presents the pseudo-code of the Awareness Annotation Algorithm (AAA), which calculates the node awareness states associated with the nodes of AWMs. The AAA has the classic structure of fixed-point algorithms on CFGs, and it begins with an initialization phase that assigns for each node and to all participants an initial “non-involved” AW\_fireable value. The function \( P(chor) \), defined in Figure 21, returns the set of participant identifiers appearing in the choreography \( chor \) by traversing its parsing tree. For each participant \( p \), the AW\_fired of a node \( n \) is calculated by the function \( f(n, p) \), which is discussed in detail in Section 5.2.3. In a nutshell, the function \( f(n, p) \) calculates how the firing of the event represented by the AWM node \( n \) affects the participant awareness state of the participant \( p \).

After the initialization phase, the algorithm iterates until a fixed point is reached, i.e., until further iterations produce no modifications of the node awareness states. At each iteration step, a node \( n \) is chosen at random to be processed. (Although, implementations of the AAA may employ heuristics e.g. based on the cycles in the CFGs; however, such heuristics fall outside the scope of this work, as they add nothing to the concepts here presented.) The processing of \( n \) consists of the following two steps, which are repeated for each participant \( p \) specified in the choreography. First, the AW\_fireable value of \( n \) is recalculated as the safe approximation of all the AW\_fired values of \( n \)’s predecessors, which is calculated using the function \( \hat{\rho} \) described in Section 5.2.2. (For reasons of understandability, we discuss how the safe approximation is calculated in Section 5.2.2.) Secondly, AW\_fired \( (n, p) \) is updated with the outcome of \( f(n, p) \), which in some cases is calculated on the basis of the updated value of AW\_fireable \( (n, p) \) that results from the previous step.

Figure 22 presents the AWM resulting from the application of the AAA to the CFG of the running example shown in Figure 16. The node awareness states are represented as labels attached to the respective nodes. Notice that there are no AW\_fireable and AW\_fired values annotated on the nodes that are unreachable from the start node, which are drawn less markedly in Figure 16. The nodes unreachable from the start one are ignored when calculating
the node awareness states because they represent events that never become fireable and are never fired in any enactment of the choreography. In the pseudo-code of the AAA, this is realized by the “reachable from start” clauses on Line 9 and Line 20.

```
function generateAwarenessModel (choreography chor) returns awareness model {
    /* generate chor’s CFG as described in Section 4.1 */
    awareness model awm := generateCFG(chor);
    /* retrieve the start node of the AWM */
    node start := startNode(awm);
    /* initialization */
    foreach node n in N(awm) reachable from start do
        /* P(chor) returns the set of participants */
        /* specified by the choreography chor (see Figure 21) */
        foreach p in P(chor) do
            AW_fireable(n,p) = ni;
            AW_fired(n,p) = f(n,p);
        end foreach
    end foreach
    /* iteration until a fixed-point is reached */
    repeat
        foreach node n in N(awm) reachable from start do
            /* update AW_fireable as safe-approximation of */
            /* the AW_fired of the successors of n */
            foreach p in P(chor) do
                /* of the predecessors of n (see Section 5.2.2) */
                AW_fireable(n,p) = \bigcap_{n' \in \text{pred}(n)} (AW_fired(n',p));
                /* update output (see Section 5.2.3) */
                AW_fired(n,p) = f(n,p);
            end foreach
        end foreach
        until no AW_fired(n,p) changes
    repeat
        /* update awareness model */
        /* using the reachable from start clauses */
    until no AW_fired(n,p) changes
    return awm;
}
```

Figure 20: Pseudo-code of the Awareness Annotation Algorithm.
Figure 21: Recursive definition of the function $P(A)$ for extracting the participants involved in an activity $A$.

5.2.2 Safe Approximation of $AW_{\text{fireable}}$ Values of Nodes

In the iteration phase of AAA, the $AW_{\text{fireable}}$ value of a node is recalculated as the safe approximation of the $AW_{\text{fired}}$ values of its predecessors of $n$. The safe-approximation of $AW_{\text{fired}}$ of a set of nodes is calculated by the function $\bigwedge$ (read “meet”) defined Figure 23.\(^8\) The safe approximation is calculated on the basis of the following principles:

1. Immediate awareness implies eventual awareness (see Section 5.1);
2. Non-involvement implies unawareness (see Section 5.1);
3. When a participant is marked as unaware in any of the input awareness states, it is also marked as unaware in their safe approximation.

The last principle comes from the “nature” of the information aggregated by $\bigwedge$, i.e. participant awareness states. The goal is to safely approximate the participant awareness state of one participant with respect to a certain event becoming fireable. As discussed in Section 4.2, an event becomes fireable when one or more of its predecessors have been fired (see Definition 14). If the participant $p$ is unaware of the firing any of the predecessors of the event $e$, then there are some enactments of the choreography in which $p$ is unaware of $e$ becoming fireable. Therefore, the safe approximation is marking $p$ as unaware in $AW_{\text{fireable}}(n)$.

\(^8\)The meet function is traditionally denoted in flow-analysis algorithms by the symbol $\bigwedge$. However, we need the $\bigwedge$ symbol later in the work to represent the logic conjunction of multiple predicates; hence our unconventional choice of symbol for the meet function.
Figure 22: Awareness Model obtained by applying the Awareness Annotation Algorithm to the Control Flow Graph shown in Figure 17.
5.2.3 Updating the $\text{AW}_{\text{fired}}$ Value of Nodes

The value of $\text{AW}_{\text{fired}} (n, p)$ of a node $n$ for the participant $p$ is calculated with the function $f (n, p)$. The function $f (n, p)$ is defined in Table 5 case-based on the different types of AWM nodes. In practice, the function $f (n, p)$ specifies how the firing of the event represented by the node $n$ affects the participant awareness state of the participant $p$, and it is applied in the AAA on each node $n$ once for each of the participants in the choreography. In some cases, $f (n, p)$ simply returns the value of $\text{AW}_{\text{fireable}} (n, p)$, meaning that the traversal of nodes of those types does not modify the participant awareness state of the participant $p$. Specifically, this is the case of nodes representing the enactment of skip and throw activities, end of block activities, merge and split nodes of parallel activities, merge nodes of choice activities, and the start, end, exception handling and error propagation nodes of choreographies.

After the traversing of a node representing the dispatching of a message exchanges, the sender is immediately aware. The recipients, instead, are eventually aware because the messages are processed asynchronously. The participants that were not involved before traversing the node remain not-involved after it. Otherwise, the participants are unaware.

The traversing of nodes that represent opaque activities makes all the participants that partake them eventually aware. This comes from the assumption that the participants partaking an opaque activity reckon its completion (see Section 2.1). Participants that were not involved before the opaque activity stay not involved. Otherwise, the participants that were involved before the opaque activity, but that do not take part in it, become unaware.

The nodes representing local decisions for iteration and choice activities modify the participant awareness in the same way. Namely, since the decision is local, the decision maker is immediately aware of it. All participants

$$\alpha \in \{\text{ia, ea, ua, ni}\}$$

$$\mathcal{R}(\alpha_1, \ldots, \alpha_m) := \begin{cases} 
\text{ia} & \text{if } \forall j \in [1, m] : \alpha_j = \text{ia} \\
\text{ea} & \text{if } (\exists j \in [1, m] : \alpha_j = \text{ea}) \land \\
& (\forall i \in [1, m] : \alpha_j = \text{ia} \lor \alpha_j = \text{ea}) \\
\text{ni} & \text{if } \forall j \in [1, m] : \alpha_j = \text{ni} \\
\text{ua} & \text{otherwise}
\end{cases}$$

Figure 23: Definition of the $\mathcal{R}$ function that calculates for a participant $p$ the safe-approximation of the $\text{AW}_{\text{fired}}$ values of multiple nodes.
<table>
<thead>
<tr>
<th>AWM node $n$</th>
<th>Definition of $f(n,p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_s \rightarrow m$ to $p_{r_1}, \ldots, p_{r_n}$</td>
<td>$\begin{cases} ia &amp; \text{if } p = p_s \ ea &amp; \text{if } p \in {p_{r_1}, \ldots, p_{r_n}} \ ni &amp; \text{if } (p \notin {p_s} \cup {p_{r_1}, \ldots, p_{r_n}}) \land AWF_{\text{fireable}}(n,p) = ni \ ua &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>opaque $(p_1, \ldots, p_m)$</td>
<td>$\begin{cases} ea &amp; \text{if } p \in {p_1, \ldots, p_m} \ ni &amp; \text{if } p \notin {p_1, \ldots, p_m} \land AWF_{\text{fireable}}(n,p) = ni \ ua &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>block start</td>
<td>$\begin{cases} ua &amp; \text{if block is branch of parallel activity } A \land AWF_{\text{fireable}}(n,p) = ni \land \ &amp; p \text{ is acting participant in another branch of } A \ &amp; AWF_{\text{fireable}}(n,p) \text{ otherwise} \end{cases}$</td>
</tr>
<tr>
<td>choice $p_i$ split</td>
<td>$\begin{cases} ia &amp; \text{if } p = p_i \ ni &amp; \text{if } p \neq p_i \land AWF_{\text{fireable}}(n,p) = ni \ ua &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>iteration $p_i$</td>
<td>$\begin{cases} ia &amp; \text{if } p = p_i \ ni &amp; \text{if } p \neq p_i \land AWF_{\text{fireable}}(n,p) = ni \ ua &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>otherwise</td>
<td>$AWF_{\text{fireable}}(n,p)$</td>
</tr>
</tbody>
</table>

Table 5: The definition of $f(n,p)$ for the node $n$ and the participant $p$, case-based on the various types of AWM nodes.
that do not take the local decision remain not involved, if they were so before
the local decision is taken; otherwise, they become unaware.

Finally, the start nodes of blocks that are branches of parallel activities require special care. If the block is not a branch of a parallel activity, then no participant awareness states are modified. Otherwise, the block is a branch of a parallel activity, and we must take extra care with participants that are currently not involved. Branches of a parallel activity are enacted concurrently and, due to the flow algorithm used to annotate the participant awareness, the branches cannot “synchronize” on the participants becoming involved. That is, if the participant $p$ is not involved before a parallel activity, but it becomes involved in one of its branches, the flow algorithm would “copy” the information about $p$’s non-involvement on all its branches. Therefore, if $p$ is an acting participant in some action of a branch, marking it as non-involved in another branch would violate the “safe approximation” principle of participant awareness. In the definition of $f(n,p)$ shown in Table 5, this case is dealt with by marking a participant $p$ as unaware of the start node $n$ of a block that is a branch of a parallel activity if $p$ is non-involved in the $\text{AW}_{\text{fireable}}(n)$, and $p$ is an acting participant any other branch of the parallel activity. This is the case, for example, of the AWM node $\text{block [prl_1branch_2] start}$ in the running example shown in Figure 22. In fact, the participants $p_1$ and $p_2$ are unaware in $\text{AW}_{\text{fired}}(\text{block [prl_1branch_1] start})$ – instead of non-involved as reported in $\text{AW}_{\text{fireable}}(\text{block [prl_1branch_2] start})$ – because $p_1$ and $p_2$ are acting participants in the other branch of the parallel activity $\text{prl_1}$, i.e. the block activity $\text{prl_1branch_1}$.

5.2.4 Computational Complexity of the AAA

The AAA is a “classic” iterative fixed-point algorithm. Similar fixed-point flow algorithms, e.g. the ones for calculating dominance and post-dominance on CFGs, perform a number of iterations to update the values associated to the nodes proven to be between $O(|E(awm)| \times \log |E(awm)|)$ (see [Tar74]) and $O(|N(awm)|^2)$ (see e.g. [CHK01]), where $E(cfg)$ and $N(awm)$ denote the sets of control-flow edges and nodes of the CFG $cfg$, respectively. In the reminder we assume the latter, higher complexity, as an approximation of how many iterations are performed by the AAA to reach the fixed-point. Additionally, the AAA requires the execution of a reachability analysis between all the couples of nodes, which is known to have upper-bound $\log^2 |N(awm)|$.

Each invocation of the merge function $\cap$ has complexity linear to (1) the number participants and (2) the number of successors of the node whose input-value is being calculated. AWMs are generally structured so that each node has few successors/predecessors (see Section 4.1), so we can approximate the upper bound complexity of of $\cap$ to $O(|P(chor)|)$, where $|P(chor)|$ denotes the cardinality of the set of participants, i.e. how many
participants are altogether declared in the choreography.

The definition of the update-function $f(n, p)$ is case-based. The evaluation of all its cases can be considered to be $O(1)$, with the exception of the case in which $n$ is a block start node generated from a branch of a parallel activity; in this last case, the evaluation requires the traversal of the parse-tree of the other branches, and it can be conservatively estimated to $O(|A(chor) − 4|)$. The reason is simple: the activities that can specified in the other branches of the parallel can be at most $|A(chor) − 4|$, because for sure they do not contain the current branch, the activity that it necessarily contains (see ChorTex’s extended BNF in Section 2.1), the parallel activity in which the current branch is nested, and the root choreography.

Putting all the pieces together, the AAA has an upper-bound complexity of:

$$O(|N(awm)|^2 \times |P(chor)| \times |A(chor) − 4| + \log^2 |N(awm)|)$$

Since we are calculating the upper-bound complexity, we can simplify the estimation above as follows:

$$O(|N(awm)|^2 \times |P(chor)| \times |A(chor)|)$$

Surely, an over-approximation of 4 on the count of activities and a complexity of $\log^2 |N(awm)|$ do not matter much, given the fact that the dominant term is $|N(awm)|^2$; put formally:

$$|N(awm)|^2 \gg \log^2 |N(awm)|$$

(The sign $\gg$ reads “much bigger.”) Finally, we can observe that the number of nodes in a CFG is roughly linear with the number of activities specified by the choreography. (It is not possibly to quantify precisely the ration between activities and CFG nodes, as it depends on which type of activities are specified by the choreography.) In fact, basic- and iteration activities are mapped to sub-graphs with only one node. Choreographies, instead, are mapped to sub-graphs with at least two nodes (start and end) plus one node for each declared exception handler and propagating exception. The other activities are mapped to two nodes. In other words, the amount of nodes and activities are comparable, but there are strictly more nodes than activities (this is guaranteed by the root choreography alone). Therefore, the estimation of the upper-bound complexity of the AAA can be further conservatively simplified to:

$$O(|N(cfg)|^3 \times |P(chor)|)$$
6 Verifying the Strong Realizability of ChorTex Choreographies through Awareness Constraints

This section introduces the constraints on participant awareness states of an AWM of a ChorTex choreography that, if verified, ensure the strong realizability of the latter. We assume the goodwill of the participants, i.e. they do not knowingly violate the choreography. In other words, if a participant is aware that performing an action in the current enactment state would violate the choreography, it will not perform that action. Under the assumption of the participants’ goodwill, the strong realizability of a ChorTex choreography is verified if the three following strong-realizability requirements are met by its participants:

**Know when to act:** The acting participants of every action are immediately- or eventually aware of the latter becoming enactable;  

**Know when not to act:** The acting participants of every action are immediately aware of the latter becoming not enactable;  

**Know when the role is over:** In any enactment, each participant eventually knows when its part in it is over, i.e. if no matter how the enactment proceeds, no further actions are required by the participant and no further messages will be received.

The first two of the requirements above are of straightforward explanation. Firstly, if a participant with goodwill knows when it can perform its actions and when not, that participant will never violate the choreography. Secondly, if the performing of an action is necessary for the progress of the enactment, its acting participants know that the action can (and should) be performed, and hence they will eventually perform it.

The third requirement, i.e. that the participants when their roles are over, has a less immediate rationale and comes for the language-equivalence in terms of conversations between the composition of the roles and the choreography that is required by the definition of strong realizability. The reason is the following. As shown for example in [KP06], a choreography can be represented as a State-Transition System (STS). The roles specified by the choreography can also be modeled as separate STSs. The composition of the roles, therefore, is a composite STS the state space of which is the Cartesian product of the state spaces of the various STSs of the roles, and removing those that are not reachable to to the transitions in the single STSs. The final states of the composite STS are those in which all the separate roles

---

9 At the best of our knowledge, virtually every work in the state of the art on realizability, with the exception of [KMMN11], builds on this same assumption.  
10 The authors have pondered at length whether to name this requirement “He’s dead, Jim.” While not been a particularly academic name, it does sound eerily appropriate.
are themselves in a final state. Recall that the behavioral equivalence that is adopted in Definition 8 is language-equivalence. Put it in terms of STSs, language-equivalence requires (among other things) that all and only the conversations that lead the STS of the choreography to a final state also lead the composite STS of the roles to a final state. Naturally, a role that has not begun is in a final state because the corresponding participant has not yet been involved in the enactment. Therefore, in order for a choreography to be strongly realizable, in every possible enactment the participants that have been involved in it must know when their roles are completed.

In order to verify the strong realizability of a ChorTex choreography, the three strong-realizability requirements are “translated” to awareness constraints.

**Definition 18 (Awareness Constraints).** An *awareness constraint* is a first-order logic predicate evaluated on the node awareness states of an AWM. Besides the usual first-order logic operators, an awareness constraints admits the following predicates which test the participant awareness state of a participant $p$ with respect to the becoming fireable and the firing of the event represented by the node $n$ of the AWM:

$\text{AW}_{\text{fireable}}(n,p) = \alpha$ (Participant-awareness state in $\text{AW}_{\text{fireable}}$)

$\text{AW}_{\text{fired}}(n,p) = \alpha$ (Participant-awareness state in $\text{AW}_{\text{fired}}$)

$\text{SC}_{\text{out}}(n,p) = \text{CAN}$ ($p$ “spots” the end of its role from the node $n$)

(The latter predicate is reported here for completeness; the explanation of its meaning requires a large amount of groundwork, and is postponed to Section 6.3.) As in Section 5.2, the symbol $\alpha$ denotes one of the four participant-awareness states described in Section 5.1, i.e.:

$\alpha \in \{\text{ia, ea, ua, ni}\}$

Each awareness constraints is *attached* to one node of the AWM for allowing “short-hand” notations. In particular, in an awareness constraint attached to the AWM node $n$, the following couples of predicates are equivalent:

<table>
<thead>
<tr>
<th>Long-hand</th>
<th>Short-hand</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{AW}_{\text{fireable}}(n,p) = \alpha$</td>
<td>$\text{AW}_{\text{fireable}}(p) = \alpha$</td>
</tr>
<tr>
<td>$\text{AW}_{\text{fired}}(n,p) = \alpha$</td>
<td>$\text{AW}_{\text{fired}}(p) = \alpha$</td>
</tr>
<tr>
<td>$\text{SC}_{\text{out}}(n,p) = \text{CAN}$</td>
<td>$\text{SC}_{\text{out}}(p) = \text{CAN}$</td>
</tr>
</tbody>
</table>

That is, the short-hand version of the predicates “implies” the node whose node awareness state is tested to be the one to which the awareness constraint is attached.

Figure 24 exemplifies on the running example the types of awareness constraints that are presented in Section 6.1 through Section 6.3.
Figure 24: The Awareness Model presented in Figure 16, augmented with the awareness constraints; the satisfied awareness constraints are framed in green, the unsatisfied ones in red.
6.1 Know When to Act

The “know when to act” principle is translated to awareness constraints attached to nodes of the AWM that represent participant-activated events. There are two types of such awareness constraints, depending on whether the action associated with the participant-activated event is initiating or not.

6.1.1 Non-initiating actions

Let $n$ be an AWM node representing the firing of a participant-activated event $e_a$ associated with the non-initiating action $a$. To verify that the acting participants of $a$ “know when to act,” $n$ has attached the following awareness constraint requiring each acting participant of $a$ to be either immediately or eventually aware of $a$ becoming enactable (see Definition 13):

$$\forall p \in \text{actingParticipants}(a) : \text{AW}_{\text{fireable}}(p) \in \{\text{ia, ea}\}$$

The function $\text{actingParticipants}(a)$ returns the acting participants of the action $a$ (see Definition 1). The rationale of this awareness constraint is the following. The acting participants must be aware of when they can perform the action, i.e. when it becomes enactable (hence the constraint on $\text{AW}_{\text{fireable}}$). If the acting participants are immediately aware of when the action become enactable, they will be able to perform the action as soon as it would not violate the choreography. If they are eventually aware, they will be able to perform it at some point in the future. If the acting participants are unaware or not-involved, they can never perform that action for the risk of violating the choreography, and this constitutes a realizability defect.

An example of this type of awareness constraint is $c_3$ in Figure 24. The participant $p_3$ is unaware of the fact that it can dispatch of $m_3$, and therefore it never will, thus deadlocking the enactment. On the contrary, the decision maker $p_1$ of the choice activity $\text{choice}_1$ is eventually aware of the fact that it is required of deciding which branch to enact, and assuming that it will perform the decision in a finite amount of time, this guarantees the progress of the enactments (see awareness constraint $c_4$ in Figure 24).

6.1.2 Initiating actions

ChorTex choreographies have either:

- A single initiating action;
- Multiple, non mutually-exclusive initiating actions.

Mutually-exclusive initiating actions, i.e. initiating actions exactly one of which occurs in every possible enactment, are not possible due to the constructs of ChorTex. In fact, the only way to specify in ChorTex mutually
exclusive actions is within branches of a choice activity. However, the decision action of the choice activity *dominates* (i.e. is always performed before) the actions in all the choice’s branches. This implies that an action in a branch of a choice activity can never be initiating. As a consequence, ChorTex choreography can never have mutually-exclusive initiating actions.

In the case of a single initiating action, no awareness constraints are required to verify that participants “know when to act.” The acting participants of the one initiating action perform create the enactment by performing that action. In a sense, the initiating action is the first source of participant awareness in an enactment, and to perform it is reasonable not to require prior awareness, i.e. knowledge of what has happened before in an enactment that *did not yet exist*. It should be noted that it is outside of the scope of realizability analysis to determine how the acting participants of the initiating action decide to perform it.

On the contrary, in the case of multiple, non-exclusive initiating actions, awareness constraints are required to ensure that all the initiating actions can be performed *irrespective to which one is the first to occur in an enactment*. In the running example (see Figure 24) there are two initiating actions, namely the dispatching of $m_1$ by $p_2$ to $p_1$, and the dispatching of $m_2$ by $p_3$ to $p_1$ and $p_2$. Either of these two actions may initiate an enactment of the choreography. Moreover, both actions *must* be performed in every enactment, because the parallel activity $prl_1$ does not complete until both messages have been dispatched. In each enactment, either $m_1$ is firstly dispatched, and then $m_2$, or vice-versa. For both the senders of the two message exchanges, namely $p_2$ and $p_3$, to be able to dispatch their message with the certainty of not violating the choreography, it must be the case that the dispatching of one of the messages “provides enough awareness” to the sender of the other to be able to dispatch the other message. Unfortunately, this is not the case of the running example. The dispatching of $m_2$ by $p_3$ to $p_1$ and $p_2$ makes the sender of $m_1$, namely $p_2$, eventually aware, and therefore able to later dispatch $m_1$. On the other hand, consider the case in which the enactment is initiated by the dispatching of $m_1$. The participant $p_1$ and $p_2$ are eventually- and immediately-aware of the dispatching of a message of type $m_1$, respectively. However, since $p_3$ is not a recipient of $m_1$, it does not know that the enactment has been initiated, and therefore it will not know that it must send $m_2$. This causes a deadlock: since the message exchange activity $mex_2$ cannot be enacted (its acting participant is not aware that it can – and should – dispatch the message), the parallel activity $prl_1$ can never be completed, and the enactment is stuck.

Generalizing from the example reported above, the participant awareness resulting by the performing of each initiating action must enable the acting participants of all the other initiating actions to perform them. Therefore, enactments can be begun by any of the initiating actions without resulting in deadlocks, exactly as required by the choreography. Put formally, given
the set of initiating actions $A := a_1, \ldots, a_m$ of the choreography, each of the
nodes $n_{a_1}, \ldots, n_{a_m}$ representing the participant-activated events associated
with the initiating actions $a_1, \ldots, a_m$ has associated the following awareness
constraint:

$$\forall p \in \left( \bigcup_{a' \in A \setminus \{a\}} \text{actingParticipants} (a') \right) : \text{AW}_\text{fired} (p) \in \{\text{ia}, \text{ea}\}$$

The awareness constraint above can be paraphrased in natural language as follows: the acting participants of every initiating action other than $a$ must be eventually- or immediately aware of the performing of $a$. Thus, no matter which initiating action is the first to be performed in the enactment, all the others can be performed as well by their respective acting participants.

### 6.2 Know When Not to Act

When required to perform an action, i.e. generating a message or taking a
decision, a participant is not required to take that action immediately. In the
life-cycles of ChorTex activities presented in Section 2.2.2, this is represented
by the PENDING states of message-exchange (Figure 6), choice (Figure 10)
and iteration (Figure 11) activities. That is, some time may pass between
the moment an action becomes enactable (see Definition 13), and when it is
actually enacted. The awareness constraints that realize the “know when to act”-requirement (presented in Section 6.1) ensure that the acting partici-
pants know when their actions become enactable. However, those awareness
constraints do not account for the cases in which an action that is enactable
at one point in time becomes later non-enactable due to changes to the en-
actment state. Specifically, actions change from enactable to non-enactable
due to the throwing and propagation of exceptions (see Section 2). The
operational semantics of exception throwing specifies that, as soon as an ex-
ception is thrown, all the other activities that are being enacted as branches
of a parallel activity are terminated. In such situations, the unawareness
of acting participants with respect to the current non-enactability of their
actions may mislead them into violating the choreography.

Consider the choreography shown in Figure 25. After that the partici-
O01971001
.object participates $p_1$ has dispatched a message of type $m_1$ to $p_2$, the parallel $\text{prl}_1$ activity
is enacted, which in turn causes the enactment of the message exchange activity $\text{mex}_2$ and the choice activity $c_1$. Currently, there are two enactable
actions: the dispatching of a message of type $m_2$ by $p_2$ to $p_1$, and the decision
by $p_1$ on which branch of the choice activity $c_1$ to enact. In an enactment
in which $p_1$ decides “true” in the choice activity $c_1$ before $p_2$ dispatches
the message $m_2$, the latter action becomes non-enactable. In this case, if
the sender of $\text{mex}_2$, i.e. $p_2$, dispatches $m_2$ to $p_1$ despite the fact that the
exception $e_1$ has been thrown, the choreography is violated.
Figure 25: A choreography in which actions can change from enactable to non-enactable due to the throwing of exceptions.

6.2.1 Identifying Concurrently-Acting Participants

To verify that a choreography does not have realizability defects such as those of the choreography in Figure 25, we need awareness constraints that verify that the participants that may act in other concurrently-enacted branches are immediately aware of the throwing of exceptions or their propagation from nested choreographies. Figure 26 presents an algorithm that derives such awareness constraints by exploiting the block-based structure of ChorTex to identify which parallel activities are terminated when a certain propagates exception event is fired in an enactment. The function \( \text{parent}(A) \) returns the activity in which \( A \) is nested. If \( A \) is the root choreography (which, by definition, has no parent activity), \( \text{parent}(A) \) returns NULL. The function \( \text{sourceActivity}(n) \), instead, returns the activity from which the CFGs node \( n \) was generated (see Section 4.1).

The algorithm in Figure 26 is divided in two phases. First, it identifies the parallel activity that encompasses all those activities that may be terminated when the propagates exception node \( n \) is traversed (i.e. when the event represented by \( n \) in the AWM is fired). This parallel activity, called outer parallel, is the last one encountered navigating upwards – i.e. from nested activities to those encompassing them – the hierarchy of parents of \( n \) until the first choreography is found. (If there are parallel activities nested into each other without choreographies to “insulate” them, they are all terminated by the throwing of an exception in any of their branches.) It could
/* n is a propagates exception node */
/* the algorithm attaches the newly generated constraint to n */
function deriveAwarenessConstraint (node n)
    returns constraint {
    /* A is the activity from which the propagates exception node */
    /* n has been generated (i.e. a throw activity or a choreography) */
    activity A := sourceActivity(n);

    /* identification of the “outer” parallel activity */
    parallel activity outerParallel := NULL;
    activity A′ := parent(A);
    /* A′ is NULL if A is the root choreography */
    /* the iteration terminates when A′ is a choreography */
    /* (not necessarily the root one) */
    while (A′ not NULL & & A′ is not choreography) do
        if (A′ is parallel do A_1 and ... and A_n) then
            outerParallel := A′;
        end if
        A′ := parent(A′);
    end while

    /* identification of the nodes that represent actions that */
    /* may be made non-enactable after the traversing of */
    /* the propagates exception node n */
    if outerParallel not NULL then
        /* the set P contains the acting participants of actions */
        /* that can be concurrently enacted */
        set P := ∅;
        /* retrieve/generate chor’s CFG sub-graph */
        /* as described in Section 4.1 */
        foreach node n′ in generateCFG(outerParallel) do
            if (not n′ dominates n & &
                not n dominates n′) then
                P := P ∪ actingParticipants(n′);
            end if
        end foreach
        return new constraint c := \(\prod_{p \in P} AW_{fired}(p) = ia\);
    end if
}

Figure 26: Pseudo-code for deriving the awareness constraint attached to
the propagates exception node n.
be the case that no outer parallel is found, i.e. if:

- the *propagates exception* node $n$ is not nested into any parallel activity (in which case the traversing of $n$ makes no action non-enactable, since none may be enacted concurrently);

- the *propagates exception* node $n$ is nested into a parallel activity, but $n$ is encapsulated into a choreography nested in the parallel activity and, therefore, the exception is handled by that choreography’s exception handlers. (However, that nested choreography might have its own *propagates exception* nodes, which will be associated with awareness constraints if necessary.)

In both the above cases, no awareness constraint is needed for $n$. The reason is that, since no outer parallel activity is found, there are no concurrently-enacted actions that might be rendered non-enactable by the propagation of the exception.

In case an outer parallel is found, the second part of the algorithm identifies which actions might be made non-enactable by the traversing of the *propagates exception* node $n$. These actions are all those represented by some node in the CFG sub-graph of the parallel activity identified before, with the exception of the nodes dominated $n$ (because they either have already been enacted before $n$ is traversed) or those that dominate $n$ (because they can be enacted only after $n$, and therefore they will never be enacted). Once all the actions that might be made non-enactable are found, the awareness constraint is composed by requiring that each acting participant of at least one of those actions is immediately-aware of the firing of the event represented by $n$.

### 6.2.2 Computational Complexity

The algorithm in Figure 26 requires the traversing of the parsing tree of the choreography, and it can be (very roughly) estimated as quadratic with respect to the number of activities in the choreography, i.e. the max depth of the parsing tree, times how many activities may be terminated by the enactment of a throw activity which, once again, we conservatively estimate it to be less or equal to the number of activities specified by the choreography. We have already estimated the number of activities to be linear to the number of nodes in the CFG, and therefore this upper-bound can be over-approximated to:

$$O(|N(awm)|^2)$$

More precise approximations are possible but not worth the effort, because we are considering the computational upper-bound, and this is not the highest among the various algorithms that compose the realizability analysis (see Section 7).
6.3 Know When the Role is Over

A final state is an enactment state (see Definition 5) in which no further action is required of the participants. Notice that a deadlock is not a final state; a deadlock, in fact, is an enactment state in which further action is required of the participants, but they cannot perform it. The reaching of a final state in the enactment of a choreography is represented in the respective AWM by the traversing of an end node. Choreographies may admit multiple final states, each representing a different outcome. As a consequence, AWMs may have multiple end nodes, each associated with a different final state. For example, the AWM shown in Figure 24 has two end nodes. The node $\text{chor [chor}_1\text{]} \text{end}$ represents the successful completion of the choreography. Instead, $\text{chor [chor}_1\text{]} \text{err e}_2$ represents the termination of the choreography because of an exception of type $e_2$ that cannot be handled and that propagates outside the root choreography.

If verified on a choreography, the awareness constraints presented in Section 6.1 and Section 6.2 guarantee that exist participant implementations the composition of which is trace-equivalent to the choreography, i.e. it can enact exactly (i.e. all and only) the conversations specified by the choreography. However, the definition of strong realizability for ChorTex choreographies requires more than trace-equivalence: it requires language-equivalence (see Definition 8). Language-equivalence is a stronger relation than trace-equivalence because it additionally requires that every conversation that leads the choreography to a final state also leads the composition of participant implementations to a final state, and vice-versa [BIM95]. This additional requirement is the reason for the the “know when the role is over”-requirement, which is verified by the awareness constraints presented in the reminder of this section.

Adopting the formalization proposed in [KP06] of choreographies, participant implementations and their compositions as STSs, a composition of participant implementations reaches a final state if and only if each of the composed participant implementation reaches a final state. The final states of a participant implementation are its internal states that represent the completion of the role in some enactments. Naturally, participants that are not yet involved in an enactment before the latter ends need not know that their role in that enactment is over, since, in fact, it had not even yet begun.

Unlike the case of choreographies, the reaching of a final state by the participants is not explicitly represented in the AWMs. This is further complicated by the fact that, when the enactment is completed, all roles are over, but not all the participants are necessarily aware of it. Consider the choreography shown in Figure 27. An enactment is completed when $p_1$ decides not to iterate the activity loop further. Since $p_1$ takes the decision that immediately results in the completion of the enactment, it knows when the enactment is over, and thus that also its role is. On the other hand, $p_2$
does not know when the enactment is over: its role is to receive an unspecified number of messages of type $m_1$, as long as they keep being sent. When $p_1$ decides not to further iterate the activity $loop$, $p_2$ will still wait indefinitely for more messages that will never be dispatched. This “hanging” of $p_2$ violates the “know when the role is over”-requirement for strong realizability, and thus constitutes a realizability defect: since $p_2$’s participant implementation cannot reach a final state by the time the enactment ends, the composition of the participant implementations of $p_1$ and $p_2$ does neither, and therefore the composition of the participant implementations and the choreography are not language-equivalent.

To avoid realizability defects like the one shown in the previous example, we need to verify that, by the time an enactment ends, all the participants involved in it know that their roles are over. A participant knows that its role in an enactment is over by implying that its role is over based on the events it can observe (see Definition 15) and on how the choreography is specified. The explanation of how participants do this requires some groundwork.

Definition 19 (Final Observable Actions and Final Observable Nodes). An action $a$ is final observable action for a participant $p$ if there is at least one valid enactment trace for the choreography in which $a$ is the latest observable action (see 15) by $p$. An AWM node $n$ is a final observable node for a participant $p$ if it represents the firing of a participant-activated event associated with the performing of a final observable action for $p$.

It is important to notice that in some enactment a participant might observe multiple final observable actions being performed, as well as one final observable action being performed multiple times. For instance, in the running example (see e.g. Figure 24) the participants $p_2$ and $p_3$ have two final actions, namely the dispatching performed when enactment the message exchange activities $mex_3$ and $mex_4$. A choreography in which one final action can be performed multiple times in an enactment is the one shown in Figure 25. In that choreography, $p_2$ has only one final action, namely $[mex_1] p_1 \rightarrow m_1 p_2$ which, depending on the decisions taken by $p_1$, can be performed any number of times.

The identification of the final observable nodes for a participant $p$ is easily achieved by adapting the classic flow-analysis method for calculat-

```plaintext
chor [chor1] ( { iteration [loop] p1 do { [mex1] p1 → m1 to p2 } } )
```

Figure 27: A choreography in which $p_2$ does not know when its role is over.
ing reaching definitions. In imperative programming languages, a reaching
definition for an instruction $i$ is an assign instruction $i'$ affecting a certain
variable $x$ that may reach $i$ without an intervening assignment to $x$ [Aho07].
For reasons of brevity we omit the details of the reaching definitions algo-
rithm, which can be found for example in [MR90]. The intuition of the
reaching definitions algorithm is the following: each node $n$ representing
an instruction in the program is annotated with pointers to those assign-
ment instructions that are reaching definitions to $n$ for some variable in the
program. These pointers to reaching definitions are grouped by the vari-
bles they affect. Depending on the CFG of the program, one node may be
annotated with multiple reaching definitions for one certain variable.

The problem of identifying the final action nodes of a participant $p$ can
be mapped to the concept of reaching definitions as follows. If $p$ is an
acting participant of $n$, it counts as an assignment of the value $n$ to the
“variable” named $p$. All the reaching definitions $n_1, \ldots, n_m$ for $p$ to an end
node $n_e$ are the final observable nodes of $p$ in all the enactments ending
with $n_e$. Naturally, it is possible that the various end nodes are associated
with different sets of reaching definitions. The overall set of final observable
nodes for $p$ is the union of the reaching definitions for $p$ on all the end nodes
of the AWM. In the reminder, the function that returns the set of final
observable nodes of the participant $p$ in an AWM is denoted by:

$$finalObservableNodes(p)$$

As previously mentioned, a participant might observe multiple final ac-
tions being performed in one enactment. One of those final actions will
actually be the last one that the participant observes in the enactment
(and therefore the one that signifies the end of that participant’s role), but
how can the participant know which one? Consider again the choreography
shown in Figure 27: the only final observable node of the participant $p_2$ is
$[mex_1] p_1 \rightarrow m_1$ to $p_2$, but the dispatching by $p_1$ of a message of type
$m_1$ to $p_2$ may happen any number of times in an enactment, depending on
the decisions of $p_1$. In such a case, in order for a participant to imply that
its role is over, the choreography must be specified so that the participant is
aware of the firing of canary events, i.e. events whose firing guarantees that
no final actions for the participant can be further performed.

**Definition 20** (Canary Events and Canary Nodes). The event $e$, the firing
of which is represented in the AWM by the node $n$, is a canary event for
the participant $p$ if no final observable node of $p$ is reachable from $n$ and $p$
is immediately- or eventually-aware of the firing of $e$. Put formally:

$$\mathcal{E} (n, p) := (\exists n' \in finalObservableNodes(p) : n'$ is reachable from $n) \land
\left(\text{AW}_\text{fired}(n, p) \in \{\text{ia}, \text{ea}\}\right)$$

An AWM node that represents a canary event is a canary node.
In other words, the observation of the firing of a canary event by a participant guarantees that no actions observable by that participant lie on any of the paths connecting the respective canary node and any end nodes.\(^{11}\) Most canary events are reactive events (see Definition 11), i.e., those events that are triggered “on-cascade” by the performing of some action.

As discussed in Section 6.2, because of parallel activities, an enactment may concurrently traverse multiple paths on the AWM. This means that a participant might need to observe multiple canary events before it can deduce its role is over.

**Definition 21 (Sufficient Canary Events and Nodes).** A set of canary events \(e_1, \ldots, e_m\) of the participant \(p\) is **sufficient for the event** \(e\) if, in any possible enactment, after the firing of \(e\) the participant \(p\) can observe at least one of \(e_1, \ldots, e_m\). Similarly, the set of canary nodes \(n_1, \ldots, n_m\) of the participant \(p\) is **sufficient for the node** \(n\) if every path connecting \(n\) with any of the end nodes of the AWM contains at least one of \(n_1, \ldots, n_m\).

That is, a set of canary events of a participant is sufficient for a certain event \(e\) if, no matter how the enactment evolves after the firing of \(e\), at least one of those canary events will be observed by that participant. This provides a way of verifying the “know when the role is over”-requirement for strong realizability: a participant knows when its role is over if, for each of its final observable actions, there is a sufficient set of canary events.

### 6.3.1 The Bird-Watching Algorithm

The Bird-Watching Algorithm (BWA) presented in Figure 28 checks if a participant \(p\) has sufficient canary events for each of its final observable actions.\(^{12}\) Similarly to the AAA presented in Section 5.2, the BWA presented in Figure 28 is a flow-analysis algorithm. The idea is to use a fixed-point algorithm instead of an enumeration of the paths, possibly infinitely many due to loops in the AWMs, that connect final observable- and end nodes.

---

\(^{11}\)The term “canary event” is clearly a reference to the distasteful mining practices (hopefully) of yore. In stark disregard of animal life, miners brought along caged canaries down the shafts. The canaries were (ab)used as low-tech gas detectors, as they would die sooner than the miners in the presence of toxic gases such as carbon monoxide, methane or carbon dioxide. Thus, the death of a canary meant that the shaft had to be fled really, really quickly. In our realizability analysis, the firing of canary events signals the end of a participant’s role. We like to think that our canaries deliver their signal by singing, instead of by perishing. No innocent birds need ever shed their lives on the altar of choreography realizability.

\(^{12}\)The name “Bird-Watching Algorithm” is an admittedly bad pun on the fact that the participants need to be able to “spot” canaries. Angry Birds Algorithm (ABA) has also been suggested as an alternative. While recognizing that ABA is a clearly superior name, the authors cannot run the risk of rousing the anger of a certain Finnish entertainment media company. It cannot be a good idea to cross people capable of ruthlessly achieving near world-domination (of the mobile gaming market) by means of slings and detonating avians.
/* awm is the AWM of the choreography */
/* p is the participant being checked */
/* when the BWA terminates, each node of awm reachable from */
/* the start node is annotated with input- and output values */

function birdingWatchingAlgorithm (awareness model awm, 
  participant p) returns void {
  /* retrieve the start node of the AWM */
  node start := startNode(awm);

  /* initialization; N(awm) is the set of nodes in the AWM awm */
  foreach node n in N(awm) reachable from start do
    /* “conservative” initialization */
    SC in(n) := CANNOT;
    SC out(n) := fE(p,n,SC in(n));
  end foreach

  /* iteration until a fixed-point is reached */
  repeat
    foreach node n in N(awm) do
      /* safe-approximation of the outputs of n’s successors */
      /* see Section 6.3.2 */
      SC in(n) := \bigwedge_{n’ \in \text{succ}(n)} (SC out(n’));
      /* update the output-value of n, see Section 6.3.3 */
      /* the identification of canary events performed */
      /* during fE can be cached for reuse */
      SC out(n) := fE(p,n,SC in(n));
    end foreach
    until no SC out(n) changes
  }

Figure 28: Pseudo-code of the Bird-Watching Algorithm.
The BWA is run once for each of the choreography’s participants. As opposed to the forward direction of the AAA (see Section 5), in the BWA the paths in the AWM are traversed by following the control-flow edges backwards. In the run of the BWA for a participant $p$, each node $n$ is associated in the BWA with input and output values, denoted by in this case by $SC_{\text{in}}(n)$ and $SC_{\text{out}}(n)$, respectively. ($SC$ stands for “Spots Canaries”.) $SC_{\text{in}}(n)$ represents the safe-approximation of the output-values of the successors of the node $n$, and $SC_{\text{out}}(n)$ symbolically represents the capability of $p$ to spot the end of its role after the traversal of $n$. The possible values of $SC_{\text{in}}(n)$ and $SC_{\text{out}}(n)$ are the following:

- **CAN**: $n$ is a canary node of $p$, or there are sufficient canary events of $p$ for the node $n$ (see Definition 21);
- **CANNOT**: $p$ does not have sufficient canary events for the node $n$.

Ideally, when the BWA reaches a fixed-point, all the final observable nodes of $p$ have CAN as output-value. In fact, if a final observable node $n$ has an output-value different than CAN, it means that there is some path connecting $n$ and some end node $n_e$ in which $p$ cannot spot canary events that let it imply the end of its role. Therefore, a final observable node of $p$ not annotated with CAN as output-value in the fixed-point reached by the BWA is a symptom of a realizability defect. In the AWMs, this is represented by means of the following awareness constraints attached to each of the final observable nodes $n_{f_1}, \ldots, n_{f_m}$ of $p$:

$$SC_{\text{out}}(n_{f_i}) = \text{CAN}$$

Figure 29 shows the input- and output-values associated with the nodes in the fixed-point obtained by applying the BWA on the AWM of the running example, as well as the awareness constraints that verify for $p_3$ the “know when the role is over”-requirement. (For completeness, the same awareness constraints are also reported in the AWM of the running example shown in Figure 24.) The participant $p_3$ has two final observable nodes, namely:

- $[mex_3] \ p_3 \rightarrow m_3 \to p_1, \ p_2$
- $[mex_4] \ p_1 \rightarrow m_4 \to p_2, \ p_3$

Each of them is associated with an awareness constraint that verifies if, when the BWA reaches the fixed-point, the output value associated to the node is CAN. This is the case of the $[mex_4] \ p_1 \rightarrow m_4 \to p_2, \ p_3$ node. There is only one path connecting it to the only reachable end node, namely $\text{chor} \ [\text{chor}] \ \text{end}$, and along that path each node is a canary node. (This is a special case, due to the fact that no further actions are performed by any participant along that path.) The awareness constraint associated to the node $[mex_3] \ p_3 \rightarrow m_3 \to p_1, \ p_2$ is instead unsatisfied. The reason
Figure 29: The fixed-point reached by the BWA on the AWM shown in Figure 22 for the participant \( p_3 \), with additionally the “know when the role is over” awareness constraints for \( p_3 \) (boxed in green when satisfied, red otherwise) and the highlighting of canary- and final observable nodes.
is that the node \( \text{mex}_{3} \) \( p_3 \rightarrow m_3 \) to \( p_1, p_2 \) is a final observable node, and if \( p_1 \) decides to execute the \textit{choice}_1 \textit{either} branch, there are no canary
nodes for \( p_3 \) on the path leading to \textit{chor} \[ \text{chor}_1 \] \textit{err} \( e_2 \). Therefore, in
enactments in which \( p_1 \) decides to enact the \textit{choice}_1 \textit{either} branch, \( p_3 \) cannot
“know when the role is over.”

