Analysis of Service-Oriented Computing Systems

PhD Thesis

Dragan Ivanović

November 2012
PhD Thesis

Analysis of Service-Oriented Computing Systems

presented at the Computer Science School of the Technical University of Madrid in partial fulfillment of the degree of PhD in Software and Systems

PhD Candidate: Dragan Ivanović
B.Sc. in Electrical Engineering
M.Sc. in Computer Science
Technical University of Madrid, España

Advisor: Manuel Carro
Associate Professor
Technical University of Madrid

Madrid, November 2012
Acknowledgements

Work on this thesis was made possible by many people to whom I am deeply grateful. I cannot possibly thank enough my wife Jasmina and my daughter Vedrana whose love and support never faltered during almost three years of our geographic separation. And without the selfless support to our family from my dear parents and my brother Žarko during that time, I wouldn’t have come to the stage of writing these lines.

My special thanks go to my thesis advisor, Manuel Carro who has taken me on board in the first place, and then given me help, guidance and support in every imaginable way – scientifically, professionally, and personally – for the whole duration of the exciting journey through the doctoral studies. I couldn’t have possibly hoped for more. I am glad of having had the opportunity to work on most of my publications together with Manuel Hermenegildo, the director of the CLIP laboratory as well as of the IMDEA Software institute, a person of truly dazzling intelligence and seemingly inexhaustible energy.

During the greatest part of my PhD studies I worked within and was funded by S-Cube, a EU Network of Excellence project on software and services. It was indeed an excellent and inspiring environment for doing and aligning service-related research. For very fruitful discussions, hospitality during my study stays, and joint research work I am especially thankful to Martin Treiber, Schahram Dustdar and Philipp Leitner of the Technical University of Vienna, Andreas Metzger, Eric Schmieders and Osama Sammodi of the University of Duisburg-Essen, Dimka Karastoyanova, Michele Mancioppi and Branimir Wetzstein of the University of Stuttgart, and to many other colleagues from the project.

Finally, my work was made easier, more interesting and productive with the help of my friends Rémy Hæmmerlé, José Francisco Morales, Edison Mera, Abu Naser Masud, Teresa Trigo, Guillermo Román, José Miguel Rojas, Diana Ramirez and others to whom I can thank for a friendly and inviting work environment.
Abstract

Service-Oriented Computing (SOC) is a widely accepted paradigm for development of flexible, distributed and adaptable software systems, in which service compositions perform more complex, higher-level, often cross-organizational tasks using atomic services or other service compositions. In such systems, Quality of Service (QoS) properties, such as the performance, cost, availability or security, are critical for the usability of services and their compositions in concrete applications. Analysis of these properties can become more precise and richer in information, if it employs program analysis techniques, such as the complexity and sharing analyses, which are able to simultaneously take into account both the control and the data structures, dependencies, and operations in a composition.

Computation cost analysis for service composition can support predictive monitoring and proactive adaptation by automatically inferring computation cost using the upper and lower bound functions of value or size of input messages. These cost functions can be used for adaptation by selecting service candidates that minimize total cost of the composition, based on the actual data that is passed to them. The cost functions can also be combined with the empirically collected infrastructural parameters to produce QoS bounds functions of input data that can be used to predict potential or imminent Service Level Agreement (SLA) violations at the moment of invocation. In mission-critical applications, an effective and accurate continuous QoS prediction, based on continuations, can be achieved by constraint modeling of composition QoS based on its structure, known data at runtime, and (when available) the results of complexity analysis. This approach can be applied to service orchestrations with centralized flow control, and choreographies with multiple participants with complex stateful interactions.

Sharing analysis can support adaptation actions, such as parallelization, fragmentation, and component selection, which are based on functional dependencies and information content of the composition messages, internal data, and activities, in presence of complex control constructs, such as loops, branches, and sub-workflows. Both the functional dependencies and the information content (described using user-defined attributes) can be expressed using a first-order logic (Horn clause) representation, and the analysis results can be interpreted as a lattice-based conceptual models.

Keywords: service-oriented computing; quality of service; service composition; monitoring; prediction; adaptation; cost analysis; constraints; sharing analysis.
Resumen

La computación basada en servicios (Service-Oriented Computing, SOC) se estableció como un paradigma ampliamente aceptado para el desarrollo de sistemas de software flexibles, distribuidos y adaptables, donde las composiciones de los servicios realizan las tareas más complejas o de nivel más alto, frecuentemente tareas inter-organizativas usando los servicios atómicos u otras composiciones de servicios. En tales sistemas, las propiedades de la calidad de servicio (Quality of Service, QoS), como la rapidez de procesamiento, coste, disponibilidad o seguridad, son críticas para la usabilidad de los servicios o sus composiciones en cualquier aplicación concreta. El análisis de estas propiedades se puede realizar de una forma más precisa y rica en información si se utilizan las técnicas de análisis de programas, como el análisis de complejidad o de compartición de datos, que son capaces de analizar simultáneamente tanto las estructuras de control como las de datos, dependencias y operaciones en una composición.

El análisis de coste computacional para la composición de servicios puede ayudar a una monitorización predictiva así como a una adaptación proactiva a través de una inferencia automática de coste computacional, usando los limites altos y bajos como funciones del valor o del tamaño de los mensajes de entrada. Tales funciones de coste se pueden usar para adaptación en la forma de selección de los candidatos entre los servicios que minimizan el coste total de la composición, basado en los datos reales que se pasan al servicio. Las funciones de coste también pueden ser combinadas con los parámetros extraídos empíricamente desde la infraestructura, para producir las funciones de los límites de QoS sobre los datos de entrada, cuales se pueden usar para previsar, en el momento de invocación, las violaciones de los compromisos al nivel de servicios (Service Level Agreements, SLA) potenciales or inminentes. En las composiciones críticas, una previsión continua de QoS bastante eficaz y precisa se puede basar en el modelado con restricciones de QoS desde la estructura de la composition, datos empíricos en tiempo de ejecución y (cuando estén disponibles) los resultados del análisis de complejidad. Este enfoque se puede aplicar a las orquestaciones de servicios con un control centralizado del flujo, así como a las coreografías con participantes múltiples, siguiendo unas interacciones complejas que modifican su estado.

El análisis del compartición de datos puede servir de apoyo para acciones de adaptación, como la paralelización, fragmentación y selección de los componentes, las cuales son basadas en dependencias funcionales y en el contenido de información en los mensajes, datos internos y las actividades de la composición, cuando se usan construcciones de control complejas, como bucles, bifurcaciones y flujos anidados. Tanto las dependencias funcionales como el contenido de información (descrito a través de algunos atributos definidos por el usuario) se pueden expresar usando una representación basada en la lógica de primer orden (claúsulas de Horn), y los resultados del análisis se pueden interpretar como modelos conceptuales basados en retículos.
## Contents

1 Introduction ................................................. 1
   1.1 Thesis Objectives ................................... 2
   1.2 Structure of the Thesis .............................. 3
   1.3 Contributions ......................................... 5

2 Background .................................................. 9
   2.1 Services and Service Compositions ................. 9
   2.2 A Primer on Logic Programs ........................ 11
       2.2.1 Syntax ........................................ 12
       2.2.2 Semantics ................................... 14
   2.3 Computation Cost Analysis ............................ 17
   2.4 Sharing Analysis for Logic Programs ............... 18

3 Supporting Predictive Monitoring and Proactive Adaptation With Automatic Complexity Analysis .... 21
   3.1 Related Work .......................................... 22
   3.2 Computation Cost Analysis in the Service World .. 23
       3.2.1 Impact of Data on Composition QoS .......... 24
       3.2.2 Computation Cost of Service Networks ...... 25
       3.2.3 Approximating Actual Behavior .............. 27
       3.2.4 Data-Dependent Upper and Lower Bounds ...... 29
       3.2.5 Cost Functions for Monitoring ............... 32
   3.3 Analysis of Orchestrations ............................ 34
       3.3.1 Overview of the Translation .................. 35
## 5 A Constraint-Based Approach to Quality Assurance in Service Choreographies

5.1 Motivation ....................................................... 86

5.2 Constraint-Based QoS Modeling for Choreographies ................. 89

5.2.1 Modeling Cumulative QoS Metrics ................................ 90

5.2.2 QoS Models of Participants and Continuations .................. 90

5.2.3 Automatic Derivation of the QoS Constraint Model for a Participant 92

5.2.4 Analysis of Message Types With Size Constraints ............... 94

5.2.5 Centralized and Distributed Processing of QoS Constraints ...... 96

5.3 Examples of Application ......................................... 98

5.3.1 Supporting SLA Negotiation For Classes of Input Data .......... 98

5.3.2 Predicting SLA Violations at Run Time .......................... 100

5.3.3 SLA Compliance Checking, Dynamic Binding and Adaptation .... 101

5.4 Conclusions ...................................................... 103

## 6 A Sharing-Based Approach to Supporting Adaptation in Service Compositions

6.1 Motivation ....................................................... 107

6.2 Functional Dependencies and Data Attributes Through Sharing ...... 110

6.2.1 Functional Dependencies in Service Compositions ............... 111

6.2.2 Data Attributes in Service Compositions .......................... 114

6.2.3 Representing Dependencies With Substitutions .................. 119

6.2.4 Variable Sharing .............................................. 122

6.3 Sharing Analysis ................................................ 124

6.3.1 Derivation of Control and Data Dependencies .................... 125

6.3.2 Validity of Control Dependencies ................................ 127

6.3.3 Generating Horn Clause Representations .......................... 128

6.3.4 Sharing Analysis Proper ....................................... 133

6.3.5 Interpretation of Sharing Results .................................. 134

6.3.6 Complexity And Precision of the Sharing Analysis ............... 135

6.4 Examples of Application ......................................... 136

6.4.1 Parallelization ................................................. 136
List of Figures

2.1 The basic roles and operations for SOA components. ...................... 10
2.2 Basic elements of Prolog syntax ..................................................... 12
2.3 An example route finding program in Prolog. .............................. 14
2.4 A modified route finding predicate. .............................................. 16
2.5 An illustration of the relationship between a concrete and an abstract substitution .......................................................... 20

3.1 Simplified car part reservation system. ........................................ 24
3.2 Invoking other services. .............................................................. 26
3.3 Computational cost, services $A + B_1$ and $A + B_2$. .................. 26
3.4 Separating factors that affect composition QoS. ............................ 28
3.5 Levels of information and precision in describing service QoS. ......... 30
3.6 Using upper and lower bounds. .................................................... 31
3.7 Actual and predicted QoS throughout history. .............................. 33
3.8 The overall process. ................................................................. 34
3.9 Translation of types. ................................................................. 37
3.10 Translation example. ............................................................... 44
3.11 Ratio of true and false positives for two environmental regimes. .... 46
3.12 Two-tier simulation setting. ....................................................... 48
3.13 Upper bounds for computational costs. ........................................ 49
3.14 Simulation results across failure rates using the same noise distribution .. 51
3.15 Simulation results across failure rates using different noise distributions . 52

4.1 An example orchestration to reconfigure content provided to a user. .... 57
4.2 Architecture of the QoS Analysis Framework. ............................. 61
4.3 Abstract syntax for orchestrations. ................................................. 63
4.4 Orchestration for Fig. 4.1 (left) and its associated running time constraints (right). ................................................................. 63
4.5 Lack of accuracy and precision in the estimation of QoS bounds. ....... 74
4.6 Comparative measures of prediction quality for inaccurate and imprecise assumptions. ................................................................. 79
4.7 Distribution of negative prediction times (ms). ............................... 81
4.8 Comparative true negative/positive prediction times for inaccurate and imprecise QoS assumptions. ................................................. 81
4.9 Distribution of positive prediction time leads ahead of failure (ms). ..... 82
4.10 Adaptation-related metrics under inaccurate or imprecise QoS assumptions. ................................................................. 83

5.1 An example choreography for purchase ordering. ......................... 87
5.2 Message exchange sequence and a component model for a choreography session. .................................................. 88
5.3 Abstract syntax of the participant continuation language. ............ 92
5.4 Structurally derived QoS constraint model for participant A. .......... 93
5.5 A simple typing system for messages with size constraints ......... 95
5.6 Analysis of types with size constraints for participant B. .......... 96
5.7 Centralized (left) and distributed (right) processing of choreography QoS constraints. ................................................................. 96
5.8 An example of SLA failure prediction zones. ................................. 101
5.9 Adaptation need detection in B. ................................................ 102

6.1 A sample purchasing end-to-end service composition in BPMN notation. 108
6.2 A conceptual model of functional dependencies in composition activities. 113
6.3 Structural data description example. ............................................. 116
6.4 Tests, rules and attributes for data structures from Fig. 6.3. ......... 117
6.5 A sample concept lattice for the context from Figure 6.4. ............. 119
6.6 An example workflow. Arrows indicate control dependencies. .... 126
6.7 An example of deadlock dependency on logic formula: • can be either ∧ or ∨. 127
6.8 Fragments of translation to Horn clause form of the composition from Fig. 6.1 ................................................................. 131
6.9 Conceptual lattice (laid out horizontally for convenience) grouping together activities and data based on functional dependencies. 137
6.10 Effect on the information on service statefulness on the level of parallelization. 139
6.11 Attribute inheritance arising from by Θ in Table 6.3. 141
6.12 A sample fragmentation scheme between four domains. 142
6.13 A drug prescription workflow in BPMN from E-Health scenario. 143
6.14 BPMN diagram of the composite activity a₄ from Fig. 6.13. 143
6.15 The resulting context for the drug prescription workflow analysis. 144
6.16 An example fragmentation for the drug prescription workflow. 145
6.17 Alternative context and fragmentation scheme for the drug prescription workflow. 145
6.18 A hypothetical concept lattice of credit approval procedures. 147
6.19 Adaptive search for loan approval service candidates. 148
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Abstract orchestration elements.</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>Translating simple constructs.</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Translating fault handling primitives.</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Translating invocations.</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Translating scopes.</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Translating control structures.</td>
<td>42</td>
</tr>
<tr>
<td>3.7</td>
<td>Resource analysis results for the group reservation service.</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Sample QoS prediction results.</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparative prediction quality indicators for the most precise and accurate component QoS assumptions.</td>
<td>78</td>
</tr>
<tr>
<td>5.1</td>
<td>Experimental inputs and outputs of the execution time model.</td>
<td>99</td>
</tr>
<tr>
<td>6.1</td>
<td>Data dependencies in the example service composition.</td>
<td>110</td>
</tr>
<tr>
<td>6.2</td>
<td>A representation of a sharing result $\Theta$ as a context.</td>
<td>137</td>
</tr>
<tr>
<td>6.3</td>
<td>The sharing result $\Theta$ for a case of fragmentation based on data sharing.</td>
<td>141</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

During the past decade, the Service-Oriented Computing (SOC) [PTDL07] has established itself as a widely accepted paradigm for development of flexible, distributed, and adaptable applications. While Web services [Pap07] — the basic building blocks of Service-Oriented Applications (SOA) — are just one among many varieties of reusable software components present today [SGM02], several distinguishing characteristics of SOC pose new challenges to software architects, designers, developers, and maintainers. Besides, the promise of standards-based, platform (i.e., operating system and programming language) independent distributed computing based on loose coupling between components also comes at the cost of adopting many new, service-specific technologies, development methodologies and engineering approaches. In this thesis we address some of the challenges brought about by SOC, related to service composition, Quality of Service (QoS) and service adaptation.