### 6.3.2 Merge of Input Values

The merge of output values of multiple successors is handled by the function \( \bigwedge \) defined in Figure 30.\(^{13}\) The function \( \bigwedge \) takes as arguments one or
more output-values of nodes, and returns \texttt{can} if all the values are \texttt{can}, and
\texttt{cannot} otherwise.

\[
\bigwedge (v_1, \ldots, v_m) := \begin{cases} 
\texttt{can} & \text{if } \forall i \in [1, m] : v_i = \texttt{can} \\
\texttt{cannot} & \text{otherwise}
\end{cases}
\]

Figure 30: Recursive definition of the function \( \bigwedge \) which merges output
values of the predecessors of a node in the BWA.

### 6.3.3 Update of Output Values

The output-value of nodes is calculated using the function \( f^* \) defined in
Figure 31. The predicate “\( n \) is canary node of \( p \)” is defined in Definition 20.
The set \( N^{\text{end}}(\text{chor}) \) is the set of end nodes of the choreography \textit{chor}. The
function first \((A)\) returns the AWM node generated from the activity \( A \) and
labeled as “first.” Finally, and the function \( \text{source.Activity}(n) \) returns the
activity from which the AWM node \( n \) has been generated (see Section 4.1).

The outcome of \( f^* (p, n, v) \) is \texttt{can} in the following three cases:

- \( n \) is a canary node for \( p \);
- \( v \) is \texttt{can} (due to the “otherwise” case);
- \( n \) is a parallel split node, and every end node reachable from \( n \)
is also reachable from the first node of one or more branches of the
parallel activity that have \texttt{can} as output value.

\(^{13}\)Apologies are extended to those readers who, like the authors, find the \( \bigwedge \) symbol
somewhat creepy. But an eye symbol is just \textit{too} appropriate in the context of the Bird-
Watching Algorithm.
\[ v \in \{\text{CAN, CANNOT}\} \]

\[
 f^\bowtie (p,n,v) := \begin{cases} 
 \text{CAN if } f(n,p) \\
 \text{CAN if } \neg f(n,p) \land n = \text{parallel split} \land \\
 \quad \text{sourceActivity}(n) = \text{parallel do } A_1 \text{ and } \ldots \text{ and } A_m \land \\
 \quad \forall n_e \in \mathbb{N}^\text{end} \text{ (chor) } \exists i \in [1,n] : n_e \text{ reachable from first } (A_i) \land \\
 \quad \text{SC}_{\text{out}} \text{ (first } (A_i)) = \text{CAN} \\
 v & \text{otherwise}
\end{cases}
\]

Figure 31: Definition of the function \( f^\bowtie \) which updates output values in the BWA.

The rationale of the last case is the following. Recall that, since there are possible multiple end nodes for the choreography, it can be the case that more than one end node is reachable from the parallel split node, i.e. in the case that one or more exceptions can propagate from the branches of the parallel activity. Since all the branches of a parallel activity are concurrently enacted, it is not necessary for the participant \( p \) to observe along each branch canary events for each end node that can be reached from the parallel split. Instead, given one end node, it is sufficient that for \( p \) to observe a canary event – and therefore deduce the end of its role – along any of them. In the definition of the function \( f^\bowtie \), this is specified by checking that for each end node reachable from the parallel split node, the parallel activity has at least one branch whose “first node” is annotated with CAN as output-value (i.e. canary events will be spotted along that branch), and that the end node is reachable from it.

### 6.3.4 Computational Complexity of the BWA

The BWA is a fixed-point flow algorithm that is performed once per participant in the choreography. Each invocation of the merge function \( \lambda^\bowtie \) has complexity linear to the number of successors of the node whose input-value is being calculated. AWMs are generally structured so that each nodes has few successors/predecessors, so we can approximate the evaluation of \( \lambda^\bowtie \) to \( O(1) \).

The definition of the update-function \( f^\bowtie \) is case-based. The upper-bound complexity of evaluation of its first case, i.e. if the node \( n \) is a canary node, is linear with with the amount of final observable nodes (see Definition 20). The same argument we have made for \( \lambda^\bowtie \) can be applied here, and the evaluation of this case is approximated to \( O(1) \). The second case
depends on the amount of end nodes of the choreography and the number of branches of a parallel activity. The amount of end nodes of a choreography and branches of parallel activities is usually negligible with respect to the overall amount of activities specified, therefore we also approximate the evaluation of this case to $\mathcal{O}(1)$.

Notice that a reachability analysis has already performed when performing the AAA algorithm (see Section 5.2.4), and since the AWM nodes and edges have not since changed, its outcome can be reused.

The identification of the final observable nodes is performed by “re-purposing” the classic flow-analysis algorithm to calculate reaching definitions, and its upper-bound complexity is thus $(|\mathbb{N}(awm)|^2)$. The final observable nodes can be calculated only once per participant, and the outcome of this analysis can be used in both the BWA and the identification of canary nodes performed in $f^\infty$.

The initialization phase of the BWA is performed once per node; since we have estimated the upper-bound complexity of the function $f^\infty$ to $\mathcal{O}(1)$, the upper-bound complexity of the initialization phase is:

$$\mathcal{O}(\mathbb{N}(awm))$$

To reach the fixed-point, the BWA performs $\mathcal{O}(|\mathbb{N}(awm)|^2)$ iterations, each one with complexity of $\mathcal{O}(1)$ (remember that both $\lambda^\infty$ and $f^\infty$ have upper-bound complexity of $\mathcal{O}(1)$). Therefore, the overall upper-bound complexity of a run of the BWA for a given participant is:

$$\mathcal{O}(|\mathbb{N}(awm)| + |\mathbb{N}(awm)|^2) \approx \mathcal{O}(|\mathbb{N}(awm)|^2)$$

Since the BWA is performed once per participant, the overall upper-bound complexity of all its runs in the scope of one realizability analysis is the following:

$$\mathcal{O}(|\mathbb{N}(awm)|^2 \times |\mathbb{P}(chor)|)$$

7 Discussion

To the best of our knowledge, the concept of awareness has been previously adopted to investigate the realizability of interaction choreographies only in our previous work [MCvdHP08] and by Desai and Singh in [DS08]. Both previous approaches assume synchronous communication between the participants and, as a consequence, the notion of awareness therein explored is on a “yes/no” basis, i.e. a participant is either (immediately) aware of an event, or unaware of it. In this work we reworked the notion of awareness to accommodate asynchronous messaging. Since participants involved in an action may become aware of its completion at different points in time (e.g. the sender and the recipients of one message exchange), we have introduced the novel concept of “eventual awareness,” i.e. that a participant will become aware at some later time that an event has been or could be fired.
7.1 Overall Upper-Bound Computational Complexity of the Awareness-Based Realizability Analysis

The diagram in Figure 32 combines the information on the upper-bound complexity of the Awareness Annotation Algorithm (Section 5.2.4), the Bird-Watching Algorithm (Section 6.3.4), and the unnamed algorithm presented in Section 6.2.1 with the information on the complexity of generating the CFG and generating and verifying the awareness constraints that is discussed in the remainder of the present section.

The complexity of generating a CFG from a choreography is linear with respect to the number of activities therein defined (see Section 4.1). As discussed in Section 5.2.4, the amount of nodes in the CFG is comparable (but strictly bigger) than the amount of activities specified in the choreography.

The upper-bound complexity of the AAA was estimated in Section 5.2.4 to the following:

\[ O(|N(awm)|^3 \times |P(chor)|) \]

The notation \(|N(awm)|\) denotes the amount of nodes in the AWM \(awm\). Since the upper-bound complexity above is higher than those of all the other algorithms employed in the realizability analysis, it can be assumed to be the overall upper-bound complexity.

Once generated, the evaluation of awareness constraints is mostly a matter of looking up a pre-calculated value annotated on the AWM nodes, and it is therefore negligible. The generation of the awareness constraints, on the other hand, is significant in terms of computational complexity, and depends on the type of awareness constraints taken into account. In the case of the awareness constraints that realize the “know when to act”-requirement, it is a matter of scanning all the nodes of the AWM and looking up their acting participants; therefore, it has an upper-bound computational complexity of \(O(|N(awm)|)\). The upper-bound complexity of the algorithm to generate the “know when to act” has been estimated in Section 6.2.2 to:

\[ O(|N(awm)|^2) \]

Finally, the generation of the awareness constraints for verifying the “know when the role is over”-requirement consists in performing the BWA once for every participant in the choreography. Its upper-bound complexity has already been estimated in Section 6.3.4 to:

\[ O(|N(awm)|^2 \times |P(chor)|) \]

Combining the upper-bound computational complexity of all its parts, the overall upper-bound computational complexity of the awareness-based realizability analysis presented in this work is shown in Figure 33. The final outcome is approximated keeping into account that (1) we are looking for the upper-bound complexity (hence only the “biggest” term in the sum really...
Figure 32: Diagram that summarizes the awareness-based realizability analysis, and correlates the upper-bound complexity computational complexity of its different parts.
\[ O(\|A(\text{chor})\|) + \quad \text{(Generation CFG)} \]
\[ O(\|N(awm)\|) + \quad \text{(Domination Analysis)} \]
\[ O(\log^2 |N(awm)|) + \quad \text{(Reachability Analysis)} \]
\[ O(|N(awm)|^3 \times |\mathcal{P}(\text{chor})|) + \quad \text{(Awareness Annotation Algorithm)} \]
\[ O(|N(awm)|) + \quad \text{(Generation “know when to act” constraints)} \]
\[ O(|A(\text{chor})|^2) + \quad \text{(Generation “know when not to act” constraints)} \]
\[ O(|N(awm)|^2 \times |\mathcal{P}(\text{chor})|) \approx \quad \text{(Generation “know when the role is over” constraints)} \]

\[ O(|N(awm)|^4) \quad \text{(Overall Upper-Bound Complexity)} \]

Figure 33: Overall upper-bound computational complexity of the awareness-based realizability analysis of ChorTex choreographies.
counts), and that the number of activities in the choreography is always smaller than the nodes in the respective AWM (as discussed earlier in this section).

Given the variability of realizability definitions in the state of the art (see Section 3), and their dependency on particular choreography modeling languages (see Section 3.1), it makes little sense to compare realizability analysis methods devised for different languages. Nevertheless, the computational complexity of our method compares very favorably with the methods based on automata [FBS05, KP06, MFEH07, HB10] or Petri-Nets [DW07, LW09], which resort to PSPACE-complete language-equivalence checks (see also [LW11]). The computational complexity of our realizability analysis method comes mostly from the adoption of eventual awareness. Intuitively, eventual awareness is an “expedient” to reduce the size of the state-space of the realizability problem by hiding the actions of receiving messages performed by the recipients.

7.2 A Case Against Exception Handling in Choreographies

The awareness constraints realizing the “know when not to act”-requirement make a compelling case against the adoption in choreography modeling languages that allow asynchronous communication of exception handling constructs that are closely inspired from Object-Oriented (OO) programming languages. The prevention of violations caused by the throwing of exceptions imposes strong limitations on the design of ChorTex choreographies. In fact, exception throwing is “safe” only if, whenever it may occur, the actions that may be concurrently performed have the same, unique acting participant. An intuitive proof is based on the observation that at most one acting participant is immediately aware after the performing of any one action (see Table 5). Since all participants that may act concurrently to the throwing of an exception must be immediately aware of the latter ( “know when not to act”-requirement, see Section 6.2), and that only one can be so at any time, it follows that there can be only one unique acting participant in parts of strongly-realizable choreographies that can be terminated by the throwing or propagation of exceptions.

Interestingly, a similar observation is found in the BPMN v2.0 Choreography specification [OMG11, p. 344] with respect to termination end events:

“[Terminate end events can be used in a Choreography, however] there would be no specific ability to terminate the Choreography, since there is no controlling system. In this case, all Participants in the Choreography would understand that when the Terminate End Event is reached (actually when the Message that precedes it occurs), then no further messages will be expected in the Choreography, even if there were parallel paths. The use of
the Terminate End Event really only works when there are only two Participants. If there are more than two Participants, then any Participant that was not involved in the last Choreography Task would not necessarily know that the Terminate End Event had been reached.”

This is not surprising, given the fact that a BPMN v2.0 termination end event has the same effect of the throwing of an exception that terminates an entire ChorTex choreography. Exception handling and termination events are constructs that work well in service orchestrations, where all the activities are executed by the same actor (e.g. the orchestration engine).

Given the tight relationship between service orchestrations and choreographies, we understand how tempting it is to “port” service orchestration constructs to service choreographies. After all, similar constructs in service orchestrations and choreographies intuitively simplifies the projection of the roles [YZCQ07] and softens the learning curve for the modelers. However, the distributed nature of service choreographies most likely requires a radical re-thinking of how error handling is performed, see e.g. [KWL09].

References


[BCC+06] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve


Automated Attribute Inference in Complex Service Workflows Based on Sharing Analysis

Dragan Ivanović
Universidad Politécnica de Madrid (UPM)
idragan@clip.dia.fi.upm.es

Manuel Carro
Universidad Politécnica de Madrid (UPM) and
IMDEA Software Institute
{mcarro,hermen}@fi.upm.es
manuel.{carro,hermenegildo}@imdea.org

Abstract—The properties of data and activities in business processes can be used to greatly facilitate several relevant tasks performed at design- and run-time, such as fragmentation, compliance checking, or top-down design. Business processes are often described using workflows. We present an approach for mechanically inferring business domain-specific attributes of workflow components (including data items, activities, and elements of sub-workflows), taking as starting point known attributes of workflow inputs and the structure of the workflow. We achieve this by modeling these components as concepts and applying sharing analysis to a Horn clause-based representation of the workflow. The analysis is applicable to workflows featuring complex control and data dependencies, embedded control constructs, such as loops and branches, and embedded component services.

Keywords—workflow; business process; service composition; horn clause; static analysis;

I. INTRODUCTION

Service-Oriented Computing stresses interoperability among services, i.e., among loosely coupled and platform-independent software components with standardized data type descriptions and interfaces. While back-end services essentially implement indivisible operations, service compositions express higher-level, potentially long-running business processes in an executable form, often across organizational boundaries. Compositions are often described by specifying workflows that describe links between activities and the routing of data. This is done using a language that allows process modelers and designers to capture the essential elements of business logic and processing requirements [17], [9], [21], [20].

Inferring properties of workflows is valuable for several important design- and run-time activities such as transforming and refactoring workflows, identifying patterns, facilitating run-time instrumentation, reshaping, or performing distributed enactment of service compositions [15].

In this paper we focus on the inference of user-defined business domain-specific attributes for data items and activities in service orchestration workflows featuring rich control structures (branching, loops, and-split-join, (x)or-split-join), data dependencies, and component services with a (partially) known structure. The user-defined attributes that describe input data are organized into contexts and concept lattices from Formal Concept Analysis (FCA) [5]. Inference based on static program analysis techniques (see [16] for a general introduction) allows us to further enrich attribute information by automatically deriving emerging properties which are implicit in, but not evident from, the input data attributes and the workflow structure. Static analysis results are mapped back to the FCA representation framework, and can then be used to feed, for instance, fragmentation algorithms [10], [11].

This paper is a natural continuation of earlier work [8] which presented the basis of the application of sharing analysis to workflows. However that work did not specify from where and how the entities subject to sharing were obtained, or where its results were applied.

Work related to our approach includes several proposals that address the problem of information control flow from the viewpoint of decentralized process/workflow execution in complex scientific [22] or business domains [4], [23]. In that work the attributes of data items and activities carry the particular semantics of privacy or confidentiality levels from the viewpoint of multiple participants. With respect to that work, this paper adds two dimensions of flexibility. First, we employ a type of static analysis, sharing analysis, based on abstract interpretation [2], [6], which is applicable to a generalized user-defined semantics of the data and activity attributes (e.g., describing data components or some form of “data quality” for the activities and the entire workflow). Second, we relax (sometimes significantly) the constraints on the control structure of workflows that can be treated, by allowing complex and nested control structures that are commonly found in practice (such as loops, branches, splits and joins). In that respect, the workflow representation we use avoids on purpose depending on a particular business process language unlike some other analyses [12]. References to related work connected to particular aspects of our approach are given throughout the subsequent sections.

The paper proceeds as follows. Section II motivates and outlines the approach, and introduces the medication prescription workflow example that is used throughout the text. Section III introduces the notion of concept lattices and describes how such lattices are derived from an assignment of user-defined attributes to a set of input data objects. Section IV deals with turning workflow definitions and the concept lattices
into a form amenable to sharing analysis, and addresses the analysis itself. Section V explains how the results from sharing analysis are interpreted in terms of contexts and the user-defined attributes. Finally, Section VI presents our conclusions.

II. Motivation and Sketch of the Approach

Fig. 1 depicts a simplified example of a drug prescription workflow in BPMN [17]. The process is initiated by the arrival of a patient with an appropriate identification (labeled as x in the figure). Next, two parallel activities (a1 and a2) are run to retrieve the patient’s medical history and medication record. The data items resulting from these two activities are respectively marked y and z. Additionally, while retrieving the medical history, activity a1 informs about the stability of the health of the patient. Depending on it, either the last prescription is continued (activity a3) or new medication is selected (activity a4). Finally, the treatment of the patient is logged (activity a5).

Some relevant questions can be raised. For example, is the medical history (y) available to activity a2? This will depend on what activities a1 and a4 do with y (note that a4 internally executes a loop) and on whether a3 or a4 is executed. If a5 needs that information and it is not available, we have a correctness problem which may lead the workflow to failure. If a5 should not have that information (for, e.g., privacy reasons) but it can be leaked, there is a potential problem too. More generally, design-time analysis of the characteristics of data and activities in the given workflow can be used to obtain results that are useful for several tasks, of which we highlight just a few:

a) Fragmentation: A workflow can be split into several fragments that can be executed within different organizations. The assignment of activities to fragments can be done according to different criteria. In our example, a healthcare organization may wish to delegate parts of the workflow to business partners. Confidentiality requirements may require that either the medical history or the medication record (or both) be hidden from some of the partners. That would mean separating the activities in Fig. 1 into several swimlanes corresponding to different organizational domains, and assigning activities accordingly.

b) Data Compliance: When services are composed into a business process (such as a1...a5 in our example), organizations need to ensure that the information content used by a component service is adequate to implement the desired behavior, and often syntactic compliance with XML message formats is not enough. For example, the Patient ID (which may be a national identity card, a driving license, a passport, etc.) may or may not contain at runtime information enough to retrieve, e.g., a medical history. Some potential problems can be detected at design time by tracking data flow between activities and analyzing which pieces of information are shared between activities and which are needed by them.

c) Robust Top-Down Development: Business process modeling can start at a high level, and elaborate components in a top-down fashion. Some components may contain complex structured constructs, or be developed into separate, reusable services. Data flow analysis at the process level can help us identify required features of inputs and outputs from such components. Additional results can be obtained for sub-activities of a component. For instance, in Fig. 2, which exposes a possible structure for workflow component a4, it would be interesting to derive the attributes of criterion c. The process can be repeatedly applied to the components a41 and a42 of the sub-workflow.

Fig. 3 depicts our approach for inferring domain-specific attributes for entities in a workflow which can later be used by design-time tools. From the users’ perspective, the starting point is a description of the workflow using an appropriate formalism (BPMN, in our example), and an input data context (see Fig. 4 and Section III-A); a list of business domain objects which are input to the workflow, described using relevant attributes from the domain. The result is presented as a context that assigns attributes inferred by the analysis to data items and activities. This context can then be inspected by a user or a tool and be used for further analysis of the workflow, transformation, or other design-time tasks.

The initial context is used to set up a concept lattice (Sec. III), which is the main vehicle to prepare the input to the sharing analysis and to interpret its results. The sharing analysis works on a logic program generated from the translation of the workflow definition and the initial concept lattice into Horn clauses and logical variable substitutions, respectively (Sec. IV). Finally (Sec. V), the abstract substitutions which result from the sharing analysis are used to produce a result lattice which in turn is used to generate the final context.

III. From Contexts to Concept Lattices

We will use FCA [3, 5], a mathematical formalism used to represent and analyze data using contexts and concept lattices, to describe properties of the data and activities in the workflow. Due to space constraints, the reader is kindly referred to the existing literature for a more in-depth introduction to FCA.

A. Contexts and Concept Lattices

In a given domain, a context is a relationship between (finite) sets of relevant (user-defined) objects and attributes, and is usually represented as a table (Fig. 4). If O is a set of objects and A a set of attributes, the context is a relation ρ ⊆ O × A. For any object o ∈ O, o′ denotes the set of all attributes a ∈ A such that opa. By extension, for any B ⊆ O, B′ = ∩o∈B o′ is the set of all attributes common to all objects in B. Conversely, for any D ⊆ A, D′ = {o ∈ O : D ⊆ o′} is the set of all objects that have all attributes from D.

In FCA, a set of objects B ⊆ O is said to be a concept if (B′)′ = B′ = B. If D = B′, it follows that (D′)′ = D′ = D. The operator (·)′ is a closure that, when applied to a subset of objects, gives the concept that includes these objects. Speaking intuitively, it ensures that a concept includes all objects that share the same set of attributes. In particular, (∅)′ gives the most general concept that contains all objects, including possibly those with no known attributes.
In order to make concepts useful for analysis, we need to organize them into concept lattices. A lattice is a mathematical structure \((L, \leq, \lor, \land)\) built around a set \(L\) (in our case containing concepts from a context), a partial order relation \(\leq\), the least upper bound (LUB) operation \(\lor\), and the greatest lower bound (GLB) operation \(\land\). For arbitrary \(x, y \in L\), the element \(x \lor y = z\) has the property \(x \leq z\) and \(y \leq z\), but it is also the least such element, i.e., for any other \(w \in L\) such that \(x \leq w\) and \(y \leq w\), we have \(z \leq w\). The case for the greatest lower bound operation \(\land\) is symmetric. In this paper, we deal only with finite and complete lattices, where for any arbitrary non-empty subset of lattice elements the LUB and the GLB exist in \(L\); such lattices have unique greatest (\(\top\)) and least (\(\bot\)) elements.

For concept lattices, the ordering relation \(\leq\) between two concepts \(B_1\) and \(B_2\) holds iff \(B_1 \subseteq B_2\), or, equivalently, iff \(B_2' \subseteq B_1'\): a higher concept includes all objects from a lower (or derived) one; lower concepts are derived from higher ones by adding attributes. Consequently, the LUB is obtained using \(B_1 \cap B_2\), and the GLB using \(B_1 \cup B_2\).

Context lattices are usually represented using a variant of Hasse diagrams (Fig. 5). Nodes correspond to concepts, with the top concept visually on the top, and the bottom concept placed accordingly. The annotations associated with a concept (using dashed lines) show the attributes introduced by the concept (besides the derived attributes from the higher concepts) above the line, and the objects that belong to that concept, but not to any of its derived concepts, below the line. Concepts may have one or both parts of the annotation empty; in the latter case, the annotation is not shown.

Fig. 5 presents the concept lattices for the medical database contexts from Fig. 4. The most general concepts are shown on top of the lattices, and the most specific (empty in both cases) at the bottom.

### B. Describing Data with Concept Lattices

The data items that are input to the workflow need to be mapped to the appropriate objects in the input concept lattice. In the case of our example (Fig. 1), we would need to map the Patient ID input data item to either Passport, National ID, Driving License, or Social Security Card. In our example, each of those objects maps to a different concept in the lattice, but in general several objects can map to the same concept.

The prerequisite in order to use concept lattices is to create an adequate context at a level of abstraction that captures enough information to represent all relevant concepts and their attributes. Complex models can always be simplified by keeping only those attributes that really discriminate between different concepts. Existing tools (e.g., ConExp, Lattice Miner, Colibri and others [1]) facilitate the process of eliciting and exploring knowledge using FCA.

A relevant point is that some data sources may not appear explicitly as workflow inputs. In Fig. 1, activities \(a_1\) and \(a_2\) need to access some external source to extract records using the input Patient ID. The attributes of the retrieved records depend on properties of these data sources and therefore, they
need to be mapped to appropriate objects (in this case the Medical history and the Medication record from Fig. 5(a)).

IV. APPLYING SHARING ANALYSIS

Our application of sharing analysis to elicit new knowledge about attributes of the workflow entities is based on three points: (a) representing the control structures of the workflow in a language amenable to analysis, (b) representing data links and activities in the workflow as explicit variables, and (c) representing attributes of these entities as additional hidden variables which can share with the variables set up in (b). Two variables share if there is some object which is reachable from both, maybe following a reference chain. By inferring how runtime variables can share in the programming language representation of the workflow we deduce the runtime attributes of data items and activities in the workflow.

A. Workflows as Horn Clauses

The sharing analysis tools we will use [7], [6] work on logic programs, and therefore the workflow under consideration needs to be represented in the form of a logic program [14]: a series of logical implications which can be operationally understood as stating which subgoals are needed to accomplish a given goal. Note that the translation into a logic program does not need to be operationally equivalent to the initial workflow; it only needs to represent the data flow and data aliasing correctly. We translate data flow into parameter passing and data aliasing into unification of logical variables. Here we build on [8], where examples of translations from an abstract notation for workflows into Horn clauses are given. Due to size constraints we cannot reproduce here the description of the translation. Instead, we kindly direct the reader to [8] for more details. A key ingredient of the translation is representing, for each activity in the workflow, the sets of data items read and written. This vantage point in workflow modeling is shared with the existing approaches to the analysis of soundness of Workflow Nets with Data (WFD nets) [18], as well as with the approaches to verifying validity of business process specifications using data-flow matrices [19]. However, unlike those higher-level conceptual views that are mainly concerned with various aspects of business process management, in our case we aim at inferring properties on a more technical level that takes into account details of (possibly complex and nested) control flow and data operations. For that purpose, WFD nets or UML activity diagrams are not sufficiently informative, while Horn clauses provide an adequate computation paradigm that has been extensively studied.

As an illustration, we give here a commented translation of our workflow written in BPMN (Figs. 1 and 2) into Horn clauses. The translation for this case is given in Fig. 6 using Prolog syntax, and will be explained in the following text.

Lines 1-8 are a Horn clause that defines the predicate w for the workflow with a list of comma-separated goals in the body (lines 2-8) following the definition symbol “:-”. Character “%;

Fig. 3. Overview of the approach.
introduces a comment line and helps relate parts of the body with activities from Fig. 1. Arguments to w are listed inside parentheses in line 1; following the Prolog syntax convention, variable names start with an uppercase letter. These variables correspond to the names of activities and data items from the BPMN diagrams, including those data sources which are not explicit in the diagram: in this case the databases from which the medical history and medical record are retrieved from. These databases have to be characterized in the result lattice, and in order to do so they are assigned variables D and E. To expose results for the sub-workflow from Fig. 2, the arguments to w include all activities and data items from the sub-workflow.

Simple activities (including external service invocation) are translated as goals of the shape $\alpha = \varphi[\Gamma]$, where $\alpha$ stands for an activity, $\Gamma$ is a sequence of all data items used as inputs by the activity, and $\varphi$ is an uninterpreted function symbol (whose particular name is not relevant for sharing analysis, and has been chosen to recall the activity name). This is followed by goals of the same shape where the left-hand side of “=” stands for data item produced by the activity, and the $\Gamma$ part on the right hand side includes data items used in the computation of the data item. For instance, goals $A_1=f_1(X,D)$ and $Y=f_1^{-1}(X,D)$ in lines 2 and 3 represent the fact that $a_1$ uses data items $x$ and $d$ as inputs, to produce data item $y$. The only exception in $w$ is the goal for sub-workflow $a_4$ (line 7) to be discussed below.

The ordering of activities in the body of a clause must respect data dependencies, in the sense that data items should appear as arguments in a goal only if they are produced by a preceding activity. The ordering also needs to respect control dependencies arising from explicit sequences and joins (AND and OR). Otherwise, as in the AND-split case, the relative order of activities as goals in the body of a Horn clause is not significant from the sharing analysis point of view [8], and one such ordering can always be found, unless there is a race condition between potentially parallelized activities that try to read/write the same data item. This is not the case in our example and the possibility of this happening can be statically detected from the structure of the workflow. Also note that we include both branches of the XOR-split, since the data in $a_5$ can be affected by either one of them. The workflow for the component activity $a_4$ is effectively a repeat-until loop, and its body (activities $a_{4,1}$ and $a_{4,2}$) is translated in lines 19-23 in the same manner as $w$.

The goal for $a_4$ in the definition of $w$ (line 7) is a call to a predicate $a_4$ defined in lines 10-13. Its loop structure is translated by introducing auxiliary clauses in lines 15-17 that represent the case of loop exit (line 15) and the loop iteration by means of a recursive call. The call to the body of the loop ($w_2$ in line 11) is translated before the call to the auxiliary predicate $a_{4x}$.

B. Input Substitutions

An input substitution sets up the initial sharing (and therefore which attributes are shared) between the input top-level variables. It is a mapping from the variables that represent the data items given as input to the workflow to subsets of the “hidden” variables which represent attributes.

Variable sharing can be represented as a lattice where nodes represent variable sets which share a unique, hidden variable. The structure of the sharing lattice can be directly derived from the input concept lattice by assigning a hidden variable to each attribute in the input context. For clarity, hidden variables are named after the corresponding attributes. Next, the top-level variables are mapped to objects from the input context, and,
Fig. 7. Initial substitution for the two cases.

to subsets of the hidden input variables (which are nodes in the sharing lattice). The ordering \( a \sqsubseteq b \) between top-level variables \( a \) and \( b \) in the sharing lattice holds iff \( A \sqsubseteq B \), where \( A \) and \( B \) are the corresponding subsets of the associated hidden input variables. It directly follows that \( \sqsubset \) in the sharing lattice is the exact opposite of \( \sqsubseteq \) in the concept lattice.

In the text that follows, we will use two cases for input substitutions:

**Case 1** Patient ID (item \( x \)) maps to the Passport object (has the attributes Name and PIN).

**Case 2** Patient ID (item \( x \)) maps to the Social Security Card object (has the attributes Name, Address and SSN).

In both cases, the data source \( d \) for medical histories maps to the Medical history DB object (attributes Symptoms and Tests), and the data source \( e \) for medication records maps to the Medication record DB (attributes Symptoms and Coverage).

The input substitution is easy to produce: the input variables are made equal to (i.e., made to share with) terms that contain exactly the associated hidden variables from the input sharing lattice (Fig. 7). The actual shape of the terms is not significant, and therefore we just use lists of variables associated to the attributes.

**C. Obtaining Sharing Analysis Results**

The sharing analysis is applied to the program resulting from the translation of the workflow (Fig. 6) and the code that sets up the initial substitutions (Fig. 7). The underlying theoretical framework we use is abstract interpretation [2], which interprets a program by mapping concrete, possibly infinite sets of values onto (usually finite) abstract domains and reinterprets the operations of the language in a way that respects the original semantics of the language. The abstract approximations of the concrete behavior are safe in the sense that properties proved in the abstract domain necessarily hold in the concrete case. However, its precision depends in general on the problem and on the choice of the abstract domain.

The analysis is run using CiaoPP [7], [6], a tool for the analysis and transformation of logic programs featuring, among others, a powerful sharing analysis. While the sharing analysis we used is, in pathological cases, exponential in the number of variables in a clause, in our experience it exhibits a reasonable speed in practice.

The output of the analysis is an abstract substitution (Fig. 8(a)), which is common to both cases of input data mapping. The difference will arise in the interpretation, as described in the next section. Each row (1-7) contains a subset of the top-level variables (representing items and activities in the workflow) that share at least one unique hidden variable. The minimal set of hidden variables which can explain that workflow (that share at least one unique hidden variable). The minimal set of hidden variables which can explain that workflow (that share at least one unique hidden variable). The minimal set of hidden variables which can explain that workflow (that share at least one unique hidden variable).

V. From Sharing back to Concepts

The mapping of intermediate variables (those which are not initial top-level variables) to subsets of the hidden output variables carries the information on the relationship between these variables that the sharing analysis inferred. However, the

---

1The results presented here were obtained in 1.192ms using CiaoPP running on an Apple MacBook computer with Intel Core Duo processor, 2GB of RAM and MacOS X 10.6.5.
meaning of these output hidden variables has to be interpreted in terms of the original attributes — starting with those of the input data items. The sharing analysis of course preserves the original relationship among the input top-level variables [8]: if two variables \( a \) and \( b \) were associated in the input sharing lattice to subsets of attributes \( A \) and \( B \), respectively, such that \( A \subseteq B \), then for the corresponding subsets \( A_1 \) and \( B_1 \) to which \( a \) and \( b \) map in the resulting sharing lattice, it is the case that \( A_1 \subseteq B_1 \).

The next step is to construct a result concept lattice (Fig. 9) based on the sharing analysis results where data items and activities are considered as objects and the hidden variables in the result are considered as a new set of attributes. The activities are highlighted and framed, and the input data items from the input concept lattice are set in boldface. In this lattice we first assign the original attributes to the input data items, and then pass them down to all the lower-level concepts. We then obtain the resulting contexts (Fig. 10) for the two initial cases aforementioned. Note that only the attributes that are associated with some input data item may appear.

It should be noted that the construction of the resulting concept lattice can be done in polynomial time with respect to the number of objects (data items, activities) and attributes [13]. Different algorithms for construction of concept lattices differ in performance over different types of sparse contexts.

We want to note that in the most general case sharing analysis is undecidable, and the results of the analyzer can be a safe over-approximation which can indicate sharing between variables when it could not be proved that there is definitely no sharing. However, when it indicates no sharing, then this is definitely the case. The assignment of attributes to the workflow elements should be interpreted accordingly: the absence of an attribute is always certain, but its presence is not guaranteed.

We can now go back to the application cases mentioned in Section II and illustrate how the information in the contexts in Fig. 10 can be applied.

d) Fragmentation: The organization responsible for medicine prescription may want to split the workflow among several partners, based on what kind of information they are allowed to handle. The basis for fragmentation is the resulting contexts from Fig. 10. An example of fragmentation is shown in Fig. 11. The swim lanes correspond to the health organization and its partners. Registry and Archive cannot handle Symptoms, Tests, or Coverage data, and is therefore assigned activity \( a_5 \). Medical examiners can at most see Symptoms and Tests, and are thus assigned the activities \( a_1 \) and \( a_{42} \). Medication providers can only take care of Symptoms and Coverage, and are assigned activity \( a_2 \). All other activities (\( a_3 \), \( a_4 \) and \( a_{41} \)) need full access and remain centrally handled by the health organization.

e) Data Compliance: It may be known that a particular kind of information identifying a patient, such as his/her SSN, is required for retrieving the patient’s medication record (activity \( a_2 \)), and that the patient’s address is required for sending the results of tests (activity \( a_{42} \)). It can therefore be detected at design time that unless the patient is identified with a Social Security Card, these activities will fail. The designer may either restrict the use of the workflow by requiring the card, or select implementations of the mentioned activities with weaker success preconditions.

f) Robust Top-Down Development: Based on the characterization of the input data items, designers can derive the attributes of the data items in nested workflows. For instance, the attributes of the medicine search criterion (\( c \)) and the prescription candidate (\( p \)) are inferred in Fig. 10 in a safe way.

VI. CONCLUSIONS

We have shown how an FCA-based characterization of input data to a workflow can be enriched to include intermediate data items and internal activities. These are annotated with attributes which are inferred from emergent properties of the workflow which stem from the workflow structure and relationships between input data. We have shown how this task can be automated by translating (a) an initial FCA into a lattice from which sharing conditions are derived and (b) the workflow structure into a logic program. Then, (a) and (b) are subjected to a sharing analysis, and the results are mapped back to a resulting lattice and that to a resulting context, whose information can be used as a starting point for a number of other tasks. We have illustrated this methodology with a worked example.

As future work, we plan to address the development of automatic translations from common business process specification languages (BPEL, XPDL, YAWL, etc.) into logic programs amenable to sharing analysis in order to further test and refine the techniques proposed herein. Besides, we plan to explore other applications of the concept of sharing to services, aiming not only at (local) data sharing between activities, but also looking towards the representation of stateful service conversations and quality aspects of services.

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>Address</th>
<th>SSN</th>
<th>Symp.</th>
<th>Tests</th>
<th>Cover.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>Name</th>
<th>PIN</th>
<th>Symp. Tests Cover.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>p</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community’s 7th Framework Programme under the NoE S-Cube (Grant Agreement no 215483). The authors were also partially supported by Spanish MEC project 2008-05624/TIN DOVES and CM project P2009/TIC1465 (PROMETIDOS).

REFERENCES

Deriving Specifications for Composite Web Services

George Baryannis  
Department of Computer Science  
University of Crete  
GR 71409 Heraklion, Greece  
Email: gmparg@csd.uoc.gr

Manuel Carro  
School of Computer Science  
Universidad Politécnica de Madrid (UPM) and IMDEA Software Institute, Madrid, Spain  
Email: manuel.carro@imdea.org

Dimitris Plexousakis  
Department of Computer Science  
University of Crete  
GR 71409 Heraklion, Greece  
Email: dp@csd.uoc.gr

Abstract—We address the problem of synthesizing specifications for composite Web services, starting from those of their component services. Unlike related work in programming languages, we assume the definition of the component services (i.e. their code) to be unavailable — at best, they are known by a specification which (safely) approximates their functional behavior. Within this scenario, we deduce general formula schemes to derive specifications for basic constructs such as sequential, parallel compositions and conditionals and provide details on how to handle the special cases of loops and asynchronous execution. The resulting specifications facilitate service verification and service evolution as well as auditing processes, promoting trust between the involved partners.

Keywords—specification of service compositions, inference of specifications, service composition

I. INTRODUCTION

Service composition enables service-based systems to be built using accepted engineering principles, such as (service) reusability and composability. Composite services provide value-added services that achieve functionality otherwise unattainable by atomic services. In order to fully achieve these goals, composite services should be made available to consumers in the same way as atomic services are, abstracting away complex details of the way participating services are orchestrated to achieve the required functionality. This allows service consumers to invoke services regardless of the way they are implemented (i.e. as an atomic service or as a composition of services). This can be accomplished by providing formal specifications of composite services which present to the end user the minimum information required to understand the functionality offered, often by describing the inputs, outputs, preconditions and effects (collectively known as IOPEs) of the composite service.

Formal specifications are indispensable in a variety of service-related activities. Similarly to the case of programming specifications, service specifications could be used as a basis to construct a service based on a set of requirements agreed upon by the parties involved, or to check that some existing specification meets a set of requirements. Furthermore, they can assist in auditing processes that check third party or legacy code conformance to specifications, promoting trust between digital society partners, since specification conformance is one step towards trustworthiness.

Specifications also play a major role in verification techniques. Verification involves checking whether a system (such as a service or a service composition) satisfies a property given the particular property and a formal description (i.e. a specification) of the system. Moreover, specifications are important when evaluating the results of service adaptation or service evolution [1]. For instance, it is fundamental to ensure that a new version of a composite service adheres to either the original specification or an evolved specification that has the same or fewer requirements (equal or weaker preconditions) and produces the same or more results (equal or stronger postconditions).

Composite specifications also offer great assistance when one attempts to deduce whether a set of services can actually be composed in a meaningful way. During the process of creating the composite specifications, inconsistencies may be detected between preconditions and/or postconditions of the participating services, rendering that particular set of services not composable. Thus, such problems can be prevented before the composite service is delivered to the end user, so that they may be resolved by replacing the service or services that cause the inconsistencies.

While existing service description frameworks attempt to describe service compositions using a variety of composition models ranging from orchestrations to choreographies to Finite State Machines, no attempt (to the best of our knowledge) has been made to handle the problem of automatically producing specifications for a composite service, based on the specifications of the participating services. The same is true for automated Web service composition approaches: while each of them offers a way of automatically or semi-automatically producing the composition schema, as well as the control and data flows of the composite services, none attempts to derive a complete specification of the service that is to be delivered to the service consumer.

Traditional tasks involving specifications, especially in the field of programming languages, include generating code out of specifications, using specification languages such as VDM [2], Z [3], B [4] or Event-B [5], generating specifications from existing code (i.e. the weakest precondition calculus [6]) or checking that a particular code conforms to a specification. The case handled in this paper is different:
we aim at inferring specifications for a set of individual components, functioning as a whole, which are only known through their specifications. Therefore we cannot rely on the implementation in order to ensure that the service behavior agrees with the specification or use the implementation to deduce preconditions based on some predicate transformer. The only available knowledge is the specifications of the participating services and the composition schema. Our approach involves characterizing the meaning of the particular control structure used in the composition by means of the existing preconditions and postconditions, followed by a definition of a composite specification which is syntactically similar to the atomic specifications, in order to be able to recursively reason with it in a homogeneous way.

The rest of this paper is organized as follows. Section II offers a motivating example that illustrates the issues behind creating a composite specification. Section III provides an analytical description of the derivation process for most fundamental control constructs. Section IV deals with the cases of loops and asynchronous interaction. Section V offers a brief description of work related to specification derivation and Section VI concludes and points out topics for future work.

II. MOTIVATION

In this section, we present a rather indicative motivating example that attempts to illustrate the need for service composition specification in a service evolution scenario, as well as the issues behind deriving a composite specification, given the specifications of the participating services. The example is based on the E-Government case study of the European Network of Excellence S-Cube [7].

In this case study, citizens submit applications to request some government-related service, such as obtaining government-issued documents. A fee is required to obtain a particular document, so a mechanism that executes payment transactions is involved. Moreover, the citizen may request that the resulting document be authenticated. To that end, a digital signature certification mechanism is provided. A typical process to obtain a document is illustrated in Figure 1.

Users log into the system and fill in forms regarding their request as well as payment details, which are then simultaneously processed before the payment process can begin. If users demand authentication for their documents, then a certification process is executed, resulting in the delivery of a certified document to the user. Otherwise, an uncertified document is delivered. For reasons that will be clarified in the sequel, we have labeled the states before and after particular points in the process. For instance, state 1 is the state after the completion of Login and before beginning execution of services CheckRequest and CheckPayment, while state 2 is the state following CheckRequest/CheckPayment and before invoking the ExecutePayment service.

Let us assume that the individual tasks described above are implemented as Web services. Table I offers a possible specification of the services involved in the process, in terms of their preconditions and postconditions, expressed in first-order logic. $s_i$ and $s_o$ denote the states before and after execution of the particular service respectively. Suppose that, at first, we have a composite service $S_1$ that is implemented according to a specification $T_1$ in order to handle the document purchase process we described, but without certification, as shown in Figure 1. Then, it is decided that some documents should be certified with a digital signature, so the initial specification is augmented to $T_2$ to take that into consideration. In order to meet the new requirements, service $S_1$ needs to be evolved into a new composite service $S_2$. We need to check if the evolved service $S_2$ meets the new specification $T_2$. What we can do is derive a composite specification $I(S_2)$ based only on the information at hand (the orchestration definition of $S_2$ and the specifications of the participating services) and check if $I(S_2)$ subsumes $T_2$.

The composite specification should explicitly state all conditions that must be true before the execution of the whole composite service, as well as all conditions that are true after a successful execution. While we have preconditions and postconditions for each participating service, there is no obvious way of deciding which part of them will be included in the composite specification. The resulting
specification should be based on the way the services are orchestrated, taking into account the control and data flow of the composition.

We propose a derivation process that is based on structural induction and attempts to construct the composite specification using a bottom-up approach. The approach is applicable on any block-structured process, as well as graph-based ones, provided they can be transformed to block-structured equivalents [8]. The approach is based on the availability of the composition schema, which can be obtained, for instance, from the BPEL document of the composite service. In our example, the composite process is actually a sequence of services, which are either atomic (the Login and ExecutePayment services) or composite themselves (an AND-Split/AND-Join and an If-Then-Else execution). We need to first derive the specifications for the two inner compositions and then move a step up and derive the final composite specification, given the specifications for all four services of the sequence. In order to achieve this, we need to formulate the derivation for all fundamental control constructs, which we handle in the following section.

III. CALCULATING PRE- AND POST-CONDITIONS

Formal specifications have been extensively used in computer science in order to rigorously describe what a system should do and can also similarly be used to offer a formal presentation of what a Web service provides and under which circumstances. A traditional format for a specification contains the conditions that should be met prior to execution (called preconditions, which we will denote by $P$) and the conditions that result after a successful execution of the program (called postconditions or results, denoted by $Q$).

In contrast to program specifications where preconditions are usually the weakest possible ones (and postconditions the strongest possible), in the case of services, $P$ and $Q$ can be expected to be safe approximations, e.g., $P$ can be stronger than the weakest possible precondition for that particular service. $P$ can therefore disallow invocations in cases where the actual code would work, but it would not allow invocations in a state not entailed by the weakest precondition. Note that if the approximation were done in the opposite direction, i.e., with $P$ being weaker than the weakest precondition, executions allowed by $P$ could be erroneous.

A. Specification Semantics

A FOL semantics for a service specification with regard to its preconditions and postconditions is:

$$
\forall x \cdot (P(x, s_i) \Rightarrow \exists y \cdot Q(x, y, s_o))
$$

$P(x, s_i)$ and $Q(x, y, s_o)$ are the (approximations of) preconditions and postconditions, respectively, using predicates, where $x$ and $y$ are vector variables that represent accordingly the input fed to the service and the returned output. $s_i$ and $s_o$ are fixed for a given composition schema and denote execution points. The reason for using such state identifiers as additional arguments to the predicates is to differentiate the truth value of predicates based on when they are evaluated, without having to carry around a usually cumbersome notion of state of the world. This allows us to express fluency in predicate values in a lean way. Other formalisms could be employed, such as the situation calculus (and variants, such as the fluent calculus), that are specifically designed for the description of dynamic domains. However, it should be noted that situation calculus can be encoded as a logic program [9], [10], which has equivalent expressive power to FOL. Therefore by choosing FOL, we are not losing any power, while at the same time staying in a widely known formalism. Moreover, while the logical consequence in FOL may be semi-decidable, automated theorem provers for FOL, such as Prover9 [11] which is employed for the proofs in this work, are mature enough to provide high performance in practice.

Given similar specifications for the services participating in a composition, we want to construct a specification for the composite service $c$, which essentially involves calculating a set $P_c$ of preconditions and a set $Q_c$ of postconditions such that the following holds:

$$
\forall x \cdot (P_c(x, s_i) \Rightarrow \exists y \cdot Q_c(x, y, s_o))
$$

where $P_c(x, s_i)$ and $Q_c(x, y, s_o)$ are built using the preconditions and postconditions of the component services.

We insist that the derived specifications maintain the approximation that we mentioned earlier: preconditions for $c$ derived from preconditions that are not the weakest themselves should be stronger than (or at least as strong as) the weakest possible precondition for the composition. We will return to this issue in Section V. In the following subsections, we will show how to calculate preconditions and postconditions for the most fundamental control constructs: sequences, AND-Split/AND-Join, OR-Split/OR-Join, XOR-Split/XOR-Join and conditionals [12], [13]. In all cases, we

<table>
<thead>
<tr>
<th>Service</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Login</td>
<td>Valid(user, s) ∧ ¬LoggedIn(user, s)</td>
</tr>
<tr>
<td>CheckRequest</td>
<td>FilledIn(request, si) ∧ LoggedIn(user, si)</td>
</tr>
<tr>
<td>CheckPayment</td>
<td>FilledIn(payForm, si) ∧ LoggedIn(user, si)</td>
</tr>
<tr>
<td>ExecutePayment</td>
<td>Valid(payForm, si)</td>
</tr>
<tr>
<td>CreateCertified</td>
<td>PayCompleted(doc, user, si)</td>
</tr>
<tr>
<td>CreateUncertified</td>
<td>Delivered(certifDoc, user, si)</td>
</tr>
<tr>
<td>Service</td>
<td>Postconditions</td>
</tr>
<tr>
<td>Login</td>
<td>LoggedIn(user, so)</td>
</tr>
<tr>
<td>CheckRequest</td>
<td>Valid(request, so)</td>
</tr>
<tr>
<td>CheckPayment</td>
<td>Valid(payForm, so)</td>
</tr>
<tr>
<td>ExecutePayment</td>
<td>PayCompleted(doc, user, so)</td>
</tr>
<tr>
<td>CreateCertified</td>
<td>Delivered(certifDoc, so)</td>
</tr>
</tbody>
</table>

| Table 1 | ATOMIC SERVICE SPECIFICATIONS |
consider compositions of two services, but it is straightforward to extend our work to more complex cases.

B. Sequence

We denote sequential invocation by $A(x, z); B(z, y)$, where all variables $z$ that constitute the input of service $B$ are produced as an output of service $A$. This includes variables which are input for $B$ and which do not result from the execution of $A$, but come directly from sources external to the sequence. For the purposes of the specification we consider them to be routed untouched through $A$, $a$, $b$, and $c$ respectively denote the state before the execution of $A$, after the execution of $A$ and before $B$, and after the execution of $B$ (Fig. 2). The semantics of the sequential composition would be

$$\forall x \exists z \cdot ((P_A(x, a) \Rightarrow Q_A(x, z, b)) \land (P_B(z, b) \Rightarrow \exists y \cdot Q_B(z, y, c)))$$

(1)

From Eq. (1) we can deduce:

$$\forall x \exists z \cdot (P_A(x, a) \land P_B(z, b) \Rightarrow \exists y \cdot (Q_A(x, z, b) \land Q_B(z, y, c)))$$

(2)

However, Eq. (2) exposes internal variable $z$ to the precondition. This is not desirable, since preconditions should be externally checkable and depend only on the input data to the composition. We can use the postcondition of $A$ to eliminate this shortcoming:

$$\forall x \exists z \cdot (P_A(x, a) \land Q_A(x, z, b) \land P_B(z, b) \Rightarrow \exists y \cdot (Q_A(x, z, b) \land Q_B(z, y, c)))$$

(3)

In Eq. (3) the precondition can be checked exclusively based on $x$.

The derived specification shows what conditions must be met before executing the sequence $A; B$ and which conditions will hold after a successful execution. However, it does not state clearly which conditions must hold for the composition to be valid. For a sequential composition to be valid there should be at least one case where it is applicable: the precondition of the first service should hold and the precondition of the second one should be true when applied to the result of the first service. Expressed in FOL, this validity condition is as follows:

$$\exists x, z, y \cdot (P_A(x) \land Q_A(x, z, b) \Rightarrow P_B(z, b))$$

(4)

Note that there is a close connection with Hoare’s triples [14] (which we explore in more depth in Section V); in that formalism our notion of an empty domain would correspond to inferring a false precondition for a piece of code.

C. AND-Split/AND-Join

In the AND-Split/AND-Join composition pattern, which we denote by $A(x, z) \land B(w, y)$, there are two (or more) diverging branches of activities that are executed concurrently. Eventually the two branches converge into one branch, but only after activities on both branches have completed successfully.

For the composite service of Fig. 3, if we consider the AND-Split/AND-Join case, the following holds:

$$\forall x \cdot (P_A(x, a) \Rightarrow \exists z \cdot Q_A(x, z, c)) \land \forall w \cdot (P_B(w, b) \Rightarrow \exists y \cdot Q_B(w, y, d))$$

(5)

Note that it is possible for states $a$ and $b$ (and for states $c$ and $d$ as well) to be equivalent, but we leave equations in their general form. In a similar way to the sequential case, we can deduce from Eq. (5) the following:

$$\forall x \forall w \cdot (P_A(x, a) \land P_B(w, b) \Rightarrow \exists z, y \cdot (Q_A(x, z, c) \land Q_B(w, y, d)))$$

(6)

In this case, there is no need for further steps, as all input and output variables should be externally visible. As far as the validity condition is concerned, we only need to ensure that there is a case where the preconditions of both services are true:

$$\exists x, w \cdot (P_A(x, a) \land P_B(w, b))$$

(7)

The same validity condition applies to all parallel composition patterns. Now that we have derived specifications for the sequential and AND-Split/AND-Join composition patterns, we will attempt to apply the derivation to the motivating example. The parallel execution of services CheckRequest and CheckPayment, results in the following specification, based on Eq. (6):

$$\forall request, payForm, user \cdot$$

$$\begin{equation}
(FilledIn(request, 1) \land FilledIn(payForm, 1)) \\
\land LoggedIn(user, 1) \Rightarrow \\
Valid(request, 2) \land Valid(payForm, 2)
\end{equation}$$

(8)
Here, we use the state identifiers included in Figure 1. For example, $\text{LoggedIN}(\text{user}, 1)$ is true if user is logged in before the execution of $\text{Check} \_ \text{Request/CheckPayment}$. Given the above specification, we can now derive the specification for the composite service up to the $\text{ExecutePayment}$ service, which is a sequence of 3 services: $\text{Login}$, $\text{Check} \_ \text{Request/CheckPayment}$ and $\text{ExecutePayment}$. The specification is derived by first producing the specification for the subsequence of the first 2 services, based on Eq. (2):

$$
\forall \text{request, payForm, user}.
(\text{Valid(user, 0)} \land \neg \text{LoggedIN(user, 0)}
\land \text{FilledIN(request, 1)} \land \text{FilledIN(payForm, 1)}
\land \text{LoggedIN(user, 1)} \Rightarrow
\text{LoggedIN(user, 1)} \land \text{Valid(payForm, 1)}
\land \text{Valid(payForm, 2)})
$$

(9)

Notice that we use Eq. (2) instead of Eq. (3) because no internal variables are exposed. Adding the $\text{ExecutePayment}$ service to the sequence, results in the following specification:

$$
\forall \text{request, payForm, user}.
(\text{Valid(user, 0)} \land \neg \text{LoggedIN(user, 0)}
\land \text{FilledIN(request, 1)} \land \text{FilledIN(payForm, 1)}
\land \text{LoggedIN(user, 1)} \land \text{Valid(payForm, 2)} \land \text{Valid(\text{payForm, 2})}
\land \exists \text{doc} \cdot \text{PayCompleted(doc, user, 3)})
$$

(10)

Note that $\text{LoggedIN(user, 1)}$ and $\text{Valid(payForm, 2)}$ appear on both sides of the implication, and therefore can be removed from the right hand side without changing the meaning of the formula. This is an example of specification simplification, which will be discussed in Section III-G.

D. OR-Split/OR-Join

The OR-Split/OR-Join composition pattern, which we denote by $A(x, z) \lor B(w, y)$, is similar to the AND-Split/AND-Join pattern but with two fundamental differences. First, not all of the diverging branches are necessarily activated. Instead, a mechanism selects one or more of them to be executed each time. Second, at the merging stage there is no need for synchronization between the converging branches.

For the composite service of Figure 3, if we consider the OR-Split/OR-Join case, the following holds:

$$
\forall x \cdot (P_A(x, a) \Rightarrow \exists z \cdot Q_A(x, z, c)) \lor \\
\forall w \cdot (P_B(w, b) \Rightarrow \exists y \cdot Q_B(w, y, d))
$$

(11)

From Eq. (11), we can deduce the following:

$$
\forall x \forall w \cdot (P_A(x, a) \land P_B(w, b) \Rightarrow \\
\exists z, y \cdot (Q_A(x, z, c) \lor Q_B(w, y, d))
$$

(12)

As we mentioned previously, the validity condition is the same as in the AND-Split/AND-Join case (Eq. 7). Intuitively, the reason is that we do not know which branch is going to be executed, and therefore we must require that all of them are eligible. This validity condition may appear too strong for an OR (or XOR) parallelism, however it can’t be weakened without any extra knowledge about the particular parallel execution. While this may lead us to label a composition as invalid, when eventually the branch that caused the invalidity is not executed, it guarantees that no invalid composition is mislabeled as valid, which is far more important.

E. XOR-Split/XOR-Join

The XOR-Split/XOR-Join composition pattern, that can be denoted by $A(x, z) \oplus B(w, y)$, differs from the previous pattern in that it allows only one of the diverging branches to be executed each time. Hence, when the branches converge, only one of the branches is expected to provide results.

For the composite service of Figure 3, if we consider the XOR-Split/XOR-Join case, the following holds:

$$
\forall x \cdot (P_A(x, a) \Rightarrow \exists z \cdot Q_A(x, z, c)) \oplus \\
\forall w \cdot (P_B(w, b) \Rightarrow \exists y \cdot Q_B(w, y, d))
$$

(13)

XOR between two operands can be expressed as a conjunction of an OR between the operands and a negated AND between the same operands. Using the results of the calculations in the previous cases we result in Eq (14):

$$
\forall x, w \cdot (P_A(x, a) \land P_B(w, b) \Rightarrow \\
\exists z, y \cdot (Q_A(x, z, c) \lor Q_B(w, y, d))
$$

(14)

From Eq. (14), we can deduce the following:

$$
\forall x, w \cdot (P_A(x, a) \land P_B(w, b) \Rightarrow \\
\exists z, y \cdot (Q_A(x, z, c) \land Q_B(w, y, d)))
$$

(15)

The validity condition is once again expressed by Eq. 7.

F. Conditional Constructs

Conditional constructs, such as if-then-else or switch statements, evaluate a condition in order to decide which branch will be executed. Similarly to the XOR-Split/XOR-Join pattern, only one of the branches is selected, based on the truth value of the condition.

In an if-then-else composition of the form $IF \ a = c \ THEN \ A(x, y) \ ELSE \ B(x, y)$, as seen in Figure 4, if the condition $a$ is true, then this implies that $A$ is executed; if the condition is false, then this implies that $B$ is executed. Input variable $x$ refers to either of the two services since the branches are exclusive and the same is true for output variable $y$. $x$
also contains the terms that are involved in the condition C. Hence, the following should hold:

\[ \forall x \cdot (C(x, e) \Rightarrow \exists y \cdot (P_A(x, a) \Rightarrow Q_A(x, y, c))) \land \\
(\neg C(x, e) \Rightarrow \exists y \cdot (P_B(x, b) \Rightarrow Q_B(x, y, d))) \]

(16)

From Eq. (16), we can deduce the following:

\[ \forall x \exists y : \\
[(C(x, e) \land P_A(x, a)) \lor (\neg C(x, e) \land P_B(x, b))] \Rightarrow \\
[(C(x, e) \land Q_A(x, y, c)) \lor (\neg C(x, e) \land Q_B(x, y, d)))] \]

(17)

Determining whether a conditional composition is valid depends on finding a case where the precondition derived above is valid, resulting in the following validity check:

\[ \exists x \cdot ((C(x, e) \land P_A(x, a)) \lor (\neg C(x, e) \land P_B(x, b))) \]

(18)

We can now return and complete the specification for our example, by deriving first the specification for the conditional execution of CreateCertified and CreateUncertified based on Eq. (17) and given the condition \( \text{ReqCertif}(\text{doc, user}, 3) \):

\[ \forall \text{doc, user}, \exists \text{certifDoc} : \\
[\text{ReqCertif}(\text{doc, user}, 3) \land \\
\text{PayCompleted}(\text{doc, user, 4}) \lor \\
(\neg \text{ReqCertif}(\text{doc, user, 3}) \land \\
\text{PayCompleted}(\text{doc, user, 5})) \\
\Rightarrow [(\text{ReqCertif}(\text{doc, user, 3}) \land \\
\text{CertifCompleted}(\text{doc, user, 6}) \land \\
\text{Delivered}(\text{certifDoc, user, 6})) \lor \\
(\neg \text{ReqCertif}(\text{doc, user, 3}) \land \\
\text{Delivered}(\text{doc, user, 6})))] \]

(19)

Table II shows the derived preconditions and postconditions for the basic constructs that we examined in this Section.

### G. Simplifying the Derived Specification

As compositions become larger and more complicated, both with regard to the number of services and the composition schema, the derived specification will, in turn, grow similarly. This should have become obvious through the motivating example, where the final specification can be considered large, although still easily processable by a theorem prover. In even more complex composite services, the need to somehow simplify and compact the resulting specification becomes more crucial, demanding the formulation of a simplification process that should follow the derivation.

Simplifying a specification involves a series of tasks, ranging from generic ones such as dealing with duplicate...
predicates (for instance, predicates that appear in both sides of an implication, as mentioned at the end of Section III-C) and applying known equivalences, to tasks that depend on specific knowledge on the particular composite service. For instance, in the final specification as expressed in Eq. (20), the preconidion:

\( (\neg \text{ReqCertif}(\text{doc, user, 3}) \land \text{PayCompleted}(\text{doc, user, 4}) \lor (\neg \text{ReqCertif}(\text{doc, user, 3}) \land \text{PayCompleted}(\text{doc, user, 5})) \)

under the monotonicity constraint that once a payment process is completed, it remains completed thereafter (which comes from domain knowledge)

\( \forall x \exists y \cdot (y > x) \land \text{Valid}(\text{payForm}, x) \Rightarrow \text{PayCompleted}(\text{doc, user, y}) \)

is equivalent to \( \text{PayCompleted}(\text{doc, user, 4}) \), which is reasonable due to the nature of the preconditions of the services that form the if-then-else part of the composition and the equivalence of states 4 and 5.

**IV. HANDLING LOOPS AND ASYNCHRONOUS EXECUTION**

The loop structure was excluded from the discussion in Section III. Loops allow for the repeated execution of a task or a process until a condition (the loop guard) ceases to hold. This poses a significant challenge as there is no a priori knowledge of how many iterations will be performed. Due to that fact, the state identifiers that we used in all other constructs to differentiate predicate evaluations are rendered inapplicable.

Without knowledge of its precondition and postcondition, a possible way to specify a loop is by formulating the specification based on an upper limit on the number of iterations. For a looped execution under condition \( C \) and for a maximum number of \( k \) iterations, knowing that the looped commands are specified by preconditions \( P(x) \) and \( Q(x, y) \), where \( x \) and \( y \) are the input and output variables, yields the following recursive loop specification:

\[
\begin{align*}
L(x, x', 0) & \iff (\neg C(x) \land x = x') \\
L(x, x', k) & \iff k > 0 \land C(x) \land (P(x) \Rightarrow Q(x, y)) \land L(y, x', k - 1)
\end{align*}
\]

(21)

\( L(x, x', k) \) denotes the \( k \)-th iteration of a loop with input variables \( x \) and output variables \( x' \). Recursive specifications like (Eq. 21), while rather expressive and concise, are difficult to work with, especially by theorem provers. Also, such a specification wouldn’t be useful as part of a service specification, particularly in the case of asynchronous interaction, discussed later in this Section. Moreover, it depends on the ability to determine ahead of time an upper limit on the number of iterations, which cannot be expected to be always available.

Without knowledge of an upper limit on the number of iterations, a means to characterize a loop is through its invariant. A loop invariant \( I \) is a statement that is true before and after each iteration of the loop, thus it stays unaffected by the loop execution. By definition, the loop invariant is a loop precondition \( (I \Rightarrow P) \). Moreover, a loop postcondition can be derived through the following implication: \( I \land \neg C \Rightarrow Q \), where \( C \) is the loop guard, the condition that must be true for the iteration to continue [15].

Several issues are raised in the discussion of using invariants to generate loop specifications. A fundamental one is which of the possible statements that may be produced by a loop invariant generation process is the most suitable choice. The selected loop invariant should be at least strong enough to imply a successful execution of the loop if it was limited to a single iteration. For instance, if \( P_1 \Rightarrow Q_1 \) describes the successful execution of the looped services for a single iteration, then we need the loop invariant to be at least strong enough so that \( I \Rightarrow P_1 \) and \( I \land \neg C \Rightarrow Q_1 \) hold.

**A. Generating Loop Invariants**

As far as the loop invariant generation process is concerned, we once again have to consider the special characteristics of the service-oriented world. In the traditional programming languages case, generating invariants is based on the commands that form the body of the loop, whereas in the services case, we can only rely on an approximate specification of the body. Hence, we can expect that a generated invariant is an approximation too. The correct direction of the approximation needs to be determined.

Suppose that we have an invariant \( I_w \) that is a weaker safe approximation of the invariant \( I \) that would be generated based on the actual code of the loop, i.e. \( I \Rightarrow I_w \). If we use that invariant as a precondition, then we may allow invalid executions, which is unacceptable. On the other hand, if we use it to derive a postcondition \( Q_w \) by applying the implication \( I_w \land \neg C \Rightarrow Q_w \), then \( Q \Rightarrow Q_w \), meaning that we will get a weaker postcondition, which is acceptable, since the specified results of the execution are more than the actual ones.

Suppose now that we have an invariant \( I_s \) that is a stronger safe approximation of \( I \), i.e. \( I_w \Rightarrow I \). This will lead to a stronger precondition, which is what we expect from an approximate specification of a service, but also a stronger postcondition (if \( I_s \land \neg C \Rightarrow Q_s \), then \( Q_s \Rightarrow Q_w \)), which is problematic, because some of the actual results of the service may not be specified. Consequently, we need a stronger approximation of the invariant in order to derive a useful precondition, and a weaker approximation in order to derive a useful postcondition.

Another issue concerning invariant generation is the peculiar characteristics of our case. While, in general, invariant generation is based on a set of commands (the loop program), in our case we only have an approximate specification.
of the commands of the loop, so the generation process must be based on this information. Essentially, the invariant generator must take into account the preconditions of the looped commands, so that the resulting invariant at least implies these preconditions, as mentioned at the beginning of this section.

Furia and Meyer [16] provide a concise summary on the different methods that have been proposed in literature to generate loop invariants. Of the works mentioned, only static methods that do not depend on executing the program and do not rely on existing program annotations can be applied in our case since we actually need the invariant as a means to specify the loop and not the other way round. Methods such as abstract interpretation [17], [18], [19], [20] and constraint-based techniques [21], [22] are applicable, although they should be adapted in order to take into account the discussion in this section.

B. Specifying Asynchronous Services

So far, we have made the implicit assumption that all service executions are synchronous: a service receives a request, the client waits for the service to handle the request and the service returns a response. However, it is very common in Service-Oriented Computing to employ services that interact in an asynchronous manner: the client invokes the service but does not wait for the response, which may take more time to be produced than in the synchronous case. It is important to determine how this asynchronous interaction affects the derivation process we have described so far.

As far as preconditions are concerned, there is no difference: whether it is a synchronous or an asynchronous interaction, preconditions need to be true at the moment the request is received. However, the evaluation of postconditions is affected, because in the asynchronous case the response is received in a state which may differ from that in which the invocation was performed. Hence, care must be taken to ensure that postconditions are expressed and evaluated in the correct context.

In order to deal with this issue, we borrow the property of Static Single Assignment form (SSA) [23] from compiler design. SSA states that there is exactly one assignment for each distinct variable in a program. To implement assignments to the same variable, variable renaming is employed, so that if, for instance, a variable \( y \) is involved in two assignments, it is renamed to \( y_1 \) and \( y_2 \).

Let’s return to the motivating example. Suppose that the \texttt{ExecutePayment} service is executed asynchronously. This means that after checking the precondition and invoking the service, the composite process continues to the certification phase. Let’s assume that we have a more detailed version of the precondition, \( \text{Valid}(\text{payForm}, \text{user}, s_1) \), which is true when the particular user has correctly filled in the corresponding payment form. If, after the invocation of \texttt{ExecutePayment}, another service has to modify the variable \( \text{user} \), it is renamed to \( \text{user}' \). Thus, when the asynchronous service completes execution and the postcondition \( \text{PayCompleted}(\text{doc}, \text{user}, s_0) \) has to be checked, it will be evaluated against the value of variable \( \text{user} \) that matches the value used in the evaluation of the precondition.

In the case of loops, expressing specifications using the SSA form requires that we know an upper limit for the number of iterations, otherwise it is not possible to apply the variable renaming scheme. In absence of such an upper limit, loop specification derivation must follow the detailed discussion in this section, in order to produce simpler specifications for which SSA can be easily applied.

V. Related Work

Formal specifications have been used in computer science in order to describe what a system should do. Specifications can be used to drive the system’s implementation and to verify whether existing systems (or design plans) are correct with respect to the specification that was agreed upon. A traditional format of a program specification contains the conditions that should be met prior to execution (called preconditions) and the conditions that result after a successful execution of the program (called postconditions or results). Hoare [14] introduced the well-known triple notation \( P[S]Q \) which expresses that if preconditions \( P \) are met before initiating execution of program \( S \), then when the execution completes postconditions \( Q \) will be true.

Hoare’s notation expresses a sufficient set of conditions for a program to have a desired set of results. Dijkstra [6] expanded on this by focusing on necessary and sufficient (called weakest) preconditions, that also guarantee the desired result. The notation he introduced, \( \text{wp}(S, Q) \) denotes the weakest precondition for program \( S \), which is “the set of all states such that execution of \( S \) begun in anyone of them is guaranteed to terminate in a finite amount of time in a state satisfying \( Q \)” [15]. Dijkstra then defined the weakest preconditions for basic statements such as assignment, conditionals, or loops and showed that one can derive formally a set of statements that, if executed, will lead to a specified result, by using structural induction on the basic weakest preconditions.

Our work attempts to bring Dijkstra’s derivation of program specifications in the field of Service-Oriented Computing, hence it differs in some important points. The defining difference is that Dijkstra’s derivation process is driven by the program implementation. In the case of services, we have no access to the implementation, thus we cannot use it to either drive the derivation process or verify the resulting specification. Another important difference is that the weakest precondition derivation relies on specifications that aren’t approximations: in order to derive the weakest precondition for a composition using the \( \text{wp} \) operator, we need to have the weakest precondition for all participating services. As
we have already mentioned, service specifications are most often approximations and as a result, they cannot be used in combination with the \textit{wp} operator. Finally, a further differentiating point in comparison to programming specifications is the asynchronous interaction that we encounter in services, which is not addressed in programming specifications.

Despite the differences outlined above, it is important to make sure that our approach does not contradict Dijkstra’s. In other words, we need to ensure that our approach does not infer a precondition that is weaker than the one calculated by the \textit{wp} operator. We now provide an intuitive explanation supporting this for the case of the sequential composition. Let us reason by contradiction and assume that the precondition produced by our approach, \( P \), is not stronger or as strong as the weakest precondition \( P_{\text{wp}} \): \( \neg( P \Rightarrow P_{\text{wp}} ) \) — or, equivalently, \( P \land \neg P_{\text{wp}} \), i.e., there are states in which the weakest precondition \( P_{\text{wp}} \) does not hold, but in which our precondition \( P \) holds, which by definition makes \( P \) incorrect.

Since the initial assumption is that \( P \land \neg P_{\text{wp}} \) is true, then \( P_A \land P_B \land \neg P_{\text{wp}} \) must also be true. From the discussion at the beginning of Section III, we know that the approximated preconditions \( P_A \) and \( P_B \) are stronger than or equivalent to the corresponding weakest preconditions \( P_{\text{wp}A} \) and \( P_{\text{wp}B} \), i.e., \( P_A \Rightarrow P_{\text{wp}A} \land P_B \Rightarrow P_{\text{wp}B} \). With that into account, we need \( P_{\text{wp}A} \land P_{\text{wp}B} \land \neg P_{\text{wp}} \) to be true.

The result is contradictory since we want at the same time the precondition of a composite service to be false and the preconditions of the services it contains to be true. In particular, the precondition of the \textit{leftmost} service, \( P_A \), has to be true, since it is the “starting point” of the composition. Hence our assumption was incorrect, meaning that our approach will never produce a precondition that is weakest than the one derived by the \textit{wp} operator. In a similar manner, we can prove that our result holds for all control constructs handled in this work, since in almost all of them, \( P \) contains the conjunction of \( P_A \) and \( P_B \). The only exception is the conditional case, where the derived precondition contains a disjunction of \( P_A \) and \( P_B \), depending on the condition truth value. Regardless, the contradiction we have proven still holds, since in any case either \( P_A \) or \( P_B \) will have to be true.

Another work related to specification derivation is that of Ghezzi et al. [24], [25] which focuses on methods for specification recovery. [24] proposes a method to infer algebraic specifications of abstract data types, given the related class and its methods and with no access to the source code. Behavior models are extracted based on the run-time behavior of the class to be specified and are used to drive the generation and selection of possible actions performed by the class, described as terms, i.e. sequences of legal method applications with fixed actual parameters, starting from a constructor. [25] similarly creates behavior models by observing the input-output relationships after executing the methods of the class. Then, graph transformation rules are applied in order to result in a generalization of the initial behavior models, which is the inferred specification for the class. These works rely on the run-time behavior of a component in order to derive its specification, which is different from our approach, in which we rely on the specifications of sub-components and the control flow between them.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an approach for inferring composite service specifications given the specifications of the services participating in the composition (in the form of sets of preconditions and postconditions) and the composition schema. The approach attempts to construct the specification by using structural induction based on derivation rules defined for most fundamental control constructs. The resulting specification can be used to formally describe the composite service in terms of its preconditions and postconditions without requiring any knowledge of the internals of the composition, allowing for an actual “black box” view of the whole process.

The nature of the proposed approach facilitates a possible implementation: structural induction lends itself to be written as a recursive algorithm. Hence, it would be straightforward to create an automated process that takes a set of service specifications and a composition schema and produces the specification for the composite service of the schema. Such specifications can then be used to prove desired properties of the composite service or be fed to automated composition approaches that accept preconditions and postconditions as input [26].

Future work includes implementing the proposed approach and evaluating it for compositions of varying complexity. Concerning specification simplification, we plan to look into the work of Douglas Smith [27] on simplifying precondition formulas and determine whether the actions he proposes may be applied in our case. Finally, it would be interesting to explore whether the resulting specifications suffer from the frame problem and related issues as examined in [28].

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube). Manuel Carro has also been partially funded by Spanish MICINN project TIN-2008-05624 DOVES and CAM project S2009TIC-1465 PROMETIDOS.

REFERENCES

Preventing SLA Violations in Service Compositions Using Aspect-Based Fragment Substitution

Philipp Leitner\textsuperscript{1}, Branimir Wetzstein\textsuperscript{2}, Dimka Karastoyanova\textsuperscript{2}, Waldemar Hummer\textsuperscript{1}, Schahram Dustdar\textsuperscript{1}, and Frank Leymann\textsuperscript{2}

\textsuperscript{1} Distributed Systems Group, Vienna University of Technology
Argentinierstrasse 8, 1040 Wien, Austria
lastname@infosys.tuwien.ac.at

\textsuperscript{2} Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätstraße 38, 70569 Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

Abstract. In this paper we show how the application of the aspect-oriented programming paradigm to runtime adaptation of service compositions can be used to prevent SLA violations. Adaptations are triggered by predicted violations, and are implemented as substitutions of fragments in the service composition. Fragments are full-fledged standalone compositions, and are linked into the original composition via special activities, which we refer to as virtual activities. Before substitution we evaluate fragments with respect to their expected impact on the performance of the composition, and choose those fragments which are best suited to prevent a predicted violation. We show how our approach can be implemented using Windows Workflow Foundation technology, and discuss our work based on an illustrative case study.

1 Introduction

As more and more companies shift towards a service-based model \cite{1} of doing business, e.g., by providing coarse-grained value-added services as compositions of existing (external) Web services, management of service level agreements (SLAs) \cite{2} is becoming increasingly important. SLAs are contractual agreements between a service provider and its customers, which govern the quality that the customers can expect. Violating SLAs is often associated with monetary penalties for the provider, i.e., the service provider generally has a strong interest in preventing SLA violations.

To this end, research in the area of SLA monitoring and compliance management \cite{2–4} has so far mostly focused on detecting and explaining SLA violations after they have happened. While this is very useful to optimize service compositions in the long run, it does not prevent the problem in the first place. Therefore, we see the need for mechanisms to prevent violations at runtime, before they have happened. Basically, such mechanisms need both, a way to predict violations ahead of time, and a means to actually adapt the problematic composition instance in such a way that the violation is prevented. The former has already been covered in earlier work \cite{5,6}.

The main contribution of this paper is a proposed solution for the latter problem. We apply the aspect-oriented programming (AOP) approach \cite{7} to
adaptation of running composition instances. Adaptations are triggered by predicted violations. Unlike in earlier work [8], aspects can contain composition fragments of arbitrary complexity, which can be applied before, after or instead of any subset of the original composition. We evaluate potential fragments based on their expected impact on SLA conformance, in order to identify the fragments which are best suited to prevent a predicted violation. Note that in this work we focus on performance-related service level objectives (SLOs). Our work is not directly applicable for most qualitative SLOs, such as security.

The rest of this paper is structured as follows. In Section 2 we present an example case, which we use as an illustrative example in the remainder of the paper. In Section 3 we present our approach to aspect-based adaptation in detail. In Section 4 we explain how our approach can be implemented using Windows Workflow Foundation [9] (Windows WF) technology. Section 5 contains an evaluation of our approach. In Section 6 we discuss related scientific work. Finally, Section 7 concludes the paper with a summary and an outlook on future work.

2 Case Study and Motivation

To illustrate the core ideas of the paper we use an order processing scenario. The scenario consists of a reseller who offers products to its customers. As shown in Figure 1 (left part of the figure), the reseller composes services from other providers (supplier, shipper, banking) to implement its process. After receiving the customer order, the list of needed parts is determined and parts which are not in stock are ordered from a supplier. After all the parts are gathered, the product is assembled and shipped to the customer. The reseller guarantees its customers a certain order processing time via an SLA. The goal of the reseller is to prevent cases of SLA violations, as this would lead to reduced customer satisfaction as well as penalty payments. The reseller can use SLA monitoring and prediction techniques as discussed in our previous work [5] to predict at process runtime whether the SLA with the customer is likely be violated, i.e., in our case, whether the order processing time will exceed the agreed value. If SLA prediction shows that the SLA with the customer will be violated, the reseller wants to adapt the service composition instance by using alternative, better performing services. Assume that in our scenario there are two alternative suppliers who offer faster delivery times, but do not provide all needed product types on their own. The full product range offered to the customer can only be realized by using both alternative suppliers in conjunction. Figure 1 shows a composition fragment consisting of a switch between those two suppliers, whereby supplier 2 is used if supplier 1 is unable to provide a certain part. Even though not the default case, this composition fragment can be used at runtime instead of the original supplier invocation if a given instance is likely to exceed the maximum processing time as promised to the customer in the SLA.

There are two approaches for supporting the execution of alternative composition fragments: (1) the straight-forward approach is to model all alternative fragments already at design time as part of the composition, e.g., using if-else branches; (2) the approach of this paper is to model alternative fragments separately from the original composition model, and dynamically substitute them based on prediction results at composition runtime. We will now explain this approach and its advantages in detail.
3 Aspect-Based Adaptation

In this paper we use the notion of aspect-based fragment substitution to model how service composition instances can be adapted at runtime in order to prevent predicted SLA violations. Our approach is general, in the sense that it is not specific to any concrete composition model. Instead, it can be applied to many existing block-structured composition models (for instance WS-BPEL or Windows Workflow Foundation [9]). In our work we reuse well-known AOP terminology to describe adaptations of service compositions. The most important of these terms are aspects (cross-cutting concerns, which are turned off and on at design or run-time, e.g., logging), advices (business logic which implements aspects, e.g., the code to implement logging), joinpoints (points in the application code where advices can potentially be inserted), and weaving (the process of dynamically inserting advices in joinpoints). Note that in literature AOP is often discussed as both a design time and a run time technology, i.e., weaving can happen both statically or dynamically. In this paper we only consider weaving at runtime (at running instances), since our primary concern is the adaptation of composition instances, without modification of the underlying definition.

The main concepts of our work are summarized in Figure 1. Aspects are defined as an adaptation trigger, which is based on a predicted SLA violation, and any number of advices. Every advice in turn contains exactly one composition fragment, one impact model containing any number of impact clauses, a list of constraints on other advices of the same aspect, and any optional other properties. The fragment is linked to the service composition to adapt (denoted as target composition in the following) using two types of joinpoints – in-joinpoints mark the beginning of the composition segment to replace, while out-joinpoints...
mark the end of the segment. We will now discuss these components in more detail.

3.1 Adaptation Triggers

As discussed extensively in Section 1 and Section 2, our approach is motivated by the need to prevent SLA violations. Therefore, runtime adaptation is generally triggered by predictions of such violations. In the remainder of this paper we will assume that some means of prediction are available. This may be powerful prediction tooling as presented in [5] or [6], or simply some estimations provided by a human domain expert. The actual approach to prediction is out of scope of this paper, however, for the sake of completeness we give a very brief overview of our own earlier work.

![Fig. 2. Generating Predictions From Runtime Data](image)

We generate predictions using regression from runtime data. This data is collected using an event-based approach (i.e., components in the service-based system emit status events, which are collected and correlated). The actual regression is implemented as a black-box function, using methods from the field of machine learning, more precisely artificial neural networks [10] (ANNs). We sketch this in Figure 2. This prediction is carried out in predefined places in the service composition, the so-called checkpoints. Checkpoints should be selected in such a way that enough data is already available to generate useful predictions. Note that there is a strong relationship between the checkpoint selection and which actions can be associated with an advice – in earlier checkpoints a lot of adaptation actions are still available, while later checkpoints allow for more accurate predictions because of more data being available. The problem of selecting checkpoints is discussed in more detail elsewhere [5].

3.2 Composition Fragments

Composition fragments can be considered the core of our adaptation approach. In essence, fragments are full-fledged, even if usually small, service compositions. That is, fragments may contain variables, branches, Web service invocations, parallel execution, loops, scopes, fault handling, compensation, or any other construct which is legal in the composition model used. However, they do not have to follow the same syntactic and semantic rules as the target composition. For example, if WS-BPEL is used as composition model, designers of fragments may access e.g., variables defined in the target composition, even if the respective data is undefined in the fragment itself (syntactic rule). Also, they could specify
a receive activity without a corresponding reply activity (semantic rule). The reason for this is that during weaving the fragment will be inserted into the composition model of the target composition, essentially becoming part of the composition itself. A fragment definition is valid if it results in an executable composition after weaving, which cannot be checked in isolation.

In addition to all activities provided by the composition metamodel, fragments may contain three additional activity types (\texttt{FRAGMENT\_START}, \texttt{end}, and \texttt{TRANSPARENT\_BLOCK}) with a semantic specific to our approach. We refer to these activities as virtual activities, because they are never actually executed. Instead, virtual activities are dropped or replaced during weaving. Virtual activities are solely responsible for defining the joinpoints between the fragment and the target composition, marking the segment of the target composition to substitute.

Every fragment starts with exactly one \texttt{start} activity and ends with exactly one \texttt{end} activity. In-joinpoints, defined via the \texttt{start} activity, represent the start of substitution, and out-joinpoints, defined via the \texttt{end} activity, represent the end of substitution. All joinpoints can reference any activity in the service composition, either before or after the execution of the activity (i.e., both “immediately before executing Get List of Parts” and “immediately after executing Get List of Parts” are valid joinpoints). However, the in- and out-joinpoint of a fragment need to reference activities in the same sequence in the target composition, i.e., the joinpoints defined in Figure 1 are correct, but, for example, it would not be possible to move the in-joint point to the activity “Get List of Parts”. The reason for this limitation is that semantic problems arise if in- and out-joinpoints are situated in different sequences. In the example, the branching activity “part in stock?” would be removed, but not the actual branches, rendering it impossible to decide which branch to execute.

It is not only possible to replace a segment of the target composition, even though this is the general case we discuss. Trivially, one may also just insert the
fragment at a specific joinpoint (the in- and out-joinpoints are identical, and
the fragment is non-empty), or remove a segment (substitution with an empty
fragment). We refer to the sum of all joinpoints of a fragment as the linking
of the fragment to the target composition. Figure 3 summarizes this linking. The
start activity specifies that the fragment should be inserted before the activity
“Schedule Assembling”, while the end specifies that the end of the substitution
segment is before the activity “Billing & Shipping”. On the right-hand side,
Figure 3 shows the dynamically constructed instance after the fragment has
been woven into the target composition. Activities depicted with the prefix T
originate from the target composition, while activities with the prefix F are
specified in the fragment.

Transparents are more complicated than start or end. They are a place-
holder representing a part of the target composition in the fragment. This part
is defined in the same way as the substitution segment, i.e., transparents have
both out- and in-joinpoints. Additionally, the same restrictions apply (in and
out-joinpoints need to reference activities in the same sequence). At runtime,
transparents are replaced by a copy of the part that they represent. The pur-
pose of transparent activities is threefold. Firstly, they allow for the definition
of fragments which substitute segments, while still retaining some of this seg-
ment’s functionality. One example of this usage is depicted in Figure 3, where
the “Check for Faults” activity from the target composition is retained in the
fragment. Note that it is not mandatory that a transparent references only a
single activity. Secondly, transparent activities allow to essentially duplicate ac-
tivities in the target composition. This is because transparents are in fact free
to reference any part of the target composition, not only parts which are in the
substitution segment (and hence removed during weaving). Additionally, many
transparents may copy the same activities, multiplying them even further.
Thirdly, transparent activities allow for the definition of generic fragments.

3.3 Generic Fragments

Generic fragments are (unlike the fragments discussed so far) not developed
specifically for a given target composition. Instead, they can be applied to a num-
ber of compositions. Therefore, generic fragments do not contain any concrete
case-specific business logics. They are used to implement adaptation scenarios
which can be useful across several concrete target compositions and domains.
Figure 4 exemplifies three generic fragments. The main property of generic frag-
ments is that they consist only of virtual activities and control flow constructs,
i.e., they do not contain any concrete activities such as Web service invocations.
These generic fragments are instantiated by defining the linking (i.e., all in-
and out-joinpoints) to concrete target compositions. As soon as this linking is
defined, the fragment stops being generic, and is as case-specific as any other
fragment.

The first and most simple generic fragment in Figure 4 (Remove) has been
mentioned before – it is an empty fragment consisting only of a start and end ac-
tivity. Using this generic fragment any segment of the target composition can be
deleted. The second example is a generic fragment named Reorder2. It consists of
start, end, and two transparents (“after” and “before”). Using this fragment
two segments in the target composition can be rearranged, e.g., exchanging their
order. Trivially, one can also implement similar generic fragments \texttt{ReorderX}, rearranging \textit{X} segments instead of just two. Finally, \texttt{Parallelize2} consists again of \texttt{start}, \texttt{end}, and two \texttt{transparents} ("branch\textsubscript{1}" and "branch\textsubscript{2}"), however, this time "branch\textsubscript{1}" and "branch\textsubscript{2}" are executed in parallel. Using this generic fragment one can parallelize two segments from the target composition (which presumably have been executed in serial before). Of course, it is again possible to define \texttt{ParallelizeX} fragments to parallelize more than two segments at the same time.

### 3.4 Dynamic Weaving

At run-time, one or more previously selected fragments are weaved into the running instance of the target composition. The selection procedure will be discussed in Section 3.5. As we have sketched in Figure 5, the general weaving algorithm is a simple 2-step procedure. Firstly, the fragment is pre-processed, i.e., for each \texttt{transparent} in the fragment the linking to the target composition is resolved, and the \texttt{transparent} is replaced by a deep copy of the segment that it represents. Secondly, the linking of the fragment itself is resolved, and the \texttt{start} and \texttt{end} virtual activities are removed from the fragment (they are not needed anymore). Finally, the segment of the target composition (indicated by the linking) is removed, and the fragment is inserted instead.

Weaving can be done either online or offline. For offline weaving the composition instance is halted while the adaptation is applied (see Line 4-5 in Listing 5), and resumed when the adaptation is finished (Lines 19-20). If online adaptation is used the instance continues running during weaving. This has the advantage that weaving does not introduce additional execution time overhead. However, if after weaving the running instance has already passed the entry point of the fragment (the linking of the fragment’s \texttt{start} activity) the weaving fails and is rolled back. This is because our system needs to guarantee that a fragment is either executed as a whole, or not at all (which cannot be guaranteed after the instance has begun executing the substitution segment in the target composition). Our system falls back to offline adaptation as soon as at least one advice which needs to be applied requires it (i.e., if many advices are applied and only one of them requires offline weaving, we still need to suspend the composition instance before adaptation).

Generally, if more than one advice needs to be applied, we use recursive one-by-one weaving, that is, we start by weaving the first fragment into the instance (ignoring any other fragments). The result of this first weaving pro-
Fig. 5. Weaving Algorithm

cess is then the input to the weaving of the second fragment. This is continued until all fragments are weaved. The order in which fragments are applied is unimportant as long as all fragments are independent (i.e., as long as none of the segments indicated by any linking of either fragments or transparencies overlaps). If this is not the case the user can specify a defined ordering of advices as part of the advice definition. The ordering can be defined using five different order predicates (REQUIRES BEFORE, REQUIRES AFTER, IF PRESENT BEFORE, IF PRESENT AFTER, and CONFLICTS WITH). REQUIRES [BEFORE|AFTER] specifies that a given advice has to be applied before or after this advice (otherwise the advice cannot be applied at all). IF PRESENT [BEFORE|AFTER] specifies that if the other advice is present, it has to be applied before or after this one (but the other advice can also simply not be applied). Using REQUIRES BEFORE one can specify complex fragments, whose linking does not actually point to the target composition itself, but to another fragment. This is possible since we can rely that the referenced fragment has already been weaved into the target composition before before the weaving of the dependent fragment starts. Another type of ordering predicate is CONFLICTS WITH. This predicate specifies that two fragments are mutually exclusive, i.e., they cannot be applied together. At runtime, we construct a forest of directed graphs from these predicates, whose nodes are advices and whose edges are “is executed after” relationships. If the graphs in this forest are acyclic there is at least one allowed order of advices, which can be constructed using topological ordering. If the graphs contains at least one cycle the definition of advices is invalid, since the definition contains at least one cyclic dependency.

3.5 Impact Model and Advice Selection

As described briefly in Section 3.1, we build upon a predictor which estimates SLO values by assessing a set of lower level metrics. Examples include ordered product types, duration of branches of the composition, expected delivery times of suppliers and shipper, or QoS of services used. In order to being able to evaluate whether a given advice will actually help preventing the predicted SLA
violation, we need to specify for each advice its impact on those lower-level base metrics (impact model). The impact model is used to identify which concrete advice (from all advices designed within an aspect) should be applied, i.e., which advices are best suited to prevent a predicted violation (advice selection).

The impact model contains a non-empty set of impact clauses. An impact clause relates to one base metric and specifies the expected value of that metric after adaptation (i.e., after this fragment has been applied). This value can be determined in several ways: (1) based on measured history data if the corresponding advice has already been used in past composition instances, e.g., using data mining techniques; (2) based on SLAs with external providers; or (3) by using QoS aggregation techniques as discussed in earlier research [11]. In QoS aggregation, based on the composition fragment structure, the properties of atomic activities are recursively aggregated (e.g., the duration of a sequence of activities is the sum of durations of those activities, the duration of the parallel execution of activities is given by the duration of the longest activity etc. [11]). The impact model should specify impact clauses for all metrics which the advice affects.

The impact model is specified as part of advice definition. Advice selection at runtime is performed as follows. If in a checkpoint a violation is predicted, we obtain the set of advices defined for this checkpoint. For each allowed combination of advices we evaluate if the usage of these advices would prevent the SLA violation, i.e., all impact clauses are applied to the data which has originally been used to generate the prediction. The updated data (which essentially represents the state after adaptation) is then again fed to the predictor, to regenerate the prediction after adaptation. The difference between the original prediction and the new prediction is the estimated impact of applying these advices. This is sketched in Figure 6. If more than one advice should be applied at the same time, the impact clauses are applied in the same order to the data as the weaving order of the fragments would be. If the predicted value complies to the SLA, that advice or advice combination is put into a candidate set. In the next step, we then select the best alternative from the candidate set by looking at additionally specified criteria (in addition to the concrete predicted value). In this step, complex evaluations can take place, taking into account and weighting different dimensions (e.g., cost, customer satisfaction, reliability) according to a user-defined utility function, which is currently left for future work. At the moment, we simply choose the candidate which brings the SLO value closest to the
target value, i.e., we apply “just as much” adaptation as necessary to prevent the predicted violation.