Individual (atomic or back-end) services are usually highly specialized for a particular task, and their interfaces typically include of one or more functionally cohesive sets of operations. However, the true power of SOC shows in complex, cross-domain and cross-organizational settings, where service compositions put together several service components to perform higher-level or more complex tasks. In turn, exposing service compositions themselves as services and enabling their use by other compositions enables the development of highly complex, large scale, distributed, and flexible software systems. In SOC, the component services are typically provided and maintained by third parties [DNGM+08].
QoS properties of individual services and service compositions are critical for their overall usability in a real-world context, and typically express non-functional properties related to performance, cost, availability, security and other relevant aspects of quality [KPP+13]. Many of these properties can be derived from formal models of computation that express, on an appropriate level of abstraction, the semantics of the concrete executable artifacts of which a Service-Based Application (SBA) consists. Thus, in principle, there is a potential for improvement of QoS analysis precision and information richness when we progress from an analysis that relies only on empirical measurements, to one that takes into account the composition control structure (expressed in an abstract or an executable form), and finally to one that also takes into account data structure, content and operations. This progression opens the space for fruitful cross-fertilization between SOC and the program analysis techniques [NNH05] (such as the complexity and sharing analyses) that have been developed for the conventional models of sequential and concurrent computation.

Service adaptation, especially on the composition level, aimed at meeting the evolving user and business process requirements, is a commonplace in SOC [DNGM+08]. In particular, there is a huge interest in techniques and approaches that enable automatic, runtime self-adaptation without user intervention. In many cases, such an adaptation can be triggered by a prediction of a potential or an imminent violation of the desired QoS limits, and may involve measures for avoiding or mitigating its effects. In other cases, the adaptation may aim at maximizing distributed execution, or at controlling information flow between services from different business domains. The criteria for correctness of adaptation need to ensure that the interventions on the composition control structures, service bindings, and data preserve the key correctness properties, i.e., avoid race conditions, uninitialized data and other semantic errors. Thus, in the field of adaptation, a space again opens for an application of program analysis techniques, which are able to infer adaptation options from both control and data structures and dependencies.

1.1 Thesis Objectives

The overall objective of this thesis is to leverage program analysis tools and techniques developed for the analysis of (constraint) logic programs, and apply them to problems
that arise in the area of Service-Oriented Computing (SOC). In particular, that includes
the application of computation cost (i.e., complexity or resource usage) and the sharing
analyses for logic programs to service compositions, as well as the development of con-
straint/logic programming models of services, service compositions, and their Quality of
Service (QoS) attributes and properties.

The methodology used in pursuit of these objectives focused on verifying applicability
and usability of the program analysis and dynamic system analysis approaches in the
real world context of Service-Oriented Architectures, such as: QoS-based matchmaking
and adaptation, QoS estimation, and QoS-oriented predictive monitoring. The underly-
ing formalisms for describing the service based systems and reasoning about them in-
clude formulation of constraint satisfaction and logical inference problems. To that end,
several prototypes of tools for service execution, monitoring, etc., were developed to al-
low both the simulation and empirical collection and analysis of data from experimental
executions in real-world based scenarios.

1.2 Structure of the Thesis

Besides the current, introductory chapter, this thesis has the following parts:

**Chapter 2 Background** gives an overview of the key concepts related to Service-
Oriented Computing (SOC) and the program analysis techniques used in the re-
main ing parts of the thesis.

**Chapter 3 Supporting Predictive Monitoring and Proactive Adaptation With
Automatic Complexity Analysis** presents an approach for increasing the accu-
cracy of Quality of Service (QoS) analysis for service orchestrations. We focus here
on the notion of computational cost and resource usage, in a wide sense, which can
be linked to QoS characteristics. To attain more accuracy, computational cost / re-
source usage are expressed as functions on input data (or appropriate abstractions
thereof) and we show how these functions can be used to make more informed
decisions when performing composition, proactive adaptation, and predictive mon-
itoring. We describe how to automatically synthesize these functions from orches-
trations, as well as how to effectively use them to increase the quality of non-trivial
service-based systems with data-dependent behavior.
Chapter 4 *Constraint-Based Runtime Prediction of SLA Violations in Service Orchestrations* addresses the problem of predicting violations of Service Level Agreements (SLAs) in highly dynamic, adaptive and mission-critical applications that require continuous, instance level monitoring of the executing service orchestrations. The proposed approach derives the constraints that model SLA conformance and violation from the structure of the orchestration and the assumed component QoS ranges, in principle at any given point of the execution of a service composition. The results can be used to perform optimized service matching or trigger preventive adaptation or healing. For greater accuracy, this approach can be combined with the computation cost analysis from Chapter 3. We study the impact of different factors, such as the inaccuracy or imprecision of the component QoS assumptions on the quality of the constraint-based prediction.

Chapter 5 *A Constraint-Based Approach to Quality Assurance in Service Choreographies* extends and enriches the constraint based prediction techniques from the previous chapter to treat service choreographies. While end-to-end SLAs are well suited for request-reply interactions, more complex, decentralized, multi-participant compositions (service choreographies) typically need multiple message exchanges between stateful parties and the corresponding SLAs thus involve several cooperating parties with interdependent QoS. The usual approaches to determining QoS ranges structurally (which are by construction easily composable) are not applicable in this scenario. Additionally, the intervening SLAs may depend on the exchanged data. The chapter presents an approach to data-aware QoS assurance in choreographies through the automatic derivation of composable QoS models from participant descriptions. Such models are based on a message typing system with size constraints and are derived using abstract interpretation. The models obtained have multiple uses including run-time prediction, adaptive participant selection, or design-time compliance checking.

Chapter 6 *A Sharing-Based Approach to Supporting Adaptation in Service Compositions* presents an approach for automated inference of data properties based on sharing analysis, which is able to handle service compositions with complex control structures, involving loops and sub-workflows. The properties inferred
can include functional data dependencies, information content, domain-defined attributes, privacy or confidentiality levels, and others. The analysis produces characterizations of the data and the activities in the composition in terms of minimal and maximal sharing, which can then be used to verify compliance of potential adaptation actions, or as supporting information in their generation. This sharing analysis approach can be used both at design time and at run time. In the latter case, the results of analysis can be refined using the composition traces (execution logs) at the point of execution, in order to support run-time adaptation.

Chapter 7 Conclusions and Future Work presents the concluding remarks and an outline of future work.

1.3 Contributions

The main contributions of this thesis include:

- **Automatic computation cost analysis for service orchestrations**, based on translation into the form of a logic program amenable to analysis using the existing tools. The scheme accepts a description of orchestration written in a subset of BPEL [JEA+07b], with a subset of XPath as an expression language, and expresses the computation cost in terms of number of simple activities (steps) and the component service invocations as a function of input data size.

The initial proposal was presented in MONA+ 2009, and has dealt with using the derived computation cost bounds functions to predict the QoS bounds for an orchestration instance at the point of invocation:


This was later extended into a proposal for computation-cost based dynamic adaptation presented in ICWS-2010, where the complexity functions are used as a cri-
teria for dynamic selection of component services with the aim of minimizing the total composition cost:


These two papers are the basis for Chapter 3.

- Accurate and efficient QoS prediction techniques based on constraints for run-time, instance-level continuous monitoring of adaptable service compositions.

The paper presented in ICSOC-2011 (which received the Best paper award):


proposes an approach to continuous prediction based on continuations and a structural derivation of QoS constraints (for one or more QoS attributes, such as the execution time or availability) from the building blocks of an orchestration (its basic and complex activities), and the empirically derived QoS ranges for the component services. The approach is able not only to predict possible SLA violations, but also the structural parameters (e.g., branch conditions and loop iteration counts) that correspond to the cases of SLA compliance, possible SLA violations, and imminent SLA violations, respectively. The prediction approach is not probabilistic, but rather based on (crisp) logical reasoning on constraint satisfiability for the three cases. Predictions can be made at any step in the execution of the orchestration (for which a continuation is available) with a minimal processing overhead.

Some effects of inaccurate and imprecise information about the QoS ranges for the component services are explored within an empirical evaluation framework in the paper presented in PESOS-2012:
These two papers form the basis for Chapter 4.

The paper to be presented in ICSOC-2012:


extends the constraint-based prediction approach to cover service choreographies. It handles several new challenges, such as modeling complex interaction patterns between participants in a choreography in a modular way, and handling stateful conversations. Besides the constraint based modeling for participants, this involves message size analysis and joint constraint solving. This paper is the basis for Chapter 5.

• Sharing analysis for service workflows, with complex control structures (loops, branches, sub-workflows) and data dependencies.

The paper presented in ICSOC-2010:


presents a scheme for translation of an abstractly defined workflow into the form of a Horn clause (logic) program, which is then amenable to sharing analysis using the existing tools developed for sharing analysis of Prolog programs, such as
CiaoPP [HPBLG05]. It uses a lattice characterization of workflow inputs and interprets the results of the sharing analysis as a lattice that describes a partial ordering of activities in the workflow relative to the input data lattice, which can then be used for dividing the original orchestration into fragments that can be executed in a distributed manner.

The paper presented in SCC-2011:


expands the sharing approach by proposing a design-time methodology for analyzing user-defined data attributes in service orchestrations based on sharing. The inferred attributes can be used for information-flow driven fragmentation, but also for other tasks, such as component selection and compliance checking. The attributes are represented and analyzed using concept lattices, a notion from Formal Concept Analysis [GSW05], which enables easy and intuitive handling of the analysis inputs and outputs from the point of view of a service designer.

An extended version of these two publications has been accepted for publication in the Special Issue on Adaptation of the Springer’s Computing journal. It further extends the approach to the analysis of functional dependencies based on sharing, and is the basis for Chapter 6.
Chapter 2

Background

The aim of this chapter is to present the basic background concepts and approaches upon which the matter of the following chapters is based. The treatment of these concepts here is rather informal. More formal treatment can be found in the references mentioned throughout this chapter, and the subsequent chapters contain additional information where necessary. We start with the concepts from service-oriented computing, continue with logic (Horn clause) programs, and then briefly present the complexity analysis and the sharing analysis for logic programs.

2.1 Services and Service Compositions

As mentioned in Chapter 1, Service-Oriented Computing (SOC) has become a widely established paradigm for development of flexible, distributed, complex software systems [PTDL07]. SOC involves different aspects: the Service-Oriented Architecture (SOA) for conceptually organizing software systems, various development methodologies, engineering principles for Service-Based Applications (SBAs), as well as a set of standard implementation technologies: SOAP/HTTP for message transfer [GHM+07], WSDL for describing services [CCMW01], and UDDI for service registries [CHvRR04].

Figure 2.1 shows the basic roles for the components in SOA. Service providers publish their services, typically using some sort of service registry. Service clients (which are

---

1 Other service technologies, such as RESTful services [PW10], have been successfully proposed and applied in many areas, but in this background overview we stick to the “classical,” WSDL-based services.
At the core of SOA are the Web services, which are self-contained modules that implement a coherent set of tasks. Web services are primarily meant to be used as software components by machines, and are available as resources on the Internet. They are self-describing (using WSDL) and discoverable, and their interfaces, types, addressing, etc., is standards based, independent on the programming language or the operating system used for implementation and deployment.

Basic (or atomic) Web services are typically interfaces to back-end software components that perform particular processing tasks. These are typically coarse-grained components that group together a logically coherent set of basic operations, which do not depend on other services. Service compositions, on the other hand, realize more complex tasks or business processes on top of the basic services, and these tasks and processes can span organizational and information system bounds. Compositions can be modeled using a number of formalisms, including Petri Nets [EL02], π-calculus [Mil91], timed state automata [KPP06], and a number of business process modeling notations and approaches, such as UML or BPMN [Obj11]. Executable specification languages for business processes include BPEL [JEA+07b], CDL [Wor05], Yawl [vdAtH05], DecSerFlow [vdAP06a], and many others.

In SOC, two standard views of a service composition are called the orchestration and the choreography. The former expresses the composition as a computational procedure with a centralized flow of control, while the latter models it as a set of interacting participants that communicate using some pre-defined message exchange protocol. The business process languages, such as BPEL, are therefore more oriented to orchestrations and
concerned with expressing program control constructs (with sequential, as well as parallel flows), state variables and operations on them, while the orchestration languages, such as CDL, tend to be more oriented towards the issues of coordination, transactional consistency, and message passing protocols.