4 Prototype Implementation

In our prototype we consider the aspect-based adaptation of service compositions implemented using the Windows Workflow Foundation [9] (WF) composition model. More concretely, our system can be applied to WF Sequential Workflows. WF Sequential Workflows are similar to e.g., Web service compositions implemented using WS-BPEL. However, unlike most WS-BPEL engines, WF is deeply integrated with Microsoft .NET (starting with version 3.0), along with strong tool support for developing compositions. Additionally, and most importantly for this paper, Microsoft .NET supports the dynamic adaptation of WF instances via an explicit API, the WorkflowChanges API. This API allows us to suspend, modify, and resume any running composition instance. Additionally, activities in the composition can easily be replicated. Implementation of the weaving algorithm as discussed in Section 3 is, therefore, straightforward. Another important advantage of building the prototype based on WF is that we can reuse the tooling integrated in Visual Studio to support the development of fragments.

We have implemented the approach discussed in this paper within the larger VRESCo SOA runtime environment project. VRESCo is discussed in detail elsewhere [12], and will not be covered here. To trigger adaptations as briefly discussed in Section 3.1, we utilize our earlier work on prediction of violations, as discussed in [5]. The prototype has been designed to fit into PREVENT, an autonomous system for prevention of SLA violations [13]. The interested reader may download a recent snapshot of our VRESCo prototype, which includes an implementation of the case study used in this paper.

5 Evaluation

We will now evaluate our approach in two different ways. Firstly, we will qualitatively analyse the expressiveness of our approach by comparison with previously published adaptation patterns [14]. Secondly, we will have a look at performance implications. This is done by monitoring the weaving time in our prototype system, as well as comparing the execution time of dynamically weaved and statically defined composition instances.

5.1 Coverage of Adaptation Patterns

In order to discuss the expressiveness of our approach we have used the adaptation patterns defined in [14]. In this work, 14 patterns of structural changes in processes are identified. Using our approach 9 of these patterns are fully supported.

5 http://sourceforge.net/projects/vresco/
We have summarized the coverage of adaptation patterns in Table 1. The patterns AP1, AP2 and AP4 are the core feature of our approach, and can be implemented trivially. For AP2 and AP5 we specifically described a generic fragment in Section 3. Similarly, all of AP3, AP5, AP8, AP9, AP10 AP14 can be implemented rather elegantly using transparents. The patterns concerning subprocesses (AP6 and AP7) cannot be implemented since our approach does not support linking to more than one composition at the same time. AP11 and AP12 are in simple cases implementable using transparents, but our approach does not provide any explicit support for it, making the implementation rather cumbersome. Similarly, AP13 can be implemented by replacing the branching node as a whole, but we do not consider this solution as in line with the idea of this pattern.

5.2 Performance Analysis

In a second step we have evaluated the runtime implications of our prototype. For this, we monitored the average execution time of dynamically weaved composition instances with an increasing number of activities, and compare them to the same instance defined statically. We also compare online and offline weaving, and distinguish between fragments defined using transparents and fragments defined without. For simplicity, all compositions and fragments are sequences of “wait” activities. Using different types of activities would not have an impact on the evaluation outcome, since our adaptation approach handles all non-virtual activities the same way, i.e., weaving an “invoke” activity has a similar overhead than weaving a “wait” activity. To minimize external influences all results are the average of 50 independent test runs. We have also repeated the evaluation multiple times to make sure that the outcome is reproducible. The outcomes of these experiments are summarized in Figure 7(a).

As can be seen, online weaved compositions exhibit very little overhead as compared to statically defined compositions. Of course, offline weaving introduces some overhead, which stems from the time necessary to select the fragments, to implement the actual weaving, and to suspend and unsuspend the composition. In our experiments, the largest part of these factors is the actual

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern Name</th>
<th>Covered</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP1</td>
<td>Insert Fragment</td>
<td>✓</td>
</tr>
<tr>
<td>AP2</td>
<td>Delete Fragment</td>
<td>✓</td>
</tr>
<tr>
<td>AP3</td>
<td>Move Fragment</td>
<td>✓</td>
</tr>
<tr>
<td>AP4</td>
<td>Replace Fragment</td>
<td>✓</td>
</tr>
<tr>
<td>AP5</td>
<td>Swap Fragment</td>
<td>✓</td>
</tr>
<tr>
<td>AP6</td>
<td>Extract Sub Process</td>
<td>✗</td>
</tr>
<tr>
<td>AP7</td>
<td>Inline Sub Process</td>
<td>✗</td>
</tr>
<tr>
<td>AP8</td>
<td>Embed Fragment in Loop</td>
<td>✓</td>
</tr>
<tr>
<td>AP9</td>
<td>Parallelize Activities</td>
<td>✓</td>
</tr>
<tr>
<td>AP10</td>
<td>Embed Fragment in Conditional</td>
<td>✓</td>
</tr>
<tr>
<td>AP11</td>
<td>Add Control Dependency</td>
<td>✗</td>
</tr>
<tr>
<td>AP12</td>
<td>Remove Control Dependency</td>
<td>✗</td>
</tr>
<tr>
<td>AP13</td>
<td>Update Condition</td>
<td>✗</td>
</tr>
<tr>
<td>AP14</td>
<td>Copy Fragment</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1. Coverage of Adaptation Patterns
weaving time. Therefore, we have further analyzed this factor in Figure 7(b). We depict the weaving time depending on the number of activities to weave. Generally, concrete activities are faster to weave than transparent activities (since the logics of weaving transparents is more complicated), and offline weaving is faster than online weaving (since, in the online case, some additional sanity checks are done by the Windows Workflow runtime). In general, this increased weaving time for online weaving does not matter too much, since the online weaving time does not directly impact the execution time of the process. Overall, the overhead introduced by weaving is relatively constant in $[45 : 80]$ ms, even for large fragments (more than 80 activities).

Summarizing, we can see that dynamic weaving does not introduce a big overhead, especially if online weaving is possible. If offline weaving has to be used, an additional weaving overhead, which is generally in $[45 : 80]$ ms, is introduced. We argue that for most application areas this overhead is still far from being dramatic. Even though the concrete numbers are specific for our prototype implementation, they still show that implementing our ideas efficiently is well possible.

6 Related Work

In this paper we apply the AOP paradigm to adaptation of service compositions. On the level of atomic services earlier work in this direction has been presented by Kongdenfha et al [15]. In this work, they use the AOP paradigm to adapt the implementation of atomic Web services. A comparable approach has also been presented by Song et al. [16], who use the AOP approach to weave cross-cutting concerns, such as security, into all atomic Web services in a composition. A similar track has also been followed by Narendra et al., who used AOP-based adaptation of services in a composition to propagate changes in non-functional properties through the composition [17]. Of course, all of these approaches assume that the developer has access to the implementation of these atomic services.

The general scope of our work is similar to work presented by Gmach et al. [18]. However, the focus of our contribution is purely on adaptation of service
compositions, while Gmach et al. adapt on service infrastructure level (i.e., by moving services to different hosts, or by re-scheduling requests in the service bus). Finally, adaptation with the explicit goal of preventing SLA violations has been discussed by various authors, e.g., our own earlier work on PREVENT [13] or recent work by Metzger et al. [19]. The concrete execution of adaptation of compositions has in the past been covered by research in various directions. Earlier approaches often did not consider the adaptation of the composition structure at all, instead focusing solely on service rebinding. In a simplistic manner such adaptations can in fact be carried out using WS-BPEL alone, by using the Dynamic Partner Link feature. However, practical problems such as finding the right service to bind to (often based on QoS), or the need to resolve interface differences, demand for more sophisticated service rebinding approaches. Examples of such work include the WS-Binder [20] or the PAWS [21] frameworks. More advanced service rebinding was also one of the contributions of Moser et al. in [22]. Finally, some work on service rebinding (dealing also with stateful services) has been presented by Mosincat and Binder in [23].

An early approach towards structural adaptation of compositions has been discussed in [24]. However, in this work no free-form adaptation is possible. Instead, predefined parameterizations are applied if certain conditions hold. Arguably, the AOP paradigm can provide a more powerful abstraction for adaptation in compositions. This idea has first been introduced by Charfi et al. [25, 26]. However, unlike our work, Charfi et al. focus on the traditional AOP idea of weaving crosscutting concerns into the composition, while we apply the AOP paradigm with a different goal (adaptation for SLA compliance) in mind. Using aspects for runtime adaptation in WS-BPEL has been covered by the BPEL’n’-Aspects framework [8]. Our main contribution over this work is that in our case aspects can be composition fragments, while BPEL’n’Aspects supports only single Web service invocations as aspects. Work with similar goals, but specific to the telecommunications domain, has been presented by Niemöller et al. [27]. An approach which deals with process fragment composition is presented by Eberle et al. [28]. Their idea is to exploit the redundancy in separately modeled composition fragments and use those redundant overlapping fragment parts to merge the fragments. In our approach we model how fragments should be merged explicitly by using virtual activities.

7 Conclusions and Future Work

In this paper we have presented an approach to runtime adaptation of service compositions for preventing SLA violations. The adaptation is based on composition fragments which are dynamically substituted at runtime using AOP techniques. Composition fragments are modeled separately and are explicitly linked into the original composition using virtual activities. In addition to their process logic, fragments specify also their expected impact on the composition performance. This is necessary in order to be able to choose the best fitting fragments for preventing a predicted SLA violation at composition runtime. We have implemented the approach using Windows Workflow Foundation technology and experiments show that the performance impact of dynamic weaving is acceptable.

While the current status of the approach is promising, there are still some open issues left for future work. Firstly, we do not take into account that adap-
tation which prevents the violation of one SLA metric could easily lead to the violation of another. In particular, we currently do not take the costs of adaptations into account (e.g., increased costs by using more expensive services in the weaved fragment) which in some cases could be higher than the gain of not violating the SLA. Therefore, we will extend the impact model and its evaluation in our future work. Secondly, we currently assume that the number of possible combinations of advices to apply is small, so that finding the best combination via full enumeration is possible. In future work we plan to embrace heuristic optimization for cases where full enumeration is not feasible.

Acknowledgments

The research leading to these results has received funding from the European Community’s 7th Framework Programme under the Network of Excellence S-Cube (Grant Agreement no. 215483).

References


Preventing KPI Violations in Business Processes based on Decision Tree Learning and Proactive Runtime Adaptation

Branimir Wetzstein¹, Asli Zengin², Raman Kazhamiakin³, Annapaola Marconi², Marco Pistore², Dimka Karastoyanova¹, Frank Leymann¹

¹Institute of Architecture of Application Systems (IAAS), University of Stuttgart, Germany
branimir.wetzstein@iaas.uni-stuttgart.de
karastoyanova@iaas.uni-stuttgart.de
leymann@iaas.uni-stuttgart.de

²Fondazione Bruno Kessler IRST, Italy
zengin@fbk.eu
marconi@fbk.eu
pistore@fbk.eu

³SayService srl, Trento, Italy
raman@sayservice.it

Abstract: The performance of business processes is measured and monitored in terms of Key Performance Indicators (KPIs). If the monitoring results show that the KPI targets are violated, the underlying reasons have to be identified and the process should be adapted accordingly to address the violations. In this paper we propose an integrated monitoring, prediction and adaptation approach for preventing KPI violations of business process instances. KPIs are monitored continuously while the process is executed. Additionally, based on KPI measurements of historical process instances we use decision tree learning to construct classification models which are then used to predict the KPI value of an instance while it is still running. If a KPI violation is predicted, we identify adaptation requirements and adaptation strategies in order to prevent the violation.

Key words: Business Activity Monitoring, Business Process Intelligence, WS-BPEL, Decision Tree, Process Adaptation.

1. Introduction

In recent years, the industry experienced a wide adoption of the service-oriented architecture for the implementation of business processes [15]. Service-based applications realize such processes by modeling and deploying a complex, distributed and layered system, where the business model of an application is implemented through a service composition which orchestrates services running on a service infrastructure [12, 14]. To be effective, such applications should meet certain business goals, traditionally expressed as Key Performance Indicators (KPIs) of the business processes. These KPIs are typically continuously monitored at run-time using business activity monitoring techniques.

If monitoring shows that KPI targets are not reached, then it is necessary to identify the factors which strongly influence the KPI and cause KPI target violations most often. In complex business processes, the relations between the overall business process performance and lower-level influential factors and their combination are neither explicit nor easy to reveal. In addition to identifying the influential factors based on historical process executions, it is desirable to be able to predict for a running process instance whether it will reach the KPI target. This allows us to react in a timely fashion and possibly prevent a predicted KPI target violation by identifying an adaptation strategy which can potentially improve the performance of the running process instance.

In this paper we present an integrated monitoring, prediction and adaptation approach that aims to address the above problems. The execution of the business process is continuously monitored based on runtime events published by the process execution middleware. Based on monitoring data of historical process instances, we use decision tree algorithms in order to learn the dependencies between the KPI and the influencing lower-level metrics. The resulting KPI dependency tree is used for KPI prediction in future process instances. If for a running process instance, a KPI target violation is predicted, adaptation requirements are extracted from the decision tree specifying predicates on the...
metric values that should be improved. In the next step, we identify adaptation strategies consisting of adaptation actions which should be performed in order to satisfy the adaptation requirements. In a subsequent step, we filter and rank adaptation strategies based on a constraints and preferences model. Finally, the process instance is pro-actively adapted in order to prevent the KPI target violation. The presented work builds on the work presented in [16] which focused on monitoring and KPI dependency analysis, and extends, refines and evaluates our preliminary ideas presented in [7], where the overall monitoring, analysis and adaptation framework has been described on a higher level.

The paper is organized as follows. We begin with a motivating scenario that describes the problem and which we use in the rest of the paper to present our solution. Section 3 gives an overview of the approach by describing its lifecycle. Section 4 presents the different types of artifact models created at design time. Section 5 describes the runtime phases consisting of learning of KPI dependency trees, KPI prediction, and adaptation. Section 6 describes the implementation of the approach and presents results of an experimental evaluation. Finally, we give a summary of related work and conclude the paper together with the directions for future work.

2. Scenario and Motivation

In this section we introduce a scenario that we use in the following sections for explaining our approach. As shown in Figure 1, the scenario consists of a purchase order process implemented by a reseller who offers products to its customers and interacts with external supplier, banking, and shipment services for processing the order. Furthermore, the reseller uses warehouse and packaging services, which are internal to the organization.

![Fig. 1: Purchase Order Process](image)

For measuring the performance of its business process in terms of quality dimensions such as time and cost, the reseller defines a set of Key Performance Indicators (KPIs). A KPI is based on a metric and specifies based on business goals a target value function which maps values of that metric to a set of nominal values, a.k.a. KPI classes. For instance, in our scenario the reseller could choose “order fulfillment lead time” (process duration from order receipt until the shipment arrives at the customer) as the KPI metric, and then specify a “traffic light function” with three KPI classes that have a business meaning to him, e.g.: m < 4 days is “green”, 4 days < m < 7 days is “yellow”, otherwise “red”.

The defined KPIs have to be measured and calculated based on executed process instances. If after a while the monitoring shows an unsatisfactory result, i.e., undesirable KPI classes (KPI violations) are reached for many instances (i.e. purchase orders are late, in our case), the reseller wants to find out the influential factors which lead to those KPI classes. Automatically identifying the influential factors is not trivial. Let us consider for instance our reference example: understanding the reasons why certain orders are delivered on time and others are not is a complex task, as the KPI depends on the
PREVENTING KPI VIOLATIONS IN BUSINESS PROCESSES BASED ON DECISION TREE LEARNING AND PROACTIVE RUNTIME ADAPTATION

combination of several factors such as ordered product types and amounts (input data of the process), duration and availability of the internal services, duration, reliability, SLA conformance of external services etc. After the KPI dependencies have been understood based on historical process instances, they can be used for predicting the KPI classes of future instances.

The next step is then to try to prevent the KPI violations in future. This can be done using runtime process instance adaptation. Therefore, we assume that there is a set of alternatives in the process execution which can be chosen dynamically to proactively adapt the process instance. Assume for example that there is a set of alternative services for a set of service types used in the process. For example, there might be several alternative shippers which offer different service levels 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, reputation, etc.) Based on the prediction, the goal is then to choose a shipper which will be likely to lead to a desirable KPI class.

3. Solution Overview and Methodology

In this section we give an overview of our approach by describing its lifecycle as shown in Figure 2.

![Fig. 2: Lifecycle of the Approach](image)

The supporting architecture and implementation are described in Section 6. The lifecycle consists of the following phases:

1. **Modeling:** In the modeling phase (not shown explicitly in Figure 2), several models are created: (i) application model, which in our case consists of a deployable process model implemented as a service orchestration, (ii) metrics model, defining process and QoS metrics which have to be measured, and the KPIs specifying targets on key metrics based on business goals (iii) adaptation action model, specifying the available adaptation actions, (iv) check point model, defining at which points in the process the prediction and potential adaptation should take place, (v) preferences and constraints model, needed for selection of an adaptation strategy.

2. **Monitoring:** In the monitoring phase, all metrics specified in the metrics model are monitored. That includes the KPIs but also lower-level metrics of the potential influential factors. As a result, metric values for a set of executed process instances are obtained.

3. **KPI Dependency Analysis:** After a certain number of executed process instances, for each KPI at each checkpoint a decision tree is trained which helps to understand the dependencies of that KPI on lower-level metrics. The resulting KPI dependency trees serve from now on as
classification models for future process instances and are used for KPI prediction. The learning of the decision trees is performed “offline” (in the background) based on history data and does not affect the execution of running process instances.

4. **KPI Prediction:** When a running process instance reaches a checkpoint, it halts its execution. The metric values which have been measured until the checkpoint for that instance are gathered and used as input to the classification model(s) learned in phase 3. The prediction result per KPI is (in the special case) a predicted KPI class (e.g., “good”, “medium” or “bad”) 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 metrics should be improved to reach a specific KPI class and serves thus as basis for adaptation.

5. **Identification of Adaptation Requirements and Adaptation Strategies:** Adaptation requirements are identified by extracting the metrics that should be improved from the instance tree(s). Based on the adaptation requirements, a set of alternative adaptation strategies is identified by taking into account available adaptation actions. An adaptation strategy thus consists of a set of adaptation actions that should be used in the process instance in order to reach a certain desired KPI class.

6. **Selection of an Adaptation Strategy:** The list of alternative adaptation strategies is filtered and ranked based on a constraints and preferences model. Constraints allow defining conditions which should never be violated, while preferences are specified as weights on different KPIs and metrics and lead to a strategy score number determined based on multiple attribute decision making.

7. **Adaptation Enactment:** The adaptation strategy with the best score is enacted by executing the adaptation actions. The process instance is unblocked and continues its execution while taking into account the performed adaptation.

After the steps 4-7 have been performed for a certain number of instances, the effectiveness of the adaptations can be evaluated by checking how many KPI violations have been prevented and how many instances still violate their KPIs. This might lead to (re-)adjustment of the models, e.g., adjustment of KPI targets, (re)moving or adding of checkpoints, and adjustment of the constraints and preferences model.

### 4. Design Time: Modeling for Monitoring, Prediction and Adaptation

In this section, we describe the different types of models created at design time: metrics model, adaptation actions model, check point model, and constraints and preferences model. These models are used as input to the runtime phases (Figure 2). An overview of the overall metamodel is shown in Figure 3.

---

**Fig. 3: Design-Time Artifacts Metamodel**
4.1 Metrics and KPIs

The metrics model contains (i) KPIs and the underlying KPI metrics, i.e. key metrics reflecting the time, cost, and quality dimensions of the process which help to assess the process performance, (ii) metrics which KPIs potentially depend on, i.e., lower level metrics used during KPI dependency analysis and prediction.

A metric definition contains in particular the following elements:

- **data domain**, i.e. all unique values the metric can contain (e.g., real numbers, nominal values)
- **entity** characterized by the metric (e.g., a process instance, activity instance, service endpoint)
- **measurement definition** which specifies how the metric value is to be obtained. It therefore uses one or more measurement mechanisms, e.g., a probe, or an event processing engine. This definition can be based on other metrics.

The metrics model is deployed on the monitoring infrastructure and is used to monitor process instances during their execution in order to obtain metric values for the defined metrics. The metric measurement can be realized based on diverse monitoring mechanisms. In our prototype, we use an event-based approach receiving process events from the process execution engine and the service infrastructure and correlating and aggregating those events based on complex event processing (see Section 6).

In our scenario, we define among others the metric **Order Fulfillment Time** to measure the duration of each process instance of the reseller process. The metric value is calculated by receiving and correlating two corresponding events of a reseller process instance (start of the activity “Receive PO” and end of the activity “Shipment”) and subtracting their timestamps. The correlation is performed based on a process instance ID which every corresponding process event contains. The corresponding metric values (one per process instance) are stored in a Metric Database.

In order to be able to assess the monitored metric values in respect to business goals, a set of KPIs is defined. A KPI is defined based on a metric (KPI metric) and maps value ranges of that metric to a set of nominal values (KPI classes) which allow evaluating whether that metric conforms to business goals.

A Key Performance Indicator (KPI) definition contains the following elements:

- the underlying KPI metric
- a set of nominal values (>=2) representing KPI classes (e.g., “good”, “medium”, “bad”)
- a target value function which maps values of the KPI metric to KPI classes

The KPI is itself a metric and is defined in the metrics model.

In the scenario, we use the metric Order Fulfillment Time as the KPI metric, specify three KPI classes “green”, “yellow”, “red”, and then define a target value function as follows: < 4 days $\rightarrow$ “green”, > 4 days and < 7 days $\rightarrow$ “yellow”, otherwise “red”. Note that in this case, the KPI class is evaluated per process instance as the underlying metric is evaluated per process instance. The KPI could however also be based on a metric that is calculated based on several process instances in a period (e.g., average order fulfillment time in the last month).

4.2 Adaptation Actions

The adaptation actions model defines (i) a set of adaptable entities in a process and (ii) a corresponding set of adaptation actions which implement alternative realizations of adaptable entities.

An adaptable entity definition contains the following elements:

- **entity** which can be adapted (e.g., in a BPEL process [13] that could be a particular partner link instance, activity instance, variable instance)
- a set of metrics which characterize this entity

In our scenario, we define the shipment and supplier partner links in the reseller process as adaptable entities, as there are alternative shipment and supplier services available. For the shipper partner link, for example, we define “shipment delivery time”, “shipment cost”, “shipper reputation” as characterizing metrics.

An adaptation action (AA) definition contains the following elements:

- adaptable entity targeted by this action
adaptation specification which defines how the adaptable entity is to be adapted, e.g., substitution of another service, skipping of a process activity, process variable value change etc.

- a set of metric effects which specify the impact of the adaptation action on metric values of the adaptable entity. The impact is specified as a predicate on metric values (e.g., delivery time < 3 days). The metric effects can be derived from past measurements; if no such measurements are available then they have to be estimated or in some cases can be derived from SLAs (e.g., in case of service substitutions).

We have predefined three adaptation action types in our prototype which can be used for adapting a running BPEL process instance after it has been halted at a check point (Section 4.3): (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, which 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.

In our concrete scenario, we have assumed that there is a set of alternative shipment and supplier services with different QoS characteristics. For each alternative, we have created an adaptation action and specified its effects on metrics of the corresponding adaptable entity. Thereby, we assume that the effects can be derived from SLAs, or estimated based on experience if no measurement data is yet available on those services. For example, an AA which substitutes a new shipment service defines its effects on the shipment delivery time, shipment cost, and shipper reputation.

4.3 Check Points

For performing prediction and adaptation, one defines one or more checkpoints in the process. A checkpoint definition contains the following elements:

- a trigger defined as a process runtime event (or derived event from 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.

- a set of available metrics from the metrics model whose metric values are available at this checkpoint and which should be used as explanatory attributes for creating the classification model (one per KPI) for this checkpoint (see Section 5.1). The set of available metrics for a check point is created automatically (by analyzing the process model and deriving data and time metrics available at a check point) and provides suitable results in most cases but can be adjusted by an “advanced user” if he wants to influence the classification learning process (see [16] for a more general discussion on this topic).

- a set of available adaptation actions from the adaptation actions model which can be used to adapt the process at this checkpoint.

Obviously, the set of available metrics 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 actions will decrease, and thus there will be 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 check points in a process model.

In our example, after the “check in stock” activity at the beginning of the process, available metrics are e.g. the ordered product types and amounts, the customer, the process duration until that activity, and whether the ordered items are in stock. Available adaptation actions are supplier and shipment service substitution.

4.4 Constraints and Preferences

When several alternative adaptation strategies are identified, we need to make a decision which of those alternatives is to be selected. Thereby, we address two aspects: (i) adaptation strategies might violate certain rules or thresholds which should always be avoided, (ii) adaptation strategies have
different effects on a set of competing KPIs and metrics (e.g., time vs. cost). The former aspect is addressed via constraints, the latter via preferences.

A constraint defines a boolean-valued predicate over one or more metrics measured for the running instance. If during the selection of a strategy a constraint evaluation results in the value “false” for a strategy, then that strategy is removed from the set of alternatives.

In our scenario, we use constraints for defining which KPI classes should be prevented in any case (KPI target violations), e.g., by specifying that the predicted class of a KPI !="red". We use constraints also on metrics of adaptable entities, e.g., maximal cost of supplier and shipper service < x.

Preferences are used for ranking of adaptation strategies according to a score represented by a number between 0 and 1. Therefore, we use Simple Additive Weighting as part of Multiple Attribute Decision Making [5]. At design time, the user has to assign a weight between 0 and 1 to each KPI and metric of an adaptable entity, whereby the sum of all weights should be 1. At runtime, then a score is calculated as discussed in Section 5.4.

In the scenario, where we have specified one KPI and three metrics for each of the two adaptable entities, we thus have to assign weights to seven metrics in total.

The constraints and preferences model can be used at design-time for creating a default configuration of the adaptable entities (i.e., provide them with initial values). In our case, we can select a combination of a supplier and shipper service which has the highest score according to the subset of preferences and constraints which can be evaluated at design time (e.g., KPI class is not known before runtime, however the delivery times of the shippers and suppliers are known at design time from the modeled metric effects and can be used to create a default configuration). This default configuration can then be changed at runtime based on KPI prediction results.

5. Runtime: KPI Dependency Tree Learning, Prediction and Adaptation

In this section, we present the runtime phases in detail. Figure 4 shows an overview of the artifacts created at runtime. After a set of process instances and corresponding metric values have been monitored as defined in the metrics model, a KPI Dependency Tree is learned for each KPI at each check point based on classification learning techniques. When a checkpoint is triggered for a running process instance thus creating a check point instance, the KPI Dependency Tree is used to predict the class of the corresponding KPI in the running process instance by inserting the available metrics values at that check point instance into the tree. The result is an instance tree which shows the predicted KPI class in relation to metrics which can be adapted. The instance tree is then used to derive adaptation requirements and corresponding adaptation strategies consisting of adaptation actions.

![Fig. 4: Runtime Artifacts Metamodel](image-url)
5.1 Creating KPI Dependency Trees based on Classification Learning

The KPI metric value and the corresponding KPI class depend typically on the combination of a set of influential factors (alternatively, influential metrics), e.g., input data to the process (ordered product types and amounts), service outputs (e.g., ordered products available in stock) and processing duration of used services (e.g., shipment delivery time).

In order to find out these dependencies, we use classification learning known from machine learning and data mining [17]. In a classification problem, a dataset is given consisting of a set of examples (a.k.a., instances) described in terms of a set of explanatory attributes (a.k.a., predictive variables) and a categorical target attribute. The explanatory attributes may be partly categorical and partly numerical. By using a learning algorithm, based on the example dataset (a.k.a., training set) a classification model is learned (a.k.a., supervised learning), whose purpose is to identify recurring relationships among the explanatory variables which describe the examples belonging to the same class of the target attribute. The so created classification model can be used to explain the dependencies in past instances but in particular also to predict the class of (future) instances for which only the values of the explanatory attributes are known.

We map the KPI Dependency Analysis to a classification problem by defining the KPI as the categorical target attribute with categorical values as KPI classes, and a set of lower-level metrics (potential influential factors) which serve as explanatory attributes for this KPI at the check point. Classification learning for a KPI is then performed for each check point separately as the set of available explanatory attributes is different for each check point. This set consists of two types of metrics: (i) metrics whose values are available at the check point; this set is part of the check point definition (ii) metrics whose values cannot be measured until the checkpoint but which are affected by the available adaptation actions of the check point. The latter group of metrics is important as we want to learn how the KPI class depends on metrics which are affected by adaptations. This will allow us to extract adaptation requirements from the tree (Section 5.3). If a tree would be trained only based on available metrics at a check point, then the prediction would yield the predicted KPI class as a result, however we would not know how to adapt the process in case a bad KPI class is predicted.

At process runtime, after a set of process instances have been executed, we construct a data set for a KPI at a check point as follows. For each instance, we create a data item consisting of (i) the metric values of the available metrics and “adaptable” metrics defined for that check point, (ii) the KPI class of the KPI metric value for this instance. Based on this data set resulting from monitoring, a classification problem consists now of identifying a classification model that can optimally describe the relationship between the metrics and the KPI class.

There are different types of algorithms for classification model learning and prediction, e.g., artificial neural networks, classification rules, and support vector machines [17]. We have decided to use decision trees because of their following advantages in our context: (i) They constitute a white box model as they show explicitly the relationships between explanatory attribute value ranges and categorical target attributes (i.e., KPI classes). Thus they are easy to understand and interpret for people and enable human support in the learning and adaptation phases. (ii) They support both explanation and prediction. (iii) In particular, they support extraction of adaptation requirements from the tree paths (Section 5.3). (iv) Furthermore, decision trees support both numeric (typically, time based metrics) and categorical explanatory attributes (typically, process data based metrics).
A decision tree algorithm works by splitting the instance set into subsets by selecting an explanatory attribute (new node in the tree) and corresponding splitting predicates on the values of that attribute (branches). This process is then repeated on each derived subset in a recursive manner until all instances of the subset at a node have the same value of the target attribute or when splitting does not improve the prediction accuracy. There are different types of decision tree algorithms. They differ, for example, in how they select predictive attributes for splitting (e.g., based on information entropy), or splitting predicates, e.g. whether the tree is binary, or can have more than two outgoing edges per node. The algorithm automatically performs a validation of the learned classification model (e.g., using cross-validation) and calculates quality metrics of the tree, in particular its accuracy, i.e. the percentage of correctly classified instances (based on a test set). A KPI dependency tree is learned automatically at runtime for each KPI per checkpoint. It can be configured after how many instances the tree should be learned and when it should again be retrained.

In our approach, we have used the popular J48 algorithm to generate the KPI dependency tree [17].

A **KPI Dependency Tree (J48)** consists of a (possibly empty) set of non-leaf nodes representing metrics and a non-empty set of leaf nodes representing KPI classes. Thereby, a particular metric or KPI class can be present in the tree zero to several times. An outgoing edge of a tree node defines a predicate on the values of the metric of that node. The metric values on outgoing edges of a node are disjoint. Each leaf node contains 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, we learn which metric values lead to a particular KPI class, and for how many instances that was the case in the past.

An example tree is shown in Figure 5. It has been generated for the Order Fulfillment KPI at the checkpoint defined right after the “check in stock” activity on the basis of 100 instances. The tree contains available metrics (order in stock, item quantity) and “adaptable” metrics at the checkpoint (shipment delivery time, supplier delivery time). It shows, for example, that for the combination “order in stock=true” and “item quantity <= 20” the KPI class “green” has always been reached in the past (which was the case for 30 instances). Overall, the tree shows that the order fulfillment time KPI mainly depends on whether the ordered items are available in stock. In the positive case, 45% (30+15) of all instances reached “green”, and 12% reached “red”. In the other case (order in stock = false), many KPI violations have occurred (36+23) and the KPI class mainly depends on the shipment delivery time and supplier delivery time.

### 5.2 Runtime Prediction based on KPI Dependency Trees

At process runtime, after a sufficiently large set of instances has been executed and monitored, based on the checkpoint definition, for each checkpoint a decision tree is learned. It explains how the KPI classes of those history instances depend on influential factor metrics.

![Fig. 5: A KPI Dependency Tree for the Order Fulfillment Time KPI](image)
In the next step the decision tree can be used for prediction. When the process instance execution reaches a checkpoint which is signaled by the specified event, the available metrics for that instance until the checkpoint are gathered and "inserted" into the decision tree for that checkpoint. Therefore we traverse the tree breadth-first: if the current node corresponds to an available metric, we follow the outgoing branch whose predicate is satisfied by the measured metric value and replace the current node with the target node of that branch; otherwise, if the metric is not available (but affected by available adaptation actions) we leave the node in the tree (and continue with its children until a leaf node is reached).

As the result we get 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 which correspond to metrics which are affected by the available adaptation actions. In the latter (general) case, the KPI class is thus predicted in relation to the values of metrics which can be adapted.

![Instance Tree of the above KPI Dependency Tree](image)

**Fig. 6: An Instance Tree of the above KPI Dependency Tree**

Figure 6 shows an instance tree created from the original tree shown in Figure 5 assuming that we have measured "order in stock=false". This tree consists now only of metrics which are yet unknown but are affected by the available adaptation actions of this checkpoint. It is used for extraction of adaptation requirements as discussed in the next section.

### 5.3 Identification of Adaptation Requirements

At each checkpoint, after obtaining an instance tree for each KPI, we have to decide whether adaptation is needed, and if yes, which metrics should be improved and how. An instance tree shows how the KPI class of the running instance depends on the metrics affected by available adaptation actions (adaptable metrics).

If the instance tree contains only one leaf denoting the KPI class, then the predicted KPI class is independent of the adaptable metrics 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 6), then the non-leaf nodes correspond to influential factor metrics which are adaptable by the predefined adaptation actions and the tree shows how we should adapt. For example, if we ensure a supplier delivery time below 3 days and a shipment delivery time below 2.2 days we will very likely (assuming that the classification model has a high accuracy, see Section 6) reach a "green" KPI class.

The idea towards adaptation is thus (i) to extract those paths and the corresponding metric predicates, which lead to desirable KPI classes, and then (ii) select adaptation actions, which will lead to satisfaction of those metric predicates. We call the paths of the instance tree which lead to desirable KPI classes safe paths. The user can configure which KPI classes are desirable for an instance (e.g., "green", and "yellow" could be desirable, while "red" is to be avoided) via constraints in the constraints and preferences model. If we ensure one of the safe paths, then we avoid all of the undesirable paths, i.e. KPI target violations. Thus, eventually each safe path (consisting of a conjunction of metric predicates) is an alternative adaptation requirement for the corresponding KPI.
An adaptation requirement (AR) is extracted from a safe path as follows: from each branch on the path we extract the metric predicate and add it to the adaptation requirement. The predicates are combined by using logical conjunction, i.e., all predicates have to be true in order to satisfy the requirement. Finally, in the last step if there are predicates which are satisfied with semantically worse metric values (e.g., if a predicate is “supplier delivery time > 2 days”), then it can be ignored and removed from the requirement because the value does not have to be improved.

In case more than one KPI has been defined, alternative adaptation requirement sets are extracted from each instance tree separately and then combined by building a Cartesian product between them, whereby some of the resulting ARs can contain contradictory metric predicates and are then removed.

An adaptation requirement (AR) specifies:

- the predicted desirable KPI class for one or more KPIs
- a conjunction of metric predicates which should be achieved in order to reach those desirable KPI classes

As a result we get a set of alternative adaptation requirements each consisting of a conjunction of predicates over adaptable metrics which have to be satisfied.

For the example instance tree (Figure 6), we can extract two adaptation requirements as shown in Table 1 (first two columns), one for the KPI class “green” and one for the KPI class “yellow”. We assume here that we have specified only one KPI, and that a constraint has been defined specifying the KPI class “red” to be undesirable.

5.4 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 (AS) consists of a set of adaptive actions which satisfy the metric predicates of an adaptation requirement and the constraints in the constraints and preferences model.

A valid adaptation strategy (AS) consists of the following elements:

- the adaptation requirement addressed by the strategy
- a set of adaptation actions which should be enacted for this strategy, whereby all metric predicates of the corresponding adaptation requirement are satisfied by the metric effects of the adaptation actions and all constraints are satisfied
- a score number, calculated on the basis of the specified preferences

In the first step, for each adaptation requirement a set of alternative strategies is identified as follows: (i) for each metric predicate of the AR we enumerate (alternative) adaptation actions which satisfy those predicates according to their metric effects; (ii) if there are adaptable entities where at least one of their metrics is not part of the AR, then all AAs for each such adaptable entity are also enumerated; (iii) the sets of AAs created in (i) and (ii) are combined using Cartesian product creating a set of alternative adaptation strategies for this AR. The result is a set of alternative adaptation strategies which would all according to the AR solve the KPI class(es) of the corresponding adaptation requirement. Finally, the resulting set of alternative adaptation strategies is the sum of adaptation strategies for each (alternative) AR.

In the second step, that set is further filtered according to the constraints defined in the constraints and preferences 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.

In the third step, the strategies are finally ranked according to a 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 a metric set \( M_w = \{ m_1, m_2, \ldots, m_n \} \) consisting of KPIs and metrics of the adaptable entities (Section 4.4). For each adaptation strategy \( x \) and metric \( y \) in \( M_w \) we can determine the value \( v_{xy} \) (either from measurements or metric effects). Before applying the simple additive weighting (SAW) [5], we have to normalize these metric values to make the different metrics comparable. The normalized metric value \( n_{v_{xy}} \) can be calculated by using the division by maximum value method:

\[
 n_{v_{xy}} = \begin{cases} 
 \frac{v_{xy}}{\max \{ v_{1y}, v_{2y}, \ldots, v_{py} \}} & \text{if "higher is better"} \\
 \frac{v_{xy}^{-1}}{\max \{ v_{1y}^{-1}, v_{2y}^{-1}, \ldots, v_{py}^{-1} \}} & \text{if "lower is better"} 
\end{cases}
\]
The normalized metric values \( n_{xy} \) are in the range between 0 and 1, whereby the value 1 is always given to the best metric value. We thereby have to distinguish between metrics where a higher value is better (e.g., reputation) and metrics where a lower value is better (e.g., cost). Note that for KPIs and any other non-quantitative metrics, the categorical values have to be mapped to a cardinal scale to enable proper calculation of a normalized value; this mapping has to be provided by the user.

Finally, for each strategy we can calculate a score by summing up the weighted metric values:

\[
\text{Score}_x = \frac{1}{p} \sum_{y=1}^{p} w_y n_{xy}
\]

Finally, the best ranked strategy is selected and enacted.

Tab. 1: Identification and Ranking of Adaptation Strategies

<table>
<thead>
<tr>
<th>Conditions</th>
<th>KPI</th>
<th>Strategy</th>
<th>Adaptation Strategies</th>
<th>Constraints</th>
<th>Score</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shipment Deliv. Time &lt; 2.2</td>
<td>green</td>
<td>Sh1 Premium + Su1 Premium</td>
<td>(1.0, 0.9, 0.2, 1.0, 1.0, 0.3, 0.9)</td>
<td>ok</td>
<td>0.61</td>
<td>3</td>
</tr>
<tr>
<td>Supplier Deliv. Time &lt; 3.0</td>
<td></td>
<td>Sh1 Premium + Su2 Premium</td>
<td>(1.0, 0.9, 0.2, 1.0, 0.8, 0.5, 0.8)</td>
<td>ok</td>
<td>0.64</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sh2 Premium + Su1 Premium</td>
<td>(1.0, 1.0, 0.1, 0.8, 1.0, 0.3, 0.9)</td>
<td>nok</td>
<td>0.57</td>
<td>-</td>
</tr>
<tr>
<td>Shipment Deliv. Time &lt; 2.2</td>
<td>yellow</td>
<td>Sh1 Premium + Su1 Standard</td>
<td>(0.5, 0.9, 0.2, 1.0, 0.8, 0.8, 0.8)</td>
<td>ok</td>
<td>0.615</td>
<td>2</td>
</tr>
<tr>
<td>Supplier Deliv. Time &lt; 7.5</td>
<td></td>
<td>Sh1 Premium + Su2 Standard</td>
<td>(0.5, 0.9, 0.2, 1.0, 0.7, 0.7, 0.6)</td>
<td>ok</td>
<td>0.565</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sh1 Premium + Su1 Premium</td>
<td>(0.5, 0.9, 0.2, 1.0, 1.0, 0.3, 0.9)</td>
<td>ok</td>
<td>0.51</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1 shows the two adaptation requirements extracted from the instance tree (Figure 6) and the identified alternative strategies per requirement. Each strategy consists here of a combination of a shipper service and supplier service with different metric 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 reputation metrics for the shipper and the supplier, respectively. Based on the weight distribution (0.2, 0.05, 0.25, 0.1, 0.05, 0.25, 1.0) in the preferences model, the score for each strategy is calculated and used for ranking.

6. Prototype Implementation and Experimental Evaluation

We have implemented the approach as shown in Figure 7. Our prototype uses Apache ODE\(^1\) as the business process execution engine which executes BPEL processes [13]. The monitoring is performed based on the ESPER complex event processing (CEP) framework\(^2\) which calculates metrics based on events which are published by the process engine and a QoS monitor as already described in [16]. The classification model learner is based on the WEKA suite\(^3\) which provides decision tree algorithm implementations. For the implementation of check points and instance adaptation, we use a framework which extends the Apache ODE BPEL engine [8]. The check points are supported via blocking events which stop process instance execution until they are explicitly unblocked by a corresponding incoming event coming from our framework. The adaptation actions are supported by the same mechanism of incoming events whereby our framework populates the corresponding incoming event with the new partner link value, variable value, or the state of an activity and sends it to the process engine.

\(^1\) http://ode.apache.org/
\(^2\) http://esper.codehaus.org/
\(^3\) http://www.cs.waikato.ac.nz/ml/weka
6.1 Experimental Evaluation

We have implemented the scenario from Section 2 as a BPEL process interacting with six Web services. The Web services have been implemented in Java and for experimental purposes simulate certain influential factors (e.g., duration and output are made dependent on factors such as product types and amounts, and random behavior). For experimentation, we have deployed all these components on a single desktop PC. We define Order Fulfillment Lead Time as the KPI to be analyzed and define two checkpoints in the process (after “Check Stock” (i.e., both supplier and shipper can still be selected), and before “Shipment” thus allowing only the shipper to be selected). We create a set of overall 30 service candidates with different QoS characteristics (specified as mean values) and create a configuration which simulates the behavior of those services according to their QoS characteristics, but with deviations.

Tab. 2: Experimental Results

<table>
<thead>
<tr>
<th>Check Point</th>
<th>Learning Decision Tree Accuracy</th>
<th>Weights</th>
<th>Prediction and Prevention (200 instances per run)</th>
<th>KPI Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>KPI/time</td>
<td>No Need</td>
<td>Adapted and No Need</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cost/rept.</td>
<td>(predict/measured)</td>
<td>(predicted/successful)</td>
</tr>
<tr>
<td>None</td>
<td>N/A</td>
<td>0.2/0.1/0.5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.2/0.5/1.0</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Warehouse</td>
<td>88.2%</td>
<td>0.2/0.1/0.5</td>
<td>102/63</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.2/0.5/1.0</td>
<td>108/105</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>85/85</td>
<td>6/6</td>
<td>109/92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>105/103</td>
<td>5/5</td>
<td>90/88</td>
</tr>
</tbody>
</table>

**KPI Dependency Tree Learning Phase:** We trigger the execution of 500 process instances using a test client. For each of these instances we select the concrete supplier service and shipper service randomly in order to ensure that history data used for learning contains metrics data on each of these services and on most of their combinations. During process instance execution, the previously specified metrics are measured and saved in the metrics database. Then, for each checkpoint a decision tree is learned using the J48 algorithm [17]. The results and aspects to consider are as follows:

- **Duration:** The performance of the learning of a tree is about 15 seconds for 500 instances. As learning can be done in the background it does not affect the instance execution.
- **Accuracy:** The quality of the trained tree as a classification model can be assessed in terms of its accuracy, i.e., the percentage of correctly classified instances (from a test set) by the
model. This metric is provided by the decision tree algorithm after validation of the model (cross-validation) [17]. In general, the later the checkpoint is defined in the process the better the tree quality will be, because there are more known metrics that can be used for training the tree (see column tree accuracy in Table 2). In our approach, we assume that a classification model with a reasonably high accuracy has been created and do not take the accuracy into account in the following prediction and adaptation phases.

- **Historical Data**: The dependency tree is generated based on metric values resulting from historical process executions and corresponding adaptations. Metric effects resulting from adaptation actions which have *never* been used before are not reflected in the tree and would not later be considered during extraction of adaptation requirements. Thus, there should be a “bootstrapping” phase where several adaptations are performed either randomly (as in our experiments) or based on some other criteria in order to create historical instance data used for learning.

**Prediction and Adaptation Phase**: For the prediction and adaptation, we use two different constraints and preference models, one preferring lower cost, the other lower duration. For each model, we perform three experimental runs with 200 instances per run. The first run is performed using the default configuration (optimal according to the preferences model) without using the prediction and adaptation framework. In the other two runs, the prediction and (potential) prevention is performed at two different checkpoints. We evaluate for each instance what is predicted and whether the prediction has been correct (“measured”); this is done for the prediction types “No (Adaptation) 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). The results are as follows (Table 2):

- **Duration and Cost of Adaptation**: The prediction and adaptation time together are below a second, thus making it only a performance impact factor for very short running processes. This duration metric and potentially other metrics reflecting the “cost of adaptation” could be modeled as “adaptable metrics” and given a weight in the preferences model. Then they would be taken into account during selection of an adaptation strategy.