Quality of Service (QoS) plays an important role in SOC. It covers mostly non-functional attributes (properties) of services (and their compositions), such as the execution time, monetary cost, availability, and network bandwidth consumed [KPP+13, Gro05]. Since services are typically provided and controlled by third parties (i.e., they cannot be downloaded or modified by service clients), to meet their specific requirements the clients often need to take into account the QoS of the service (composition) they use. While low-end services are typically provided on the best-effort basis (i.e., without any guarantees by the provider), real-world business application typically use the Service Level Agreements (SLAs) [SZL06] that specify the QoS levels for service rendition agreed between the providers and the clients.

QoS analysis for services and service compositions has several important applications in SOC. Firstly, enriching service descriptions with QoS information can help reasoning about and selection of those services that meet client’s particular QoS requirements [KP07, DSS+04, GZ07, ZBC10]. Secondly, since the QoS-related information often does not come pre-computed, different QoS analysis approaches for service compositions have been proposed for automating that process [MBC+10, BBC+10, MJN08, BMF+08, FPR07, RSD06, JRG04]. Thirdly, QoS for service composition can be predicted at runtime, so that the composition can be adapted to avoid or mitigate SLA violations [ACM+07, ZLC07, HT05, LWK+10, LMRD10, LWR+09]. And fourthly, QoS itself can be used as a criteria for automated generation of QoS compositions [AR09, AP07, CLJW06, CPEV05a, CPEV05b, FHW+05, GJ05, JMG05, Car02].

2.2 A Primer on Logic Programs

Logic programming is a declarative programming paradigm which represents the computation task as a logic inference problem, which is solved by applying an appropriate inference (i.e., program execution) mechanism [CH11]. Prolog [PRO94, PRO00], with its many variants and implementations [HBC+12, Wie10, SSW93, SCDR11], is certainly
the most widely spread and applied logic programming language. Over the past several decades, the initial Prolog-based concept of logic programming (LP) [Llo87] has been gradually extended into constraint logic programming (CLP) and concurrent constraint logic programming (CCLP) [AF03], along with the addition of tabling [RRS+95], constraint handling rules (CHR) [Fw09], distributed and parallel execution constructs [CH95], functional programming techniques [Llo94], advanced module systems and object orientation [Mou00], and many other syntactic and semantic extensions. In this section we concentrate on some key aspects of a fragment of the classic Prolog (without negations) that suffice in the scope of this thesis.

### 2.2.1 Syntax

Figure 2.2 shows the basic elements of Prolog syntax, using mathematical notation. The concrete notation elements will be typed in a fixed width font as we go. A program consists of a sequence of clauses, of the form \((H ← G)\), where in the concrete syntax “−” is written as “:−”, the enclosing braces are omitted, and the clause is terminated with a full stop. Literal \(H\) is called the head of the clause, and is always some predicate named \(p\) applied to \(n\) arguments. Goal \(G\) is called the body of the clause, and it can be another lit-
eral, a comma-separated conjunction (sequence) of goals, a semicolon-separated disjunction (choice) of goals, or a special goal $\top$ which is always trivially true. In the concrete syntax, the braces surrounding the conjunctions and disjunctions are normally omitted, with the understanding that a comma binds stronger than a semicolon, and both associate to the right. Thus, “$G_1,G_2,G_3;G_4;G_5$” is the same as “$(G_1,(G_2,G_3));(G_4;G_5)$.” Also, $\top$ is written “true”.

The arguments in a literal are called terms, and they represent data in Prolog programs. A term $t$ can be either a numeric constant, an atom (or symbolic constant), a variable symbol that represents some (or an arbitrary) term, or a structured term of the form $f(t_1,t_2,...,t_n)$, $(n \geq 1)$, where $f$ is a function symbol, and $t_1,t_2,...,t_n$ are the argument terms. Unlike most programming languages, in pure Prolog, $f(a,b)$ is not a function call that invokes computation of the value of $f$ for given $a$ and $b$, but can be seen instead as record named $f$ with two fields containing $a$ and $b$, respectively. A term that does not contain variable symbols is said to be ground.

In the concrete syntax, predicate names, atoms and function symbols are written either as identifiers starting with a lowercase letter (e.g., “a”, “append”, “x_call_2b”), strings of operator characters (e.g., “+”, “>=”, “@=<”), or arbitrary strings enclosed within single quotes. As a special case, “[]” is also a legal symbolic name, usually used to denote an empty list. To disambiguate these symbolic names from variable names, the latter are always written as identifiers that start with a capital letter or an underscore (e.g., “X”, “Next”, “_x” or “_23”).

The recursive syntax for the structured terms allows creation of arbitrary data structures in Prolog. Some of them, such as lists have a special syntax representation using square brackets. A list with the first element (head) $t_1$ and the remaining elements (tail) $t_2$ can be written either as “.(t_1,t_2)”, using function symbol “.” (dot), or more readably as $[t_1|t_2]$. A list of three elements $t_1$, $t_2$ and $t_3$ can be written as “[t_1,t_2,t_3]”, “[t_1,t_2,t_3][]”, “[t_1|[t_2,t_3]]”, and in several other equivalent ways. Besides, Prolog allows use of prefix, infix, and postfix operators which allow us to write “+(a,+(b,x))” in a more familiar way as “a + b * x”. However, the cannonical syntax for structure terms can always be used.

Figure 2.3 shows a simple program in the concrete Prolog syntax for deciding whether a route exists between two airports in a directed graph of connections. The first five lines are facts describing the predicate leg/2 (i.e., of two arguments). Their
Figure 2.3: An example route finding program in Prolog.

body is always true, so the notation $H \leftarrow G$ is abbreviated into $H$. Comments start with “%” and extend to the end of the line. The last two lines are the rules for the predicate route/2, whose meaning we discuss in the following subsection.

### 2.2.2 Semantics

Two usual formal ways of defining the semantics of logic programs written in Prolog are the fixpoint semantics and the operational semantics based on SLD resolution. The former tries to answer what is the set of all (ground) literals that can be inferred from the given program. The latter deals with the Prolog's mechanism for executing programs step by step. The details of the two semantics approaches have been extensively studied and can be found in the literature [Llo87].

In our less formal approach, we first describe a logical reading (the declarative semantics) of Prolog clauses. A clause “$H \leftarrow G$” can be understood as an implication:

$$
\forall \bar{x} \cdot (\exists \bar{y} \cdot G) \rightarrow H
$$

where $\bar{x}$ are all the variables appearing in $H$, and $\bar{y}$ are the remaining variables in the clause. When $\bar{x}$ or $\bar{y}$ is empty, the corresponding quantifier is omitted. Formulas of the form (2.1), where $G$ is a single literal or a conjunction of literals are called Horn clauses.

The clauses of route/2 in Figure 2.3 can therefore be written as two logic formulas:

$$
\forall x, y \cdot [\text{leg}(x, y) \rightarrow \text{route}(x, y)]
$$

$$
\forall x, y \cdot [(\exists z \cdot \text{leg}(x, z) \land \text{route}(z, y)) \rightarrow \text{route}(x, y)]
$$

which use conjunction to logically model the comma-separated sequences of goals. This
logical reading is always based on the *closed world assumption*, which means that the only conclusions that can be derived from the logical reading of the program are those that are either directly present in it as facts, or can be derived from the program facts and rules. In the context of the program from Figure 2.3, this means that we cannot infer that there exists a route from vie to fra, because the appropriate facts are not present in the program.

On the operational side, Prolog programs are normally written with the aim not only to check whether some literal is true or not, but more interestingly to find the values for which it holds. In the execution of a logic program, variables serve as placeholders for terms that satisfy the logical conditions encoded in the program. The execution is triggered with a goal, e.g. `route(mad,X)` which can be interpreted as “find x such that route(mad,x) can be proven from the program.”

The standard Prolog SLD execution method can be understood as a systematic, depth-first search for a proof of the goal. It searches for the first clause that corresponds to the goal, and then traverses the goals in the body from left to right. It uses chronological backtracking to retry the next program clause of a predicate to look for another solution. Whenever a clause is selected, it uses a new set of uniquely named variables. When successful, the execution returns a *substitution* that maps variable symbols from the goal to terms that make it true (i.e., provable from the program).

Each successful execution step may map a previously unmapped variable to a term. Thus, a substitution $\theta$ is a finite set of mappings of the form $x \rightarrow t$ where $x$ is a variable and $t$ is a term. Each variable can appear only on one side of the mappings in a substitution. At any point during execution, the current value of the variables in the program is obtained by applying the *current substitution* to the variable symbols. If $\theta = \{x \rightarrow 1, y \rightarrow g(z)\}$, and $t = f(x, y)$, then the application $t\theta$ gives $f(1, g(z))$. Substitutions at successive program steps are composed together to produce an aggregated substitution for the next program point (or the final substitution on exit). E.g., for the previous $\theta$ and $\theta' = \{z \rightarrow a + 1\}$ (with $a$ being an atom, not a variable), we have $\theta\theta' = \{x \rightarrow 1, y \rightarrow g(a + 1)\}$.

Substitutions are generated by *unifications*, which take place between the corresponding arguments in the current goal and clause the selected clause head, as well as using a special predicate `/2`. Unifying $t_1$ and $t_2$ gives a substitution $\theta$ which ensures
route_t(X,Y,[X,Y]) :- leg(X,Y).
route_t(X,Y,[X|R]) :- leg(X,Z), route_t(Z,Y,R).

Figure 2.4: A modified route finding predicate.

that $t_1\theta$ and $t_2\theta$ are identical terms by introducing the least amount of new information. Unifying $x$ and $f(y)$ gives $\theta = \{x \mapsto f(y)\}$; unifying $f(x,a+1)$ and $f(1,z+y)$ gives $\theta = \{x \mapsto 1, z \mapsto a, y \mapsto 1\}$; and the attempt to unify $f(x)$ and $g(y)$ fails, because the functors are different.

As an example of the SLD semantics, we take the predicate route_t/3 from Figure 2.4, such that route(x,y,z) means “there is a route from x to y and z is a list of points along that route.” The initial goal is route_t(mad,vie,R). In the first step, the first clause of route_t/2 is chosen with fresh variables:

$$\text{route}_t(X_1,Y_1,[X_1,Y_1]) :- \text{leg}(X_1,Y_1)$$

and the unification between the goal and the head of the clause produces substitution $\theta_1 = \{X_1 \mapsto \text{mad}, Y_1 \mapsto \text{vie}, R \mapsto [\text{mad}, \text{vie}]\}$. The current goal now becomes the body of the clause under the current substitution: $\text{leg}(X_1,Y_1)\theta_1 \equiv \text{leg}([\text{mad}, \text{vie}])$. However, this cannot be unified with any fact in the program about leg/2, so $\theta_1$ has to be abandoned and the execution backtracks to try the second clause for route_t/3:

$$\text{route}_t(X_2,Y_2,[X_2|R_2]) :- \text{leg}(X_2,Z_2), \text{route}_t(Z_2,Y_2,R_2)$$

and the unification with the head gives $\theta_2 = \{X_2 \mapsto \text{mad}, Y_2 \mapsto \text{vie}, R \mapsto [\text{mad}, \text{vie}]\}$. The first goal in the body, $\text{leg}(X_2,Z_2)\theta_2 \equiv \text{leg}(\text{mad}, \text{vie})$ unifies with the first clause for $\text{leg}/2$, giving $\theta_3 = \{X_2 \mapsto \text{mad}, Y_2 \mapsto \text{vie}, R \mapsto [\text{mad}, \text{vie}], Z_2 \mapsto \text{fra}\}$. The execution moves to the next goal in the body, $\text{route}_t(Z_2,Y_2,R_2)\theta_3 \equiv \text{route}_t(\text{fra}, \text{vie}, R_2)$. This is now a recursive call to route_t/2. Again, a fresh copy of the first clause is selected:

$$\text{route}_t(X_3,Y_3,[X_3,Y_3]) :- \text{leg}(X_3,Y_3)$$

and the unification with the head gives $\theta_4 = \{X_2 \mapsto \text{mad}, Y_2 \mapsto \text{vie}, X_3 \mapsto \text{fra}, Y_3 \mapsto \text{vie}, R_2 \mapsto [\text{fra}, \text{vie}], R \mapsto [\text{mad}, \text{fra}, \text{vie}], Z_2 \mapsto \text{fra}\}$. The next goal is $\text{leg}(X_3,Y_3)\theta_4 \equiv \text{leg}(\text{fra}, \text{vie})$ which is directly found in the program facts. Thus, the execution completes successfully, with the response $R = [\text{mad}, \text{fra}, \text{vie}]$. The user can force backtracking to find another
solution, \( R = [\text{mad}, \text{muc}, \text{fra}, \text{vie}] \), and the final one, \( R = [\text{mad}, \text{muc}, \text{vie}] \), after which backtracking fails, and the Prolog engine reports no more solutions.

Note that the SLD resolution is affected by the order of clauses in the program, and can result both in the infinite number of solutions and the infinite loops that blocks a solution. The former can be caused by e.g., adding \( \text{leg(fra,mad)} \) as the last fact for \( \text{leg/2} \) in Figure 2.3, and the latter by e.g., placing \( \text{leg(mad,mad)} \) as the first fact.

### 2.3 Computation Cost Analysis

The computation complexity analysis is one of the traditional areas of Computer Science [AB09] and one of the cornerstones of algorithm analysis and design [AHU74, GBY91, KT05]. The dimensions of complexity most frequently treated are the execution time and memory space, and the usual analysis of computational complexity is typically interested in finding the asymptotic complexity of solving a problem in question, and identifying whether it is tractable.

While the classical computation complexity analysis pervades all areas of computing, it is still pretty much a human task, which in many cases involves theorem proving and other techniques that are hard to automate. The need for automated cost analysis of larger software systems, for which the traditional methods of human driven complexity analyses are impractical, has given rise to the discipline of computation cost analysis, also known as the automatic complexity analysis, or resource consumption analysis. The most common variety is the static cost analysis, which, for a given class of inputs to a procedure, tries to infer the bounds of some interesting measure of cost (processing steps, function/method calls, floating point operations, etc.), without actually executing the procedure.

The static cost analysis approaches commonly measure computation cost using some higher-level abstractions, relatively independently of the performance characteristics of the underlying hardware. In contrast, a related approach called Worst-Case Execution Time (WCET) analysis [WEE+08], while also partly relying on static analysis, is closely related to the physical, low-level parameters, and the real-time guarantees that the mission-critical production systems need to offer.

In general, the static cost analysis approaches have two main phases [AAGP11]. In
the first phase, the code is analyzed and transformed into a set of cost relations between the measures of input data (e.g., list and string sizes, integer values) and the complexity measures. In the second phase, the cost relations are solved to produce the cost bounds as functions of input data. Both phases are non-trivial and may require use of other program analysis techniques, such as partial evaluation and abstract interpretation.

One of the main challenges of the static cost analysis is its precision, since in general the problem of inferring complexity is equivalent to the Halting problem, and thus undecidable. The sources of imprecision are in both phases, and depend on the ability to capture the relevant aspects of data (e.g., treating arrays of objects or primitive values in Java, advanced control and concurrency constructs), as well as on the ability to solve the cost relations and obtain their closed form. Obviously, a strong influential factor in this respect is the richness and complexity of semantics of the programming language. Object orientation, code instrumentation and complex modularization bring additional challenges in this respect [AAG’11].

Logic programs have a mathematically well-founded semantics that makes them easier to analyze. The basic approaches for cost analysis of logic programs have been laid out relatively early [DLGHL94, Deb95], and were followed by practical implementation of the cost analysis tools for logic programs that form the basis of the present cost analyzers in the CiaoPP system [BLGPH06]. This analysis was later expanded into a user-definable resource usage bounds analysis [NMLGH08] for logic programs and Java bytecode [NMLH09].

2.4 Sharing Analysis for Logic Programs

Sharing analysis for logic programs we discuss here belongs to a group of static analysis techniques known as the aliasing analyses, which are in general concerned with detecting when two pointers (or object references) point to the same data structure or a region in the memory heap, or when two data structures contain or reference (that is, share) a same data object. In imperative programming languages, aliasing analysis can help optimize heap usage or detect potentially unsafe operations, such as potential memory leaks or cases of freeing the same memory block twice.

In logic programs, pointer operations are not directly permitted, but logic variables
can be seen as a kind of “sanitized pointers” that may under substitutions refer to other terms, which in turn may contain other variables, etc. Mapping a variable to a non-variable term during the execution of a logic program affects all terms that refer to that variable. A term \( t \) becomes more instantiated under a substitution \( \theta \) if \( t \) and \( t\theta \) cannot be made identical by means of a variable renaming. This can only happen if \( \theta \) contains a mapping of the form \( x \rightarrow v \), where \( x \) appears in \( t \) and \( v \) is a non-variable term.

At some point in execution of a logic program, characterized by the current substitution \( \theta \), two terms \( t_1 \) and \( t_2 \) are said to share if there exists some variable \( x \) which appears in both \( t_1\theta \) and \( t_2\theta \). A point in the program text can be reached from different execution traces, and in case of recursive calls in the general case the number of traces entering the same point in the program (e.g., selecting the same clause for execution) may be impossible to decide or infinite. Besides, there may be an infinite number of terms that can be passed as the predicate arguments. All of that suggests that the problem of sharing cannot be tackled by trying to enumerate possible substitutions at a point in code.

The sharing analysis for logic programs therefore uses the techniques of abstract interpretation [CC77a], which interprets a program by mapping concrete, possibly infinite sets of values (in this case the substitutions) onto usually finite abstract domains, together with operations, in a way that is correct with respect to the original semantics of the programming language. In the abstract domain, computations usually becomes finite and easier to analyze, at the cost of precision, because abstract values typically cover (sometimes infinite) subsets of the concrete values. However, the abstract approximations of the concrete behavior are safe, in the sense that properties proven in the abstract domain necessarily hold in the concrete case. Whether abstract interpretation is precise enough for proving a given property depends on the problem and on the choice of the abstract domain. Yet, abstract interpretation provides a convenient and finite method for calculating approximations of otherwise, and in general, infinite fixpoint program semantics, as is typically the case in the presence of loops and/or recursion.

We here focus on the abstract interpretation-based sharing, freeness, and groundness analysis for logic programs [MH92]. Instead of analyzing the potentially infinite universe of possible substitutions during execution of a logic program, sharing analysis is concerned just with the question of which variables may possibly share under the possible substitution.
Figure 2.5: An illustration of the relationship between a concrete and an abstract substitution.

Figure 2.5 shows an example relationship between an abstract and a concrete substitution. The abstract substitutions is a set of so-called sharing settings, where each setting groups together the variables that share some other variable(s). Obviously, abstract substitutions completely abstract from the shape of the terms, function symbols, structure nesting, or the choice of the actual shared variables. Each abstract substitution corresponds to an infinite number of concrete substitutions, and constructing one concrete substitution from the abstract one (as a witness) is easy. More formal treatment of abstract substitutions can be found in Chapter 6.
Chapter 3

Supporting Predictive Monitoring and Proactive Adaptation With Automatic Complexity Analysis

One distinguishing feature of the Service Oriented Computing (SOC) systems is that they are expected to be active during long periods of time and span across geographical and administrative boundaries. These characteristics require having monitoring and adaptation capabilities at the heart of SOC. Monitoring compares the actual and expected system behavior. If too large a deviation is detected, an adaptation process (which may involve, e.g., rebinding to another service provider) may be triggered. When deviations can be predicted before they actually happen, both monitoring and adaptation can act ahead of time (being termed, respectively, predictive and proactive), performing prevention instead of healing.

Comparing the actual and the expected level of the Quality of Service (QoS) of a composition — even assuming the composition is static — is far from trivial. Detecting deviations requires a behavioral model, which is used to check the current behavior or to predict a future behavior. Naturally, the more precise a model is, the better adaptation / monitoring results will be achieved. In this chapter we will develop and evaluate models which, based on a combination of static analysis and actual run-time data, achieve increased accuracy by providing upper and lower approximations of computational cost.
resource usage measures which can be related to QoS characteristics. For example, the number of service invocations can be related to execution time when information about network speed is available.

Two factors, at least, have to be considered when estimating QoS behavior: the structure of the composition itself, i.e., what it does with incoming requests and which other services it invokes and with what data, and the variations on the environment, such as network links going down or external services not meeting the expected deadlines. In this chapter, we present an approach that uses automated computation cost analysis to infer the complexity bounds, in terms of composition activities, based on an (abstraction of) input requests. In Section 3.2 we motivate the use of computation cost analysis in the service world by discussing the benefits it can bring in terms of prediction precision. Section 3.3 presents the approach, by outlining the process of translation into an intermediate language amenable to cost analysis by the existing tools. Sections 3.4 and 3.5 present the results of experimental evaluation of the approach in the context of predictive monitoring and proactive adaptation, respectively.

3.1 Related Work

Impact of the environmental factors (e.g., the system load, network latency or availability) on the composition QoS been extensively studied by several authors who have proposed several frameworks for QoS analysis in the context of executable BPEL processes [MJN08, JEA+07b], real-time applications [BMF+08, FPR07], and model-driven development [WY06]. Some more recent proposals use data-mining [LHD11] and runtime verification [SM11] techniques to predict possible violations of the Service Level Agreements. Most of these approaches do not relate QoS with input data and operations on them, and represent QoS levels using a probabilistic aggregation of the expected (average) values, therefore providing limited information about their bounds and dispersion. In this chapter, we argue that explicit handling of data-aware QoS bounds provides an additional level of precision.

In his work on foundations of complexity analysis for service workflows and executable BPEL processes, Cardoso [Car05, Car07] has recognized the relevance of taking into account the actual message contents and the actual data managed by the composi-
tion. It has been recognized that those prediction techniques which do not take run-time data and other parameters into account are potentially inaccurate. However, this aspect of the analysis has not been addressed adequately, mostly due to the fact that an integrated analysis of both control and data flow represents a highly non-trivial problem in itself.

In this chapter we focus on applying a methodology, based on previous experience on automatic complexity analysis [DL93, NMLGH07, MLNH07], which can generate correct approximations of complexity functions via translation to an intermediate language (Section 3.3). These functions use (abstractions of) incoming messages in order to derive safe upper and lower bounds which depend on the input data and which are potentially more accurate than data-unaware approximations. In Section 3.2.5 we show how these functions can be used by monitoring to make better decisions and in Section 3.4 we make an experimental evaluation.

Correct data-aware computation cost functions for services and service compositions can be useful in any situation where a more informed QoS estimation is an advantage. In particular, QoS-driven service composition [CPEV05a, ZBN+04, CLJW06] can use them to select better service providers for an expected kind of request, and adaptation mechanisms can also benefit from that knowledge.

### 3.2 Computation Cost Analysis in the Service World

In this section, we motivate the use of computation cost analysis as a useful tool for improving our ability to analyze and predict QoS attributes of a service composition. We start with a simple example that elucidates the significant impact of data with which a service composition is invoked on its complexity. We then look at a general framework for computational cost analysis of service orchestrations, and how it can be used to approximate the actual composition behavior in terms of QoS. Next, we look at the benefits of using data-dependent upper and lower bounds for describing QoS and how the computational cost bounds can be used to approximate them. Finally, we look at the usage of the computational cost function in the context of run-time monitoring.
3.2.1 Impact of Data on Composition QoS

A number of factors can influence QoS attributes of a service composition at run time. One group, which is usually not under control of the service designer, includes infrastructural and environmental factors, such as network speeds and latencies, system load, or processing speed. The second group of factors depends on the structure (i.e., the logic) of the composition, and is usually under full or at least partial control of the designer. Here, we focus on the latter, and for simplicity we look at service orchestrations, which are usually provided either as stand-alone artifacts, or projections of a multi-participant composition (a choreography) from a single-participant point of view.

Structure of a composition typically specifies control structures and dependencies (such as sequences, branches, parallel flows, loops, and external invocations), whose behavior may in some very simple and special cases be fixed statically, but is, in general, driven by data that is received upon invocation, or obtained as a result of processing.

Fig. 3.1 shows a fragment of a (simplified) part reservation system for car production. A part Provider serves its Client by reserving a number of part types from a pool of part Makers. The protocol only allows the Provider to reserve one part type per service invocation to a Maker. An invoked Maker replies with an “ok” message if the requested part type is available, and with an “not ok” otherwise; in the latter case the Provider goes to another Maker. If he Provider cannot reserve some car part from any of the Makers, it cancels all previously reserved part types by sending “cancel” messages. We assume that the Client is a car production line and that the parts it requires from the Provider are always specified in quantities that are sufficient for production of a series of
cars of some fixed size. This is also an example of a service composition that may take a longer time to execute and cannot be wrapped inside a single database transaction, thus creating the need for programmatic cancellation.

In this example, it is quite obvious that the value of a QoS attribute of a Provider service composition, such as the execution time or a monetary cost, strongly depends on data. In this case the size $N$ of the list of part types to be reserved influences the number of messages exchanged between the Provider and the pool of $K$ Makers. In the best case, when the first selected maker can provide all requested parts, the number of the exchanged messages is $2N$ (one reservation and one “ok” message for each part). In the worst case, the provider may reserve $N-1$ parts by trying each of $K$ Makers, fail to reserve the $N$-th part, and finally canceling the $N-1$ previous reservations. In the latter case, the number of exchanged messages is $N(2K+1)-1$.

To summarize, in our example the number of messages exchanged between the Provider and the pool of Makers grows linearly with the size of the input request $N$, and it is the key factor that drives execution time and costs, if we assume that the rest of computations by the Provider are take a constant time or cost.

### 3.2.2 Computation Cost of Service Networks

If a Provider can choose between different Makers, then the values of QoS attributes, such as the execution time, will in general depend on both the internal logic of the selected service, and the data that is passed to it. Computational cost analysis aims at statically determining the computational cost (in terms of e.g., number of execution steps or events) of a given procedure for some abstraction of input data, such as its size. The function which results from the analysis of the computational cost depends on the internal logic of the service composition (the Provider, in our example), but also on the behavior of the invoked services (the Makers), as they may, in turn, send additional messages which add to the overall count.

Fig. 3.2 depicts this scenario in some detail. The input message is abstracted in this example as a parameter $n$ (i.e., the number of car part types in our example) on which some measure of computational cost depends. The cost of service $A$ is $T_A(n)$. As $A$ invokes $n$ times another service, (whose cost is represented by a generic $S$), for which $B_1$ and $B_2$ are two candidates with different computational cost, its overall computational
cost depends as well on which service is selected to perform the composition. Using the $T(n)$ values from Fig. 3.2, the computational cost corresponding to these two options would be:

\[
T_{A_1}(n) = 2n + 3 + n(n + 1) = n^2 + 3n + 3 \quad A + B_1
\]
\[
T_{A_2}(n) = 2n + 3 + n(.1n + 7) = 0.1n^2 + 9n + 3 \quad A + B_2
\]

and to decide between $B_1$ or $B_2$, $T_{A_1}$ and $T_{A_2}$ have to be compared (Fig. 3.3). This opens up the possibility of taking into account the size $n$ of the data to select a configuration.
depending on the expected usage, and it requires information about $B_1$ and $B_2$ in order to automatically work out the resulting overall computational cost.

The computational cost-related information for $B_1$ and $B_2$ can be made available in much the same way as other service-related information (e.g., interfaces or XML schemes) is published. It needs to include, at least, the expected computational cost (preferably as a function of input data characteristics) and (possibly) the relationship between the sizes of the input and output data for every operation in the interface. The availability of these descriptions can make it possible to automatically work out $T_{A_1}$ and $T_{A_2}$ to compare them. In turn, $A$ should publish the information it synthesizes, so that it can then be used by other compositions. In our view, this repeated process of synthesis, comparison, and publishing, is a step forward towards simultaneously achieving true dynamicity and optimal selection in the creation and adaptation of service networks.