- **Prevention Effectiveness**: The KPI performance (column “KPI Evaluation”) has been considerably improved by using our framework (run 2 and 3 outperform run 1). For example, for the first preference model the number of violations (KPI class = “red”) has been reduced from 64 to 49 and 23, respectively.

- **Effects of Preferences Model Settings**: The prevention effectiveness depends on settings in the preferences model and is in our case obviously much better when the preference is set on duration rather than cost. This is because substituted services are not always behaving as expected from their specified metric effects (i.e., not satisfying the corresponding AR predicates). Thus, when choosing services which just so satisfy the AR predicates, the risk of a violation of that AR is higher.

- **Effects of Check Point Positioning**: The later the checkpoint is chosen, the higher the prevention effectiveness as the prediction accuracy is higher. On the other hand, there is an increasing risk that it is too late to adapt ("Too Late" column). Of course, for even better performance, we could predict and adapt at both checkpoints for each process instance.

7. **Related Work**

In the area of process performance monitoring and analysis, most closely related to our approach is iBOM [4] which is a platform for monitoring and analysis of business processes based on machine learning techniques. It supports similar analysis mechanisms as in our approach such as decision trees, but does not deal with adaptation, i.e., extraction of adaptation requirements from the decision trees and derivation of adaptation strategies as in our approach. [18] presents an integrated KPI monitoring and prediction approach which uses machine learning techniques for prediction. It supports not only instance level KPI prediction as in our approach but also time series based prediction across process instances. It however does not deal with adaptation. We do not exploit information on process structure during dependency analysis, as the approach described in [2], but rely on machine learning algorithms to find those dependencies supporting not only numerical but also process data based metrics. [9] deals with prediction of numerical metric values based on artificial neural networks and introduces the concept of a checkpoint used for prediction which we have reused in our approach. Like us, [20] considers SBA layer dependencies for adaptation. While they support functional dependencies as well as the non-functional ones, they model the dependencies at design-time rather than extracting them through an analysis.
[10], [11] also cover the phases monitoring, prediction and adaptation as in our approach focusing on prevention of SLO violations by adapting the process via service substitution and fragment substitution, respectively. The best adaptation strategy is selected by performing a numerical KPI prediction for each adaptation strategy alternative separately and then selecting the one with the best prediction result. Our (analysis and) adaptation approach is different, as we use decision trees which as a white box classification model enable explicit extraction of adaptation requirements and strategies from the classification model. We in addition support adaptation in relation to several KPIs based on specified constraints and preferences.

There are several existing works in the context of QoS-aware service composition [6], [19] which describe how to create service compositions which conform to global and local QoS constraints taking into account process structure when aggregating QoS values of atomic services. We have reused concepts from those works when it comes to the definition of the constraints and preferences model and calculation of QoS scores [6]. Currently, we are simply enumerating all combinations of services when identifying adaptation strategies; that is only feasible for a small number of service types and could be optimized if needed as described, for example, in [6]. Furthermore, these approaches can be used for QoS-based adaptation by replanning the service composition during monitoring [3]. In [1] the PAWS (Processes and Adaptive Web Services) framework is presented which takes into account local and global QoS constraints for selection of Web services at composition runtime. If at runtime a QoS requirement cannot be met, the framework chooses among a set of recovery actions such as retry, substitute, and compensate. Our approach is different in that we do not look at the process structure for prediction or for dependency analysis, but use machine learning techniques instead. This has the advantage that we support also process data-based metrics during analysis in addition to numerical metrics (such as duration and cost) and the approach is extensible towards taking account influential factors which go beyond process flow, e.g., infrastructure-level metrics.

8. Conclusions

In this paper we have presented an integrated monitoring, prediction and adaptation approach for preventing KPI violations in service compositions. At checkpoints, the KPI class of a running process instance is predicted based on the learned KPI dependency tree and metric data gathered for that instance until the checkpoint. In order to prevent KPI violations, adaptation requirements are extracted from the tree and then a set of alternative adaptation strategies is identified which can satisfy those requirements. The identified adaptation strategies are filtered and ranked according to constraints and preferences. The experimental evaluation of the approach has shown that the KPI violations are reduced and that the effectiveness in particular depends on the conformance of metric effect definitions (in the adaptation actions) to the adaptation requirement predicates and the related settings in the constraints and preferences model.

In our future work we will implement additional types of adaptation actions on different applications layers of a service-based application. In that context, one could think of infrastructural reconfigurations on the service layer. We will in particular address the cross-layer aspect by looking at how adaptation actions on different layers influence each other, e.g., a reconfiguration of the infrastructure has an impact on all services and process instances running on that infrastructure. That has to be taken into account during identification and selection of adaptation strategies.

Acknowledgments

The research leading to these results has received funding from the European Community’s 7th Framework Programme under the Network of Excellence S-Cube - Grant Agreement no. 215483.

References


[16] Author, 2010


This article should be cited as:
Abstract—Run-time adaptability is a key feature of dynamic business environments, where the processes need to be constantly refined and restructured to deal with exceptional situations and changing requirements. The execution of such a system results in a set of adapted process variants instantiated on the same process model but dynamically restructured to handle specific contexts. Process evolution exploits the information on process variants to identify the best performing recurring adaptations and adopt them as general solutions in the process model. However, process variants are strictly related to specific execution contexts and cannot be adopted as general solutions. We propose a framework supporting context-aware evolution of business processes based on process instance execution and adaptation history. Instead of looking for recurring adaptations, we propose to look for recurring adaptation needs (i.e., process instances with the same context constraint violation and system configuration). Based on the analysis of adapted instances, we automatically construct and rank corrective evolution variants which can handle the problematic context. At the same time, we try to identify preventative evolution variants by constructing process variants which can prevent the adaptation need. We demonstrate the benefits of our approach using a car logistics scenario.

Keywords—Business Processes, Evolution, Context-awareness

I. INTRODUCTION

Adaptability is recognized as a key feature of dynamic business environments, where operational excellence requires to continuously re-structure business processes to adapt to changes in the execution context [1]. These adaptation needs may be triggered by specific cases to be handled, unexpected situations depending on environmental conditions, or changing requirements.

This need for continuous adaptation results in a system characterized by a huge set of process executions that, although instantiated on the same process model, strongly differ in terms of process structure. In such a dynamic environment, the process models cannot remain unchanged; the short-term adaptations applied to process instances should be used to derive long-term changes in the process models. Providing support for process model evolution is becoming one of the main requirements for managing the lifecycle of dynamic processes [2]. In particular, the set of adapted process instances together with the information concerning their execution should be used as training cases for evolution mechanisms in order to progressively improve process models that are then used to instantiate future process instances.

Most existing approaches addressing this problem (e.g., [3], [4]) derive model-level changes by analyzing frequently occurring changes at the instance-level. In other words, if an instance-level change/adaptation occurs more frequently then a predefined threshold, the change will be propagated at the model-level. These evolution approaches present two major drawbacks. First, an instance-level adaptation variant is not good "in general", it is good for a specific context/situation, and thus cannot simply be propagated to the process model without taking into account the adaptation need it was devised for. Moreover, plugging-in adaptation variants in the original process model is not always a good solution, since it may result in embedding fault-handling activities rather than trying to solve the problem that required runtime adaptation.

To overcome these limitations we present a context-aware evolution framework that, instead of searching for recurring process changes, searches for recurring adaptation needs. At run-time, we may determine that a certain context which at design-time was assumed to occur rarely, actually occurs for a high percentage of the process instances. In this situation, we analyze the instance-level adaptations that have been used for handling the unexpected context. Based on these adaptations, we determine the changes that must be performed on the process model, in order to handle the new context. On the basis of the analysis results, the framework automatically proposes process variants that either embed general corrective solutions derived from context-specific adaptations, or restructure the original model to prevent the recurring adaptation need from occurring.

The enactment of context-aware evolution requires a process modeling notation that on the one hand allows to model contextual information that are relevant to the process execution, and on the other hand allows to easily embed adaptation variants within the process structure. As a reference model for adaptive pervasive systems, we adopt the approach described in the ALLOW project 1, which focuses on modeling and building adaptable pervasive systems. In particular, we model processes as Adaptable Pervasive Flows (APFs) [5], [6], [7], an extension of traditional workflow concepts [8] which make them more suited for adaptation and execution in dynamic environments.

The paper is structured as follows. We describe the mod-

1ALLOW Project, http://www.allow-project.eu/.
eling artifacts in our framework using a car logistics scenario in Section II. In Section III we introduce the evolution framework, focusing on the three main phases of the evolution lifecycle: execution, analysis, and evolution. For each lifecycle phase, we use our car logistics scenario to illustrate the new concepts. We present the related work in Section IV, followed by conclusions and future work in Section V.

II. BACKGROUND

In this section, we describe the modeling artifacts of our evolution framework, as well as their lifecycle (Figure 1).

To illustrate the role of these artifacts, we will use a car logistics scenario. We consider the concrete case of the automobile terminal of Bremerhaven. In our scenario, cars arrive at the seaport and are initially added on stock and stored. Once a delivery order for a certain car is received, the car is removed from stock and prepared according to a treatment list specified in the delivery order. The treatment list may include treatments such as installation of special equipment, paintwork, washing. When the treatment is complete, the car is delivered to the corresponding dealer or directly to the customer.

In a dynamic business environment, the role of the context is fundamental in realizing the adaptation activities [9] as it enables identifying when the process adaptation needed and what should be done. At design-time, along with the definition of a business process the business context is modeled by the Domain Expert as a set of context properties. A context property represents some important characteristic of the environment that can change over time. The Domain Expert model the evolution of a context property with a context property diagram, which is a state transition system. Here states correspond to possible configuration of a property and transitions stand for possible property evolution. Each transition is labeled with an event that characterizes the changes. It is important to note that a context property may evolve as an effect of the business process execution, which corresponds to the "normal" behavior of the domain, but also as a result of volatile - "unexpected" - changes. For this reason we decide to model and manage the context as a separate concern respect to the business processes. For a complete formal notation for context properties please refer to [10]. From the vast domain knowledge in the car logistics domain, we will focus on the handling of cars and their context properties. We consider a subset of these properties: the health status of the surface of the car, the location of the car on the seaport, and whether the car is covered or not. These context properties are displayed in Figure 2. The transitions on events can be guarded. For example, in the carSurfaceHealth the transitions on minorDamage are possible only if the carStatus is in configuration uncovered.

Here, we assume that business processes are modeled using Adaptable Pervasive Flows (APFs) [5], [6], [7]. APFs are business processes, consisting of activities and an execution order specified using control elements. The advantage of using APFs is that they include special modeling elements from the pervasive domain, such as context events and human interaction activities. Further, activities in an APF can be related to context properties, a relation which is realized through preconditions, effects, and context constraints. By using preconditions, we can require context properties to be in certain specified configurations. Context constraints are similar to preconditions, except that they can be defined on a scope consisting of more than one activity. Finally, by using effects, we can specify that the execution of an activity changes certain context properties.

We implement the sequence of steps that the car undergoes on the terminal as a business process (Fig. 3) inspired by [11]. Activities in the process model include unloading the car from the ship, inspecting it for possible transport damages, storing it, applying treatment procedures, and delivering it to the customer.

The activities in the process model are related to the car context properties through a context constraint. Here, we use the notation s*p to denote the fact that the context property p is in configuration s, and e*p the fact that event e has occurred for property p. The constraint then requires that the carSurfaceHealth is in state ok from the moment it is inspected and until it is delivered to the customer. In case the car surface gets damaged while executing one of the activities in the scope of the context constraint, the process instance will be adapted dynamically.

Our process models have associated goals. The goal associated to the car process model is to have the car delivered to the client in perfect condition. If this is not possible because the surface of the car got damaged and we cannot repair it on site, we at least want the car to be brought to a disposal area, from which it can be returned to the manufacturer or disposed:

- primary goal: ok*(sh) ∧ atDestination*(loc)
- recovery goal: unrepairableLocally*(sh) ∧ disposalArea*(loc)

This intuitive description can be formally modeled using the language for expressing goals with preferences introduced in [10].

A third artifact in our application is a list of Key Performance Indicators (KPIs), which are defined at design-time by a Business Analyst. These KPIs can be used to evaluate the performance of process instances.

In our scenario, we consider the following list of KPIs:

- KPI: Percentage of cars delivered on time
  Measurement: We consider all the car process instances for which the execution log contains an expected delivery time (if there are multiple values, then we consider the last), as well as the time the execution of the process instance was finished. On these process instances, we compute the percentage for which the finishing time was before or at the expected delivery time.
  Target: Above 95%
- KPI: Waiting time at technical treatment stations
  Measurement: Total time spent by a vehicle in queues, parking spaces, or service car ramps, while waiting to be
processed.
Target: Reduce the waiting time at technical treatment stations to at most 10 hours per vehicle.
- KPI: Average number of on-site surface damages
  Measurement: For each car process instance, we count the number of times the events \texttt{minorDamage} and \texttt{severeDamage} occur in the associated \texttt{carSurfaceHealth} property. The damages are considered to be on-site only if they occur after executing the \texttt{Check Car} activity. Target: Below 0.003
- KPI: Percentage of repaired car surface damages
  Measurement: We consider all the process instances for which \texttt{carSurfaceHealth} was logged during the execution of the instance as being in one of the damaged configurations. We measure the percentage for which the \texttt{carSurfaceHealth} was in configuration \texttt{ok} upon completion of the instance execution (i.e., a car surface is considered repaired only if it was \texttt{ok} at delivery time). Target: Above 95%

At run-time, both process models and context properties are instantiated. Process instances are then executed and possibly adapted. The decision to adapt a process instance is based on contextual information provided by the instantiated context properties, which are continuously monitored. For example, process adaptation can be initiated when the observed values of the context properties violate some constraint.

The set of adapted process instances together with the information concerning their success and their execution context will be used as training cases for the \textit{Performance Analysis} in order to progressively improve the process models with respect to the KPIs. The results of this analysis can be used to generate a set of evolution variants which optimize the existing process model with respect to the KPIs. The evolution variants are then proposed to the process designer. In case the process designer decides to evolve the process model, all new process instances will be based on the evolved model.

III. Evolution Framework

While in previous Section we introduced the modeling artifacts for evolvable process-based applications, in the following we present in details the proposed framework for context-aware evolution. Figure 4 shows an architectural overview of the framework presenting the relations between the different components and positioning them with respect to the three main phases of the evolution lifecycle, namely (1) execution phase, (2) analysis phase, and (3) evolution phase.

During the first phase, the framework is responsible for managing the execution and adaptation of the system and for logging all the information that may be useful to the other phases (e.g., execution traces, adaptation needs, adaptation variants, execution performances). During the analysis phase, the framework controls and evaluates the quality of execution
of the processes with respect to the KPIs, decides the need for evolution for a certain process model, and, on the basis of the execution history, identifies the contextual evolution problem in terms of recurring system configuration that required adaptation. Finally, in the evolution phase, the framework uses the information obtained from the analysis phase to compute process model variants that either embed the best performing (with respect to KPIs) adaptation variants or prevent the violation of the context constraint. These evolution variants are then presented to the process designer, who decides whether they should be adopted for future executions. The process designer obtains the evolved process model using a set of supporting tools.

In the following, we present in detail the different phases in the evolution lifecycle.

A. Execution phase

During the execution phase, process models are instantiated and the resulting process instances are executed on the Process Engine. The Execution Manager is responsible for keeping the system configuration up to date and for consistently aligning the status of context properties to the execution of the processes and to the context events received by the Context Manager.

The system configuration is used by the Execution Manager to monitor context constraints associated to running process instances and to trigger adaptation in case of violation. The Adaptation Manager receives in input the description of the adaptation problem (i.e., process instance to be adapted, system configuration, and constraint violation) and computes an adapted process instance that deals with the constraint violation (i.e., bringing the process to a “stable” state that does not violate the constraint, skipping/replacing some original process activities to deal with the context, compensating and terminating the process in case there is no means to deal with the constraint violation).

The information about the system execution and adaptation is recorded in the Execution Log. Examples of stored information are the traces of process instances and of context properties execution and the adaptation history in terms of adaptation problems and adopted adaptation variant.

Consider for example that while executing instances of the process model in Figure 3, the context constraint \( \text{ok}^\text{sh}(\text{sh}) \) can be violated for some of them. For each of these violations the context configuration is given by the state of the context property diagrams, for example: \( \text{carSurfaceHealth} \) in state \( \text{lightlyDamaged} \), \( \text{carLocation} \) in state \( \text{storageArea} \), and \( \text{carStatus} \) in state \( \text{uncovered} \). For each problematic process instance, the Adaptation Manager, considering the precise context configuration, can provide different adaptation solutions as the following:

- **Eager repair** (Figure 5a). In this case, the runtime adaptation is to first block the execution of the Receive delivery order activity. We bring the car to the treatment area and repair the damage (problem fixing part), and then return to the storage area (redo part). Finally, we resume waiting for the delivery order.
- **Lazy repair** (Figure 6a). In this case, the adaptation is to complete the execution of the Receive delivery order and add surface damage repair to the list of treatments...
to be performed. We can then proceed to the treatment area and perform the treatments on the list. Here, note the different context constraint. The idea is that until the car is repaired, we need to allow the carSurfaceHealth to be also in the state \textit{lightlyDamaged}, but we also need to ensure that it is not \textit{unrepairableLocally}.

\section*{B. Analysis phase}

The Process Analyzer is responsible for detecting the need for evolution and for identifying the contextual evolution problem.

In particular, the evolution need is triggered by a problem in the system performance with respect to the KPIs of the process models. The evolution need is identified by the Process Analyzer through the analysis of the information stored in the Execution Log. According to the specific techniques used by the Analyzer, we can distinguish between proactive and reactive analysis. In proactive analysis, the Analyzer predicts future KPI violations (see [12]) and evolution techniques can be applied to prevent these violations. Reactive analysis detects the fact that the KPI has already been violated and evolution techniques can be used to avoid future violations.

Given the specific evolution need (KPI violation for a certain process model), the Analyzer considers all the process instances that contributed to the KPI violation and looks for recurring adaptation needs examining their adaptation history in the Execution Log. The retrieved information is used to identify the specific \textit{evolution problem} which is characterized by the process model to be evolved, the critical KPI, the recurring adaptation need (system configuration and constraint violation), and the history of all adapted process instances.

Consider for example the following evolution need: the violation of the KPI “Average number of on-site damages” for the car process model. The process instances which contributed to the KPI violation are all the instances for which the corresponding carSurfaceHealth reached one of the configurations \textit{lightlyDamaged} or \textit{unrepairableLocally} during the execution of the scope of the context constraint \textit{ok*(sh)}.

The recurring adaptation need consists of the violation of the context constraint \textit{ok*(sh)} and the following context configuration: carSurfaceHealth in \textit{lightlyDamaged}, carLocation in \textit{storageArea}, and carStatus in \textit{uncovered}. The evolution problem will consist of the car process model, the KPI “Average number of on-site damages”, the recurring adaptation need, and a history of all adapted process instances which had the same adaptation need. In our example, this is a subset of the process instances which contributed to the KPI violation.

In general, the adapted instances included in the evolution problem can be also instances which do not contribute to the KPI violation. Such instances are particularly useful in the evolution, since they address the adaptation need and also meet the target for the KPI.

\section*{C. Evolution phase}

The aim of this phase is to propose a set of process variants that deal with the identified evolution problem and to support
For what concerns the proposal of evolution variants, we distinguish two different approaches: corrective evolution and preventive evolution.

The aim of corrective evolution is to propose a set of adaptation variants that, according to the execution history, can deal with the recurring adaptation need and exhibit good performances. The Evolution Manager looks for adaptation variants that solve the same adaptation problem and that have good performances with respect to the KPI. The identified adaptation variants are then ranked according to execution performances (e.g., performances with respect to the other KPIs and confidence) and proposed as evolution variants to be plugged-in in the original process model.

As it can be seen also from the examples, the correspondence between instance-level adaptation variants and evolution variants is not immediate. The challenge here is to understand which of the built-in adaptation tools can be used and how it can be used in order to capture the effects of the instance-level adaptation, and limit these effects to the problematic context.

The evolution enactment providing facilities for embedding the variants within the original flow model.

The adaptation variants can have different performance results. For example, it can be the case that the “Eager repair” strategy performs better then the “Lazy repair” with respect to the KPI “Percentage of cars delivered on time”, since it avoids the delay caused by the extra treatment. However, the “Lazy repair” may perform better with respect to the KPI “Waiting time at technical treatment stations”, since the treatment tasks can be rescheduled considering the availability of the treatment stations. We use these performance results of the adaptation variants, as well as a confidence parameter (the number of instances), to rank the evolution variants.

Preventive evolution proposes evolution variants that prevent the recurring adaptation need from being triggered. This approach is more disruptive since it requires a restructuring of the original process model such that its executions guarantee that the system configuration requiring adaptation is avoided. The idea here is to apply AI planning techniques similar to that of [13], which, given the process model, the context properties, the critical system configuration, and the process goal (in terms of control flow requirements on system configurations) look for a re-planned process model whose traces never reach the critical configuration and yet guarantee to satisfy the process goal. The evolution variant resulting from preventive evolution cannot be analyzed in terms of performances since it consists of a new process model for which no information is available in the logs.

In our example, we can have an activity Apply Cover which requires as precondition: uncovered*(cs) ∧ storageArea*(loc) (the car covers are available only in the storage area), and triggers the effect: cover*(cs). Similarly, we have also an activity Remove Cover. Then, by replanning the
process model, we will find that the process model displayed in Figure 7 satisfies the process goal and prevents the adaptation need.

The Evolution Enactment module is responsible for obtaining the evolved process model by choosing the evolution variants to be adopted for future executions and for embedding them within the existing process model.

In most application domains thinking of completely automating this phase is unrealistic. As a matter of fact, modifying the process models is often critical and understanding the goodness and impact of the process variants on the system requires a comprehensive knowledge of the domain.

In our framework we adopt a man-in-the-loop approach, where the Evolution Enactment module presents the evolution variants to the process designer and provides her with a set of tools that support the definition of the evolved process. In particular, we exploit the built-in adaptation tools defined in [7], which allow us to embed context-specific adaptation variants within a business process.

In our example, the process designer may find that switching completely to the preventive evolution variant is too costly, since it involves more human resources than are currently available. For regular cars, the best performing correcting evolution variant is sufficient. However, changing to the preventive evolution variant is reasonable in the case of luxury cars, since these are comparably few, and the costs for repairing even minor damages is very high. The process designer can then use the Contextual If built-in adaptation tool to text for luxury cars and integrate the two evolution variants.

IV. RELATED WORK

The problem of supporting the evolution of business processes models has been addressed by several works. Some of them are able to capture a precise sets of contexts and to use for each of them a predefined process variants [14], [15], others support the evolution of processes by analysing previous executions and adaptations [3], [4], [16], [17], [18].

Most existing approaches focus on the problem of extracting useful information from the adaptation logs. The approaches differ both in what they log and in the techniques that they use to analyze the logs (e.g., [3], [4], [17], [18], [19]). These approaches may be used in the analysis phase of the proposed evolution framework.

One direction is to use process mining techniques, as in [4], considering large collections of structurally different process variants created from the same process model. The authors use a heuristic search to find a new process model such that the weighted average distance between the new model and the variants is minimal.

Also using mining techniques to analyze change logs is the approach in [17]. The evolution result of this approach is an abstract change process consisting of change operations and causal relations between them. These change processes can be used as an analysis tool to understand when and why changes were necessary.

A different direction is that of [3]. Here, concepts and methods from case-based reasoning (CBR) are used in order to log, together with the change operations, also the reasons for and context of each change. Change information is stored as cases in a case-base specific to the process model. The case-bases are used to support process actors in reusing information about
similar ad-hoc changes, and are also continuously monitored to automatically derive suggestions for process model changes.

Change analysis and reuse is also relevant for loosely specified process models. [18] facilitates change reuse by providing a search interface for the repository of process variants. For declarative processes, [19] supports users through recommendations, which are generated based on similar past executions and considering certain optimization goals.

The approaches in [3], [4] are closest to our work, since they also generate new process models based on the information from the adaptation logs. A limitation of the approach in [4] is that it does not consider the context requiring adaptation. An instance-level adaptation may be useful only for a particular context, and therefore should not be included in the process model even if it occurs relatively often. Although [3] considers also the context of changes, this is specified as natural language question-answer facts. These are useful for supporting process actors in identifying the existing cases with the same context. They are also useful for the process engineer, who can manually determine the context for a new process model change. However, these tasks cannot be done automatically.

Further, in both [3], [4] the decision whether to integrate a change operation in the process model is determined by the frequency of the change operation. However, it can be the case that the need for adaptation was a fault in the process, and the adaptation contains fault handling and compensating activities. In these cases, rather than including the adaptation in the process model, we may be interested in avoiding the adaptation need altogether.

In contrast to [3], [4], we use the context as the main driver for evolution. This allows us to determine if an instance-level adaptation is useful for a particular context, and it also helps to search for an alternative process model which avoids a problematic context.

The importance of context for improving process models is recognized also in [16]. Here, the authors propose a context-aware process management cycle, with context-awareness spanning all the stages of the process lifecycle. While [16] remains at a very general and abstract level, we are taking the approach one step further, and provide concrete ideas for implementing the context-aware process management cycle.

V. CONCLUSIONS AND FUTURE WORK

We have presented a framework for evolving process models based on a history of process instance executions and adaptations. Our approach is context-driven. If the need to evolve the process model is detected, we analyze the relevant adapted process instances and look for recurring adaptation needs (i.e., the same constraint violation and system configuration). This allows us to construct and rank evolution variants which can handle the problematic context (corrective evolution). It also allows us to construct evolution variants which can prevent the adaptation need (preventive evolution).

In our future work, we will develop concrete solutions for corrective and preventive evolution. For corrective evolution we plan to develop techniques to automatically transform instance-level adaptation variants into evolution variants using the built-in adaptation tools. For preventive evolution we will apply AI planning techniques to re-plan the process model in order to avoid the critical configurations. We will implement and evaluate our solutions on realistic scenarios, such as the car logistic scenario introduced in this work.

Acknowledgements.

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube) and from the FP7 EU-FET project 213339 ALLOW.

REFERENCES


CAptEvo: Context-aware Adaptation and Evolution of Business Processes *

Antonio Bucchiarone, Annapaola Marconi, Marco Pistore, and Heorhi Raik

FBK-Irst, via Sommarive 18, 38123, Trento, Italy
[bucchiarone,marconi,pistore,raik]@fbk.eu

Abstract. CAptEvo is a framework for the adaptation and evolution of service-based business processes operating in dynamic execution environments. In this demonstration, we apply the CAptEvo to a case study from the logistics domain and show its advantages in handling highly complex dynamic real-world business applications.

1 The CAptEvo Framework

Adaptability is a key problem in dynamic business environments, where operational excellence requires to model and execute business processes taking into account a dynamic, open and non-deterministic execution context. These adaptation needs may be triggered by specific execution cases, dynamic service availability, non-controllable situations depending on environmental conditions, or changing requirements. Moreover, this need for continuous adaptation results in a system characterized by a huge set of process executions that, although instantiated on the same process model, strongly differ in terms of process structure. In such a dynamic environment, the short-term adaptations applied to process instances should be used to derive long-term changes that progressively improve the process models.

The CAptEvo Framework, developed within the ASTRO project ¹, integrates sophisticated techniques for managing the execution, adaptation, and evolution of context-aware business processes. The framework exploits a modeling approach for Service Based Applications where adaptability and context-awareness are key embedded characteristics of the business application [2, 3]. During the execution phase, process models are instantiated and the corresponding process instances are executed. The Execution Manager is responsible for keeping the system configuration up to date and for consistently aligning the status of context properties to the execution of the processes and to the context events. The system configuration is used by the Execution Manager to monitor context constraints associated to running process instances and to trigger adaptation in case of violation. The Adaptation Manager supports two types of AI-planning based dynamic adaptation: vertical and horizontal. The aim of vertical adaptation is to refine abstract activities of a process by automatically composing available services and obtaining a concrete process that can be executed. Horizontal adaptation results in a structural modification of the process instance by adding, changing or removing process activities to retain the reachability of its original goals in case of a changing environment. The information about the system execution and adaptation is recorded in the Execution Log. Examples of stored information are the traces of process instances and of context properties execution and the adaptation history in terms of adaptation problems and adopted adaptation variant. The set of adapted process instances together with the information concerning their execution are used as training cases

⭐ The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (Network of Excellence S-Cube)

¹ www.astroproject.org
for evolution mechanisms in order to progressively improve process models that are then used to instantiate future process instances. The evolution need is triggered by a problem in the system performance with respect to the KPIs of the process models. Given the specific evolution need (KPI violation for a certain process model [5]), the Analyzer considers all the process instances that contributed to the KPI violation and looks for recurring adaptation needs examining their adaptation history. The Evolution Manager looks for adaptation variants that solve the same adaptation problem and that have good performance with respect to the KPI. The identified adaptation variants are then ranked according to execution performance (e.g., performance with respect to the other KPIs and confidence) and proposed as evolution variants to be plugged-in in the original process model. In our framework we adopt a man-in-the-loop approach, where evolution variants, together with their performance, are presented to the process designer that, through a set of built-in adaptation tools [4], can embed them in the evolved process.

2 Demonstration

To demonstrate the CAptEvo framework in action, we use a real-world scenario from the domain of logistics [1]. The scenario is based on business processes used in the terminal of the Bremerhaven sea port, where cars arriving by ship have to be delivered to retailers. Before the cars can be delivered, a series of activities needs to be completed such as customization procedures, car shipment and repair, etc. The management of car delivery is a highly complex process, as each car requires an individual treatment, and the process execution might be affected by changes in the execution context such as car damages. This requires sophisticated modeling that allows for run-time adaptation, and evolution of the application. In our demonstration, we illustrate the CAptEvo framework in action and present the outcome of our algorithms to the end users. We have created a visualization environment enabling interaction between the framework and the user and simulating execution, adaptation and evolution of business processes in our case study. In particular, it can:

- Run the reference “Car Logistics” scenario and simulate the execution and adaptation of each business process attached to each car.
- View the different adaptation strategies supported by our framework (i.e., vertical and horizontal) and how they are used during the scenario execution.
- Inspect the behavior of the system in terms of process performance and adaptation history.
- View the process evolution results and choose the process variants to be embedded in the system.

The goal of this demonstration is to show the novel concepts and advantages of the CAptEvo Framework when applied to a real and complex pervasive system.

References

Dynamic Composition of
Pervasive Process Fragments

Adina Sirbu, Annapaola Marconi, Marco Pistore
Fondazione Bruno Kessler
via Sommarive 18, 38123 Trento, Italy
[sirbu,marconi,pistore]@fbk.eu

Hanna Eberle, Frank Leymann, Tobias Unger
Institute of Architecture of Application Systems
Universitätsstraße 38, 70569 Stuttgart, Germany
[eberle,leymann,unger]@iaas.uni-stuttgart.de

Abstract—A critical aspect for pervasive computing is the possibility to discover and use process knowledge at run time depending on the specific context. This can be achieved by using an underlying service-based application and exploiting its features in terms of dynamic service discovery, selection, and composition. Pervasive process fragments represent a service-based tool that allows to model incomplete and contextual knowledge. We provide a solution to automatically compose such fragments into complete processes, according to a specific context and specific goals. We compute the solution by encoding process knowledge, domain knowledge and goals into an AI planning problem. We evaluate our approach on different scenarios stress testing the main characteristics of pervasive process fragments.

I. INTRODUCTION

Service-based applications are a promising solution for the dynamic and heterogeneous domain of pervasive computing. Service-based applications are compositions of software services, possibly offered by different third-party organizations. A pervasive computing environment is a physical environment populated by small networked processing devices that interact with users. In such a setting, it is critical to allow for processes to be discovered at runtime and to be used depending on the context (e.g., depending on location, time, situation, people). This can be achieved with an underlying service-based application, since the software services can be discovered, selected and composed dynamically, while the application is running.

An approach going in this direction is that of process fragments [1]. Process fragments represent a tool for modeling incomplete and local process knowledge. The knowledge is incomplete since the modeler is allowed to specify just one aspect of the entire process, and even to leave gaps in the process specification. Fragments can be modeled by different people, and therefore may reflect different perspectives on the same process. Further, the process knowledge is local, since the availability and usability of a fragment is determined by context. For example, the execution of a process fragment may be bound to a certain location.

Process fragment knowledge can be integrated dynamically at design time or at run time. This requires enriching processes with goals which specify what is pursued by the process execution. It also requires enriching fragments with information on how they contribute to the outcome of the process.

We apply the concept of process fragments to the concrete case of adaptable pervasive flows, introduced within the European project ALLOW (http://www.allow-project.eu). An adaptable pervasive flow is a dynamic workflow modeling the behavior and goals of a physical entity, such as an artifact or a person. Flows are context-aware: during their execution, they can sense the environment related to their entity. For this purpose, flows include special modeling elements explicitly representing context information. To distinguish the fragments of adaptable pervasive flows from process fragments, we will refer to the former as pervasive process fragments (or simply, fragments). The fact that they represent incomplete and local process knowledge which can be integrated dynamically at runtime, coupled with the fact that they are context-aware, makes pervasive process fragments a particularly suitable framework for pervasive applications.

Consider the following scenario inspired by airport checking and taxing procedures (Fig. 1). In this scenario, boxes containing goods arrive at the airport and go through customs before being released to their owners. Boxes will be treated and taxed differently depending on country of origin, content, etc. If a box contains forbidden items it will not be released to its owner, and will instead be disposed by authorized personnel.

The process knowledge for handling a box is distributed at different locations in the airport and depends on context information. Therefore, the complete flow model for treating a particular box is not known from the beginning. What is known is the goal of the flow: to release the box to its owner. The precise flow model that can achieve this goal is created at execution time, based on the available fragments.

We propose an approach for composing pervasive process fragments according to complex goals. Our approach is based on previous results from Web service composition [2], and in...
particular on the work of [3], which addressed the problem of Web service composition with complex composition requirements. The problem of fragment composition is conceptually different from Web service composition, since the components are not orchestrated, but integrated into a complete flow model. Due to this difference, important issues in Web service composition, such as the need for Web services to communicate via message exchange, do not appear in fragment composition. On the other hand, fragment composition introduces new problems. For example, the fragments can overlap, or can have gaps in the specification. To deal with these differences, we have significantly modified and extended the approach in [3].

Our approach is based on a three-layer representation, where the first layer captures the specific domain knowledge required for the composition task, the second layer describes the abstract flow in terms of the goals it should achieve, and the third layer is the concrete, context-dependent part of the flow definition, represented using fragments.

The paper is structured as follows. In Section II, we describe our three-layer representation and the elements involved: entities, goals, and fragments. We describe a fragment composition technique based on planning in Section III, and evaluate it in Section IV. We position our work in a broader context in Section V and discuss future work in Section VI.

II. APPLICATION REPRESENTATION

For modeling an application, we consider that there exist three layers, arising from two distinctions (Fig. 2). First, we distinguish between domain knowledge, or knowledge about the entities in the domain, and process knowledge, or knowledge about business logic. Second, we distinguish between concrete, context-dependent knowledge and knowledge that is common, abstract, independent of context. The first layer is thus the domain knowledge, which is stable and abstract. The second layer is the stable part of the process knowledge, represented using goals, while the third is the dynamic and concrete part, represented using pervasive process fragments.

The domain knowledge consists of the types of entities in the domain. An entity type includes a set of properties (e.g., position, content) and a set of events that represent changes of these properties (e.g., if the content is checked it may be approved or rejected). Properties are states that last for a period of time, while events are actions that are in effect at a certain point and make the entity evolve from one state to another.

We use the domain knowledge to define our goals. A goal can be used to model a flow or a part of a flow for which the exact content is not known at design-time. The exact content may remain unknown until the flow instance is executed. At that point, the goal will be substituted with a concrete realization, by composing available fragments. Using goals, we can specify the target state for our flow, as well as coordination requirements. A target state is a situation we want to achieve at the end of the flow execution, whereas a coordination requirement is a property we want to ensure during the entire execution.

To model entities and goals, we borrow the formalization proposed in [3]. In particular, we use object diagrams to represent entities, and a simple language for expressing goals with preferences for representing goals. Despite the different application domain, this formalization perfectly captures the core characteristics of our contextual entities and goals.

We use the domain knowledge also for annotating pervasive process fragments. We encode inside fragments the relation to one or more entities, and mark the points when the execution of the fragment would trigger also the evolution of the entities. Fragments consist of activities and control elements. To relate fragments to entities we use preconditions and effects, which can be defined on certain activities in the fragment. Preconditions are properties that have to hold for the activity to be applied in the composition. Effects are properties that are made true by applying the corresponding activity.

In this representation, both object diagrams and goals are specified independently from fragments. Therefore, using the same object diagrams and goals, we can achieve different fragment compositions for different contexts.

A. Object diagrams

An object diagram is a simple state transition system containing states which encode properties of an entity, and transitions between states triggered by events.

Definition 1 (Object Diagram): An object diagram representing an entity $E$ is a tuple $(L, L_0, E, T)$, where:

- $L$ is a finite set of configurations and $L_0 \subseteq L$ is a set of initial configurations,
- $E$ is a set of events,
- $T \subseteq L \times E \times L$ is a transition relation that defines the evolution of the entity, based on events.

Fig. 3 displays the object diagrams in our scenario. Note that the tax invoice includes also the creation of the entity.

B. Goals

We express goals in terms of entities and their evolution. Goals can be used to specify desirable situations to be reached at the end of the composition, as well as rules to be maintained throughout the composition.

Definition 2 (Goal): A goal is defined with the generic constraint template $\varphi \Rightarrow (\varphi_1 \cdots \varphi_n)$, where

\[
\varphi \equiv T \mid s^t(o) \mid e^t(o) \mid \varphi \lor \varphi \mid \varphi \land \varphi.
\]
Fig. 3: Object diagrams

Here, $s^o(o)$ defines the fact that diagram $o$ is in configuration $s$, and $e^o(e)$ the fact that event $e$ of $o$ has taken place.

If the left side of the requirement is empty ($\top$), the rule specifies the need to unconditionally reach the state defined by the right side. Otherwise, the rule specifies that whenever the state in the left side occurs, the composition should try to reach the state defined in the right side. In both cases, the states on the right side are ordered using a preference operator ($\succ$), from the most preferred to the least preferred.

In our scenario, the goal is for box to reach the configuration READY. If this is not possible, we at least want to have the box disposed of, therefore in configuration DISPOSED:

$$\top \implies ready^o(box) \succ disposed^o(box) \quad (G_1)$$

C. Pervasive process fragments

Pervasive process fragments are the result of applying the process fragment definition from [1] to adaptable pervasive flows. They can be integrated into complete flow models by means of composition, and as such can be seen as the building blocks of flow models. In the following, we give a brief overview of adaptable pervasive flows and process fragments.

Adaptable pervasive flows are similar to the well-known workflows. They consist of activities and a corresponding execution order specified using control elements such as sequence, choice, parallel operators. Flows have associated constraints and address specific goals. We call flow instance a particular execution of a flow model. For describing flows we use a specialized language called Adaptable Pervasive Flow Language (APFL). The nucleus of APFL is BPEL [4], which has been extended to cover issues from the pervasive domain. APFL includes standard BPEL basic and structured activities (e.g., receive, reply, invoke, control constructs). It also includes APFL-specific activities defined as BPEL extensions (e.g., human interaction activity, context event). Further details on adaptable pervasive flows and APFL can be found in [5], [6].

Process fragments are a modeling approach which allows to model incomplete process knowledge. Here, there are three options. First, in a process fragment it is possible to have control connectors to have either no source or no target activity. Second, the modeler has the freedom to not model control connectors at all. Third, process fragments can contain gaps, modeled using a special element called Region. A region helps to define an ordering between activities, when it is not clear what needs to happen in between.

Fig. 4: Annotated fragments available for composition

The pervasive process fragments incorporate a subset of the properties of process fragments. We exemplify these properties on the fragments in our scenario, displayed in Fig. 4. First, fragments are not required to have a start activity, which must otherwise be included in a complete flow model. This is in fact the case for all the fragments in our scenario. Second, the pick connectors may be left dangling. A pick may contain several onMessage branches, but these branches are not necessarily followed by an activity, like in the case of the pick in the fragment Check box origin.

Finally, fragments can also contain regions. Regions can appear anywhere within a fragment, except on a parallel branch. Consider the region in the fragment Check box content. Here, the modeler knew that in case the content of the box is approved (Content ack is executed), the box will be taxed (Charge tax) and finally marked (Mark box), but did not know the steps in between, concerning the taxing procedure.

D. Relating pervasive process fragments to object diagrams

We extend the fragment definition with relations to object diagrams: preconditions and effects. These can be attached to basic activities and to message handlers (onMessage, onHumanInteraction).

The preconditions (denoted with $P$) are propositional formulas over the set of propositions $\{s_i^o(o_i)\}$, where $o_i$ are
object diagrams and $s_j$ configurations. An activity annotated with a precondition requires the diagrams to be in particular configurations. If the precondition does not hold, the activity cannot be applied in the composition. For example, in the fragment Check box origin, activity Check EU origin requires box to be in configuration UNLOADED.

The effects (denoted with $E$) are sets of propositions from \( \{ e_i^s(o_i) \} \), where $o_i$ are object diagrams and $e_i$ events. An activity annotated with effects encodes the fact that diagrams may move to different configurations as a result of executing the activity. In the fragment Check box content, activity Content nack triggers the event reject on the diagram box. This can happen only if in the current configuration of box there exists a transition on reject, i.e., if box is in configuration UNLOADED. For readability, we have made this condition explicit using the precondition. However, such conditions can also be left implicit. From now on, by preconditions we will refer to the conjunction of explicit and implicit conditions.

E. Overlapping activities

A key issue about pervasive process fragments is that they can include overlapping activities. The reason is that fragment models have only a local view of the entire flow and may therefore model the same information.

Informally, two activities are overlapping if and only if there exists at least one object diagram for which they have the same effects, and their preconditions and effects are consistent. Two preconditions are consistent if they do not require any diagram to be in different configurations. Two effects are consistent if they do not trigger different transitions in the same diagram.

We introduce a helper formula $\text{Xor}$. For an object diagram $o = (L, L_0, \varepsilon, T)$, $\text{Xor}(o)$ states that $o$ can be in exactly one configuration at a time:

$$\text{Xor}(o) = (\bigvee_{i \in L} s_i^o(o)) \land \bigwedge_{i \in L, s_i \neq s_j} (\neg s_i^o(o) \lor \neg s_j^o(o))$$

Let $a_1$ be an activity with preconditions $P_1$ and effects $E_1$ defined on a set of object diagrams $o_1, \ldots, o_{k_1}, o_{k_1+1}, \ldots, o_m$. Let $a_2$ be a second activity with preconditions $P_2$ and effects $E_2$ defined on $o_1, \ldots, o_{k_1}, o_{k_1+1}, \ldots, o_m$. We say that:

- $P_1$ and $P_2$ are consistent iff $P_1 \land P_2 \land \bigwedge_{1 \leq i \leq k} \text{Xor}(o_i)$ is satisfiable;
- $E_1$ and $E_2$ are consistent iff for all $o_i \in \{o_1, \ldots, o_k\}$, $o_i = (L, L_0, \varepsilon, T)$, $E(o_i) = \{e^\varepsilon(a_i) \mid \forall \varepsilon \in \varepsilon\}$, we have $E_1 \cap E(o_i) = E_2 \cap E(o_i)$.

Further, $a_1$ and $a_2$ are overlapping if:

- $k \geq 1$;
- $P_1$ and $P_2$, respectively $E_1$ and $E_2$, are consistent;
- there exists $o_i \in \{o_1, \ldots, o_k\}$, $o_i = (L, L_0, \varepsilon, T)$, $E(o_i) = \{e^\varepsilon(a_i) \mid \forall \varepsilon \in \varepsilon\}$, such that $E_1 \cap E(o_i) \neq \emptyset$.

Consider the activities Charge tax and Charge tax with invoice in the second and third fragment from Fig. 4. These activities are overlapping, even though they do not have the same name, preconditions, or effects. The activities have in common the object diagram box, and the consistency requirements are satisfied, since they both require box to be in configuration EVALUATE and trigger the same event tax.

III. Solution

The fragment composition problem can be stated as follows. Given a set of pervasive process fragments, a set of object diagrams, and a set of composition goals, the problem is to integrate a subset of the pervasive process fragments into an adaptable pervasive flow that achieves the goals.

Our solution is presented schematically in Fig. 5. First, we encode fragments, object diagrams and goals into a planning domain $\Sigma$ (steps 1-4). We then create the planning goal $\rho$ based on the goals given as input (step 5). On the domain $\Sigma$ and goal $\rho$ we apply the approach presented in [2], which generates a controller $\Sigma_c$ for controlling the domain $\Sigma$ in such a way as to satisfy the goal $\rho$ (step 6). In difference to [2], we are not interested to retrieve the controller $\Sigma_c$, but to analyze the controlled domain. If $\Sigma_c$ exists, then the controlled domain can be used to generate a synthesis of the fragments which achieves the composition goals (step 7).

The planning domain $\Sigma$ is defined as a state transition system (STS). An STS contains a set of states, of which some are marked as initial and/or final. Each state is labeled with sets of properties that hold in that state. The STS can evolve to new states as a result of performing actions. The actions can be either input (controllable) or output (not controllable).