Note that these abstract descriptions do not compromise the privacy of the implementation of the service being described, as they act as high-level contracts on the behavior of the service. Besides, in an open ecosystem of services, those which publish such descriptions would have a competitive advantage, as they make it possible for customers to make better decisions on which services to bind to.

Given a service $A$, if we assume that any services it invokes have a constant computational cost $T_{B_i}(n) = 0$, then the computational cost obtained for $A$ measures how much its structure alone contributes to the total computational cost. We have termed this the structural computational cost of a service, and it will be used later as an approximation of the real computational cost.

Two key questions are: to which extent functions expressing the cost of the computations are applicable to determining QoS, and to which point these functions can be automatically (and effectively) inferred for service compositions.

### 3.2.3 Approximating Actual Behavior

Basing the analysis of composition QoS on computational cost helps separation of the structural and the environmental factors that affect it. In the standard, statistical or data-mining approach to building models of composition QoS, where we depend on historically recorded data, without looking at the structure, it may be very difficult to separate the structural and the environmental variations. Since both are simultaneously
varying in the model, as in Figure 3.4(a) it is very hard to say what would be the effect of
a change in just one of them: say, of a faster network or processor. Even if we assume that
the historically recorded data on which they are based is representative, such statistical
composition QoS models inherently depends on the historic environmental fluctuations,
and may not necessarily hold at run time.

The computational cost approach, on the other hand, is based on rigorous reasoning
on the structure of the orchestration and its data, and reduces the level of variability of
the composition QoS for given data, as shown in Figure 3.4(b), to only the environmen-
tal factors that are outside the designer’s control, using logically sound computational
cost bounds (explained in the next subsection). Thus, one can plug the current, up-to-
date measured environmental factors into the computational cost QoS model, without
requiring longer term environment stability and an additional period of statistical data
collection.

The computational cost measures we will use count relevant events which are deter-
ministically related to the input data: processing steps, number of service invocations,
size of the messages, etc. To infer such computational costs we follow the approach to
resource analysis of [NMLGH07] which, given data on how much a few selected basic
operations contribute to the usage of some resource, tracks how many times such basic
operations are performed through loops and computes the overall consumption of the
resource for a complete computation. Since the number of loop iterations typically de-
pends on the input, the overall consumption is given as a function that, for each input
data size, returns (possibly upper and lower bounds to) the overall usage made of such
resource for a complete computation.
We assume that different instances of the same event type within an orchestration execute in the same kind of runtime environment, and thus contribute equally to the overall computational cost. Different higher-level QoS characteristics can then be derived from computational cost functions, by combining them with QoS parameters that are observed on the level of composition by means of monitoring [ZLC07]. E.g., execution time can be approximated by aggregating the number of basic activities executed and the number of invocations, and multiplying them by an estimation of the time every (type of) activity and invocation takes, as proposed in [ICH10a]. The availability of a composed service can be expressed as the product of the availability of the services it invokes (assuming independence between them) and, therefore, the availability of the composition will depend on which services are invoked and how many times they are invoked, which in turn depends on the input data.

Estimations of the time used, availability, etc. of basic components are approximate and they thus introduce some noise which also makes the derived QoS functions approximations. However, because they are functions on input data they are likely to predict more accurately the behavior for a given input than a global statistical measure (we return to this later). Besides, for cases where the comparison between two different QoS functions (and not their absolute value) is relevant, as in Fig. 3.2, the noise introduced can be expected to mutually cancel to some extent and therefore it can be ignored.

### 3.2.4 Data-Dependent Upper and Lower Bounds

The precision of a composition QoS model can increase when we add more information about the distribution of values of a QoS attributes for a service component. The most common case in the service world, as shown in the top-left graph in Figure 3.5, is to provide a fixed, average QoS metric value (e.g., for execution time). This is usually fine if we are concentrating on the average case alone, but may not be informative enough when looking at a particular executing instance, because it does not give any idea on how these values are spread around the average. Additional information can be provided along two dimensions. Firstly, instead of (or along with) the average value, we may be provided with bounds that have been calculated to contain some percentage (e.g., 99% or 95%) of the values, which gives us much better idea of the range of QoS values that can be expected from a particular executing instance of the component. Secondly,
Figure 3.5: Levels of information and precision in describing service QoS.

we can be provided with a function that shows how the QoS varies depending on the input data, thus decreasing the overall data-insensitive variability. If we combine both improvements, we end up with the data-dependent upper and lower QoS bounds that provide the highest level of precision (the lower-right graph in Figure 3.5).

Automatically inferred computational cost functions can sometimes be exact, but in general only safe upper and lower bounds can be generated. These are guaranteed to be smaller than or equal to (resp. greater than or equal to) the function they approximate. This can be traced back to limitations of the static analysis, to the actual function depending on more parameters than, e.g., data size, and others. When these bound functions are combined with estimations to determine QoS from computational cost functions, data-aware approximations of the actual bounds are created.

While this may seem to be a disadvantage when it comes to predicting future behavior, upper / lower bounds of the actual computational cost are actually useful to actually ensure that some QoS characteristic is met by making sure it stays above / below the predicted threshold. As an example, Fig. 3.6 portrays upper and lower bound computational cost functions for two compositions for some QoS characteristic which depends on input data. Depending on the QoS meaning, we may want to make sure that we stay above or below some value. The former case needs to consider the upper bound and, con-
versely, the latter requires considering the lower bound. Note also that, in the example portrayed in the figure, which service will give better results depends on the actual data size at run-time.

For some QoS metrics, an interval of lower and upper bounds depending on the input data can be expressed as

$$QoS_{(L,U)}(n) = \langle cost_L(n) \oplus env_L, cost_U(n) \oplus env_U \rangle$$

(3.1)

where the tuple components are the expected lower and upper bounds for the quality of service, $cost_X(n)$ is a function representing a lower / upper bound on resource consumption, $env_X$ represents the minimum and maximum influence of the environment on the QoS attribute at hand, and $\oplus$ is an operation which combines together cost functions and environment conditions.

For example, in the case of the execution time of a single process, $cost_X(n)$ can be the number of activities executed, $\langle env_L, env_U \rangle$ the minimum / maximum time a single activity can take and $\oplus$ would be just multiplication. Since $\langle cost_L(n), cost_U(n) \rangle$ are lower and upper bounds, if $\langle env_L, env_U \rangle$ are also safe lower and upper bounds, the calculated QoS bounds will be safe lower and upper bounds of the actual (runtime) QoS values.
This generic scheme can admit variations: for example, a more accurate approximation can be constructed by assigning different weights to activities so that $env_X$ is an array with a component for the execution time for every type of activity, $cost_X(n)$ is an array counting how many times every type of activity is executed, and $\oplus$ is the vector dot product.

### 3.2.5 Cost Functions for Monitoring

In the previous subsection, we explained how the statically inferred upper- and lower-bound computational cost functions can be used to model composition QoS. Now, we turn to using these functions at run-time to predict QoS for some given executing instance, at run-time, based on the available monitoring data.

In general, the cost of computation is inferred from the structural building blocks of an orchestration: atomic activities, external invocations, and control structures such as branches, loops, and parallel flows. At any point in the execution of the orchestration, there is a notion of the work that remains to be done until it finishes. As an example, the upper bound of an if-then-else activity is the upper bound of the condition plus the maximum of the upper bounds of the then and else branches.

We can associate to every point in the composition execution a measure of how much resources “remain to be spent”. In the if-then-else example, once the if part is over, what remains is the maximum of the upper bounds of the then and the else parts. This value depends on the point in the service composition it is measured and also on the values of the data at that moment. In a loop, where the same activity is executed several times, less “mileage” is left until the end of the execution after every iteration, even in the same point of the composition. The difference comes from the different state of the variables, and this is one of the reasons why taking data into account is beneficial.

There is, therefore, a notion of “remaining” QoS: for example, from the activities still to be executed and the expected time of every activity, the time remaining for the completion of the composition can be derived. Measuring this remaining time is relevant from a monitoring point of view. Assuming that we have a faithful predictor of the QoS remaining until the end of the execution (such as that given by resource consumption functions and environment conditions), then deviations of the environmental characteristics can be used to predict more accurately what will be the QoS at some future point by dynam-
ically combining the cost / resource consumption functions with the actual environment conditions.

Figure 3.7 exemplifies such a situation. Let us assume we are interested in some QoS metric of a composition, whose value must not exceed Max. The cost functions and environment characteristics representing safe upper and lower bounds can be used here: if the **upper** bound is smaller than Max, then we have the guarantee that we do not violate the QoS boundary. If the **lower** bound is larger than Max, then we have the guarantee that we will violate the expected QoS. If none of these hold, then we cannot say anything for sure.

We designate four points (A, B, C, and D) in the execution of some composition and we will focus on how monitoring at these points can be predictively done with the use of cost functions. In Figure 3.7 the solid line represents the QoS initially predicted with the statically inferred cost functions and the expected environment conditions, while the dashed line represents the observed QoS.

At point B, the actual quality has deviated with respect to the predicted one. This can be detected after-the-fact from a measurement at that moment, or in advance by using some other prediction technique. Since the composition has not changed, and thus neither have the cost functions, we can conclude that the deviation can only be due to a change in the environment behavior (e.g., additional load on a server or a faulty network). An updated prediction for the future can be done by using the environment influence observed so far and the existing cost function. This new prediction curve (densely
dotted) still ends, at point D, within the limits of the acceptable range Max. However, at point C a new observation gives yet higher values for the QoS attribute. Yet another function and associated plot curve (sparsely dotted) can be constructed which predicts that the execution will violate the expected QoS. Therefore, at point C we have detected a problem before it appears and we can raise an alarm and maybe trigger an adaptation procedure. In order for this technique to work in complex service compositions with loops, different response times depending on invocations, etc. it is necessary to take data into account from the beginning.

In this chapter we focus on the complexity functions that can be used for giving the initial prediction. In the next two chapters, we show a technique for continuous monitoring of orchestrations and choreographies, respectively, which is able to revise the prediction based on the actual measurements.

### 3.3 Analysis of Orchestrations

Our approach is based on translating process definitions into a language for which automatic computational cost analysis tools are available. We will now give details on this process, sketched in Fig. 3.8.
Table 3.1: Abstract orchestration elements.

**Declarations and definitions**

<table>
<thead>
<tr>
<th>Complex type definition</th>
<th>:-struct(QName, Members).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port type definition</td>
<td>:-port(QName, Operations).</td>
</tr>
<tr>
<td>External service</td>
<td>:-service(PortName, Operation, TrustedProperties).</td>
</tr>
<tr>
<td>Service definition</td>
<td>service(Port, Operation, InMsg, OutMsg)=Activity.</td>
</tr>
</tbody>
</table>

**Activities**

<table>
<thead>
<tr>
<th>Variable assignment</th>
<th>Var := Expr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service invocation</td>
<td>invoke(PortName, Operation, OutMsg, InMsg).</td>
</tr>
<tr>
<td>Reply and exit</td>
<td>reply(OutMsg)</td>
</tr>
<tr>
<td>Sequence</td>
<td>Activity1, Activity2</td>
</tr>
<tr>
<td>Conditional execution</td>
<td>if(Cond, ActThen, ActElse)</td>
</tr>
<tr>
<td>While loop</td>
<td>while(Cond, Activity)</td>
</tr>
<tr>
<td>Repeat-until loop</td>
<td>repeatUntil(Activity, Cond)</td>
</tr>
<tr>
<td>For-each loop</td>
<td>forEach(Var, Start, End, Activity)</td>
</tr>
<tr>
<td>Scope</td>
<td>scope(VarDecl, ActivityList)</td>
</tr>
<tr>
<td>Scope fault handler</td>
<td>handler(FaultName, Activity)</td>
</tr>
<tr>
<td>Parallel flow</td>
<td>flow(LinkDecl, Activities)</td>
</tr>
<tr>
<td>Activity in a flow</td>
<td>float(Attributes, Activity)</td>
</tr>
</tbody>
</table>

### 3.3.1 Overview of the Translation

Our input languages are a subset of BPEL 2.0 for the process definitions and of WSDL for the associated meta-information. These are translated into an intermediate language (Table 3.1) which can also be used to cover other orchestration languages. This intermediate representation is then translated into the Ciao logic programming language [HBC+08], which includes assertions to express types and input / output modes for arguments, as well as resource definitions and functions describing resource usage bounds. The resulting logic program is then analyzed by the CiaoPP tool [HPBLG05], which is able to infer upper and lower bounds for computational costs [NMLGH07], among other analyses.

A BPEL process definition is translated into a service definition which associates a port name and an operation with an activity that represents the orchestration body. BPEL processes forming a service network are translated into predicates which call each

---

1Although it currently models mainly BPEL constructs.
other to mimic service invocations.

The intermediate language can describe namespace prefixes, XML schema-derived data types for messages, service port types, and also known properties of external services of interest to the analysis (when such services are not directly analyzed). The activities supported by the intermediate language include generic constructs (assignment, sequences, loops...) and specific constructs to model orchestration workflows: `flow`, `float`, `scope/handler`, and `invoke`. `flow` corresponds to the similarly named BPEL activity, while the `float` construct annotates an activity within a `flow` with a description of outgoing links and their values, join conditions based on incoming links, and a specification of the behavior in case of a join failure.

A relevant observation regarding the translation is that it does not need to follow strictly the operational semantics of the orchestration language: it has to capture enough of it to ensure that the analyzers will infer correct information while minimizing precision loss due to the translation. Despite this, in our case the translated program is executable, and mirrors quite closely (but not exactly) the operational semantics of the BPEL process under analysis.