**Definition 3 (STS):** Let $P$ be a set of proposition symbols and $\text{Bool}(P)$ the set of boolean expressions over $P$. A state transition system is a tuple $(\mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{S}^F, \mathcal{F})$, where:

- $\mathcal{S}$ is the set of states and $\mathcal{S}^0 \subseteq \mathcal{S}$ the set of initial states;
- $\mathcal{I}$ and $\mathcal{O}$ are the input and respectively output actions;
- $\mathcal{R} \subseteq \mathcal{S} \times \text{Bool}(P) \times (\mathcal{I} \cup \mathcal{O}) \times \mathcal{S}$ is the transition relation;
- $\mathcal{S}^F \subseteq \mathcal{S}$ is the set of accepting states;
- $\mathcal{F} : \mathcal{S} \rightarrow 2^P$ is the labeling function.

In our planning domain $\Sigma$, the set $\mathcal{P}$ consists of all propositions $\{s_j^o(o_i)\}$, encoding the fact that the diagram $o_i$ is in configuration $s_j$. The labeling function $\mathcal{F}$ determines whether a boolean expression $b \in \text{Bool}(P)$ holds in a particular state $s$. We write $s, \mathcal{F} \models b$ to denote that $b$ is satisfied at state $s$ given $\mathcal{F}$. Satisfiability of a formula is determined according to the following standard inductive rules:
• \(s, F \models T\);
• \(s, F \models p\), iff \(p \in F(s)\);
• \(s, F \models \lnot b\), iff \(s, F \not\models b\);
• \(s, F \models b_1 \lor b_2\), iff \(s, F \models b_1\) or \(s, F \models b_2\).

The transitions in the STS are guarded: a transition \((s, b, a, s')\) is possible in state \(s\) only if the guard expression \(b\) is satisfied in that state, i.e., if \(s, F \models b\).

To create our planning domain, we first transform fragments, object diagrams and goals into STSs. The planning domain will then be the parallel product of these STSs, capturing their simultaneous evolution. After building the planning domain and the planning goal, we determine if a controller exists such that the controlled domain satisfies the goal. We use the following notion of a controlled system.

**Definition 4 (Controlled System):** Let \(\Sigma = (S, S_0, I, O, R, S^F, F)\) and \(\Sigma_c = (S_c, S_0^c, I, O, R_c, S_c^F, F_c)\) be two STSs. STS \(\Sigma_c \supset \Sigma\), describing the behaviors of system \(\Sigma\) when controlled by \(\Sigma_c\), is defined as follows:

\[\Sigma_c \supset \Sigma = (S_c \times S, S_0^c \times S_0, I, O, R_c \supset R, S_c^F \times S^F, F_c \cup F)\]

where: \((s_c, s), (b, a \& b), (a, (s', s')) \in (R_c \supset R)\) if \((s_c, b, a, s') \in R_c\) and \((s, b, a, s') \in R\).

### A. Transforming the pervasive process fragments

To transform a fragment into an STS, we recursively translate its activities. Activity preconditions are copied as transition guards in the STS. We encode activities using input and output actions. Input actions correspond to activities that can be controlled by the composition, such as receive, reply, invoke, contextEvent. Output actions correspond to activities that cannot be controlled by the composition, such as the onMessage within a pick. The idea is that once a pick is selected for composition, all its onMessage branches must be included. In this sense, it is not under the control of the composition whether to include the activity onMessage or not.\(^1\)

Table I contains the APFL elements and their translation to STS. To differentiate between input and output actions, we prepend the names with “?”, respectively “!”. We denote the start state of an activity with \(s_0\), and the end states with \(s_e\) and \(s'_e\). With \(s_0 \xrightarrow{a} s_e\) we denote the recursive translation of activity \(a\). Finally, we denote the boolean formula corresponding to the precondition of an activity with \(b\).

With this translation, we lose the information encoded in the effects of activities. We capture this information with a data structure called Action Table. The action table is used later on, when transforming object diagrams and goals. We add one entry in the action table for each occurrence of an activity with non-empty effects. An entry has the form \((b, a, \varepsilon)\), where \(b\) and \(\varepsilon\) are the precondition, respectively the effects of the activity, and \(a\) is the action corresponding to the activity occurrence.

We capture the information regarding overlapping activities with a binary relation Overlap defined on the set of actions.

\(^1\)Note that this encoding as input/output actions is different from the encoding of Web services in [3]. In Web service composition the purpose is not to integrate the Web services, but to orchestrate them using an external controller. In that setting, actions are considered to be input (output) if they are controllable (non-controllable) by the external orchestrator.

<table>
<thead>
<tr>
<th>APFL Activity</th>
<th>STS Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic activity (receive, reply, invoke, contextEvent, etc.)</td>
<td>(s_0, b_1, op, s_1)</td>
</tr>
<tr>
<td>sequence activity (a_1, a_2)</td>
<td>(s_0, b_1, s_1, s_2, s_3)</td>
</tr>
<tr>
<td>pick onMessage operation=op1 activity (a_1)</td>
<td>(s_0, T, \Box \text{pick}, s_1)</td>
</tr>
<tr>
<td>onMessage operation=op2 activity (a_2)</td>
<td>(s_0, T, \Box \text{op2}, s_2)</td>
</tr>
<tr>
<td>apfPick onHumanInteraction op1 activity (a_1)</td>
<td>(s_0, \Box \text{op}, s_1)</td>
</tr>
<tr>
<td>onHumanInteraction op2 activity (a_2)</td>
<td>(s_0, \Box \text{op2}, s_2)</td>
</tr>
<tr>
<td>switch case condition=(c)</td>
<td></td>
</tr>
<tr>
<td>activity (a_1)</td>
<td></td>
</tr>
<tr>
<td>otherwise activity (a_2)</td>
<td>(s_0, T, \Box \text{switch}, s_1)</td>
</tr>
<tr>
<td>(s_0, \rightarrow \Box \text{case}, s_1)</td>
<td></td>
</tr>
<tr>
<td>(s_0, \rightarrow \Box \text{switch}, s_1)</td>
<td></td>
</tr>
<tr>
<td>flow activity (a_1) activity (a_2)</td>
<td>(s_0, T, \Box \text{flow}, s_1)</td>
</tr>
<tr>
<td>(s_0, T, \Box \text{flow}, s_1)</td>
<td></td>
</tr>
<tr>
<td>region</td>
<td>(s_0, T, \Box \text{region}, s_1)</td>
</tr>
</tbody>
</table>

**TABLE I: Translating APFL to STS**

For every pair \((a_1, a_2) \in \text{Overlap}, a_1\) and \(a_2\) are actions corresponding to two overlapping activities. We use the Overlap relation also for encoding the fact that regions are substitutes for one or more activities. If \(a_e\) is an action corresponding to a region in the fragment \(F\), we add Overlap relations between \(a_e\) and every action \(a\) that corresponds to a fragment \(F' \neq F\).

Fragments can be used also partially, and therefore by default all states are accepting (i.e., \(S^F = S\)). However, when a fragment includes the structured activity flow, we remove from \(S^F\) all the states included in the translation of the activity, except for the initial and final state.

The translation of the fragment Bring to claim from Fig. 4 is an STS \(\Sigma = (S, S^0, I, O, R, S^F, F)\), where:

- \(S = \{s_0, s_1, s_2\}\)
- \(S^0 = \{s_0\}\)
- \(I = \{?F_0, \text{Take to baggage claim}, \text{?F_0, Box at baggage claim}\}\)
- \(O = \emptyset\)
- \(R\) contains the following transitions:
  - \((s_0, \text{approved}(*), box), ?F_0, \text{Take to baggage claim}, s_1)\)
  - \((s_1, \text{approved}(*), box), ?F_0, \text{Box at baggage claim}, s_2)\)
- \(S^F = \{s_0, s_1, s_2\}\)
- \(F = \emptyset\)

For this fragment, we also add one action table entry:

\(\text{approved}(*), ?F_0, \text{Box at baggage claim}, \text{release}(\text{box})\)

**B. Transforming the object diagrams**

Given an object diagram \(o = (L, L_0, E, T)\), we define an STS \((S, S^0, I, O, R, S^F, F)\), where \(S = L, S^0 = L_0\), and all
states are accepting ($S^F = S$). Further, we label states with the corresponding propositions (i.e., $\forall s \in S : \mathcal{F}(s) = \{ s^p(o) \})$.

We build the transitions in the new STS using the action table. For each transition $(l, e, l') \in T$, we consider all the entries $(b, a, e)$ in the action table, such that $e^b(a) \in \varepsilon$. For each such entry $(b, a, e)$, we add to $\mathcal{R}$ the transition $(l, b, a, l')$.

For example, in the STS of box, we use the original transition ($\text{APPROVED, release, READY}$) and the entry $\text{approved(box),} ?\text{F_0. Box_at_baggage_claim,} \{\text{release(box)}\}$ to create the transition ($\text{APPROVED, approved(box),} ?\text{F_0. Box_at_baggage_claim,} \text{READY}$).

C. Transforming the goals

For each goal, we construct the STSs that correspond to the satisfiability of the goal. For every formula $\varphi$, we define a single output action $e_\varphi$ which gets triggered when the formula is satisfied. We use these completion actions for composing the formulas. The preconditions on the activities will be carried over as guards also in the goal STSs. We shortly describe the STSs for the building blocks of goal formulas:

- the STS for $\top$ has one transition on the completion action;
- the STS for $s^p(a)$ has one transition guarded with the corresponding proposition;
- the STS for $e^p(a)$ waits for any activity that contains the event in its effects. For each action entry $(b, a, e)$ such that $e^b(a) \in \varepsilon$, the STS has transition on $a$ guarded by $b$.
- The STS for $\varphi_1 \lor \varphi_2$ waits for any of $e_{\varphi_1}$ and $e_{\varphi_2}$, while the STS for $\varphi_1 \land \varphi_2$ waits for both.

The STS for a goal formula $\varphi \rightarrow (\varphi_1 \rightarrow \ldots \rightarrow \varphi_n)$ is as follows. If the premise takes place ($e_\varphi$ is reported), it moves to a non-accepting state and waits for any of $e_{\varphi_1}, \ldots, e_{\varphi_n}$ to be reported. The goal with preferences has the form: $\rho_c = (s_0, s_1, \ldots, s_n)$, where $s_0$ is the initial state of the STS, and $s_1, \ldots, s_n$ are the states reached with the transitions corresponding to $e_{\varphi_1}, \ldots, e_{\varphi_n}$.

Fig. 6 presents the STSs for the goal $G_1$ introduced in Section II-B. The goal with preferences is $\rho_c = (s_0, s_1, s_2)$.

D. Generating the composed adaptable pervasive flow

We build the planning domain by taking the parallel product of all the STSs of fragments, object diagrams and goals.

One key idea here is to enforce actions to be applied in parallel if they overlap and their preconditions hold. Slightly abuse the notation, we write $a \parallel a'$ to denote a new action which corresponds to actions $a$ and $a'$ being performed in parallel. We call $a \parallel a'$ a parallel action. Note that $a$ and $a'$ can themselves be parallel actions.

The overlap relation can now be extended to cover parallel actions. Given two actions $a \equiv a_1 \ldots a_n$ and $a' \equiv a_1' \ldots a_m'$, if for all $1 \leq i \leq n$ and $1 \leq j \leq m$, we have $(a_i, a_j) \in \text{Overlap}$, then also $(a_i, a_j') \in \text{Overlap}$.

We use a simplifying observation when constructing the parallel product: that an action always appears with the same guard. By construction, each action corresponds to at most one occurrence of an activity in a fragment. In case the action corresponds to an activity, the guard is the precondition of this activity. Otherwise, the guard is equal to $\top$. The action-guard relation is maintained when transforming object diagrams and goals, since we always add the action together with the guard.

Definition 5 (Parallel Product): Let $\Sigma_1 = (S_1, S_0^1, I_1, O_1, R_1, S^F_1, \mathcal{F}_1)$ and $\Sigma_2 = (S_2, S_0^2, I_2, O_2, R_2, S^F_2, \mathcal{F}_2)$ be two STSs. Let $\text{Overlap}$ be a binary relation. The parallel product $\Sigma_1 \parallel \Sigma_2$ is defined as

$$(S_1 \times S_2, S_0^1 \times S_0^2, I_2 \cup I_2, O_1 \cup O_2, R_1 \cap R_2, S^F_1 \times S^F_2, \mathcal{F}_1 \lor \mathcal{F}_2)$$

where $(\mathcal{F}_1 \lor \mathcal{F}_2)(s_1, s_2) = \mathcal{F}_1(s_1) \lor \mathcal{F}_2(s_2)$ and for every state $(s_1, s_2) \in S_1 \times S_2$:

- if $(s_1, b, a, s_1') \in R_1$ and $(s_2, b, a, s_2') \in R_2$, then $(s_1, s_2, b, a, s_1', s_2') \in R$, $s_1 \neq a_2$ and $(a_1, a_2) \notin \text{Overlap}$, then $(s_1, s_2, b, a_1, s_1', s_2') \in R$.
- if $(s_2, b_2, a, s_2') \in R_2$ and $\forall (s_1, b_2, a_2, s_2') \in R_2, a_1 \neq a_2$ and $(a_1, a_2) \notin \text{Overlap}$, then $(s_1, s_2, b_2, a_2, s_1', s_2') \in R$.
- otherwise, if $(s_1, b_1, a_1, s_1') \in R_1, (s_2, b_2, a_2, s_2') \in R_2$ and $(a_1, a_2) \notin \text{Overlap}$, then $(s_1, s_2, b_1, a_1, s_1', s_2') \in R$.

We construct our planning domain $\Sigma$ as the parallel product of fragment STSs $\Sigma_{F_1}, \ldots, \Sigma_{F_n}$, object diagram STSs $\Sigma_{O_1}, \ldots, \Sigma_{O_m}$, and goal STSs $\Sigma_{C_1}, \ldots, \Sigma_{C_k}$:

$$\Sigma = \Sigma_{F_1} \parallel \ldots \parallel \Sigma_{F_n} \parallel \Sigma_{O_1} \parallel \ldots \parallel \Sigma_{O_m} \parallel \Sigma_{C_1} \parallel \ldots \parallel \Sigma_{C_k}$$

We simplify $\Sigma$ by removing the transitions $(s, a, b, s') \notin R$ if $a'$ corresponds to a region. Let $R' \subseteq R$ be the updated transition relation. This condition comes from the fact that region actions should not be applied independently, since we are interested in obtaining a region-free composition.

Further, we remove all transitions $(s, a, b, s') \notin R'$ which can never fire, i.e., for which $s, a, b \notin R$. On the resulting STS, we can then remove the guards as well as the labeling function. The simplified planning domain, which we denote with $\Sigma_*$, is a tuple $\langle S, S^F, I, O, R', S^F' \rangle$, where $R'' \subseteq S \times (I \cup O) \times S$ is the transition relation after the updates.

We then construct the planning goal $\rho$ by combining the composition goals: $\rho = \bigwedge \rho_c$.

Given the domain $\Sigma_*$ and the goal $\rho$, we apply the technique presented in [2], which generates a controller $\Sigma_c$ such that $\Sigma_c \parallel \Sigma_* \models \rho$. If such a controller exists, then the controlled domain $\Sigma_c \parallel \Sigma_*$ corresponds to a synthesis of the fragments which achieves the composition goals.

We obtain the composed flow model by translating to APFL the STS of the controlled domain, to which we add potentially
missing connectors and start activities. The translation to APFL is conceptually simple: from the construction of STSs, each action name is unique and corresponds to at most one appearance of an activity in a fragment. Such actions can be mapped back to their corresponding activities. In the case of parallel actions, we introduce new activities, which result from merging the original overlapping activities.

Fig. 7 presents the result of composition in our scenario. Note that the scenario has no solution for any of the separate goals \( \top \Rightarrow \text{ready}^a(\text{box}) \), respectively \( \top \Rightarrow \text{disposed}^a(\text{box}) \).

IV. EVALUATION

We implemented our approach into a prototype tool. The tool translates object diagrams (XML), fragments and goals (APFL) into a planning problem. The planning problem is given as input to WSYNTH, one of the tools in the ASTRO toolset (http://www.astroproject.org). The output returned by WSYNTH is the controlled domain \( \Sigma _c \triangleright \Sigma _s \), which we then translate back to APFL.

We evaluated our tool using a dual-core CPU running at 2.26GHz, with 3GB memory. For each experiment, we report the averages over 20 runs. The tool takes 0.58 seconds to solve our Box at the airport scenario, with the fragments from Fig. 4.

We consider two features specific to fragment composition. First, there is a tradeoff between designing fragments with a large number of activities (a higher burden on the designer) and with a small number (longer composition time). Second, the set of available fragments can contain more fragments than actually necessary for composition.

In the first experiment, we evaluate the impact of fragment sizes on performance. Intuitively, larger fragments should be especially useful for hard composition problems. We therefore consider the following scenario. Assume that a group of \( k \) people is traveling to a conference. The conference has events that overlap in time, with up to \( k \) events running concurrently. The goal is to ensure that every event is attended by one person. This scenario is equivalent to a graph \( k \)-coloring problem, which is NP-complete for \( k \geq 3 \). We encode the scenario for \( k = 3 \). Activities assign events to people, with the time overlap constraints represented as preconditions. At each step, we increase the number of events and measure the performance for fragments of different sizes, starting with fragments of one. For this purpose, we randomly group activities into semantically correct fragments. Note that only a third of the fragments are actually composed. Fig. 8a presents the composition time for up to 36 activities. We observe that larger fragments lead to a significant speedup. Using fragments of two or more activities can lead to a time increase of 20% in the worst case, and a decrease of up to 95% in the best case. For this domain involving complex composition problems, larger fragments provide a clear benefit in terms of composition time.

In the second experiment, we test how the number of available fragments influences the performance. For this purpose, we vary the total number of fragments, while keeping constant the number of fragments actually composed. We use our Box at the airport scenario. At each step, we generate new fragments as copies of random fragments from our original set. Fig. 8b presents the composition time for up to 28 fragments. As expected, the composition time grows exponentially in the number of fragments, due to the hardness of the planning problem. However, it is a promising result that the composition takes a reasonable time for a fairly large number of fragments, since we can use existing selection techniques to reduce this.

V. RELATED WORK

All business processes are designed to achieve a business goal. The goal can be represented using either an imperative or a declarative language. Imperative process models (e.g., [4]) focus on how: they describe the way to achieve the goal, and assume that the environment is stable. Declarative process models (e.g., [7]) focus on what: they specify the goal through constraints which approximate the desired behavior. Such descriptions are suitable for frequently changing environments, but cannot be executed completely automatically.

We use both process description techniques to achieve a greater flexibility with respect to the environment. The declarative part corresponds to goals, and is globally available. The imperative, concrete part corresponds to pervasive process fragments, and is available only locally. Another approach combining the two techniques is [8], which introduces a framework for defining semantic constraints over processes. If the imperative definition of a process instance is changed during execution, the framework allows to check whether it is still compliant to the declarative, semantic definition.

Fragment composition is also an implementation of process flexibility, as classified in [9]. Our approach supports flexibility by underspecification, when the process model contains placeholders for which a concrete realization is provided at runtime. This realization is selected in case of late binding (e.g., [10], [11]), and constructed for late modeling (e.g., [12]). In our approach the placeholder is the goal, and both late binding and
late modeling can be achieved. Differently from our approach, [10] and [11] do not compose fragments, they select and dynamically bind complete process models. In [12], instances are progressively built during execution, based on constraints which specify how and when the fragments can be composed. Such concrete constraints are absent in our approach.

[13]–[15] are close to our work from a modeling perspective. In [13], the authors model small processes as ‘proclets’, a modeling metaphor similar to process fragments, based on the same assumption that process knowledge is distributed. Unlike fragments, proclets get integrated into a business process by interaction. In [14], process models are generated starting from object life cycles. While [14] uses object life cycle conformance and coverage requirements, we use the more expressive goals with preferences. [15] discusses the artifact-centric paradigm and its dimensions: (i) business artifacts, (ii) artifact lifecycles, (iii) services/tasks which make transactional changes to artifacts, and (iv) constraints on these changes. Our approach can also be seen as artifact-centric, with object diagrams, effects, and preconditions corresponding to (ii)-(iv).

Closely related to fragment composition is the problem of Web service composition, that of generating a composite Web service starting from service interfaces and composition requirements. The result is an executable implementation which satisfies the requirements by suitably invoking the existing Web services. This is different from fragment composition, where fragments are integrated into a new flow model. ASTRO (see [2], [3]) is a Web service composition approach, which is also the starting point of our work. In particular, we have exploited the goal language and object diagrams introduced in [3] to model our application, and used the powerful planning techniques in ASTRO to implement our solution.

VI. CONCLUSIONS

We presented an approach for composing pervasive process fragments according to context and goals. We first specified the properties of domain entities using object diagrams, and used these properties to define our goals and to annotate the fragments. We then encoded object diagrams, goals and fragments into an AI planning problem. The result of composition is a complete adaptable pervasive flow which satisfies the goals and corresponds to a synthesis of the fragments.

In our future work, we will address several open issues. First, we plan to extend our composition mechanism with adaptation constructs provided by APFL, such as the abstract activity (described in terms of goals) and the built-in adaptation tools [5]. Further, we plan to evaluate our approach on a real-world scenario developed in the project ALLOW, which involves vehicle logistics. As a last issue, until now we have considered only the lifecycle aspect of the data, in the form of object diagrams. We plan to extend our object diagrams and fragments to include also an information model.

Acknowledgment: This work is partially funded by the ALLOW project. ALLOW is part of the EU 7th Framework Programme (contract no. FP7-213339).

REFERENCES

Designing future-context-aware dynamic applications with structured context prediction

Sonja Zaplata*,†, Matthias Meiners and Winfried Lamersdorf

Distributed Systems and Information Systems, Computer Science Department, University of Hamburg, Vogt-Koelln-Strasse 30, 22527 Hamburg, Germany

SUMMARY
Middleware support for advanced pervasive applications includes dealing with heterogeneous systems and dynamic changes of execution environments. If applications are context-aware, they are able to detect such changes at runtime and react to them accordingly. Furthermore, they can also proactively adapt to upcoming situations by making accurate assumptions about future contexts. However, the design of reusable middleware support for such future-context-aware applications is still challenging as a supporting prediction system has to be generic, but at the same time has to provide potential for high accuracy and efficiency.

This paper proposes a concept for the development of future-context-aware applications on the basis of the novel approach of structured context prediction. A framework, this approach allows for integrating domain-specific knowledge and facilitates application, combination, and implementation of suitable prediction methods. In addition, specific runtime mechanisms for distribution of knowledge and adaptation of prediction tasks are proposed, which enable useful context predictions even for dynamic applications. The overall development process as well as practical experiments with the prototype framework is illustrated by two use case scenarios—demonstrating that both high accuracy and efficiency of future context predictions are achievable and even ad hoc context predictions can be supported this way. Copyright © 2011 John Wiley & Sons, Ltd.

1. INTRODUCTION
Context-aware applications are evolving into major driving factors for pervasive systems. Following Dey and Abowd, context is defined as any information characterizing the situation of surrounding and situational entities relevant to an application and its users. A system is context-aware if it uses such context to provide relevant information and/or services to the user, whereas relevancy depends on the user’s task [1, pp. 3–6]. In consequence, the ability to obtain, to process, to manage, and to provide relevant context information describing the user’s environment and situation has become one of the most important requirements for such systems. In addition to that, the prediction of future context is another important step for enabling devices and applications to also proactively support the user or to enable the desired automatic execution of his tasks even in dynamic environments [2].

Currently, the development of context-aware applications can be facilitated by reusable components such as generic context management systems and middleware support [3–5], so that application developers are able to build context-aware applications on the basis of generic frameworks. As an extension to such systems, the novel approach of structured context prediction (SCP) [6] offers

---

*Correspondence to: Sonja Zaplata, Distributed Systems and Information Systems, Computer Science Department, University of Hamburg, Vogt-Koelln-Strasse 30, 22527 Hamburg, Germany.
†E-mail: sonja.zaplata@informatik.uni-hamburg.de

Copyright © 2011 John Wiley & Sons, Ltd.
reusable system support for the prediction of future contexts in order to develop context-aware applications, which do not only consider their respective current context, but which are also able to derive and use information about future situations.

On the basis of SCP, this contribution specifically focuses on the development of such future-context-aware applications and shows the realization of prediction support for different application areas. This paper especially considers dynamic applications, which require additional runtime mechanisms for distribution of knowledge and adaptation of prediction tasks. The relevance of context awareness, context predictions, and different requirements of applications are motivated by two use cases in the following.

1.1. Use case 1: proactive energy management for mobile devices

Today’s applications for energy management of mobile devices utilize context information in order to support the user with most accurate statements about remaining energy resources. Relevant information involves, for example, current energy consumption, current battery level, and current environmental conditions (e.g., temperature). However, predictions about the future usage of a mobile device and expected charging opportunities allow for advanced recommendations about the optimal time interval for reloading the battery, to save energy in an adjusted way, or to warn the user in case of critically intensive usage leading to an upcoming uncovered demand for energy [6].

In such a scenario, the application’s environment (e.g., the location, the time of day, the behavior of the user) is likely to change. Thus, such context entities are parameters to be considered as the relevant context. Provided that the mobile device is often utilized in a similar way, the application responsible for energy management can use a supporting prediction system for learning such regularities. Thus, respective future values of entity attributes can be predicted, for example, that a mobile device will probably not reach the location of a charging station for the next 10 h.

As the relevant prediction tasks are known in advance, they can be defined at design time of the application. The prediction system must thus be able to retrieve selected context data (in periodic or irregular intervals), detect characteristics within this context, and provide most accurate results for predefined prediction requests of the application. However, predictions must not affect the supported application in a negative way, for example, here, consume more energy than can be saved by the use of predictions. It is therefore required not only to customize the prediction system to the context of a consuming application but also to adapt the way a prediction is performed with respect to application-specific accuracy and efficiency requirements.

1.2. Use case 2: predicting service availabilities for mobile business processes

Mobile business processes often include the execution of geographically or logically distributed software services. In order to access such services and exploit local functionalities, processes can be migrated to other mobile or stationary devices [7]. The decision of migrating a process or not is supported by context data, which determine whether a service can be accessed from a remote system. Figure 1(a) shows a simple example where a field worker wants to share his position. Therefore, he initiates a predefined process consisting of retrieving GPS data of the current position, transforming the position data into a human-readable address, and propagating this address by sending a text message to a specified telephone number. Figure 1(b) shows a possible distributed execution of this process in dependence of the discovered context including device capabilities or network connections. Here, each of the three devices contributes to the execution of the process so that it is successfully executed in the end.

Now, let us assume that the first two activities have been successfully executed, but that the text message cannot be sent because the device that is currently responsible for the execution of the process is not connected to a short message service (SMS). Furthermore, there is currently no device with a suitable context (i.e., capability to send a text message) available in the direct vicinity. This situation is depicted in Figure 1(c), showing several possibilities: first, the process can remain on the current device. Second, the process can be migrated to a device nearby, which has another temporarily ‘useless’ context, but which will probably meet an SMS-capable device in the near future.
Third, the process can be migrated to another similar device, which will probably not meet an SMS-capable device in the near future. It is easy to see that only the migration to the first device will lead to a successful execution of the process, whereas the other two possibilities will cause the process to fail—so accurate assumptions about future contexts can have a very high impact on the success of process execution and are thus required to optimize the migration decision.

Nevertheless, prediction support for such dynamic applications is rather challenging, because the specific prediction task cannot (such as in the first use case) be determined at design time. Instead, it is dependent on the specific process to be executed at runtime and its special requirements for services, users, locations, or other criteria. Furthermore, because of distribution, it is difficult to determine a suitable basis for such predictions. In the presented example, it is not possible to simply ask other devices for a prediction about the service availability of SMS—because SMS may not have been relevant for this device before and therefore no data exist about former experiences with such services. A supporting prediction system must thus provide mechanisms for runtime distribution of application-specific knowledge and adaptation of prediction tasks.

1.3. Requirements and methodology

In summary, a general applicability of a context prediction system imposes several principal requirements: first, the system should support a preferably wide range of applications and diversity of exploitable contexts [3, p. 11] in order to maximize reusability. Furthermore, there are individual differences between users [8], which can also change continuously [9, p. 77]. Thus, the system has to adapt to the individual user at runtime by learning about the characteristics and regularities that determine the future context (cf. [8], [9, p. 77]).

Besides being generic and adaptive, the prediction system should be able to produce customized predictions for different scenarios in a reliable and satisfying way, that is, as accurately and efficiently as possible. Mobile devices especially often suffer from a lack of resources [10], so that the corresponding requirement for efficiency makes this trade-off even more challenging. Finally, the motivation to support application developers implies a preferably low effort for them.

Especially in view of middleware systems or distributed applications, however, context entities and prediction tasks may not always be known in advance (i.e., at design time) [5]. In order to support also such dynamic applications, the creation, adaptation, and distribution of prediction tasks as well as distribution of additional application-specific knowledge must thus be possible at runtime.

In order to address these requirements, the next section presents relevant background information about context prediction systems and introduces hybrid prediction techniques. Section 3 describes the approach of SCP, which partly uses such hybrid techniques while integrating domain-specific knowledge in order to achieve accuracy and efficiency. Section 4 introduces the general development process to design or enhance future-context-aware applications and illustrates the procedure.
with the presented use cases. Results are summarized in Section 5 showing both a quantitative scenario-based evaluation and a conceptual discussion. Finally, Section 6 gives a short summary and outlook.

2. BACKGROUND AND RELATED WORK

As the basis for every prediction about the future, there must be a sufficient amount of related data collected in the past. In the second use case, for example, the availability of a service and the position of the device are parameters to be considered. Both are called variables in this paper and can have different values at different points of time (e.g., position = at home and service available = false). Such values constitute so-called historical data:

**Definition 1 (Historical data)**

Historical data of variables $V_1, \ldots, V_n$ in a time interval with discrete points of time $j \in \mathbb{Z}$ consist of values $v_{i,j}$ for all variables $V_i$ and all points of time $j$.

The definition is inspired by time series and stochastic processes as both are very generic concepts. It is used in combination with stochastics in order to express uncertainty about future variable values. Furthermore, different methods may have different types of scales (e.g., nominal or ratio according to statistics). Finally, variables can also be understood as attributes of entities (e.g., the position of the user) establishing the connection to context, which is defined by Dey and Abowd on the basis of entities [1, pp. 3–4].

Being on the agenda of context awareness [3, p. 6], histories provide essential input for prediction. However, at least in the context of mobile devices, it is not recommended to explicitly store complete histories because memory capacity is often limited. Furthermore, most often, histories only present raw data as the source for the actual prediction. Therefore, this paper takes into account the principle of inductive learning (cf. [11, pp. 60–61]), which is, for example, used in data mining, machine learning, pattern recognition, and statistics. Inductive learning extracts knowledge about characteristics and regularities from observed parts of historical data (training data). The extracted knowledge is used to predict future values given both the current and recent context as historical data. As depicted in Figure 2, prediction and learning are potentially concurrent processes. This strategy is advantageous because storing or repeated processing of complete histories are avoided and thus resources of mobile devices are used more efficiently.

On the basis of this background, many existing approaches aim to support predictions for a particular domain or a specific context criteria, such as predicting locations of mobile users [12–15], network availability [16], or the near future health status of persons [17]. In contrast, approaches for reasoning about context situations can include arbitrary—and even heterogeneous—context data (e.g., [18–20]), but lack support for different types of predicted information. Required items may include, for example, the value of a specified variable or the kind of event occurring at a given point.
of time in the future, or the estimated point of time where a given event will occur (cf. [9, p. 207]), or the probability of the occurrences of specified variable values or events. Other related research works deal with the evaluation of methods to analyze which of them are appropriate for which prediction tasks, for example, [21]. Only few approaches explicitly address the design of a prediction framework to support the development of future-context-aware applications in general (cf. also [22]). Here, the approaches of Mayrhofer [23], Sigg [9], and Petzold [24] are considered to be major contributions towards generic context prediction:

Mayrhofer’s approach uses an exchangeable prediction method [23, p. 37]. The approach combines context prediction with a preceding extraction of context on a high abstraction level (high-level context [3], for example, the complex situation in a meeting) from context on a low abstraction level (low-level context [3], for example, the noise level in the current room) [23, pp. 5, 33, 62]. The extraction of high-level context is realized by clustering [23, p. 37]. The prediction step uses the results of the clustering as input, that is, predictions are made about future clusters [23, p. 40]. Context prediction is simplified by this approach because only few clusters have to be considered as possible values of only one variable. Thus, a prediction based on high-level context permits high efficiency. The effort for application developers using Mayrhofer’s approach is low [23, p. 128], because almost no knowledge about the application domain is used [23, p. 128]. A similar methodology is also presented by the authors Nurmi et al. [25] who propose an additional preprocessing phase (e.g., for normalization) in order to separate preprocessing from data acquisition [25, p. 3]. However, in both cases, only predictions about high-level context are possible, and Sigg states that such an approach is disadvantageous for accuracy [9, p. 179].

Sigg’s approach is nearly fully generic, as it is, in particular, not restricted to the prediction of high-level context [9, p. 92]. On the other hand, the simplification of prediction based on high-level context (e.g., clustering) is missing. Sigg considers a single prediction method applied at runtime, which is exchangeable at design time [9, p. 91]. The effort for application developers using Sigg’s approach is relatively low because prediction methods with demand for extensive configuration are not taken into account explicitly [9]. However, regardless of the choice of the method, it has to deal with a potentially high number of possible variable/value combinations as well as additional application-dependent and variable-dependent requirements. Such requirements could involve fast predictions of the values of a specific variable, different scale types, or the utilization of the type of dependencies among variables [9, p. 96], [23, p. 66]. All three authors state that there is no universal method fulfilling all possible requirements [9, p. 203–204], [23, pp. 86, 91], [24, p. 142]. Thus, it is expected that there are serious limitations considering accuracy and efficiency for generic context prediction as long as only a single method is used. The same argument holds true for the approach of Anagnostopoulos et al. [26], which focuses on location as an essential ingredient of predictions in mobile computing environments and thus is restricted to a set of (specially adapted) interpolation formulas [26, p. 137].

Petzold’s approach is restricted to the prediction of primary context [1], that is, time, position, identity, and activity [24, p. 141] and is therefore not fully generic. In addition to the other approaches, it allows a parallel, hybrid application of multiple methods [24, p. 87] (similar in [15]). This means that the same prediction task can be assigned to multiple methods in order to fulfill application-dependent and variable-dependent requirements in an optimized way. The advantages of multiple methods can be combined [24, p. 142], for example, different specialized methods can be utilized to address different aspects of the prediction. This allows for high accuracy and efficiency. On the other hand, the combination of methods leads to higher effort for application developers who have to select and combine the methods in order to apply the approach to their individual application domains. The use of pre-configured methods with different configurations can diminish the effort required to configure methods for the prediction of the user’s position [15]. However, pre-configuring methods is not feasible for all possible application domains.

Figure 3 summarizes the main characteristics of the major approaches—showing that there is still no approach, which is sufficiently generic and offers high accuracy and efficiency at the same time. Nevertheless, all approaches provide valuable experiences: first, the application of multiple prediction methods is interesting because it offers the possibility to achieve high accuracy and efficiency. Second, an integration of domain-specific knowledge narrows the prediction task and can
simplify the achievement of high accuracy and efficiency. At the same time, it offers generality to conveniently adapt the prediction system to the application—which is an important characteristic in view of the additional requirements for runtime distribution and ad hoc predictions, because such application-specific knowledge can also be distributed or exchanged at runtime. Prior knowledge can also help to reduce the amount of training data to be learned at runtime.

Beyond context, Hilario [27] distinguished different kinds of techniques for a hybrid application of multiple methods: the parallel application of methods, which is used by Petzold is called coprocessing. In contrast, chain processing denotes the sequential application of methods so that the prediction result of one method is used as input for another method. Both techniques expose benefits concerning accuracy and efficiency [27, pp. 21–22], [28, pp. 1–4]. In addition, Bayesian networks offer the possibility to describe a graph-based dependency structure of variables (cf. [29, pp. 101, 112–114], [30])—which is interesting to describe the connections between methods (i.e., taking the output of a method as the input of another method) (cf. Section 3.1).

Several methods could be applied for the prediction in a hybrid way (e.g., neural networks, regression, decision trees, discriminant functions, Markov chains, autoregressive moving average). However, many of them are not suitable because they require too many resources to be used on mobile devices or do not support adaptive online learning [2, p. 34], [23, p. 66], that is, are not able to update already learned knowledge and therefore need an explicit learning phase. The remaining methods come into consideration for the application-specific customization of a respective prediction framework. As such application-specific characteristics and dependencies have to be identified, structured, and specified by the application developer, the following section introduces the approach of SCP, which is based on these observations and thus presents an alternative to existing context prediction frameworks (cf. Figure 3).

3. STRUCTURED CONTEXT PREDICTION

The approach of SCP (cf. [6]) realizes a generic prediction system, which can be utilized in order to develop future-context-aware applications or enhance existing applications with the ability to make assumptions about future situations. It is based on two fundamental principles derived from the preceding analysis of existing approaches: first, knowledge about the application domain is used as valuable information incorporated by the application developer at design time or at runtime. The second principle is a hybrid application of multiple, exchangeable prediction methods. Thus, methods that are appropriate to ensure accuracy and efficiency of domain-specific predictions can be selected and combined by the application developers respectively. With this information, so-called Prediction Models can be created and incorporated at design time of the application or can be distributed at runtime. Additionally, an adaptation to the individual user is achieved by adaptive online learning as the default learning mechanism.

The following two sections introduce prediction models as a way to specify application-specific variable dependencies and present a supporting architecture for a corresponding prediction system. The practical usage of the resulting prediction framework is shown in Section 4.
3.1. Prediction model

The knowledge about an application domain is described as a prediction model, which specifies the way predictions have to be performed. Basically, it assigns a method to each relevant variable in order to predict its value. As input, the method uses the values of other variables—which are again predicted by their own methods or which are known (e.g., measured by a sensor or distributed at runtime).

The main part of a prediction model specifies how the methods and respective variables are connected. This part is called prediction net. A prediction net specifies that the value of a variable at a specific point of time is predicted using the values of other variables at the same point of time or earlier. As an example, the future position of a user can be predicted by the current position and by preceding positions. Prediction nets are formally defined as follows:

**Definition 2 (Prediction net)**

A prediction net is a finite directed graph $N = (W, E)$. The node set $W$ is a set of variables $\{V_1, \ldots, V_n\}$. An edge $V_i \xrightarrow{\Delta} V_{i'} := (V_i, \Delta, V_{i'})$ in the edge set $E \subseteq W \times \mathbb{N}_0 \times W$ expresses that the value of $V_i$ at the point of time $j - \Delta$ is used as input for the prediction of the value of $V_{i'}$ at the point of time $j$. The symbol $\Delta$ denotes a time offset. The notations $V_i \rightarrow V_{i'} := V_i \xrightarrow{0} V_{i'}$ and $V_i \xrightarrow{1, \ldots, l} V_{i'} := \bigcup_{k=1}^{l} V_i \rightarrow V_{i'}$ are allowed as abbreviations. A prediction net contains no cycles of the form $V_i \xrightarrow{\Delta_1} \cdots \xrightarrow{\Delta_l} V_i$ with $\sum_{k=1}^{l} \Delta_k = 0$.

The given definition is based on discrete time as introduced in Definition 1. The simple relationship $V_i \rightarrow V_{i'}$ denotes that the value of the variable $V_{i'}$ at a specific point of time is dependent on the value of variable $V_i$ at the same point of time. Accordingly, the relationship $V_i \xrightarrow{1, \ldots, l} V_{i'}$ describes that the value of the variable $V_{i'}$ at a specific point of time is dependent on the values of $V_i$ at the given $1, \ldots, l$ time steps earlier. The use of time differences instead of absolute points of time allows that the point of time ‘0’ can be chosen in a way which fits the prediction task. If, for example, the application requires a prediction about the value of a variable at a specific point of time, this point of time can be determined to be the point of time 0.

Figure 4 shows a simple example of a prediction net, which is mainly intended to predict the future energy consumption of a mobile phone (cf. use case 1). The different colors are used to visualize variable dependencies: the phone usage at a specific point of time can be predicted by the number of missed calls, the position and the time of day at the same point of time, and the phone usage one and two time steps earlier. The number of missed calls at a specific point of time is predicted by the phone usage and the number of missed calls both one time step earlier. The position at a specific point of time is predicted by the time of day at the same point of time and the position one and two time steps earlier. The time of day is not predicted because it is assumed to be known.

Prediction nets are inspired by dynamic Bayesian networks [31], which explicitly take into account the factor time. The main difference between Bayesian networks and prediction nets is that Bayesian networks describe dependencies between variables, and prediction nets describe connections between methods that are assigned to the variables. Prediction nets only allow predictions along the edges of the graph. On the one hand, this makes the design of such a net more complex, but on the other hand, it facilitates the predictions—which is advantageous when taking into account restricted resources of mobile devices. Thus, prediction nets are not considered to be a prediction method but are rather intended as a representation in order to integrate existing methods. It is, for example, possible to use a probability table for a variable with nominal scale type and at the same time regression for another variable with ratio scale type in order to handle a linear dependency in an efficient way with only low storage requirements.

The first step to perform a prediction on the basis of a prediction net is to generate the relevant part of the respective unfolded prediction net. An unfolded prediction net is a representation of a prediction net, which represents each variable multiple times, that is, one variable instance for each point
of time. If, for example, the position of the mobile device (cf. $V_3$ in Figure 4) should be predicted by the preceding position, the prediction net contains the variable $V_3$ and the edge $V_3 \to V_3$. The respective unfolded prediction net is thus determined as $\ldots \to V_{3,-2} \to V_{3,-1} \to V_{3,0} \to V_{3,1} \to V_{3,2} \to \ldots$ where $\ldots, -2, -1, 0, 1, 2, \ldots$ are points of time (realization of a Markov chain). Figure 5 shows an extract from the unfolded representation of the prediction net shown in Figure 4, respectively.

Prediction nets are a powerful modeling instrument; for example, they can be used as a basis for the hybrid application of methods. Figure 6 shows a simplified graphical representation for a resulting prediction model using chain processing where the output of the first method $V_2$ is used as the input of the second method in order to calculate $V_3$. This strategy can be used not only for clustering or generalization but also to prepare variables for the application of another method. As many applications need predictions based on different variables, the insertion of an intermediate step can help to reduce the total amount of stored knowledge and simplify repeated steps of prediction. Figure 7 shows the mechanism of coprocessing, respectively. Here, $V_{21}$ and $V_{22}$ are variables with different methods for the same prediction task. The methods executed in parallel can be used in competition or in cooperation and can even exchange data, such as intermediate results. Here, the different (end-)results are integrated by the method of $V_3$ (e.g., by arithmetic mean or majority vote).

The prediction model combines the knowledge about relevant variables and variable dependencies (i.e., the prediction net) with appropriate scales, methods, and measuring adapters. The general structure of the entire prediction model is depicted in Figure 8. The central element Prediction Model consists of a representation of a set of Variables and their relationships from the prediction net. Each variable has a unique Name, a Scale, and optionally the name of a Nominal Type in
case the variable has a nominal scale. If the value of the variable can be measured, an appropriate Measuring Adapter can be specified. The relationships among variables (i.e., the prediction net) are captured by a set of Parent Edges, which each represent a reference to a parent variable together with the respective Time Offset measured in configurable time steps. Such a relationship describes, for example, that variable $Y$ depends on the value of variable $X$ three time steps earlier. Furthermore, the variables define references to the methods to be used for the prediction of the variable’s value (Method Preknowledge), which can either use or not use the values of independent variables. Respectively, Learning Method Preknowledge indicates if automatic online learning is switched on for Method Preknowledge with resp. without values of independent variables. The Default Learning Distance defines the time interval between the beginning of measuring and learning a relationship (i.e., a value combination of the dependent and the independent variable). Section 4.1 presents a detailed example of these relationships on the basis of use case 1.

Finally, as general settings, the prediction model contains information about the configured Length of Time Step (e.g., measured in milliseconds), the desired Length of History (measured in time steps) expressing that values elder than the specified length are discarded, and the Frequency of History, which determines how often values of variables are measured and stored within one time step. Measuring Adapters and Method Preknowledge can be enhanced by Specific Settings and respective Implementations.

3.2. System architecture

As shown, a prediction model holds the application-specific knowledge and specifies the relevant variables, the dependencies among each other, and the assignment of methods, which are suitable for the prediction of future values. In the developed prototype system, the prediction model is represented as an XML document. In order to start the collection of training data, which allows for
adapting to the individual user and to enable predictions at runtime, the prediction model is passed to an application-independent prediction system installed on the respective (mobile) device in order to customize it to the specified prediction tasks.

Figure 9 shows the architecture proposed by the approach of SCP in order to use a prediction system as a reusable component in context-aware applications or middleware systems. **Learning** and **prediction** as concurrent processes are mapped to different layers. The two layers are linked by the **knowledge layer**, which constitutes a data layer at the bottom of the architecture. The learning layer creates and updates knowledge, and the prediction layer uses this knowledge for predictions requested by the higher-level application layer. The architecture permits that the prediction system is located on another device as the application itself (e.g., a powerful server offering location-dependent predictions). The data acquisition can also be performed remotely in order to use sensors of other devices nearby. A further possibility of distribution supported by the architecture is to share learned knowledge with other devices.

As depicted in Figure 9, all of the three mentioned layers contain parts of the methods. Each method possesses its own knowledge (e.g., frequencies of variable values in case of a probability table) and its algorithms for updating the knowledge and predicting the value of the variable associated with the method. A method knows about required input values and the respective output value as the result of the prediction, but nevertheless, it does not have to be aware of the structure of the prediction net.