### 3.3.2 Restrictions on Input Orchestrations

Our analysis is restricted to orchestrations which follow a `receive-reply` pattern, where all activities start after receiving an initial message and finish by dispatching either a reply or a fault notification. Additionally, we currently do not support the analysis of stateful service callbacks using correlation sets or WS-Addressing schemes. In the future we plan to relax both restrictions by identifying orchestration fragments that correspond to the `receive-reply` pattern.

In our intermediate language, we support a variant of the scope construct, which introduces local variables and fault / compensation handlers. We do not fully support compensation handlers, which in BPEL “undo” the effects of a successfully completed scope using snapshots of variables recorded at successful completion of the scope. Except for recording snapshots, compensation handlers can be treated as pseudo-subroutines on a scope level, and inlined at their invocation place.
3.3.3 Type Translation and Data Handling

The simple types in XML schemata are abstracted as three disjoint types: numbers, strings (translated into atoms), and booleans. Complex XML types are translated into predicates specifying how the type is built. Fig. 3.9 shows the translation corresponding to a fragment of the reservation scenario in Section 3.2. The type named ‘factory->resData’ is a structure with three fields: two numbers and a list of elements of type ‘factory->partInfo’. Each of these elements is in turn a structure with two fields (atoms).

The accepted expression language is a subset of XPath which allows node navigation only along the descendant and attribute axes. This ensures that navigation is statically decidable and XML structures can be deforested to pass the addressed components as separate arguments when necessary to improve the accuracy of the analysis. For example, the expression ‘$req.body/item[1]/@qty’ in the intermediate language refers to the attribute qty of the first item element in the body part of a message stored in variable req. A set of standard XPath operators and basic functions, such as position() and last(), are supported.

3.3.4 Service and Activity Translation Rules

The service construct from Table 3.1 is used to declare an orchestration that implements operation o on port p (declared in the service’s WSDL document). Its form:

\[ \text{service}(p, o, x) = A \]
where $A$ is the service body activity (normally a complex one), is translated into a Horn clause of the form:

$$s(X, Y) \leftarrow G$$

where $s$ is some distinguished (implementation-dependent) predicate name that is unique for the given combination of port $p$ and operation $o$, $X$ is the message with which the operation is invoked (stored in variable $x$), $Y$ is the result of the operation, and $G$ is a Prolog goal obtained by translating the body $A$ of the orchestration. The result $Y$ is either $\text{response}(M)$ (where $M$ is the contents of the reply message), done (for a successful finish without response message), or $\text{fault}(F)$ (where $F$ is a fault identifier). Following the Prolog convention, we use italicized uppercase letters (e.g., $P$, $O$, $X$, $G$) as placeholders for terms, write empty list as [], and the list with head $X$ and tail $Y$ as $[X | Y]$. Additionally, lowercase italicized letters (such as $p$, $o$, $x$ and $s$ above) are used as placeholders for atoms (symbolic constants), which should not be confused with the actual constants set in lowercase typewriter face (e.g., service, response).

To obtain goal $G$ from the orchestration body $A$, we introduce the translation operator:

$$\mathcal{T} : \text{Act}^* \times (\text{Name} \rightarrow \text{Var}) \times (\text{Name} \times \text{Act})^* \times \text{Var} \rightarrow \text{Goal}$$

so that $\mathcal{T}(\bar{A}, \eta, \chi, Y)$ produces a Prolog goal that translates the given list of activities $\bar{A}$ (possibly empty), an environment $\eta$ that maps names of orchestration variables to terms (values), a list of fault handlers $\chi$, and an output status variable $Y$. The initial environment for translating the service body consists only of the mapping $x \leftarrow X$, while other variables are introduced with a scope construct. The list of fault handlers $\chi$ is initially empty, and new handlers are introduced with the handler construct.

We use the fact that any activity $A$ can be implicitly converted to a sequence of activities $\bar{A}$ in the following way: if $A$ has the form $(A_1, A_2)$, then $\bar{A}$ is obtained by concatenating $\bar{A}_1$ and $\bar{A}_2$, otherwise simply $\bar{A} = [A]$. This conversion flattens any nested sequences.

We also assume that the variables in a nested scope are previously automatically renamed when necessary to avoid shadowing the same-name variables from the enclosing scopes. The same applies to the variables that receive thrown exceptions in the first argument of the handler construct. This allows unambiguous access to the current value
of any variable defined in the current or the enclosing scopes at any point in execution, which is required for e.g., fault handling.

Simple Constructs

The translation $\mathcal{T}(\bar{A}, \eta, \chi, Y)$ for several simple cases of $\bar{A}$ are shown in Table 3.2. The first entry in the table corresponds to the case where $\bar{A}$ is empty, i.e., nothing else remains in the given sequence of activities. The translation in this case sets the output status $Y$ to $\text{done}$, to indicate a finish without a response sent.

For non-empty $\bar{A}$, we let the shape of first activity in $\bar{A}$ to drive the translation. The empty activity (second entry in Table 3.2) is simply skipped, and the translation proceeds with the rest of the sequence, $\bar{A}'$. A $\text{reply}(v)$ is translated into setting the output variable $Y$ to the response message $M$ stored in variable $v$ in the current environment $\eta$, and the rest of the sequence is ignored – in principle, static checking should be able to issue warnings in the cases where more processing is attempted after a $\text{reply}$, just like C or Java compilers are able to give warnings on “dead code” after a return statement.

Variable assignments (the last entry in Table 3.2) are performed in two steps. First, the expression is translated into a goal $G_E$, using an expression translation operator $\mathcal{E}$ (which will be explained in the following sub-section), which leaves the result of the expression in a fresh variable $Z$. Next, the rest of the activities ($\bar{A}'$) is translated into goal $G$, using an updated environment $\eta[v \leftarrow Z]$ which is exactly the same as $\eta$, except that $v$ now maps to the result $Z$ of the expression.
Translation $\mathcal{T}(\vec{A}, \eta, \chi, Y)$

| $\vec{A}$ | $[\text{handler}(v, A_h)|\vec{A}']$ | $G'$ | $(\text{Add the handler and translate } \vec{A}')$ \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>where $G' = \mathcal{T}(\vec{A}', \eta, \chi', Y)$ and $\chi' = [\langle v, A_h \rangle</td>
<td>\chi]$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| $[\text{throw}(f)|\vec{A}']$ | $Y = \text{fault}(f)$ | $(\text{Signal fault, ignore } \vec{A}')$ \\
| if $\chi = []$ |
| $H$ | $(\text{Handle the fault})$ \\
| if $\chi = [\langle v, A_h \rangle | \chi']$ \\
| where $H = \mathcal{T}((A_h, \vec{A}'), \eta[v \rightarrow f], \chi', Y)$ |

Table 3.3: Translating fault handling primitives.

**Fault Handling**

Unlike the most of general purpose languages, the intermediate orchestration language does not have named subroutines and does not support recursion. Therefore, deciding what fault handler to use at run time can be statically done at translation time. The argument $\chi$ to the translation operator $\mathcal{T}$ keeps the list of currently defined fault handlers, with the latest handler first. Table 3.3 shows that the construct $\text{handler}(v, A_h)$, where $v$ is a new variable name and $A_h$ is the handler body, adds the pair $\langle v, A_h \rangle$ to the beginning of $\chi$ and translates the rest of the activities $\vec{A}'$.

Translating a $\text{throw}(f)$ construct, where $f$ is the fault name, depends on what handlers are defined at the given point in translation. If $\chi$ is empty (i.e., no handlers apply), the translation boils down to setting the output variable $Y$ to indicate the fault $f$. Otherwise, the first handler $\langle v, A_h \rangle$ is removed from $\chi$ and the sequence starting with the handler body $A_h$ and continuing with $\vec{A}'$ is translated with an updated environment $\eta[v \rightarrow f]$ where variable $v$ contains the fault $f$. In this way, the handler can try to remedy the exceptional situation and allow the rest of the sequence $\vec{A}'$ to continue, or it can throw exception or finish the orchestration by dispatching a response with reply.

**External Invocation**

Translation of the $\text{invoke}$ construct is shown in Table 3.4. Arguments $p'$ and $o'$ are the port and the operation that corresponds to the external service that is invoked. Variable $v$ holds the outgoing message (invocation arguments), and variable $w$ receives the answer. The translation has two steps. Firstly, predicate $s'$ that corresponds to the ex-
Table 3.4: Translating invocations.

<table>
<thead>
<tr>
<th>( \bar{\mathcal{A}} )</th>
<th>Translation ( \mathcal{T}(\bar{\mathcal{A}}, \eta, \chi, Y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{invoke}(p', o', v, w)</td>
<td>s'(M, Y'), a(Y', \eta, Y) ) (Make call and process the answer)</td>
</tr>
</tbody>
</table>
| where \( s' \) corresponds to port \( p' \) and operation \( o' \) and \( v \rightarrow M \) in \( \eta \) and \( a \) is a fresh predicate name defined as:
| \( a(\text{response}(M), \eta', Y) \) \rightarrow \( \mathcal{T}(\bar{\mathcal{A}}, \eta'[w \rightarrow M], \chi, Y) \) |
| \( a(\text{fault}(f), \eta', Y) \) \rightarrow \( \mathcal{T}(\text{throw}(f)|\bar{\mathcal{A}}'), \eta', \chi, Y) \) |
| \( a(\text{done}, \eta', Y) \) \rightarrow \( \mathcal{T}(\text{throw(no_response)}|\bar{\mathcal{A}}'), \eta', \chi, Y) \) |
| and \( \eta' = \{ v \rightarrow Z \mid Z \text{ is fresh and } \exists M, v \rightarrow M \text{ in } \eta \} \) |

Table 3.5: Translating scopes.

<table>
<thead>
<tr>
<th>( \bar{\mathcal{A}} )</th>
<th>Translation ( \mathcal{T}(\bar{\mathcal{A}}, \eta, \chi, Y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{scope}(V, A_b)</td>
<td>\bar{\mathcal{A}}' )</td>
</tr>
<tr>
<td>where ( G_b = \mathcal{T}(\bar{\mathcal{A}}'', \eta_+, \chi, Y) )</td>
<td></td>
</tr>
<tr>
<td>and ( \bar{\mathcal{A}}'' ) is concatenation of ( \bar{\mathcal{A}}_b ) and ( \text{restrict}^V.x_0</td>
<td>\bar{\mathcal{A}}' )</td>
</tr>
<tr>
<td>and ( \eta_+ = \eta \cup { v \rightarrow 0 \mid v \in V } )</td>
<td></td>
</tr>
<tr>
<td>( \text{restrict}^V.x_0</td>
<td>\bar{\mathcal{A}}' )</td>
</tr>
<tr>
<td>where ( G' = \mathcal{T}(\bar{\mathcal{A}}', \eta_-, \chi_0, Y) )</td>
<td></td>
</tr>
<tr>
<td>and ( \eta_- = { (v \rightarrow M) \in \eta \mid v \not\in V } )</td>
<td></td>
</tr>
</tbody>
</table>

Arguments to \( a \) are \( Y' \), the sequence of all values for variables in \( \eta \), denoted \( \tilde{\eta} \), and the final output status \( Y \). The clauses for \( a \) treat the three possible invocation statuses. The first clause takes a response \( M \), maps the output variable \( w \) to it, and proceeds with translating activities \( \bar{\mathcal{A}}' \) after \( \text{invoke} \). The second clause re-throws an exception reported by \( s' \). The third clause throws a special exception if no reply has been received.

Note that the variable values in the current scope (argument \( \tilde{\eta} \) in call to \( a \)) are passed from the translation of \( \text{invoke} \) to the clauses of \( a \). That is why the translations in the body of \( a \) use a “fresh” version of \( \eta \), designated \( \eta' \), where each mapping \( v \rightarrow M \) from \( \eta \) is replaced with \( v \rightarrow Z \), where \( Z \) is a fresh variable that receives the corresponding argument to \( a \).
<table>
<thead>
<tr>
<th>( \bar{A} )</th>
<th>Translation ( \mathcal{F}(\bar{A}, \eta, \chi, Y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( [\text{if}(C, A_1, A_2)</td>
<td>\bar{A}'] )</td>
</tr>
<tr>
<td>( [\text{while}(C, A_b)</td>
<td>\bar{A}'] )</td>
</tr>
<tr>
<td>( [\text{while}^{a}] )</td>
<td>( a(\bar{\eta}, Y) ) (Internal tail recursion marker)</td>
</tr>
</tbody>
</table>

Table 3.6: Translating control structures.

**Scopes**

A scope is a unit of work inside an orchestration, which can introduce new variables and fault handlers. These variables and handlers are local to the scope, and do not apply to code that comes after it. Table 3.6 shows the translation for the construct scope\((V, A_b)\), where \( V \) is a list of distinct variable names introduced in the scope, and \( A_b \) is its body. As mentioned before, we assume that the new scope variables in \( V \) are name in such a way (or automatically renamed) to prevent shadowing of same-name variables from the enclosing scopes, which can always be done automatically. The translation uses a modified environment \( \eta_+ \) which introduces newly declared variables from \( V \) that are set to zero, as the default value. The translation operates on sequence \( \bar{A}' \) which is obtained by linearizing scope body \( A_b \) and inserting a special marker restrict\(^{V, \chi} \) between \( A_b \) and the remainder of sequence \( \bar{A}' \), whose task is to ensure that the translation of \( \bar{A}' \) proceeds without the newly introduced variables from \( V \), and with the initial stack of fault handlers \( \chi \). The restrict construct itself is not a legal construct in the intermediate orchestration language from Table 3.1, and is used only internally by the translation process.
Conditionals and Loops

The first entry in Table 3.6 shows the translation of the if construct. It takes a condition that evaluates to true or false, and the first step in translation is to use the expression translation operator \( E \) to generate goal \( G_E \) that leaves the result in some fresh variable \( Z \). In the second step, an auxiliary predicate with some fresh name \( a \) is invoked to process the result of the condition evaluation. The first clause of \( a \) translates the then branch \( A_1 \), and the second the else branch \( A_2 \). Note that the rest of the sequence \( \hat{A}' \) is translated twice, once in each of the two clauses of \( a \). This ensures that any fault handlers declared in either branch are properly used by the code that follows the if-then-else construct.