The prediction layer makes use of an algorithm, which coordinates the methods. This is necessary because—unlike learning—the prediction normally cannot be performed by a single method only. For example, if the phone usage should be predicted using the prediction net in Figure 4, the position and the number of missed calls also have to be predicted by their respective methods. Accordingly, a prediction initiated by the application begins with the generation of the relevant part of the unfolded prediction net. For each relevant point of time for every relevant variable, a prediction unit called **predictor** is created. A predictor obtains input values from its parent predictors and passes them to the method associated with its variable in order to predict the required value at the given point of time.

```python
if v_{i,j} already predicted then
    return v_{i,j}
else
    if v_{i,j} known then
        return v_{i,j}
    else
        let parent-predictors predict their values
        predict v_{i,j} with own method using these values
        return v_{i,j}
    end if
end if
```
time. This procedure is performed unless the value has already been predicted before or is known (e.g., measured by the data acquisition layer). This ensures that not more than one prediction is performed by each predictor, which leads to a linear complexity. The following abstract algorithm summarizes the prediction of the value \( v_{i,j} \) of a variable \( V_i \) at the point of time \( j \):

The algorithm is executed multiple times in order to capture probabilistic behavior. This is inspired by an algorithm called stochastic simulation, which was originally developed for Bayesian networks [32, p. 189–191]. In a prediction round, each predictor and its method predict one of the possible values of the corresponding variable at the corresponding point of time. A value should be chosen with high probability only if the probability occurring in reality is also high. Thus, the prediction can be understood as a kind of simulation. The individual prediction results are used to finally obtain a probability distribution. This can either be a distribution of possible values of the variable at a specific point of time in the future or a distribution of possible points of time in the future where the variable will have a specific value. The probabilities capture uncertainty of the application domain (e.g., about the outcome when throwing a dice) as well as uncertainty arising from the prediction, which is only partly captured by now. Additionally, to the number of prediction rounds, an alternative reduced mode without repeated execution of the algorithm can be chosen. Compared with existing algorithms for Bayesian networks, this allows for a higher scalability. The reduced mode is thus, for example, suitable for resource-restricted mobile devices.

The set of usable methods can be extended by implementing new, possibly application-dependent methods. In consequence, an implementation of a prediction system according to the presented approach constitutes a framework, which can be extended by other methods as ‘plug-ins’. So far, a reference configuration of methods is established, which is mainly based on linear regression and probability tables. Here, a probability table represents a method that stores occurrence frequencies of variable/value combinations as knowledge. The properties of the two methods complement one another and are therefore well suited for hybrid application.

4. DEVELOPMENT OF FUTURE-CONTEXT-AWARE APPLICATIONS

From the perspective of an application developer, the general procedure of using the prediction system consists of two parts: the first part is determined by the development of the prediction model at design time. The second part is the retrieval of predictions by the respective application at runtime. Runtime learning is executed automatically so that most aspects of learning are hidden from the application and from the developers.

In order to create a prediction net as part of a prediction model, we show a systematic procedure in Figure 10. The procedure begins with the analysis of the application domain. Information about the domain and the kind of required predictions is integrated in order to specify what should be predicted on which basis, resulting in the design of a prediction net. While creating the net, the properties of the selected methods have to be considered because they are highly responsible for the suitability in view of the identified dependencies and consumption of resources. After the specification of this general structure, the selected methods have to be configured in detail, for example, the size of a probability table has to be determined. Depending on the configuration issue, this can be performed either in the prediction model itself or in the implementation of the method, respectively (cf. Listing 2 for examples).

![Figure 10. Procedure for systematic development of a prediction model.](image-url)
Listing 1. Java example for the integration and usage of the prediction system.

```java
   // on application startup:
   PredictionSystem predictionSystem = new PredictionSystem(predictionModel);
   predictionSystem.loadInstanceData(instanceData);
   predictionSystem.startLearning();
   predictionSystem.startUpdatingRecentHistory();
   
   // predict value of a variable after 60000 milliseconds with 300 iterations:
   FrequencyNonParameterDistribution predictedDistribution =
       (FrequencyNonParameterVariableDistribution) predictionSystem.getPredictionState("VariableName", new PointOfRealTime(60000), 300);
   double variableValueProbability = predictedDistribution.getProbabilityOfVariableValue(predictionSystem.getNominalValue("NominalTypePlan", "VariableValue"));
   
   // on application shutdown:
   predictionSystem.stopUpdatingRecentHistory();
   predictionSystem.stopLearning();
   predictionSystem.storeInstanceData(instanceDataFile);
```

Figure 11. Runtime distribution and adaptation.

The resulting prediction model is expressed in XML format, which is interpreted by the generic prediction system. The prediction system can be integrated into the ordinary life cycle of the supported application where it is, for example, initiated at application startup and terminated before the application exits. At runtime, predictions are requested by specifying the desired parameters (e.g., variable name, target point of time, prediction rounds). A typical example for using the prediction system is shown in Listing 1.

At last, the model has to be validated and steps of the procedure have to be repeated if results do not meet the requirements. As depicted in Figure 11, there are three possibilities to adapt the knowledge and the prediction system at runtime: first, the included component for adaptive online learning supports adaptation to the individual user by learning about the regularities in the context provided by the underlying local context management system. Second, the prediction model can be updated or exchanged, or entire new prediction models can be inserted. This includes adding new prediction net fragments (e.g., to include newly detected dependencies) or the exchange resp. reconfiguration of methods. If dynamic code loading is permitted, implementations also for new methods can be added to the system. Third, external knowledge can be inserted by the application and can be incorporated into existing knowledge. For example, knowledge about values of specific variables contained in the corresponding prediction net can be integrated at runtime in order to enable or advance results of a specific prediction task. Respective examples for customizing the generic prediction system are demonstrated by the two use cases (cf. Section 1) in the following.

4.1. Customization to use case 1

The integration of domain-specific knowledge can be illustrated by the application responsible for energy management of a mobile phone as an example for use case 1 (cf. Section 1.1). As speech transmission is identified as a main cause for energy consumption, it has to be predicted whether
the user will make telephone calls. Such phone usage can be represented as a boolean variable (representing whether the user makes a telephone call), which depends on other variables such as the time of day and the position of the user. The telephoning and the position are each predicted by an individual method. As an example, the telephoning can be predicted by the current time of day and the position of the user, and the position can be predicted by the time of day. The prediction model can thus make use of the actual dependencies between the variables, for example, of the fact that a user rarely telephones at night. Thus, the probability for a phone call at night is expected to be low. In contrast, the probability for phone calls is expected to increase if the user has a large number of missed incoming calls. Furthermore, it can be expected that the user rather uses the fixed line telephone network if he is at home or at the office—and that there are more outgoing calls at foreign places. A possible result of these considerations is the (simplified) prediction net shown in Figure 4.

On the basis of this net, Listing 2 shows an excerpt from the XML representation of the prediction model (cf. graphical representation in Figure 8), which handles the prediction of the mobile user’s position. The prediction model uses the reference configuration consisting of probability tables and, additionally, the decision method majority vote in order to integrate co-processed intermediate results by different methods (i.e., on the basis of the time of day or previous position). The position of the user can be modeled as discrete places (e.g., at home, at the office) or by using clustering of GPS coordinates alternatively. The same strategy can be applied for time intervals—leading to rather low static memory requirements.

As stated, this use case does not necessarily require runtime exchange of knowledge or adaptation of prediction tasks. Therefore, it is sufficient to develop the prediction model once at design time of the application, integrate the prediction system as shown in Listing 1 and use runtime learning to adapt to the individual user. Predictions about the expected future usage of the mobile phone can now be integrated to calculate the expected duration of remaining battery runtime. The application responsible for energy management can thus support the user with more accurate statements about remaining energy resources, which can help to avoid an uncovered demand for energy.

4.2. Customization to use case 2

As motivated in Section 1.2, predictions about service availability can be used to improve task assignment by always selecting the most ‘promising’ device. As an example, service availability can be predicted by forecasting the position of devices, which provide such services and/or of devices, which need to consume them. However, developing an appropriate prediction model for the introduced scenario is not trivial because predictions about arbitrary services with different characteristics have to be supported. The main problem is that the particular type of service can only be

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dmx:predictionModel
xmlns:dmx="http://context.vais.org/predictionModel"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <generalSettings>
    <timeStepLength>1000</timeStepLength>
    <recentHistoryFrequency>5</recentHistoryFrequency>
    <recentHistoryLength>7</recentHistoryLength>
  </generalSettings>
  <nominalSettings>
    <nominalType name="position">
      <allowedValue value="home"/>
      <allowedValue value="office"/>
      <allowedValue value="elsewhere"/>
    </nominalType>
  </nominalSettings>
  <measuringAdapterSet>
    <measuringAdapter name="PositionMeasuringAdapterFactory">
      <factoryClass>org.examplePhoneCall.measuring.PositionMeasuringAdapterFactory</factoryClass>
      <timeTolerance>1100</timeTolerance>
    </measuringAdapter>
  </measuringAdapterSet>
</dmx:predictionModel>
```

Listing 2. Excerpt from the XML prediction model of use case 1.

determined at runtime, that is, close to the actual prediction request, because it depends on the task to be executed.

However, the specific functionality and its underlying technology are especially the key influencing factors for service availability. As an example, a printer service is often available at the office, sometimes available at home, and mostly unavailable outdoors. In contrast, an SMS is likely to be available outdoors (because of better network connectivity). Other services are only available in a particular time slot, for example, only at daytime and on weekdays. On the basis of these observations, it is thus not possible to predict the availability for (arbitrary) services in general, but a customized prediction for each type of requested service is necessary. However, because of the
Fact that memory capacity is limited and that the number of possible services is infinite, this is not possible to realize. Additionally, because of the resulting network load, service availability cannot be measured regularly, so it is, for example, not possible to simply predict the availability of a service with a Markov chain using its past availability.

On the basis of the SCP, the described problem can be approached in the following way: Figure 12 shows a structure that is suitable to capture general dependencies for the context of service availability. Here, service availability is assumed to depend on the total number of devices within the local network as potential service providers and on the time of day and the position of the consuming device. Thus, these four variables and the dependencies between them basically constitute the basic part of the prediction net. The mapping of the original four variables of the example to multiple variables in the net results from the selected prediction methods and the use of coprocessing. The service availability (SA) at a specific point of time is predicted by using four methods in parallel. Similarly, the position at a specific point of time is predicted by using the time of day at this point of time as well as the position at the preceding point of time (Markov chain) and by extrapolating the position. As most of these variables are not numerical, the prediction model uses probability tables as the main prediction method of the reference configuration. In addition, also more specialized methods are used (e.g., such as a method realizing a majority vote and a method for determining the time of day as a periodic variable).

On the basis of the developed structure, the resulting prediction model (omitted because of space restrictions) can be divided into two parts: first, a service-independent part consisting of general knowledge about time ($V_1$–$V_3$), location ($V_4$–$V_9$), and net size ($V_{10}$) and second, a service-dependent part containing all variables, which provide information about service availability ($V_{11}$–$V_{16}$), respectively. The service-independent part (and its corresponding knowledge learned at runtime) can be re-used for each service regardless of its type or functionality. If a new service is detected, only the service-dependent part has to be updated, that is, a new entry has to be inserted for each service. The detection of services can be actively triggered in a periodic time interval or can be incorporated within the existing context management system, for example, by inserting a service passively only if it is used by the respective device anyway. The first strategy collects more information about service availability even if they are not relevant for the local application itself. The
second strategy better fits Dey’s and Abowd’s definition of context [1, p. 6] where only the relevant information should be considered (cf. Section 1).

The basic prediction model is distributed to all participating devices, for example, belonging to a group or company, and the prediction system starts to collect training data according to one of the above-mentioned strategies. However, the developed prediction model does not yet solve the problem of ad hoc predictions for unknown services. Therefore, the approach of SCP provides the possibility to integrate external knowledge at runtime (cf. Figure 11). If it is known that, for example, a printer service is always available at the office, then this knowledge can be incorporated. Considering the example, a new entry can be inserted into the probability table leading to $V_{15}$ (service availability by place), which states that the probability for service availability of printers at the office is 100%. In the case of mobile business processes, such service-dependent information can be carried along with the process description and can be distributed at runtime when a prediction about the particular service is requested. Combined with the knowledge already learned at runtime (e.g., the probability for reaching the office in the next 30 min), accurate assumptions can be made even for services that were unknown so far.

5. EVALUATION AND EXPERIENCES

Existing theoretical considerations and practical experiments showed that—compared with the application of a single method—chain processing of suitable methods can achieve considerable improvements regarding efficiency and accuracy [27, pp. 21–22]. Multiple methods used in coprocessing can achieve more accurate results if they are likely to produce different kinds of errors [28, pp. 1–4], [33, p. 656]. As an example, estimation uncertainty and method uncertainty may eliminate each other [28, pp. 1–4], [33, p. 656].

The evaluation presented here does not repeat such comparisons, but focuses on the applicability of the framework for different application contexts (i.e., static and dynamic applications in mobile environments). Therefore, a prototype of the generic prediction system according to the approach of SCP has been implemented for the Java Micro Edition™ and was applied to the two use cases motivated in Section 1. In particular, the framework is used by the existing DEMAC-middleware [5] for more than 6 months in order to enhance the distribution of mobile processes by the prediction of service availabilities. This section presents the experiences with the developed prototype prediction system and the use case elaborated in Sections 1.2 and 4.2 and concludes with a general conceptual evaluation.

On the basis of the developed prediction net (cf. Figure 12), the example scenario consists of realistic historical data about the behavior of a user and its mobile device spanning an interval of 7 days (Monday to Sunday). It contains the net size, the time of day, the position, and the service availability as values of the corresponding variables at different points of time, representing context data measured by real sensors. For the first practical experiments, two services with different behaviors have been chosen: a stationary printer service is regularly available when the user is at work (Monday to Friday, at daytime). An ad hoc file exchange service is offered spontaneously by a few mobile devices carried by other people in the direct vicinity of the user and is thus only available very infrequently.

The quantitative evaluation covers accuracy and efficiency. High efficiency means that the ratio of resource consumption and quality of results is appropriate. In general, appropriateness depends on the target platform as well as on the value of predictions for the individual application and thus has to be defined by the application developer. Here, the evaluation is run on an average notebook (1.5 GHz, Pentium M processor; Intel, Santa Clara, CA). The efficiency of the prediction system and of the created prediction model is determined by measuring the resource consumption and the accuracy of prediction results. All results are based on predictions about the availability of a service as a boolean variable at a specific point of time.

With the use of the methods of the reference configuration, the memory requirements are bounded and do not significantly increase because the instance data, that is, the knowledge learned by the prediction methods, is saved instead of measured raw context data. Here, the memory consumption is dominated by probability tables. Thus, the upper bound of memory required for the instance data
depends on the number of variables connected with the method and the number of their possible values in the prediction net. The maximum amount of memory required for the instance data in the example scenario is about 20 kB, and the processing time for learning is insignificant (i.e., considerably less than 1% CPU load). The processing time of a prediction depends on the number of variables in the prediction net, the number of prediction rounds, and the number of time steps in the time interval, which is taken into account for the prediction. Theoretical considerations show that—regarding these dependencies—the time complexity is linear. This result is also confirmed by practical experiments regarding the number of time steps in the time interval (cf. Figure 13) and similarly the number of prediction rounds (time consumption ranging from 13 ms for 50 rounds to 117 ms for 500 rounds if a prediction about service availability is requested 60 min ahead). The results indicate that also such relatively complex predictions take less than 1 s and, thus, the resource consumption is relatively well suited even for average or less powerful mobile devices (e.g., smartphones).

The analysis of accuracy begins with the ‘empty’ prediction system, which has no knowledge learned at runtime yet. In the course of time, the system learns from the current values of the historical data. Predictions are executed concurrently. The achieved accuracy is determined by comparing the predicted probability of a service’s availability with the actual availability (as boolean value) described in the historical source data. A prediction is considered to be correct if the prediction result states that service availability is probable (resp. improbable) and the service is actually available (resp. unavailable) in the future. Figure 14 shows the accuracy of predictions about the availability of the two service types at different days. Because the (more simple) ad hoc file exchange service is often unavailable, this regularity can be learned quickly and predictions about its availability already start with relatively good results, that is, predicting that the service is not available is correct in most cases. Furthermore, in the following days, the system learns to distinguish the availability of the service and the accuracy slightly increases. Also, the prediction results about the availability of the printer service improve very quickly. Thus, at Tuesday, the system is already able to predict that the printer service is available when the user is at work.

However, the predictions about the ad hoc file exchange service are still not completely satisfactory, that is, a trivial prediction approach always predicting that the service will be unavailable would not be significantly worse. Therefore, an enhanced solution makes use of the full potential of coprocessing by automatically preferring the methods with smallest uncertainty arising from the prediction and thus improve the adaption to the individual user. In the case of the ad hoc file exchange service, for example, the net size plays a more important role than the time of day and the position that both currently rather disturb predictions. Considering the printer service, it can be noticed that the accuracy of predictions at weekend is not satisfactory, because the day of week is not included in the simple version of the prediction net. The prediction net is therefore enhanced by such variables and by additional dependencies (e.g., the net size is dependent on the location and/or time) in order to further advance prediction quality.

Concerning the update of domain-specific knowledge, two cases have to be distinguished: first, data related with the current configuration can be updated on top of the current prediction model.

Figure 14. Correct predictions per day about the availability of a printer service (solid line) and an ad hoc file exchange service (dashed line).
and its values already learned. Such information can be exchanged in course of the distribution of context data among mobile devices (e.g., based on context ontologies) or can be passed by the application itself (such as applied in the mobile business process scenario, cf. Section 4.2). Except for the optional message exchange, such adaptations do not cause any notable overhead. Second, missing information on relationships within the prediction net have to be integrated by (manually) updating the prediction model and, if necessary, by adding respective prediction methods to the prediction framework. Although the prediction net can be updated at runtime (i.e., by uploading a new version of the XML prediction model), a new learning phase is necessary in order to capture a sufficient amount of data about the new relationships. To avoid such updates due to a faulty design, we propose to test the prediction model before deploying the application. Testing is supported by a quick run option of the prototype implementation on the basis of predefined dummy data. During this testing phase, the occurrence of expected results can be verified with standard testing software, e.g., JUnit, so that testing can be aligned to the individual goals of the application developer.

As introduced in Section 4, the developed prediction net provides the possibility that parts of the learned knowledge can be re-used, so that only few information has to be saved for each new service detected at runtime. Figure 15 shows the long-term application of the configured prediction system with an advanced version of the prediction net for service availability (cf. enhancements proposed above). Each day, the observed device detects a couple of services that are either known (blue blocks) or which are new (red blocks). The service-independent knowledge (dashed green line) is not influenced by the number of detected services and consumes about 100 kB for the advanced version of the prediction net. However, the amount of memory needed for the service-dependent knowledge increases only very slowly (solid green line) in comparison with the total amount of services detected as new (red line). Although such scalability is already a good result (especially regarding ad hoc networks with only a few number of services), it can further be improved by deleting old information about infrequent services that are not expected to lead to accurate prediction results anymore. As an example, old knowledge about a service that is newly detected in November and is never again detected during the next few months is—in most cases—no more relevant for a prediction in May. If such useless knowledge about infrequent services is deleted regularly, the amount of memory required for service-relevant knowledge can also be kept near constant.

In summary, the fundamental principles proposed by the approach of SCP are appropriate to fulfill most of the requirements as identified in Section 1. First, the system supports a wide range of applications and diversity of exploitable contexts. They enable a configuration of the prediction system, which meets application-dependent and variable-dependent demands (e.g., originated by different scale types and dependency types) because methods can be chosen flexibly according to the domain-specific knowledge. Additionally, variables as attributes of entities constitute a generic metamodel in order to capture the application domain. Thus, many different applications with diverse demands for their domains and possible contexts are supported and even ad hoc predictions for dynamic applications are possible. An adaptation to the user takes place by adaptive online learning. The effort for application developers is decreased, because the system coordinates the methods and offers a reference configuration of already implemented methods, which can be extended in the future.

Figure 15. Experiences from a long-term application.
Nevertheless, compared with Mayrhofer’s and Sigg’s approach, the effort for application developers is still high because domain-specific knowledge is used and has to be incorporated. However, this compromise allows for facilitating the prediction task at runtime and for enabling high accuracy and efficiency without being limited to a special application domain or to static design time configurations. The possibility to select appropriate methods for the considered application domain is a positive aspect not only in face of generality but also in view of accuracy and efficiency. For each variable, the best method to handle the characteristics and regularities determining the value of this variable can be selected. Efficiency can in particular be enhanced if no dependencies are expected between a set of variables or dependencies are known to be unstable. Finally, the development of an appropriate prediction model can ensure scalability and applicability in the context of heterogeneous devices. For example, developers have the possibility to select methods with only a small demand for resources in case the application is targeted to be run on resource-restricted mobile devices.

6. CONCLUSION

As a further step towards realizing dynamic pervasive environments, the approach of SCP proposes and realizes a context prediction system that is generic and provides potential for high accuracy and efficiency at the same time. In consequence, this approach provides a new combination of characteristics compared with existing approaches as analyzed in Section 2 (cf. Figure 3). In contrast to previous approaches, the composability of prediction methods and the integration of domain-specific knowledge as proposed here enable support for a wide range of new (or extended) applications and thus allow integration into existing middleware frameworks. As shown here, even dynamic applications can be supported because application-specific knowledge can be distributed and exchanged at runtime.

However, there are still some open research tasks regarding the usability of the developed framework. In particular, reducing efforts for application developers constitutes a major challenge to further facilitate the prediction of future contexts. Thus, the development of supporting tools for the creation of prediction nets, to facilitate selection and configuration of methods and to automatically transform the resulting prediction model into the XML representation, is an important future step. Furthermore, the conceptualization of advanced methods and appropriate tools to support the evaluation of the prediction models’ correctness and effectiveness is an interesting and necessary aspect for future work.

ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

REFERENCES

An Architecture and Methodology for a Four-Phased Approach to Green Business Process Reengineering

Alexander Nowak, Frank Leymann, David Schumm, Branimir Wetzstein
University of Stuttgart, Institute of Architecture of Applications Systems,
Universitätsstrasse. 38, 70569 Stuttgart, Germany
{Firstname.Lastname}@iaas.uni-stuttgart.de

Abstract. Sustainability and responsible resource exposure has become a major issue in everyday life. Government, customers, and increasing social responsibility force more and more organizations to positively optimize their environmental impact towards a better, livable planet. In this paper we propose a four-layered architecture and corresponding four-phased methodology to enable organizations to (1) define ecological characteristics, (2) sense and measure these ecological characteristics, (3) identify, localize and visualize their environmental impact, and (4) help them to develop appropriate adaptation strategies in order to optimize their environmental impact without neglecting the organization’s competitiveness.

Keywords: Business Processes, Process Views, Process Monitoring, Adaptation, Environmental Impact, Green Business Process Reengineering

1 Introduction

The growing interest in environmental topics and discussions reflects that sustainability in general has become a major issue for organizations over the last years. The increasing awareness of customers and the general public for sustainability and environmental impact on the one hand and legislative requirements on the other hand motivate more and more organizations to keep track on their environmental impact [1,2]. Based on this demand organizations are forced to design environmentally aware business processes and therefore trace the environmental impact caused by them. However, this first postulates that organizations know which environmental impact (e.g., carbon footprint [3]) their business processes have in order to adapt more sustainable solutions to their processes [4]. As complex business processes may consist of several hundred activities [5] it is not easy to identify the relevant parts of the process that mainly drive the overall environmental impact due to the various influence factors relevant to the processes. Therefore, organizations need adequate technologies and methodologies to make their business processes more transparent with respect to their environmental impact. Subsequently, adaptation techniques need to be employed to decrease the overall environmental impact while ensuring not to significantly worsen the organizations economic objectives.
In previous work [4] we have discussed initial concepts and techniques focusing on green Business Process Reengineering (gBPR) which extends the Business Process Reengineering (BPR) originally proposed by [6] and [7]. They describe BPR as the analysis and design of work flows and processes within and between organizations. In [7], BPR is also promoted as fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance. The problem is that currently neither BPR formerly described in [6] and [7] nor modern approaches like [9] do cope with green requirements adequately. This leads to a gap of missing interconnection between existing standalone solutions for efficient resource usage and a holistic optimization of an organization’s environmental impact. In most cases, the information gathered for traditional Key Performance Indicators (KPIs) provides insufficient data with respect to environmental aspects. Consequently, there is a need for concepts and technologies to define and monitor green efficiency metrics and to provide this information for analyzing and optimizing the processes properly. Given this information these approaches are faced with a further issue. Green requirements may end up in a trade-off with existing KPIs like costs or time and may change the current “best practices” when considering both KEI and KPI dimensions. In order to support the complex types of business objectives containing economic and ecological objectives we need to extended traditional BPR by introducing two novel perspectives in our gBPR approach. The first one contains the so called Key Ecological Indicators (KEIs). Using these KEIs allows measuring the environmental impact of business processes and parts thereof. This concerns for example the energy consumption, water consumption, CO₂ emission, carbon footprint, recycling, or regulatory compliance and thus forms the motivation for changing the business processes. The second added perspective covers additional management activities emerging from the integration and interaction of the KEI, process, and infrastructure perspectives [4]. This concerns for instance the mutual influences of the process structure and its underlying infrastructure.

In this paper we propose an architecture and methodology to address the current lack of supporting green requirements adequately. Consequently, the contribution of this paper is twofold: Firstly, we introduce an architecture that includes four layers to serve the different aspects of gBPR: (1) Strategy, (2) Sensing & Monitoring, (3) Analysis & Management, and (4) Adaptation. This architecture covers the proper monitoring, analysis and adaptation of green reengineering approaches and thus helps organizations to identify the relevant aspects for optimizing their environmental impact. The implementation of this architecture in a service-oriented environment is ongoing work. Secondly, we introduce a methodology to enable the process stakeholders to reduce the environmental impact utilizing the proposed architecture. The remainder of this paper is structured according to the phases of gBPR: Section 2 introduces the architecture in general. Section 3 explicitly describes the four architecture layers and their corresponding methodology support. In Section 4, the key concepts are applied to a concrete scenario. Section 5 positions our approach to the existing literature. Finally, Section 6 concludes the paper and outlines future work.
2 Architecture

Business processes of organizations are dependent on various internal and external parameters, such as the organizational structure or legislative regulations. Thus, in order to achieve best possible decrease in its environmental impact, it is essential to consider business processes from an end-to-end perspective, including their underlying infrastructure as well as the people or other resources that perform the associated activities. To best fit these requirements and to provide a holistic perspective on organizations’ processes we propose an extended BPR architecture and a four-phased methodology based on the initial gBPR concepts [4]. Our architecture comprises four major layers which are shown in Fig. 1: (1) Strategy, (2) Sensing & Monitoring, (3) Analysis & Management, and (4) Adaptation. The arrows between the different layers indicate that relevant data is provided from each layer to its successive layer. Details are explained in Section 3.

The first layer “Strategy” is used to identify and define appropriate KEIs which reflect the ecological objectives and traditional KPIs which reflect the economic objectives of an organization. KEIs are defined based on a set of ecological metrics (e.g., CO2 emission, water consumption etc.) to be measured and the specific thresholds that apply for a complete process or single activity, respectively.

The measurement of KEIs is performed in the “Sensing & Monitoring” layer. At this level we assume that monitoring of KPIs is done in an appropriate way using given methodologies and technologies, e.g., [11]. However, due to the wide range of possible KEIs that one might consider, the information gathered for determining those KPIs may be insufficient and additional information for determining the KEIs is needed. For the measurement of KEIs, the ecological characteristics of processes and activities have to be determined explicitly. In some cases, that information can be extracted from service or product specifications at design time. In the general case, however, special sensors are needed which monitor the ecological characteristics of, for instance, IT systems, manufacturing operations, human activities, ecosystems, facilities and buildings, or logistics at process runtime. That sensor information has to be correlated with process instances and activities which use the corresponding resources. As a result, the process instances and activities contained in them are annotated with sensed ecological metrics.

The third layer “Analysis & Management” forms the heart of making processes “greener” as it allows us to analyze processes and subsequently identify the parts of a process that cause the highest negative environmental impact. To reveal this information, we utilize process views as introduced in [10]. By means of augmenting the process model with ecological information from layer two (Sensing & Monitoring) we are able to build virtual views on a process and identify and visualize the KEIs of either the complete process or specific activities of interest. This enables analyzing the current environmental impact of a process model, identifying the main cause of defined KEI violations, and finally revealing the room for ecological improvement.
If the room for ecological improvement is identified, a business process can then be reengineered in different ways. This is represented in layer four “Adaptation”. For example, an organization may decide to use a cloud infrastructure instead of their on-premise solution, or to use a new truck (i.e., resource) performing a specific delivery activity. An organization may also decide to introduce a new structure of the process model, rethinking the existing organizational structure [7]. At this point, a very important issue concerning the reengineering of process models is to keep track on given KPIs and economic objectives. An organization mainly focusing on cost aspects, for example, may be limited in adopting different services or substituting resources. As economic KEIs are also augmented to the process model we are able to directly compare the impacts of modifications through generating different process views. Based on these different views, a decision for reengineering the process in the analyzed way can either be made automatically or manually. Consequently, the concrete restructuring of the process model can be performed. Again, based on the variety of the KEIs, the restructuring can be manifold. Depending on the kind of restructuring we can utilize approaches common in the field of adaptability, like (1) changing the flows of a process model, (2) changing the underlying infrastructure or resources, (3) add, remove or modify (groups of) activities, or (4) introduce dynamic provisioning of activities.
3 Four-Phased Environmental Impact Management

To illustrate our proposed architecture and methodology we use a motivating example describing the company *Auto Inc.* that manufactures premium cars. Due to internal policies *Auto Inc.* would like to decrease their CO₂ emissions caused by the manufacturing of each car of a series. Based on this information *Auto Inc.* must buy a proper amount of emission permits. If they exceed these emission permits, they need to buy an additional contingent from companies that require fewer permits, otherwise if they use less they can sell their permits, respectively. This is also known as "emission trading" [12]. This regulation provides a significant economic incentive for reducing an organizations collective CO₂ emission. In the following, we first use an abstract process to better describe the methodology and different steps the various process stakeholders (see Fig. 1) of *Auto Inc.* need to perform in order to “green” a process. A use case describing a simplified but concrete process is shown in Section 4.

3.1 Strategy and Sensing & Monitoring

The environmental impact of a business process can be assessed in terms of a set of KEIs. These KEIs are defined based on so called *Ecological Characteristics (ECs)* such as energy consumption, water consumption, CO₂ emission, recycling, or regulatory compliance. We define KEIs as a tuple consisting of an EC metric and a target value function based on the ecological goals one wants to achieve (defined by business strategy). For example, a KEI for a particular business process could be specified as "max CO₂ emission (of a process instance) < X_i". Therefore, the definition of a KEI is very similar to KPIs; the difference is that the underlying metric definition is based on EC characteristics and involves new information sources, while in case of KPIs the underlying metrics concern time, quality, or cost perspectives [11]. In order to assess the KEIs, the underlying metrics have to be measured for the performed business process instances in the “Sensing & Monitoring” layer. For the calculation of an EC referring the whole business process, we need to collect the needed data of each process activity. For example, in order to assess the CO₂ emission of a whole business process instance, we need to know the CO₂ emissions of each executed process activity in that process instance and then sum up those emissions. The collection of the needed measurement data per activity can be performed in different ways. In the simple case, a process activity has always the same EC metric value across all process instances and that value can be obtained dynamically from a service specification or a *Service Level Agreement (SLA)* if, for example, the process activity implementation is provided by an external service provider. Otherwise it can also be obtained statically from previous experiences or existing know-how. In that case no monitoring is needed.

If the EC metric value of an activity is not known at design time, it has to be monitored while performing the process instance. Therefore, we first have to determine the resources which are used by that activity and affect the needed EC. Then, at runtime we need to obtain and aggregate sensor data which reflects the EC consumption of those resources and correlate it with the process activity of the
specific instance. A specific correlation and differentiation has to be done if resources (e.g., IT infrastructure or transportation vehicles) are shared between different process instances and different process activities. The Sensor data can be provided in an automated fashion, in particular if sensors are able to emit events to an event bus. In that case, complex event processing technology can be used to correlate sensor events with process instance events in a timely fashion. Sensor data can also be provided manually by humans who e.g. manually determine how much water an activity has consumed; this analysis can happen after the process instance is already finished (post-mortem).

After having performed measurements for a certain number of process instances, we can determine which EC value each activity is dedicated to: (1) a static value or (2) a dynamic value, whereby the value depends on the data input to the process activity and/or the duration of the process activity, e.g., the emissions of a printer depend on the number of pages that should be printed (data input). In this case the monitoring in future can be performed on process level only, because a static factor (EC metric value / page) is combined with a dynamic factor (number of pages) obtained on the process level. The calculation function can then be determined by using regression analysis. Additionally, we have identified two more types of EC values, namely (3) a mixed value as a combination of (1) and (2), and (4) a dynamic value which depends on external factors and always should take into account appropriate sensor data. That information can be saved in a repository and used later, for example, if those activities are re-used in other processes. This would imply a change in the type of EC value of a specific activity or process fragment [13]. In order to use these different types of EC values in the subsequent analysis phase, we calculate average values based on the available process instances.

3.2 Analysis & Management

The information collected in Section 3.1 provides the basis for analyzing and managing the existing organizational processes by facilitating the identification and localization of vital KEI violations. In order to localize and finally visualize the cause of a KEI violation we use the concept of process views introduced in [10]. A Process view results from one or more specific transformations applied to a process model and therefore enables the analysis of processes from different perspectives. The transformations can be of an augmentation, structural or visual type, for example. Depending on the underlying information, the use of process views is one promising approach to address various important questions: Which activities make a significant contribution to the overall carbon footprint and energy consumption? What is the overall environmental impact and how would it change due to particular modifications of the process model? Which parts of the model are allowed to change? How can inter-organizational savings be achieved? To answer these questions we combine different transformations performed in several succeeding steps. Referring to our running example, the steps and their corresponding transformations Auto Inc. needs to perform in order to analyze a specific manufacturing process are described in the following. We now assume that Auto Inc. tries to achieve a more sustainable process and therefore the top management announces the decrease of CO₂ emission of
Process P which consists of nine activities, A₁ to A₉ (see Fig. 2, left). The management has further defined the new CO₂ emission thresholds X₁ to X₉ for each activity A₁ to A₉. Based on this information the process stakeholders from the architecture layer two (Sensing & Monitoring) provide the required information to support the KEI “CO₂ emission”. However, the data provided comprises both, economic and ecological information that are properly correlated to the process model. So, this data can also be used for the enrichment of existing business dashboards that represent the current state of the process instances and enables stakeholders to initiate proper actions when detecting KPI or KEI violations.

Now, as a first step when detecting a KEI violation the given process model needs to be augmented with related data (see Fig. 2, center). This is a fundamental step which is a prerequisite for all further steps or view transformations in general. The augmented process model now contains all relevant information about the processes’ KPIs and KEIs to proceed with the next step. In our example we will first use the information provided by the KEI data in order to identify which activities exceed the thresholds defined from the management. To visualize the activities with the highest amount of CO₂ emission we perform another transformation. First, we use a visual transformation that omits all activities where the augmented CO₂ emission is below their dedicated threshold Xₙ. As a next step, we additionally omit all activities that cannot be changed or outsourced per se. This can, for instance, be due to privacy concerns or legislative requirements and varies in each particular use case. The result of the omission of the activities is shown in Fig. 3 (center).

Based on the activities left, Auto Inc. can begin to identify and localize the activities with the highest amount of CO₂ emission. We can support the human readability by generating a so called heat-map, for example. This visual transformation changes the color of the shapes of the process view depending on their augmented CO₂ emission. A dark red color is equivalent to a high CO₂ emission and a light orange is equivalent to a lower CO₂ emission, respectively. Within this transformation step, we can also change the size of the activities and add the percentage value each activity exceeds
their threshold. The performed transformation steps and corresponding views are shown in Fig. 3.

*Auto Inc.* can now locate the activities with the highest CO₂ emission, represented by the corresponding colors, the size of the shape, and the CO₂ emission values inside the shapes. However, it might be feasible to “zoom in” deeper, i.e. to collapse activities for allowing to view or directly change the interior of an activity. As an example, we want to have a more detailed look at the big red activity in the left side of Fig. 4 which exceeds their CO₂ threshold by 15 percent. After performing the drill-down transformation the right side of Fig. 4 shows the sub-activities that are performed within the big red activity on the left side and their contribution to the overall CO₂ emission. Of course, the sub-activities can also be further sub-processes that are again shown in an aggregated way. Note that a viewing scenario that supports collapsing requires the augmentation of the process with runtime or deployment information about the actual implementation of an activity. Consequently, the visualization function could then visualize the information about the interior in resulting graph-like structures (see Fig. 4) or even drill down to the bits (in case of an IT process). An important issue concerning the drill-down methodology is to provide sufficient technologies for disaggregating and aggregating the overall KEI or KPI values. First approaches in this area are proposed by [14,15].

![Fig. 4. Activity Drill-Down](image)

### 3.3 Adaptation

Knowing the most dissipative activities with respect to the observed KEIs we can develop appropriate adaptation strategies that optimize these KEIs. In market environments, however, we need to ensure the competitiveness of an organization beyond the adaptations for “greener” and more sustainable processes. Ecological characteristics are often in sharp contrast to strategic and economic objectives. However, there may also occur situations that influence traditional KPIs, i.e., cost, quality, and time in a positive manner, sometimes even without extensive upfront investments. Using a computer-based e-Fax solution for supplier contact, for example, makes *Auto Inc.* reduce their CO₂ emission due to the abdication of extra hardware, but at the same time reduce their costs and time using this service. So, this trade-off is no novel appearance and can also be found at traditional KPI research, but now we have to consider a fourth dimension: the environment.
In our approach, process views also provide the means to develop and visualize different adaptation strategies. Within the development of adaptation strategies we first want to distinguish between structural and non-structural adaptations. A non-structural adaptation does not influence the structure or logic of an observed process model, but has influence on the augmented information (e.g., attributes, resources, or underlying infrastructure of the activities). The change to a supplier of electric energy providing green electricity, for example, may lower the CO₂ emission of particular activities without changing or restructuring them. However, the attributes augmented to the activities change. Structural adaptations on the other hand are dependent on the range and characteristics of the planned reengineering. Thereby, several process optimization techniques known from BPR are feasible to optimize the KEIs of the observed process. These include, but are not limited to: (1) New binding of services implementing a process activity, (2) changing the underlying infrastructure which better adapts the process characteristics, (3) changing the flows of a process model, (4) rearrange activities, i.e., add, remove or modify (groups of) activities, or (5) introduce dynamic provisioning of activities. Utilizing these techniques provides a wealth of opportunities for making a business process more sustainable and can therefore be fully applied to our approach. So, the adaptation strategies we may use here can constitute either of a complete reengineering approach including the creation of a new process model, the modification of specific activities or resources, or an arbitrary combination in-between. Furthermore, structural and non-structural adaptations can also be combined.

In order to determine the impact of an adaptation strategy we need to calculate the aggregated values of both, the corresponding KEIs as well as the corresponding KPIs for each adaptation strategy. The KPI values can be determined in an analogous manner and provide the means to compare the different adaptation strategies. In our running example, Auto Inc. decided to substitute the activities with the highest CO₂ emission by activities from external providers with a lower CO₂ emission. In this case, we may also consider constraints when exchanging activities. For example, a specific activity that improves the CO₂ emission might exceed a given cost threshold and therefore cannot be used as substitute. Two different adaptation strategies are shown in Fig. 5. The upper one substitutes the two activities identified at the drill-down, the lower one substitutes the complete activity identified before drill-down.

**Fig. 5. Adaptation Strategies**
The numbers shown exemplarily depict the impact of each adaptation strategy based on KEIs and KPIs.

In these adaptation scenarios, the information needed for the augmentation of the substitute activities needs to be provided either by previous analysis, certain know-how, information provided by business partners offering the alternative service (SLA), or other estimations. Note, that the comparison of different adaptation strategies is only as valid as its underlying estimations. Therefore, it is crucial that the data used for the augmentation is as accurate and current as possible. When comparing different strategies with one another, equivalent data is necessary for both processes in question. Otherwise, the comparison might lead to non-representative results. If the information concerning KEIs and KPIs is in a proper shape, a concrete adaptation strategy can be chosen. Considering the given thresholds for economic and ecological objectives an organization, for instance, can choose a strategy that satisfies the economic and even optimizes the ecological objectives. So, in our running example, Auto Inc. compares their adaptation strategies from Fig. 5 to one another, deliberates about which strategy best fits their overall economic and ecological objectives (i.e., their business strategy) and finally decides in which way to adapt the observed process model. Depending on the process characteristics (i.e., whether the observed process is an automated process or an undocumented process, for example) proper adaptation mechanisms may be selected to support the adaptation strategy in detail. In general, we are faced with similar issues known from Life Cycle Assessment (LCA) [17]. LCA is also a methodology for analyzing commensurable aspects of quantifiable systems. However, not every KEI value can be reduced to a number and augmented to a process model. In our approach this holds for recycling aspects or soil pollution, for example.

4 Use Case

To illustrate the practicability of our approach we use a concrete business process example from a car manufacturer. In order to apply our methodology, we use the car finishing process depicted in BPMN notation [18] in Fig. 6. This process is performed every time a new car has been assembled and leaves the assembly line. In the first step of the finishing process the car is put into operation making sure all systems are working. Then, in the regular case a quick check based on a predefined checklist is performed. In some cases a detailed check is performed. This part of the process first includes the transportation of the car to the test center and the preparation of the test procedure. The test procedure then starts with an engine test which is followed by a detailed visual check of interior and exterior. After the test run on a test track in the next process step, the water density is checked and the car gets cleaned and prepared for delivery or refinishing, respectively. Finally, in both cases a detailed report of the test results is created and sent to the operations management. Performing the finishing process either with a quick or a detailed test run results in different cost, quality, and duration characteristics of the complete process depending on the specific weights of those dimensions, e.g. the percentage of detailed tests that can be managed.
Now, an additional dimension, namely KEIs, is added. In the first step towards improving the environmental impact of this process we need to monitor and sense the required information in order to analyze and subsequently achieve the strategic objectives of decreasing both the CO₂ emissions and the water consumption by a certain percentage. The CO₂ emission can be estimated by identifying the means for the car transport to the test center, the fuel burned during the engine test and the test run, the electricity needed for light and apparatus of the visual check and during the water density check and cleaning, for example (note that concrete measuring methods are out of the scope of this paper). The water consumption can be estimated by water meters, respectively. The complete environmental information, beside other KPIs like cycle time and process costs, is then augmented to the process model and can be used for further process analysis.

Analyzing the augmented data we are noticing that our KEI targets are not reached with this process. To identify the activities that cause the main environmental impact we create a new process view. Therefore, we first omit those activities that must be performed as modeled according to internal guidelines. This includes the activities Prepare Detailed Check and Perform Quick Check. In order to provide a better readability we also perform a transformation that repaints the shapes depending on their CO₂ emission and water consumption. The result is depicted in Fig. 7. The intensity of the background colors indicates the amount of CO₂ emissions caused by
the corresponding activity. The thickness of the blue border line indicates the total water consumption of the corresponding activity. In Fig. 7 we can see, that the *Engine Test* as well as the *Test Run* activities produce a high amount of CO₂ emissions while the *Check Water Density and Clean Car* activity indicates both, a high CO₂ emission and water consumption. The information sign in the bottom right corner of each activity is used to display all information augmented to this activity (mouse-over). Based on this information we can identify the problematic activities and derive potential process alternatives.

In order to achieve the strategic objectives we decide to perform a structural adaptation of the process as depicted in Fig. 8. We include a new test run activity which performs the test at an existing roller dynamometer test bench and is executed as an alternative to the original test run activity. The test run is now performed in equal parts at both the test track and at the existing roller dynamometer test bench. The latter one allows a more efficient test run with respect to the test run duration and therefore reduces the fuel burned, for example. The alternative test run also eliminates the transport of the cars to the test track. Additionally, the cars are just handled indoors which reduces the amount of water and cleaning supplies for washing the cars. In order to visualize the total environmental impact of this process alternative, the augmented information of the related activities is overwritten, i.e. a new process view containing the new process model and its related information is generated. Within this use case, the information can be gathered either by some test runs of the specific activities or based on existing knowledge. Next, we also need to consider the process costs of the restructured process as well as the corresponding time and quality impacts. Before, the roller dynamometer test bench has not been used within the finishing process because it had worse impact on the KPI dimension cost and quality than the other test types. This now changes with adding the KEI dimension because we are faced to a new trade-off situation. Within the new case, the costs will slightly increase due to the higher costs of a test run at the roller dynamometer test bench. On the other side, we will save a significant amount of time for not transporting the car to the test track and the more efficient test run. What is important now is that we can also achieve savings at the water consumption and CO₂ emission. Based on the weights strategically set for those four dimensions we can try to determine the percentage of tests which should go to the new roller dynamometer test activity and configure the branching activity accordingly.
5 Related Work

Within the cross-cutting concern of this work different approaches considering the specific parts and areas of interest of this approach can be found in literature. Following [8], these approaches can be distinguished in Green IT and Green IS approaches. The green information technology (IT) is mainly focused on energy efficiency and resource utilization. Here, we can distinguish different approaches considering two main perspectives: (1) the hardware perspective [19,20] that covers the efficient use of resources, e.g., proper allocation of resources, and (2) the infrastructure perspective [21,22] that covers the efficient and target-oriented usage of an underlying infrastructure, e.g., proper management of cloud environments. Green information systems (IS), in contrast, “refer to the design and implementation of information systems that contribute to sustainable business processes” [8]. Consequently, the literature considers the software and process perspective. In [23] and [16] the authors have developed first concepts on how business processes can be optimized in a green manner. They focus on a classification of resources that influence the environmental impact and how they can be reduced during design-time of a business process. Additionally, they introduced a formal model dealing with the combination of quantitative and qualitative QoS in order to also consider non-numbered QoS. Subsequently, in [24] they focus on how these resources can be modeled. This approach contains interesting aspects regarding the green optimization of processes, however, considers only design time and is not focusing on an organization’s complete environment including the organizational structure, the processes, and the used infrastructure and resources.

A more general approach to assess environmental and social damages assignable to products and services is Life Cycle Assessment (LCA) [17] which is part of the ISO 14000 family “environmental management standards” [25]. It provides a technique to assess all impacts of a process from cradle-to-grave, i.e. from raw material to disposal or recycling. While it covers the whole product lifecycle, it can be used to optimize the environmental impact of a product or of a whole company. LCA provides a good basis for optimizing the environmental impact of an organization, however, does not focus on business processes in particular and the underlying infrastructure in general. Another interesting viewpoint is the research work done in “ecological information science” (in Germany this research area is called “Umweltinformatik”). Ecological IS deals with the modeling, simulation and analysis of ecosystems. They provide a lot of information on how harmful substances may spread or how control systems should behave to minimize the impact on ecosystems, for example. They also provide first ideas on business information systems considering ecological information in order to support operational decisions. However, so far they lack in applying their research results to the (IT) business processes layer and especially how business processes can be designed or adapted in order to prevent a negative impact on ecosystems.
6 Conclusion

The architecture presented in this paper describes fundamental layers needed to achieve more sustainable organizational environments in the cross-cutting concern of green Business Process Reengineering. We described each layer in detail, identified the roles within an organization responsible for each layer and sketched the main issues of each layer. Moreover, the corresponding methodology presented in this work describes a walk through this architecture. It helps organizations to plan and define their ecological objectives in form of Key Ecological Indicators (KEIs) and to identify and localize the most dissipative parts of their processes based on these KEIs. To realize the Analysis & Management as centerpiece layer of the architecture we used the approach of “process views” that enables a proper visualization of the process model utilizing so-called view transformations. Consequently, in the Adaptation layer organizations can derive adaptation strategies to optimize their collective environmental impact while considering both, their ecological and economic objectives. Finally, we presented a use case from automotive industry that shows the practicability of the proposed architecture and methodology. Our approach bridges the gap of missing interconnection between existing Green IT and Green IS approaches towards a holistic environmental impact analysis and optimization in organizational structures. In our future work we will investigate a classification for KEIs and their application in intra-organizational and cross-partner environments. Within this work we will also address the problem of how to sense and monitor the environmental influence factors on a per task basis. We will further develop different process view patterns that allow organizations the application of process views in a re-usable fashion and will devise algorithms that support the trade-off between KPIs and KEIs.

Acknowledgments

Parts of this work were funded by the 7th FP EU-Project S-Cube (Grant Agreement No. 215483) and the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart. We also would like to thank our colleagues at the IAAS for the fruitful discussions.

References


Tweetflows - Flexible Workflows with Twitter

Martin Treiber, Daniel Schall, Schahram Dustdar
Distributed Systems Group, TU Vienna
Argentinierstrasse 8, A-1040 Wien
{lastname}@infosys.tuwien.ac.at

Christian Scherling
ikangai solutions
Treustrasse 59/5/20
A-1220 Wien
office@ikangai.com

ABSTRACT

We present a lightweight coordination and collaboration platform, intertwining contemporary social networking platforms and SOA principles. The idea of our approach is to use Twitter as a platform for collaborations of human and software services in the context of workflows. We introduce primitives that provide SOA functionality like service discovery or service binding and illustrate how these primitives are embedded in Tweets. By using Tweets, we are able to reuse existing infrastructures and tools (e.g., twitter clients on mobile devices) for the communication between services and humans. Simultaneously, we exploit social network structures originating from Twitter follower networks in order to discover (human and software) resources that are required for the execution of a workflow. Finally, we are able to monitor the execution of workflows with Twitter, simply by following Tweets that represent the execution of a workflow.

Categories and Subject Descriptors
C.2.4 [Distributed Systems]: Distributed applications; H.3.5 [Online Information Services]: Web-based Services; H.4 [Information Systems Applications]: Miscellaneous

General Terms
Design, Human Factors, Management

Keywords
Crowdsourcing, Twitter, Social Networks

1. INTRODUCTION

Over the past years, the Web has transformed from a pool of statically linked information to a people-centric Web. Various Web-based tools and services have become available enabling people to communicate, coordinate, and collaborate in a distributed manner. Crowdsourcing has emerged as an important paradigm in human problem solving techniques on the Web. More often than noticed, software services outsource tasks to humans which are difficult to implement in source code. Applications range from enterprise environments [27] to open Internet-based platforms such as Amazon Mechanical Turk [2] (M Turk). These online platforms distribute problem-solving tasks among a group of humans. Crowdsourcing [12] follows the ‘open world’ assumption allowing humans to provide their capabilities to the platform by registering themselves as services. Some of the major challenges are: monitoring of crowd capabilities, detection of missing capabilities, strategies to gather those capabilities, and the tasks’ status tracking [3]. Service-oriented architecture [1] (SOA) enables the design of applications that are composed from the capabilities of distributed software services. In SOA, compositions are composed of Web services following the loose coupling and dynamic discovery paradigm. Unlike traditional SOA-based approaches, we consider complex service-oriented systems that are established upon the capabilities of human and software services [21]. The integration of human capabilities in a service-oriented manner is motivated by the difficulties to adopt human expertise into software implementations. Instead of dispensing with human capabilities, people handle tasks behind traditional service interfaces. In contrast to process-centric flows (top-down compositions), we advocate flexible compositions wherein services can be added at any time exhibiting new behavior properties. Here, we focus on coordination and collaboration principles using social networks. Specifically, we focus on the convergence of service-orientation in crowds and social networking platforms.

A socially-enhanced approach for combining human and software services brings a number of advantages that have not yet been exploited in existing crowdsourcing platforms: (i) Seamless coordination in crowdsourcing environments is accomplished by embedding instructions and flow structure in communications of widely used social networking platforms. (ii) The discovery of collaboration partners and services is based on social phenomena [23]. Spreading and cascading information flows in social networks provide powerful mechanisms for the discovery of resources. (iii) A task’s progress can be monitored by observing traces of social interactions. Short infos and status updates that are associated with a particular context (topic) provide information regarding the current status of people and services. Here we present means and primitives for coordination and collaboration using the Twitter

1http://twitter.com/
Twitter, a service founded in 2006, has 106 million registered users by January 2011 and is adding new users at the rate of 300,000 a day\(^2\). The underlying principle is very simple: Twitter users post short messages (Tweets) about any topic within a 140-character limit and follow others to receive their Tweets. Being a follower on Twitter lets users receive all the messages from those he/she follows. Within the Twitter community, a specialized markup language [13] has evolved: RT stands for retweet, @ addresses a user directly in a Tweet, and # followed by a word represents a hashtag [5]. This vocabulary enables them to express their Tweets with basic semantics and enables users to search for topics represented by hashtags.

This way of being able to communicate in an informal manner may play an important role for collaborative work in organizations. Twitter provides a lightweight and easy to use tool for sharing work-relevant information among employees, supports the coordination of group activities and it supports potentially novel, previously not anticipated collaboration opportunities within organizations.

Such a lightweight communication framework is certainly of interest when addressing the requirements of software systems which consist of a mix of humans, human-provided services and software services. In such a setting, services are accessed from and provided by different devices (e.g., laptops, mobile phones) and locations (workplace, during travel) and consumed either by humans or by software. Consequently, a convenient communication infrastructure is needed that allows for a flexible handling of communication environments.

In this paper, we investigate the application of Twitter as a communication infrastructure [11]. Our work focuses on how to use Tweets to enable the seamless communication between humans and (human-provided) services in different environments. In particular, we illustrate how SOA principles [6] (i.e, loose coupling, late binding, and service discovery) work with Twitter and how to organize conversations between service consumers and providers on Twitter.

Structure of this Work. We introduce a motivating example in Section 2 that illustrates some of the key challenges for Twitter-based collaborations. Based on our working example, we discuss key design principles in Section 3. We continue with Twitter conversation primitives and show how to apply these in a service-oriented setting in Section 4. We conclude with a discussion on our approach in Section 5, related work in Section 6 and an outlook on future work in Section 7.

2. MOTIVATING SCENARIO

Our motivating scenario is a real world project \(^3\) which has the goal of porting two iPhone apps (qcard and qlauncher) to the Android platform for ikangai solutions. The project team consists of three people of ikangai solutions and four undergrad students from the Vienna Technical University. The project is embedded in a university lecture at the Vienna Technical University and lasts for four months.

A typical challenge in this kind of project setting is to define the communication channels between all project team members. The team members are distributed over three Austrian cities and travel on a regular base. Consequently, the collaboration infrastructure must be accessible from mobile devices (e.g., Android, iPhone, Netbooks) and should be as lightweight as possible. Thus, project-related messages need to be short in order to be easily accessed from mobile devices. Furthermore, the communication must be capable of tracking all project related messages and providing team members the ability to filter the content according to their requirements. The sharing of information (links, documents) needs also to be considered, as well as the integration of external (human provided) services that are needed during the project. For example, team members blog during the project about the project’s progress on the ikangai blog. Because not all team members write English blog entries, ikangai uses a (human-provided) translation service that translates their entries into English before the entry is posted on the ikangai blog. The very same (human-provided) service also checks ikangai blog entries for grammar mistakes before they are published.

To address these challenges, the ikangai project team uses Twitter as a communication infrastructure backbone. Tweets are used to exchange project-related information and track messages between team members. All project related Tweets contain hashtags that add meta information. The latter are used to filter Tweets for project-related messages, to create message hierarchies and to invoke external (human-provided) services. Moreover, in order to utilize the follower structure of the team members (e.g., other students doing the same lecture) for expertise on different topics (e.g., programming, translation), all project communication is made public. By publishing service requests and retweeting them to Twitter followers (the Twitter follower crowd), we are able to tap into available external knowledge resources, human-provided or software-based services.

3. KEY DESIGN PRINCIPLES

In our approach, we intend to position crowdsourcing in a service-oriented setting, applying concepts from SOA and Web services such as dynamic discovery and late binding mechanisms. In particular, we investigate how to use microblogging services like Twitter for the purpose of crowdsourcing tasks or services.

3.1 Tweetflows and Social Computation

We consider our approach as technical to social computation [18]. As such, social computation is centered around the idea of integrating humans and software into complex systems. One major challenge is the communication between software and humans. Our approach provides the means for a seamless communication between humans and services.

Workflow descriptions are typically complex descriptions of activities that are executed in a particular order to achieve a certain goal. Languages like BPEL or YAWL [24] are used to specify each step that is necessary towards the completion of the workflow. In contrast, we strive for the support of simple, ad hoc workflows which are descriptions of activities (e.g., service requests) that are required for the completion of a task and are constructed on the fly, so called Tweetflows. After initial ad hoc workflow Tweets, which bootstrap the creation and at the same time the execution of the workflow, periodically updates extend the workflow until its comple-

\(^3\)http://www.ikangai.com/blog/advanced-software-engineering-ase
3.2 Technology Grounding

In crowdsourcing environments, complex descriptions of services (i.e., service meta data) or service invocation mechanisms are not available. Thus, the process of service publication, service discovery, service binding and service execution cannot be applied directly, but also needs to follow different principles and use different technologies.

- By using Twitter as a communication means for crowdsourcing, we impose a set of limitations concerning the length and complexity of messages that are exchanged during service discovery or service invocation. Twitter, being a microblogging service, limits the amount of data that is published to 140 characters per Tweet. Since we want to keep a simple one to one mapping between a Tweet and a service related message, we need to limit the amount of data (and meta data) in a Tweet information to an absolute minimum. In order to minimize to the space needed by meta information, we draw upon Twitter’s hashtag mechanism to mark keywords that represent meta information in Tweets [13].

- The messaging mechanism of Twitter follows a broadcast paradigm which we use to publish service requests and service announcements. This is in contrast to having a centralized service registry that collects all service information which is queried for a service. Consequently, we observe that Twitter pushes service announcements instead of letting service requestors pull for service-related information.

- The addressing of services utilizes Twitter’s build-in addressing mechanisms using the @ symbol to send messages directly to followers. Followers represent service providers and are able to forward service-related requests to other followers. This also permits to provide the basic means for service communication.

With these limitations in mind, we propose to use a set of primitives to provide for the features to find, bind, execute and monitor services in Twitter-based crowdsourcing environments. Those primitives are directly embedded into Tweets using a combination of hashtags and a priori-defined commands. We foresee the use of external resources that contain the actual data for the invocation of services, because of the aforementioned limitation of 140 characters. In our approach, Tweetflows represent conversations between Twitter users, and employ pre-defined communication primitives (e.g., service requests, service bindings) for the invocation of services and the exchange of data. We will discuss the implementation of the processes in greater detail in the following sections, in which we introduce a set of communication primitives that provide the needed functionality to publish, discover and bind services, and to execute simple Tweet-based workflows.

4. TWEETFLOW COMMUNICATIONS

The communication and message exchange in Twitter follows a publish/subscribe pattern [7]. By following other Twitter users, the follower subscribes to the Tweets of the user that is being followed. Given that Tweets are publicly visible to followers, conversations can be tracked by other users and messages can be retweeted, i.e., forwarded to other Twitter followers. This schema supports an efficient spreading of news [17]. The discovery of services is an important issue in many service-oriented systems. By using a spreading mechanism, publication of service-related information becomes straightforward, but also causes some challenges, for example spam or information overload.

The limited length of Tweets demands for a specification of a compact syntax to enable communications and control. For this purpose, we introduce a set of Twitter primitives that enable a seamless fabric of human and service interactions. Our proposed Twitter communication syntax is based on established Twitter primitives, like RT (re-tweet). It not only allows users to request a particular service, but also facilitates the discovery of services in crowdsourcing environments. Table 1 shows the most essential Tweetflow primitives which will be used throughout this section. In our discussions we will establish the correspondence of Tweetflow primitives and concepts found in SOA.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>Service Request</td>
</tr>
<tr>
<td>CS</td>
<td>Claiming Request</td>
</tr>
<tr>
<td>RE</td>
<td>Service Response</td>
</tr>
<tr>
<td>RT</td>
<td>Retweet Service Request</td>
</tr>
<tr>
<td>DS</td>
<td>Delegate Service Request</td>
</tr>
<tr>
<td>TF</td>
<td>Tweetflow Pipes</td>
</tr>
<tr>
<td>RJ</td>
<td>Reject Service Request</td>
</tr>
<tr>
<td>ST</td>
<td>Service State Request</td>
</tr>
<tr>
<td>SE</td>
<td>Service State Reply</td>
</tr>
<tr>
<td>SP</td>
<td>Service Announcement</td>
</tr>
</tbody>
</table>

Table 1: Tweetflow primitives.

4.1 Tweetflow Primitives

In this section we detail communication principles using Tweetflow primitives and provide simple examples. A complete scenario will be discussed in a later section of this paper.

Passing Data to and from Services. The limitation of Twitter messages requires special considerations concerning the access of input and output data for services. To overcome the size limitation, we propose the use of external resources that represent the input and output of service invocations. Resources are accessed with a simple HTTP GET operation and the corresponding link is stored directly in the Tweet. This allows for great flexibility, because we are able to pass arbitrary information to services. The same applies to the result of service invocations which are represented by Tweets. Listing 1 shows how a (human provided) English to Japanese translation service is called with a blog entry being the input data.


Listing 1: Passing data to a translation service.

Announcing Services with Twitter. The publication of services with Twitter consists of posting a Tweet with the service name and meta information about the service. Since the available space is limited, we use optional links.
to external taxonomies to provide meta information about the service being published, in addition to Twitter hashtags (see Listing 2). We consider hashtags to play a similar role as tags in folksonomies [20] - a distributed, bottom-up classification schema for services which are made available through Twitter. In addition, the retweet mechanism allows to spread service announcements over the Twitter network [15] providing for a social network-based service publication.

Listing 2: Announcing services on Twitter.

Discovering Services with Twitter. The discovery of services in Twitter does not follow the pull approach as with existing SOAs where a service requester searches for candidate service in repositories.

<table>
<thead>
<tr>
<th>SP</th>
<th>$service.&lt;name&gt;&quot;&quot;&lt;hashtags&gt;&quot;&quot;&lt;url&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>$service.&lt;name&gt;&quot;,&quot;&lt;operation&gt;&quot;&quot;&lt;hashtags&gt;&quot;&quot;&lt;url&gt;</td>
</tr>
<tr>
<td>DS</td>
<td>$user&gt;@service.&lt;name&gt;&quot;,&quot;&lt;operation&gt;&quot;&quot;&lt;hashtags&gt;&quot;&quot;&lt;url&gt;</td>
</tr>
<tr>
<td>KT</td>
<td>$service.&lt;name&gt;&quot;,&quot;&lt;operation&gt;&quot;&quot;&lt;hashtags&gt;&quot;&quot;&lt;url&gt;</td>
</tr>
</tbody>
</table>

Listing 3: Syntax elements related to discovery.

- **Publication of a Service Request**: With Twitter as communication platform, service discovery follows a push approach where the user Tweets a particular service request. The request, i.e., the Tweet contains meta information about the service, the operation that is required, hashtags providing meta information about the service (data) and a link to the service input data.
- **Direct Service Request**: The service request can also be directed to Twitter followers. In this case, the Tweet addresses the user directly.
- **Delegation of Service Request**: If Twitter followers are not able to handle the service request, but happen to know someone who is, the service request can be delegated to another follower.
- **Retweet Service Request**: Also, a user can retweet the service request to his followers and spread the service request to other users that are no followers of the service requestor. The retweeting or delegating of service requests leads to a dissemination of requests in the Twitter network. As in the case of service announcements, we implicitly use Twitter’s social structures during the discovery process.

Binding and Addressing Services. If a Twitter follower is able to provide the required service, the binding of a service request to the service instance happens if the follower directly answers to a service request Tweet. As the Tweet appears in the Tweetflow, the binding is complete and the service is being invoked. The actual addressing of the services uses the built-in Twitter addressing mechanism which sends Tweets directly to followers.

Listing 4: Claiming a service on Twitter.

Execution of Services and Monitoring. The execution of requests is associated with a state that can be requested. The detailed state model is not the focus of this work as such models have received sufficient attention in collaborative systems. Basically, states covered by our systems include pending, inprogress, aborted, finished, to name a few.

Listing 5: Syntax elements related to execution.

- **Service Response**: After the service has been completed, the service provider sends a message containing the service name and a link to the result of the service invocation.
- **Status Request**: During the execution of a service, the service requestor can check the current state of the service execution.
- **Status Reply**: State requests are replied by the user.
- **Reject Service Request**: It is also possible to reject a service request from another user.

Bootstraping, Executing and Monitoring. The process of creating or bootstrapping a Tweetflow consists of tweeting a set of service requests to find adequate service providers. As briefly discussed in the use case example, the porting of the iPhone applications requires a set of services, respectively service providers. These include, for instance, Android developers, translation services or advertisement services. After the discovery phase is completed (all service requests are bound) the actual execution phase of the Tweetflow starts.

The originator, i.e., the coordinator, of the Tweetflow posts a Tweet that contains a concatenation of actions which need to be executed by different users. This Tweet marks the beginning of the execution of a Tweetflow. A Tweetflow consists of a concatenation of service invocations, resulting in a service pipeline. In the pipeline, each service invocations passes the result to the next service in the pipeline. Listing 6 shows the syntax of Tweetflow Tweets.

Listing 6: Service pipes on Twitter.

It is worth noting that the control over the execution is distributed: upon completion of a service in the service pipeline, the service provider is directly responsible for tweeting the invocation of the next service by posting a service
request Tweet. Given the ad hoc character of Tweetflows, it’s possible to make modifications to the Tweetflow during its execution. For example, a service request can be delegated to another Twitter follower upon receiving a service request, allowing for service replacements on the fly. This open, distributed control approach allows to exploit crowdsourcing behavior in Tweetflows.

Since each service interaction is represented by a Tweet, we imply implicit monitoring of the service execution of the service pipeline. However, in order to track the execution of Tweetflows, we require that each Tweetflow Tweet contains a hashtag with name of the Tweetflow (see Listing 7) to be able to filter for Tweets.

### Listing 7: Tweetflow Tweets.

#### 4.2 Mapping between Tweetflows and SOA

Using Tweetflow principles, we are able to fully support the SOA-lifecycle consisting of service discovery, and service binding (interactions). However, in human-centric systems, it becomes important to support additional coordination mechanisms (i.e., routing through pipes). Similar (simple) mechanisms are already found in traditional message-oriented systems such as email. Our approach brings the benefit of seamless communication and coordination in a service-oriented manner. The analogy of these primitives to SOA concepts is summarized in Table 2.

<table>
<thead>
<tr>
<th>Tweetflow Primitives</th>
<th>SOA Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR, RT, DS, RJ</td>
<td>Service Discovery</td>
</tr>
<tr>
<td>CS</td>
<td>Service Binding</td>
</tr>
<tr>
<td>RE</td>
<td>Service Response</td>
</tr>
<tr>
<td>TF</td>
<td>Message Routing</td>
</tr>
<tr>
<td>ST, SE</td>
<td>Service Monitoring</td>
</tr>
<tr>
<td>SP</td>
<td>Service Publication</td>
</tr>
<tr>
<td>@</td>
<td>Service Addressing</td>
</tr>
</tbody>
</table>

#### Table 2: Mapping SOA principles to Twitter.

In the following, we show a simplified example of a human-provided document translation service as defined by Listing 8. Our approach utilizes the very same technologies used to implement software services such as WSDL and SOAP to support interactions with human-provided services.

First, let us briefly outline the features of the human-provided services (HPS) framework [21] that offers the building blocks to support flexible human interactions in SOA. HPS enhances the traditional SOA-triangle by enabling people to provide services with the very same technology as used by implementations of software-based services. Various operations for different collaborative activities indicate a provider’s ability (and willingness) to participate in collaborations. Three steps are performed when using HPS: (i) *Publish*: the user can create an HPS, specify its operations, and publish the service on the Web by using a registry. (ii) *Search*: the requester can perform a keyword-based search to find HPSs. Ranking is performed to find the most relevant HPSs based on, for example, the expertise of the user providing the service, or the relevance of a user’s experience for a particular request. A user’s experience is automatically calculated through interaction mining techniques. (iii) *Interact*: the framework supports automatic user interface generation using standardized XML forms that are generated according to exchanged complex data types in SOAP messages.

#### Listing 8: WSDL specification of HPS-based translation service (simplified example).

Listing 8 shows a simplified example of a HPS-based document translation service that is provided by one or more human actor(s). The WSDL description defines complex data types such as a *GenericResource* that wraps document-related information (i.e., the location of the document, expiration time, and language attributes). The *TranslationRequest* contains request-related information and additional tags applied to a request. Every request is acknowledged through *AckSupportRequest* by automatically generating a unique identifier. Further operations are supported to obtain the status of a request (e.g., *GetStatus*). Asynchronous notifications can be generated by passing the endpoint of the requester to the service (not shown in this example). The invocation of the translation service is shown in Listing 9.

#### Listing 9: Requesting the SOAP translation service.