A while loop involves repeated evaluation of a condition before each loop iteration. It is translated into two auxiliary predicates with fresh names \( a \) and \( a' \). The first one, \( a \), translates the condition using operator \( E \), and invokes the second one, \( a' \), which acts on the result of evaluation. The first clause of \( a' \) corresponds to executing the loop body: its body is formed by translating a sequence that consists of the loop body \( A_b \) and a special marker \( \text{while}^a \) that tells the translator to call back predicate \( a \) at the end of the previous iteration. This special marker is internal to the translation system, and does not appear as one of the legal constructs in Table 3.1. The second clause of \( a' \) corresponds to an exit from the loop when the condition evaluates to false, and consists of the translation of the sequence \( \hat{A}' \) that comes after the while construct. Note that any fault handlers declared in the body \( A_b \) are local to the body, and are not automatically carried over to the next iterator or to the orchestration code that follows the loop.

The translation for the construct repeatUntil\((A_b, C)\) is obtained by replacing it with a sequence of the form scope([], \( A_b \)), while\((C, A_b)\). The leading \( A_b \) is placed inside a scope to prevent the fault handlers from the first iteration to spill over to the subsequent ones and

3.3.5 A Translation Example

A translation example is presented in Fig. 3.10. Subfigure (a) is a BPEL fragment of an orchestration, (b) is the corresponding intermediate form, and (c) is the translation into a logic program. The orchestration traverses the list of part types to reserve from
(a) A BPEL code fragment

\[
\text{while( '$i'>0', (}
  \text{ '}$i$' }\leftarrow \text{ '$i - 1' },
  \text{ '}$p$' }\leftarrow \text{ '$resp.body/factory:part[$i]$' },
  \text{ invoke( factory:sales,cancelReservation,'}$p$','}$r$')
\]

\(\text{throw( factory:unableToCompleteRequest})\)

(b) The intermediate representation.

\[
\text{a_13(A,B,C,D,E):- (}$i$,$p$,$resp.body/factory:part,$r$,$Y$)\]
\[A>0, !, \text{ a_14(A,B,C,D,E)}.\]
\[
\text{a_13(A,B,C,D,E):-}
  \text{ E=fault('factory->unableToCompleteRequest').}\]
\[a_14(A,B,C,D,E):-\]
\[F \text{ is } A-1, \text{ a_15(F,B,C,D,E)}.\]
\[
\text{a_15(A,B,C,D,E):-}
  \text{ nth(A,C,F), a_16(A,F,C,D,E).}\]
\[
\text{a_16(A,B,C,D,E):-}
  \text{ 'service_factory->sales->cancelReservation'(B,F),}
  \text{ ( F=fault(G) }\rightarrow \text{ E=fault(G) }}
  \text{ ; F=reply(H) }\rightarrow \text{ a_13(A,B,C,H,E)).}\]

(c) Translation into logic program.

Figure 3.10: Translation example.

the external part maker sales service.\(^2\) If a fault arises, a fault handler tries to cancel
already made reservations before signaling failure to the client. The figure shows just
the while loop, which finishes with a reply.

\(\text{2Unlikely in the example in Section 3.2, this code does not query different factories.}\)
Table 3.7: Resource analysis results for the group reservation service.

<table>
<thead>
<tr>
<th>Resource</th>
<th>With fault handling</th>
<th>Without fault handling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lower bound</td>
<td>upper bound</td>
</tr>
<tr>
<td>Basic activities</td>
<td>2</td>
<td>$7 \times n$</td>
</tr>
<tr>
<td>Single reservations</td>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>Cancellations</td>
<td>0</td>
<td>$n - 1$</td>
</tr>
</tbody>
</table>

The resource analysis finds out how many times external service invocations will be performed during process execution, from which deducing the number of messages exchanged is easy. The results for the complete orchestration are displayed in Table 3.7, where the estimated upper and lower bounds are expressed as a function of the input message.\(^3\) We differentiate two cases: one in which fault-free execution is assumed, and another where fault handlers can be executed, which gives more cautious estimates. These two cases were obtained by turning on or off the generation of Prolog code for fault handling—the last part of Fig. 3.10 (c).

### 3.4 An experiment in Predictive Monitoring

To validate the applicability of the proposed approach to predictive monitoring, we conducted a series of experiments which simulate the behavior of a service composition which may violate some QoS attribute (time, in this case) and we try to detect this situation ahead of time using analytically derived complexity bounds and the observed environment conditions. In our scenario, service composition \(A\) is initiated with an input message, whose size (using some appropriate metric) is represented as an integer \(n\) (ranging, in this case, between 1 and 50). Upon message reception, service \(A\) iteratively invokes a partner service \(B\) and waits for a reply. The number of iterations is bounded by upper and lower bounds \(\langle E_{AU}(n), E_{AL}(n)\rangle\) which grow linearly and which range between 100 and 500 and between 50 and 250, respectively, as \(n\) goes from 1 to 50.

Each invocation of \(B\) uses some time to transmit the invocation and its reply. Time is modeled as an environmental factor with bounds \(\langle u_{AL}, u_{AU}\rangle\) that may change over time. Executing \(B\) involves a number of operations (or steps) independent from \(n\) with bounds

---

\(^3\)The analyzer took 1.811 seconds to infer this information on a Intel Core Duo 2GHz machine with 2GB RAM and Darwin Kernel v10.2.0.
Each step takes some time between the time-varying bounds $\langle u_{BL}, u_{BT} \rangle$. The initial values for the environmental factors (measured in milliseconds) are $u_{AL} = 7.5$, $u_{AH} = 20$, $u_{BL} = 3.75$, and $u_{BH} = 10$. The experiment is run under several regimes that differ on how environmental factors evolve.

The experiment assumes that the service composition $A$ has to finish in at most $T_{\text{max}} = 25,000$ ms. Violations to this requirement are detected by monitoring with the help of the complexity functions and the environment bounds, as previously described. The monitor periodically builds a running upper / lower bound estimate of the remaining execution time based on the elapsed time and the complexity / environmental factor bounds, respectively. The monitor issues a warning (Warn) when the upper bound of the estimate for the end-time of the task exceeds $T_{\text{max}}$, to flag the risk of the time constraint violation, and an alarm (Alarm) if the lower bound estimate exceeds $T_{\text{max}}$ to indicate that a violation of the time constraint is imminent under the current conditions. Alarm prevails over Warn.

We performed one hundred simulations for every value of $n$ by randomly choosing, for each $\alpha$, a concrete complexity value between the bounds for $A$ and $B$. We measure the effectiveness of the approach by empirically assessing the frequency of violation given a warning/alarm status. Figure 3.11 shows alarm / warning / violation profiles for two environmental regimes. The one on the left simulates a system reconfiguration where environmental characteristics remain at their initial values until time $T_{\text{max}}/3$, when their bounds suddenly double (i.e., delays increase). The second regime (right) simulates a gradual degradation of the system, where environmental bounds linearly increase over the period $T_{\text{max}}$ by a factor of four.

Under the first regime, composition executions for small values of $n$ take little time
to complete, so they comply with the time limit (marked by OK) and no alerts are raised. For slightly larger input sizes (e.g. \( n = 9 \)), executions still comply with the time limit, but warnings are raised (Warn/OK), since the monitor’s estimate of the upper bound running time exceeds \( T_{\text{max}} \). As \( n \) increases, the number of false warning positives decreases in favor of the true warning positives (Warn/¬OK), because the average running time increases and thus the possibility of execution being affected by sudden deterioration of the environment factors. In the same region (around \( n = 20 \)) some warnings start to be promoted to alarms, as the lower bound time estimates increasingly start to overshoot \( T_{\text{max}} \). These cases are marked with Alarm/¬OK, and they are all true positives, since the system degrades monotonically (things never get better). Further increases in \( n \) (to around 30) lead to rapid disappearance of the false warning positives. After \( n = 38 \), all executions fall into Alarm/¬OK, because the monitor is always able to detect ahead of time that the lower execution time bound overshoots the time limit.

In the second regime in Figure 3.11, featuring a gradual degradation, the upper execution time bound overshoots \( T_{\text{max}} \) in some cases even for very small input sizes (e.g. \( n = 3 \)). A warning is then raised although no actual violations happen (executions are OK). As \( n \) increases, a pattern similar to that in the first regime is followed. For large values of \( n \) (but before the point in which it happened in the previous regime) all alarms rightly correspond to the ¬OK case.

### 3.5 An Experiment in Proactive Adaptation

To validate our approach in the context of proactive adaptation, we performed a simulation to study the effectiveness of applying data-aware computational cost functions to service matchmaking. We simulate a service network (Figure 3.12) where a client \( C \) selects among a set of providers \( P_i \) to reserve \( n = 1..50 \) sets of car parts. Each set consists of \( M = 5 \) different part types. The external client chooses one \( P_i \) which in turn selects from among a set of part suppliers \( S_i \), shared between all the providers. All \( P_i \) and \( S_i \) are known to be semantically equivalent, but vary in response time (which is our target QoS attribute). Both \( P_i \) and \( S_i \) may fail with some probability \( p_f \). When this happens, adaptation is triggered and another (next-best) service from the pool is sought for.

The selection policies we have simulated are: random selection from the pool of can-
Figure 3.12: Two-tier simulation setting.

didates, fixed preferences, and data-dependent QoS prediction based on computational cost. Although not exhaustive, these selection policies are helpful for comparing the data-aware to other approaches.

In the data-aware case, we select the best candidate taking into account its upper bound complexity (i.e., worst case behavior), in terms of messages exchanged. Every second-tier service $S_i, 1 \leq i \leq 12 = N$, has a different upper bound cost function $ub_i(n)$ (Figure 3.13), where $n$ is the requested number of sets of a given part type. The bold line highlights the lowest upper bound among all the services for each $n$. Assuming that $i^*$ is the index of the second-tier service that is selected for given selection policy and $n$, the upper bound computational cost $UB_j$ for (first-tier) provider $P_j$ is computed with the expression:

$$UB_j(n) = EP_j(n) + M \times (1 + ub_{i^*}(n)).$$

which takes into account both the structural computational cost $EP_j$ (using the same family of functions as in Figure 3.13), and the cost of $M$ invocations to $S_i$ (adding one for each outgoing request).

The selection assigns a fixed time to every message exchange to convert it into execution time.\(^4\) In a real scenario, per-exchange time can be updated as execution proceeds to reflect network state, system load, etc. as in [ICH10a].

The fixed preferences policy ranks services using the expected response time for some representative input; we chose $n = 12$. Therefore all queries whose data size is 12 are

\(^4\)We are not taking into account the time associated to executing internal activities. The technique we used to infer the number of messages can infer the number of activities of every type associated to some invocation, which can be accounted for similarly to messages.
handled equally by both the fixed preferences and the data-dependent complexity cost policies (see later and Figs. 3.14 and 3.15).

For each selection policy and for each \( n \) in the range 1..50, one hundred simulations were run and averaged. Each run performs matchmaking and simulates the execution of the selected service. Besides failures, the simulated number of outgoing messages in the run is (uniformly) randomly chosen between 60% and 100% of the upper bound, to model that the number of messages may in fact be less than this upper bound. The time associated with every message exchange is padded with additional noise having a normal distribution to simulate the variations in the behavior of the network. We are, therefore, not assuming a constant time per event in the simulation.

Several sets of simulations with different time noise distribution parameters were performed, of which we have chosen two representative ones. In Fig. 3.14 all services have the same per-message average time (5 ms). In Fig. 3.15, services in both layers are assigned a different time per message whose average is in the range 4-8 ms. The figures show plots for the three selection policies and for three failure probabilities \( p_f \in \{0.001, 0.01, 0.1\} \) (left to right).

The data-dependent selection policy gives the best results in our experiments. No-
tably, it features a homogeneous and predictable behavior w.r.t. failure rates and timing noise. In an extended set of simulations (not appearing in this paper due to space constraints), the same behavior appears for even very high, almost unrealistic failure rates, which not only supports our claim that a more informed decision leads to better results, but also points to the resiliency of such a policy for extreme scenarios. In contrast, a different time per message exchange in Figs. 3.14 and 3.15 made the fixed preferences policy select the service with quadratic behavior in the former and a service with linear behavior in the latter.

3.6 Conclusions

Precision of QoS analysis for service orchestrations can be greatly improved by taking into account input data and operations on it, as well as by working with data-sensitive QoS bounds, rather than with single-point aggregated averages. Computation cost analysis can be used as an effective basis for a more precise, data-sensitive analysis when combined with the empirically collected data on environmental factors on the infrastructure level.

In this chapter we proposed an approach to derive such a data-aware computational cost functions, and presented some experimental results that illustrate its applicability in the context of predictive monitoring and proactive adaptation. The core of the approach is a translation-based scheme which, from an orchestration (written in a subset of BPEL/WSDL), generates a (logic) program that can be analyzed by existing tools to automatically derive the complexity bounds as functions of input data.

The experiments in predictive monitoring suggest that the approach is able to provide a good degree of prediction accuracy, where the predicted certain compliance with the QoS limit, as well as the predicted imminent limit violations (alarms) are sound (i.e., safe), while the percentage of false warning (indicating a possible, yet not imminent limit violations) tends to increase with the degree of volatility of the environment factors on the infrastructure level.