```xml
<definitions name="TranslationService" ...>
<types>
  <schema targetNamespace="http://...">
    <simpleType name="Language">
      <restriction base="xsd:string">
        <enumeration value="Japanese"/>
        <enumeration value="English"/>
      </restriction>
    </simpleType>
    <complexType name="GenericResource">
      <sequence>
        <element name="Location" type="anyURI"/>
        <element name="Expires" type="dateTime"/>
        <element name="Lang" type="tns:Language"/>
      </sequence>
    </complexType>
    <complexType name="TranslationRequest">
      <sequence>
        <element name="Request">
          <complexType name="GenericResource"/>
        </element>
      </sequence>
    </complexType>
    <element name="GetTranslation" type="tns:TranslationRequest"/>
    <operation name="GetTranslation"/>
    <portType name="HPSPort">
      <operation name="GetTranslation"/>
    </portType>
    <service name="TranslationService">
      <port binding="tns:HPSPort"/>
      <operation name="GetTranslation"/>
    </service>
  </schema>
</types>
</definitions>
```
5. DISCUSSION
The use of Twitter for the collaboration between team members and services has several implications. First and foremost, using Twitter for communication and coordination means that messages are immediately visible to all followers. Input data of services is also visible, since it is being linked to in the Tweets. If sensitive data needs to be passed, then direct Twitter messages can be sent or the Tweets can be marked as private to prevent non-followers from reading them. Alternatively, users can send direct messages to other users without public Tweets. By using Twitter for service related messages (e.g., discovery, publication), we implicitly make use of information cascades if the corresponding message is retweeted to other followers. As has been discussed in the literature, retweets have the potential to reach large portions of the Twitter network. However, the success rate of information cascades for service requests in an open crowdsourcing setting still needs to be asserted in greater detail.

As an experiment ikangai tweeted a request for a Japanese-English translation for the ikangai blog and received an answer from a Twitter user ikangai did not know before. A similar event was observed with a request for an English-Korean translation. In this case, a direct follower of ikangai answered stating that he knew a student able to provide this kind of service. This suggests that information cascades in Twitter can be used for service discovery purposes. Listing 10 shows parts of the Tweetflow that were used to set up the communication and development infrastructure which was explained in our working scenario.

```plaintext
SR #service:infrastructure.create #googledocs #ase_2010_11
SR #service:infrastructure.create #pivotaltracker #ase_2010_11
SR #service:document.create #kickoff #ase_2010_11
SR #service:name.create #projectname #ase_2010_11
SR #service:poll.create #meeting #ase_2010_11
SC @ikangai #service:poll.create #kickoff #ase_2010_11
SC @redali25 #service:document.create #kickoff #ase_2010_11
SC @stefanasseg #service:infrastructure.create #pivotaltracker #ase_2010_11
SC @ikangai #service:poll.create #meeting #http://bit.ly/bfdhGY
TF #ase_2010_11 ikangai service:poll.create #meeting http://bit.ly/bfdhGY
TF #ase_2010_11 @stefanasseg service:poll.create #kickoff http://bit.ly/bfdhGY
TF #ase_2010_11 @redali25 #service:document.create #kickoff
TF #ase_2010_11 @ikangai #service:name.create #projectname
TF #ase_2010_11 @ikangai #service:poll.create #meeting http://bit.ly/bfdhGY
RE @ikangai #service:infrastructure.create #googledocs | #ase_2010_11 http://bit.ly/bWcyV
RE @ikangai #service:document.create #kickoff #ase_2010_11 http://bit.ly/bWcyV
RE @ikangai #service:infrastructure.create #pivotaltracker #ase_2010_11 http://bit.ly/17vVC
RE @ikangai #service:name.create #projectname #ase_2010_11 http://bit.ly/ajp8VF
```

Listing 10: Tweetflow example.

The core follower structure of the project team was a complete graph because the project team was known a priori and followed each other on Twitter. Consequently, all project-related messages were observed by each team member. However, as discussed before, service requests that couldn’t be served were forwarded to other followers of the team members, creating a permeable information boundary. In an enterprise setting, such structures can be mapped onto development teams which work on different areas of a product and which members are highly connected. As shown in the working example, some expertise might be missing and an open messaging infrastructure can be used to extend the expertise of a group. After the kickoff meeting, a Tweetflow for coordinating the activities for the creation of the project infrastructure was initiated. The originator of the Tweet (Twitter user ikangai), bootstrapped the Tweetflow by posting a number of service requests. The service requests were answered by the team members and claimed the services. The originator bound the services to the users and started the execution of the Tweetflow by posting a new round of Tweets. As shown in the example, there were dependencies between actions, like the creation of a Google Docs account and the creation of a Google document which had to be taken into consideration. A polling service was bound by user ikangai using the Tweet TF #ase_2010_11 ikangai service:poll.create #meeting http://bit.ly/bfdhGY, integrating the Restful Doodle Poll API into the Tweetflow.

6. RELATED WORK
In service-oriented environments, standards have been established to model human-based process activities and tasks (WS-HumanTask [8]). However, these standards demand for the precise definition of interaction models between humans and services. In our approach, we combine SOA concepts and social principles. We consider open service-oriented environments wherein services can be added at any point in time. Following the open world assumption, humans actively shape the availability of services. The concept of Human-Provided Services (HPS) [21] supports flexible service-oriented collaborations across multiple organizations and domains. Similarly, emergent collectives as defined by [20] are networks of interlinked, valued nodes (services).

Open service-oriented systems are specifically relevant for future crowdsourcing applications. For example, a hybrid human-computer document translation system has been discussed by [22], however not focusing on the realization as a service-based system. While existing platforms (e.g., MTurk [2]) only support simple interaction models (tasks are assigned to individuals), social network principles support more advanced techniques such as formation and adaptive coordination. Social game-based human computation has been introduced in the context of image labeling that is performed by humans. From the technical point of view, TurKit [16] is a crowds computing framework based on MTurk.

On an architectural level, we follow the blackboard architectural pattern [9]. In our architecture, the Tweet Bus plays the role of the blackboard which holds state information, i.e., Tweets of the Tweetflows. Enterprise Service Bus Architectures (ESB) [4] have a strong similarity to our Tweet Bus architecture. Like in ESBs, our approach also uses a centralized communication channel to transport messages. Clients plug into this channel and listen to messages which are transported in a standard format. However, the pub-
lic visibility of Tweets, the 140 character limit for messages and the ability to forward (retweet) messages arbitrarily to other users not listening to the communication bus, i.e., to push messages into a community of followers, are the main differences.

7. SUMMARY AND OUTLOOK

In this paper, we presented an approach for using Twitter as a communication and coordination medium for the execution of simple workflows. A novel application of our approach are collaborations in crowdsourcing environments where people provide their skills and capabilities in a service-oriented manner. In contrast to existing crowdsourcing platforms, our approach enables the realization of collaborative crowds where individuals jointly work on activities and short-term projects.

We introduced Tweetflow primitives which are embedded in Twitter communications to control collaborations and to interact with human- and software-based services. By using Twitter, we implicitly provided a platform-agnostic communication backend, which possesses enterprise service bus characteristics: each client is able to plug into the Tweet Bus and exchange messages, independent of any implementation. Moreover, a Tweet Bus provides the means to integrate human- and software-services seamlessly. And finally, the Twitter follower structure offers the ability to exploit social structures and to utilize them for service discovery purposes as well as to tap human resources of crowds: service requests can be forwarded (i.e., retweeted) to followers and are thus distributed to the crowd. In future works, we plan to extend Tweetflow primitives to enable the creation of more complex Tweetflows. For instance, we plan to integrate branching conditions, synchronization points and execution deadlines into Tweetflows to allow for more complex Tweetflows. Our prototype will be extended with a graphical tool that facilitates the creation of Tweetflows and we intend to extend the support for SOAP-based Web services and Restful Services [19]. We will investigate Twitter user profiles to include them for service recommendations [10] or delegations.

Acknowledgment

This work received funding from the EU FP7 program under the agreements 215483 (SCube) and 215175 (COMPAS).

8. REFERENCES

A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules

Barbara Pernici¹, S. Hossein Siadat¹, Salima Benbernou², and Mourad Ouziri²

¹ Politecnico di Milano, Italy
² LIPADE, Université Paris Descartes, France

Abstract. Quality of Service (QoS) guarantees are commonly defined in Service Level Agreements (SLAs) between provider and consumer of services. Such guarantees are often violated due to various reasons. QoS violation requires a service adaptation and penalties have to be associated when promises are not met. However, there is a lack of research in defining and assessing penalties according to the degree of violation. In this paper, we provide an approach based on fuzzy logic for modelling and measuring penalties with respect to the extent of QoS violation. Penalties are assigned by means of fuzzy rules.

Key words: QoS, service level agreement, penalty, fuzzy logic

1 Introduction

QoS guarantees defined in contracts may be violated due to various reasons. This situation needs to be handled through applying adaptation techniques not to bring dissatisfaction. The concept of penalty has been used in SLAs to compensate the conditions under which guarantee terms are not met [1]. Despite some research have been done on the description, negotiation and monitoring of SLAs, however there is not much work on the definition of penalty clauses. [4] studied on WS-Agreement specification to define penalties based on different types of violation. However, penalties are assigned to violation of a single property instead of assigning penalties to violation of overall QoS. Moreover, the approach introduces a method for measuring penalties which is for fixed predefined number of violations, instead of measuring the extent of violation and assigning penalties accordingly.

One main issue is how to determine the appropriate amount of penalties as compensations from providers to satisfy customers. As quality parameters can be satisfied partially, the assessment of penalties can be based on the degree of quality violation. Understanding the violation degree is a prerequisite for assessing penalties. However, measuring such violation is yet an open research challenge. In addition, the influencing factors in defining penalties need to be identified. A static amount of penalty (manual approaches) does not reflect the extent of violation at runtime. The amount and level of penalties are related to the degree of quality violation provided from the provider side. On the other
side, the customers characteristics may also affect the amount of penalties. For example a penalty to satisfy a gold/loyal customer is different with the one for an occasional customer. To the best of our knowledge, there is no formal relation between the assigned penalty and its influencing factors. Moreover, the extent and type of penalties are not clearly expressed in related work. However, understanding such relation and providing a mapping between them are complicated issues. We argue what is missing is a suitable mechanism for modelling penalties that takes into account both provider and consumer sides. Apart from the degree of violation, we also consider the state of customer and service provider with respect to their past history (e.g. whether the service has been penalised previously) in determining the right amount of penalties. However, as the relation between a given penalty and its influencing factors is not linear, conventional mathematical techniques are not applicable for modelling penalties.

Recent approaches are dealing with the issue of partial satisfaction for quality commitments and different techniques were used such as applying soft constraint [6], fuzzy sets [3] and semantic policies [2]. Among them, [6] introduced the concept of penalties for unmet requirements. However, defining penalties and finding a relation between the assigned penalties and the violated guarantees are remained challenges in similar approaches. The goal of this paper is to apply an inference technique using fuzzy logic as a solution [5] to propose a penalty-based approach for compensating conditions in which quality guarantees are not respected. Fuzzy logic is well suited for describing QoS and measuring quality parameters [3]. We demonstrate a penalty inference model with a rule-based mechanism applying fuzzy set theory. Measuring an appropriate value for penalties with respect to the amount of violation is the main contribution of the paper.

In the following, we start by a motivating example in Section 2. In Section 3 we show the descriptions of penalties and in Section 4 we provide a rule-based system using fuzzy set theory for modelling and reasoning penalties. Section 5 shows some experiments in applying penalty for the problem of QoS dissatisfaction and we conclude the paper in Section 6.

2 Motivating Example

Let’s assume that a user is wishing to use a food delivery service. Therefore, a contract is established between the user and the service provider. The contract defines non functional criteria such as delivery time, quality of the perceived service (the quality of food during the delivery service, for example the food is maintained at the ideal temperature), and availability of the delivery service. Therefore we define a list of parameters for our example as follows: time to delivery (td), quality of delivered food (qd), availability of delivery service (ad). These quality parameters together with a list of penalty terms are defined in a contract and illustrated in Table 1.

The delivery service will be penalized if it is not able to provide the quality ranges defined in the contract. An overall QoS violation will be calculated first
A Penalty-based Approach for QoS Dissatisfaction using Fuzzy Rules

<table>
<thead>
<tr>
<th>Quality Parameters</th>
<th>time to delivery</th>
<th>between 10 to 15 min</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>quality of delivered food</td>
<td>between 0.8 to 1</td>
</tr>
<tr>
<td></td>
<td>availability</td>
<td>between 90 to 100% of the time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Penalties:</th>
<th>Minor or Null penalties</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Penalties on quality parameters</td>
</tr>
<tr>
<td></td>
<td>Extra Service penalties</td>
</tr>
<tr>
<td></td>
<td>Termination penalty</td>
</tr>
</tbody>
</table>

Table 1: motivating example

and afterwards a penalty is assigned with respect to the extent of the violation. We also take into account customer and provider perspectives by considering parameters from both parties. Parameters such as history of a delivery service and state of a customer can be involved. The history of a service shows whether the service is penalized previously. This can influence the amount of given penalties for future. The current state of a customer presents the importance of the customer for service provider. For example, minor violation of service delivery can cause a major penalty for provider in case the customer is gold (with good history). In contrast, a normal customer (with ordinary history) will not be given any extra service if the quality of delivered food is not good.

3 Definition of Penalties

In order to provide a formal model of penalties and build a reasoning mechanism to handle the penalties in the contract, in the following we try to summarize the different types of penalties that can be applied. We categorize the penalties into two main classes:

1. **Numerical penalties**: They are related to measurable qualities of service. In other words, we have to handle and work with variables of the service (e.g. the availability > 0.9, the responsetime < 0.2ms).

2. **Behavioural penalties**: They are related to the behaviour of either the customer or the service provider. Consider the following case: a merchant wishes to obtain a service for online payment by bank card. The financial institution offered a 25% off if the settlement proceeds within two days of the request. Beyond these two days, the penalty is such as the trader does not have the discount and will therefore be required to pay all fees.

A penalty clause in an SLA may be of the following types:

- Penalty Null and denoted by \( P_0 \): no penalty is triggered because all agreed QoS are satisfied or minor violation has occurred.
- Penalty on the QoS: a penalty should be triggered on one of the QoS parameters \( Q_j \) in the contract if \( Q_i \) is not fulfilled.
- Penalty on the penalty: a new penalty \( P_j \) should be triggered if the previous one \( P_i \) is unfulfilled. Such penalty will be handled through the long term contract validation. The reasoning on the time aspect of the contract is out of the scope of the paper.
– Extra service penalty: if a QoS is not fulfilled by the service provider, to penalize him, an extra service might be offered to the customer.
– Cancellation penalty: this is a dead penalty for the service provider. A service substitution occurs.

4 Modelling Penalties

We present a fuzzy model to express penalties in a rule-base system. Our fuzzy penalty model is defined by the couple $FP = <S, R>$, where $S$ is a fuzzy set on penalties and $R$ is a set of inference rules.

4.1 Fuzzy sets for penalties

Our knowledge system includes linguistic variables defined by tuple $(Q, C, H, P)$, where $Q$ is a set of QoS parameters defined by fuzzy parameters as $Q = \{t_d, q_d, a_d\}$ where $t_d$ is the time to delivery, $q_d$ is the quality of the delivered service and $a_d$ is the availability of the delivery service. $C$ is the current state of the customer, $H$ is the history of the service to show whether the service is penalized previously and $P$ is the set of penalties. We define these linguistic variables by fuzzy sets in the following.

The linguistic parameter of customer is defined by three fuzzy sets as in $C = \{\text{Normal, Silver, Gold}\}$. We define two fuzzy sets to represent the state of service with respect to previous penalties as in $H = \{\text{Penalized, Not – penalized}\}$. Finally penalties are described by five fuzzy sets to show the diverse range of penalties as in $P = \{\text{Null, Minor, Average, Major, Termination}\}$, where null is no penalty, and termination is the situation in which the customer will terminate his contract with the delivery service. A fuzzy set represents the degree to which an element belongs to a set and it is characterized by membership function $\mu_{\tilde{A}}(x) : X \mapsto [0, 1]$. A fuzzy set $\tilde{A}$ in $X$ is defined as a set of ordered pairs

$$\tilde{A} = \{(x, \mu_{\tilde{A}}(x)) \mid x \in X, \mu_{\tilde{A}}(x) \in [0, 1]\}$$

where $\mu_{\tilde{A}}(x)$ is the membership function of $x$ in $\tilde{A}$. Therefore, a membership function shows the degree of affiliation of each parameter by mapping its values to a membership value between 0 and 1.

We associate membership functions to a given fuzzy set to define the appropriate membership value of linguistic variables. We start by providing membership functions for quality parameters from the motivating example. We take an approach that calculate an overall degree of violation with respect to the violation of each quality parameters. This way, we perform a trade-off mechanism and quality parameters are not treated independently. For each quality parameter a membership function is provided to show the degree of their satisfaction. We define three linguistic variables for each parameters such that $t_d$ belongs to the set $\{\text{Slow, Normal, Perfect}\}$ and $q_d$ is in the set $\{\text{Unacceptable, Bad, Good}\}$ and $a_d$ is in the set $\{\text{Low, Medium, High}\}$. Figure 1 depicts the membership
functions of time to delivery (a) and service availability (b). The functions are defined according to the contract and by an expert of the system. For example, the time to delivery between 10 to 15 min is perfect, between 15 to 20 min is good and more than 20 min is slow. Membership functions of penalty and customer state are shown in Figure 2 in (2a) and (2b) respectively.

4.2 Inference rules on penalties

The inference rules to trigger penalties are expressed as follows:

- \( R_{Q} \) : QoS-based penalty rules. These are rules that reflect the violation of quality parameters. Penalties will be applied to a service if QoS guarantees stipulated in SLA are not fulfilled. It will be presented formally by \( R_{Q} : Q \rightarrow P \).

  For instance, in the SLA, the delivery service agreed with the customer: \( 10\text{mins} \leq \text{delivery time} \leq 15\text{mins} \) and good quality of delivered food. If the QoS delivery time is not fulfilled (partially), then penalty \( p_{e1} \) (e.g. 10% discount) will be applied. Depending on the severity of the violation a harder penalty might be applied. For example, if both QoS are not fulfilled then penalty \( p_{e2} \) (e.g. 20% discount) will be applied. The fuzzy inference system gives us such degrees for penalties. Both cases are presented respectively below by rules:
- \( R_1 \) \((t_d = \text{Slow}) \land (q_d = \text{Good}) \rightarrow p_{e1}\)
- \( R_2 \) \((t_d = \text{Slow}) \land (q_d = \text{Bad}) \rightarrow p_{e2}\)

\( R_P \): penalty on penalty rules. These rules reflect whether the service was given a penalty. If a service was penalized previously and again does not fulfill a QoS, then a penalty will be harder. It will be presented formally by \( R_P : Q \times P \rightarrow P \) such that \( R_P(q, p_1) = p_2 \Rightarrow p_1 \prec p_2 \).

For instance, let us consider a service having a penalty \( p_{e1} \) w.r.t rule \( R_1 \) and again provides a slow delivery time, then the penalty \( p_{e3} \) (e.g. 10\% discount plus free delivery) will be applied. The rule can be presented as below:
- \( R_3 \) \((t_d = \text{Slow}) \land p_{e1} \rightarrow p_{e3}\)

\( R_C \): customer-related penalty rules. The rules defined here will be adapted according to a customer qualification. Such rules will be presented formally by \( R_C : Q \times P \times C \rightarrow P \).

For instance, if the provided QoS is not fulfilled knowing that a penalty is assigned to the service, and if a customer is gold (has a good history), then extra service penalty \( p_{e4} \) (giving some extra service to the gold customer e.g. one movie ticket) will be harder than the one applied for normal customer \( p_{e3} \).

The rules can be presented as below:
- \( R_4 \) \((t_d = \text{Slow}) \land p_{e1} \land (C = \text{Normal}) \rightarrow p_{e3}\)
- \( R_5 \) \((t_d = \text{Slow}) \land p_{e1} \land (C = \text{Gold}) \rightarrow p_{e4}\)

5 Experiments and Implementation

We have simulated our approach in a simulator based on fuzzy inference system. Initial membership functions were designed based on the contract in the motivating example and fuzzy rules are defined by the expert of the system. Figure 2c illustrates membership function for QoS violation (see [3] for further details). Having defined the QoS violation, we measure the extent of penalties taken into account the state of customers and previously applied penalties for the same service. For this, fuzzy rules are defined considering all three influencing factors. Figure 3 depicts fuzzy rules for penalty based on QoS violations, customer’s state and service status with respect to previous penalties which are defined by the service-state parameter represented by fuzzy set \( \{\text{Penalized, Not - penalized}\} \).

For example rule no. 8 shows that a major penalty will be given to a silver customer if major violation occurs from defined QoS, while rule no. 7 will give a normal penalty (has lesser effect than major penalties) to the normal customer when the same amount of violation happens. The role of service-state can be seen in the rule, e.g. by comparing the rule no. 5 with the rule no. 14. In general, a harder penalty will be given to the service which is already penalized from the provider side.

The inference system calculates the degree of penalty by applying all the rules in a parallel approach for given input values of influencing factors. For example assume a QoS violation of 0.7 which has a membership degree of 0.5 for both normal and major fuzzy sets (according to their membership functions
Fig. 3: Fuzzy rules for penalty based on QoS violations, customer's state and previous penalties on the service

Fig. 4: A view of the inference system for applying penalties

presented in the figure 2c). Such a violation, can trigger all the rules that include normal and major QoS violations. Note that the result of each rule depends on the membership degrees of other linguistic variable. For this example, rules with minor QoS-violation are not triggered at all. This situation is demonstrated in Figure 4. The result of each rule is integrated with an aggregation method to include the effect of all the rules. Figure 5 depicts a plot showing the penalties regarding QoS violation and customer's state. The figure represents possible values for penalties after defuzzification for all values of QoS violation and customer's state. For example, for the QoS violation of 0.7 and customer-state of 0.4 the penalty degree is 0.66 which is shown in the figure. The relation between QoS violation and customer's state can also be seen in the figure.

6 Conclusions and Future Work

Applying penalties is a complex research issue in service oriented computing which has not been paid enough attention in the literature. In this work, we
elaborated the concept of penalty and propose a mechanism for modelling and measuring penalties. Penalties are modelled using a fuzzy approach and applying fuzzy set theory. The relation between penalties and their influencing factor are defined by fuzzy rules through an inference method. We have demonstrated the proposed penalty model through a motivating example and performed some initial result in measuring penalties.

Acknowledgements

The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

2. Pei Li, Marco Comerio, Andrea Maurino, and Flavio De Paoli. Advanced non-functional property evaluation of web services.