The experiments in proactive adaptation indicate that the service matching based on data-sensitive QoS bounds tends to consistently outperform other matching strategies, e.g., based on random choice from a set of candidate or based on choosing the first
Figure 3.14: Simulation results for $p_f = 0.001, 0.01, 0.1$ (left to right) using the same noise distribution.
Figure 3.15: Simulation results for $pf = 0.001, 0.01, 0.1$ (left to right) using different noise distributions for each service.
available candidate from a fix list of preferences.
As discussed in Chapter 3, the task of analyzing and predicting the Quality of Service (QoS) metrics for service compositions, both at design time and at the level of an executing instance, is of a great theoretical and practical importance. In this chapter, we take a somewhat different approach, based on generating a constraint model for QoS metrics of an executing composition based on its structure, the semantics of its building blocks, and its current state of execution at a given moment. While the present approach can be fruitfully combined with the results of the computation cost analysis (when available, which is discussed in Section 4.2.5), it is inspired by a different set of challenges that arise in mission-critical, highly dynamic and adaptable service-based applications.

Firstly, in such applications, service compositions can be subjected to run-time adaptations that may dynamically change not only their bindings with the component services, but also their structure and data. When that happens, all guarantees on the composition behavior need to be re-checked and cost functions recomputed, which may incur unacceptable computation overheads and even lead to outdated predictions. Secondly, the behavior of the component services used in the composition may be known only empirically, without any guarantees attached. Thirdly, for compositions that implement
mission-critical business processes, we wish to achieve continuous, close monitoring that is able to signal a possible or imminent violation of the Service Level Agreement (SLA) as soon as possible in the lifetime of the executing composition instance, and to relate, if possible, internal composition events (such as branching and loop iterations) with the cases of SLA compliance and failure.

To answer these challenges, we propose a fast and efficient structural analysis, supported by some basic empirical data, which directly builds a constraint model for the composition at a given point in execution, which is then used for prediction. The prediction, based on constraint solving, is concerned not with inferring the expected (or the most probable) values of the QoS metrics, but concentrates on the identification of the possible and imminent cases of SLA conformance and violation at each point.

In this chapter, we consider service orchestrations, which are compositions with a centralized control flow. In the following chapter, an extension to service choreographies is proposed and discussed. The orchestrations may involve a wide range of workflow patterns [vdAtHKB03] — including parallel flows, different splits/joins, loops, branches, etc. — and are usually expressed using some dedicated notation, such as BPMN [Obj09], BPEL [Jea07a], Yawl [vdAtH05] or DecSerFlow [vdAP06b], or other adequate formalism. In this chapter, we use abstract (but executable) notation for orchestrations from which we formulate a constraint satisfaction problem (CSP) [Dec03, Apt03] that models the situation of SLA conformance or violation.

The rest of the chapter is organized as follows: Section 4.1 presents a motivating example. Section 4.2 then describes how the CSP can be automatically formulated on the basis of an orchestration continuation, to take into account the known assumptions about third-party components, as well as to include internal structural parameters of branches and loops. In Section 4.3 we give an example of application, and in Section 4.4 some notes on the implementation. Sections 4.5 and 4.6 present an evaluation framework and the experimental results on the quality of prediction in a potential adaptation context. Section 4.7 presents some conclusions.
Figure 4.1: An example orchestration to reconfigure content provided to a user.
4.1 Motivation

Consider a scenario where a provider of multimedia content (text, audio and video) needs to periodically update and reconfigure program streams offered to individual clients (users), based on their historical usage patterns. That may require choosing between different mixtures of available streams (such as news, sport, entertainment, etc.) presented to a user, genres within them, and type of multimedia materials. The choice may depend on the frequency of use (casual vs. frequent users), user interests, and bandwidth adequate to serve different types of content (e.g. low quality vs. HD video). In such a scenario, the provider would run the reconfiguration process from time to time when serving user requests, although typically not for each access. Reconfiguration depends on other (usually back-end) administrative and analytic services, and should not cause noticeable glitches in content delivery. The SLA for the content delivery service does provide some window for running the reconfiguration process on top of it, but it is normally very restricted. Therefore, the running time of the reconfiguration process and its availability are of the utmost importance.

Fig. 4.1 depicts an example orchestration implementing the reconfiguration process, using BPMN notation [Obj09]. It starts with the reception of user ID (activity $a_0$), which spawns in parallel ($a_1$) the retrieval of the users’ account record ($a_2$) and the user’s usage patterns ($a_3$). If the usage pattern is stable ($a_4$), the user’s current content profile is reused ($a_5$). Otherwise, a new content profile is generated ($a_6$) based on the account record and the current usage patterns. For efficiency, first minor variations in content profile parameters are attempted; if these are not likely to fit the usage pattern ($a_7$), more radical changes are attempted, and so on. Finally, the content profile (either the current one or a new one) is written to the configuration database ($a_8$).

In this example, the configuration process may affect responsiveness of the main multimedia content delivery service, and therefore we want to continuously monitor and predict reconfiguration running time, having in mind the overall SLA. At any point in the execution of the reconfiguration orchestration, including its start, and within that particular context, there are a number of interesting objectives to aim at:

**Predicting Certain SLA Violations:** If we are able to predict that the orchestration cannot possibly meet the SLA constraints, then we can either abort it (effectively post-
poning the reconfiguration), or adapt it by switching to a simpler and/or more robust version. Conversely, if we are reasonably sure that the execution will be SLA-conformant, we can plan to use the potential slack in a productive way.

**Predicting Possible SLA Violations:** If we can predict that SLA violations may occur, but not necessarily so, and we can identify potential points of failure, then we can prepare, ahead of time, adequate adaptation and healing mechanisms, and/or try to decrease the risk of violation by using fail-safe component services.

**Inferring the Necessary Preconditions:** If we not only predict, but understand why an SLA violation may or must happen, we can use that information to identify bottlenecks, to develop criteria for selection of components, and to drive either runtime or design-time adaptation.

In this chapter we present a unified constraint-based approach and analysis framework that makes it possible to perform runtime prediction of SLA violation / conformance for service orchestrations, based on monitoring information and a constraint model of an abstract semantics of the orchestration structure. Predictions are based on and expressed in a form that describes the circumstances under which SLA violations and conformant executions of an orchestration may take place, which can be used to reason about the orchestration and its components.

### 4.2 Constraint-Based QoS Prediction

#### 4.2.1 The General Prediction Framework

An SLA typically defines, among other things, which QoS attributes are relevant in the context of the provider-client contract, and what values of these QoS attributes are acceptable. For QoS attributes expressed as numbers on a measurement scale, QoS constraints given by an SLA are often expressed as ranges of permissible values for each attribute. More complex relationships between SLA attributes — such as trade-offs between cost and speed — can be devised, but in our analysis we will assume that the QoS constraints are given as lower and upper bounds on appropriate QoS metrics.

Furthermore, we will focus on an important subset of QoS metrics that are monotonic and cumulative in the sense that they express an amount of a physical or logical
resource consumed by each activity in an orchestration, so that the amounts from sub-
sequent activities add together into an aggregate metrics. Running time is an obvious 
example of a cumulative metrics, because consumed time is never recovered. In this 
chapter we will assume, for simplicity, that metrics are accumulated by through addi-
tion (which is a fairly common case). Note that some metrics whose natural aggregation 
function is not addition can be easily mapped into additive metrics. For instance, the 
aggregation function for the availability (the probability of successful access) $p$ of $n$ sub-
sequent operations can be calculated as $\prod_{i=1}^{n} p_i$, where $0 < p_i \leq 1$ is the availability of 
the $i$-th component. Using the transformation $\lambda = -\log p$, we can transform the original 
multiplicative metric of $p$ into the additive $\lambda = \sum_i \lambda_i$.

An important feature of a cumulative QoS metrics is that, at any point in execution of 
an orchestration, its value can be calculated as a cumulative function (such as addition) 
of two components: the previously accumulated metrics and an estimate of the pending 
metrics for the remainder of the execution of the orchestration, until it finishes. For 
some metrics, their accumulated value needs to be be measured taking into account 
the history of the actual execution up to the current execution point (e.g. elapsed time 
from the start of execution), while for other metrics the current value at any execution 
point does not depend on the previous history. For example, in the case of availability 
the current metrics always represents “availability so far”. Since it is being measured 
at some execution point which has obviously been reached, the probability $p$ of being 
available up to the point of measure is 1 (and then $\lambda = 0$).

From the viewpoint of adaptation, another important aspect of a prediction scheme 
is the time interval between the detection of the possibility of an SLA violation and its 
actual occurrence. This “window” makes it possible to warn about potential or immi-
nent SLA violations ahead of time. Additionally, if a prediction technique can identify 
conditions that increase or decrease likelihood of an SLA violation, it can filter false 
positives from true positives and thus increase the reliability of prediction. These condi-
tions can be related to internal parameters of the orchestrations, such as the truth value 
of branching conditions or the number iterations in a loop. For our constraint-based 
approach, this will be illustrated in Section 4.2.5.
4.2.2 QoS Prediction Architecture

The architecture of the constraint-based QoS prediction framework is shown in Fig. 4.2. A process engine executes service orchestrations and interacts with external services by exchanging messages. In the process, it publishes lifecycle events such as signaling the start or end of a process, invocation of a component service, and reception of a reply. Also, from time to time, the process engine publishes the current point of execution of a running orchestration in the form of a continuation (explained in the following subsection). That is typically not done at each step, but at specific milestones such as service invocations, loop iterations and branches. Deciding how to determine the optimal granularity for publishing points is a matter for future work.

The events published by the process engine are sent via an event bus. The constraint-based QoS predictor can be connected to that bus and listen to lifecycle events (or a subset of events of interest). When a continuation is published, it is pushed by the event bus to the predictor. The predictor performs the analysis, and publishes QoS predictions back to the event bus, together with QoS metric bounds inferred by the analysis. That information can be accessed by an adaptation mechanism, which can use the published predictions and the QoS metrics to prepare adequate adaptation actions on the orchestration definition, an executing instance, or both. Such adaptation actions may include,
among other things, selection of components to minimize the risk of failure, changes in
the structure of the process, or intervention on the orchestration data.

### 4.2.3 Representing Orchestrations and their Continuations

In order to estimate how much the remainder of the execution can contribute to a given
QoS metrics, we need to have some knowledge about where in the execution we are
placed — or, more precisely, what remains to be executed: it is the orchestration activities
yet to be executed which need to be taken into account to predict the remainder of the
metric value. In our case we represent this still-not-executed part of the orchestration
explicitly, in the form of a *continuation*. A continuation [Rey93] is an abstract object
(such as a set of data structures or a function) that represents the control state of a
computation — i.e., the precise execution point of a program (including the associated
data) and whatever remains to be executed.

In our case we are interested in continuations of running instances of orchestrations.
A continuation is always implicit in the state of a process engine, even when the chosen
programming language does not make it accessible as such: it is determined, for exam-
ple, by the activity being executed, the representation of the orchestration and the data
in the orchestration. In our approach, we rely on keeping available at all moments an ex-

dplicit representation of the continuation, inspect its structure (which in general becomes
progressively simpler as execution proceeds) and use it to generate constraints which
model the conditions under which the execution can meet / not meet the QoS stated in
the SLA.

The (simplified) abstract syntax we will use is shown in Fig. 4.3. It is based on the
concrete syntax used by a prototype orchestration engine which we developed as experi-
mentation base for this chapter and that uses Prolog as the language to express branch
and loop conditions and elementary operations. A *simple activity* represents a basic unit
of work, such as a calculation or assignment. Similarly, *cond* encodes a logical condition
that is used for if-then-else branching or while loop iteration. List comprehension is
simplified using *foreach*. Communication with the environment is done using *invoke*
and *reply*. Besides sequences, both parallel OR and AND splits/joins are supported.
Most BPMN constructs can be translated straightforwardly. A translation of the exam-
ple process from Fig. 4.1 (with some low-level details omitted) is shown on the left of
continuation ::= a.
    a,a1,a2 ::= \{elementaryoperation\} (elementary)
    | a1,a2 (sequence)
    | \{(cond) → a1;a2\} (if-then-else)
    | a1∧a2 (and-join)
    | a1∨a2 (or-join)
    | while[(cond), a] (while loop)
    | foreach[(x, list, a)] (list comprehension)
    | invoke(partner, out, in) (invoke a service)
    | reply(out) (send a reply)
    | relax (do nothing)
    | stop (finish)

Figure 4.3: Abstract syntax for orchestrations.

Fig. 4.4.

The continuation at every point of the execution of Fig. 4.4 is not explicit in the orchestration representation, but is rather kept by the interpreter which executes it (which we do not have space to describe in detail in this chapter). This continuation represents what is left to execute after every computation step, and is updated every time a step is taken. For instance, after taking the else branch in the orchestration from Fig. 4.4 (left), the continuation is a sequence of activities in lines 6-9, 11 and 12.

4.2.4 Deriving QoS Constraints from Continuations

A constraint is a relation that restricts values of variables that, in our case, represent values of QoS metrics associated with the constructs in the orchestration and their basic

1 ( invoke(account_svc, UserID, AccRec) 500 ≤ T1 ≤ 800 (assumption: account_svc)
2 ∧ invoke(usage_svc, UserID, UsagePatt) 200 ≤ T2 ≤ 500 (assumption: usage_svc)
3 ),
4 ( stable(UsagePatt) Cond ∈ [0, 1], 0 ≤ T4 ≤ 10 (condition)
5 → invoke(reuse_svc, AccRec, Profile) 100 ≤ T5 ≤ 400 (assumption: reuse_svc)
6 ; invoke(gen_svc, (AccRec, UsagePatt), Profile), 200 ≤ T6 ≤ 600 (assumption: gen_svc)
7 while( unfit(Profile) ),
8    invoke(gen_svc, (AccRec, UsagePatt), Profile ) T9 = k×(T7 + T8) + T7 (while duration)
9 )
10 ), ( Cond = 1 ∧ T10 = T4 + T5) ∨ (Cond = 0 ∧ T10 = T4 + T6 + T9) (if
11 invoke(conf_svc, (UserID, Profile), _), 100 ≤ T11 ≤ 300 (assumption: conf_svc)
12 stop. T = T3 + T10 + T11 (total running time)

Figure 4.4: Orchestration for Fig. 4.1 (left) and its associated running time constraints (right).
components. The particular relations which are generated depend both on the QoS metric that is to be captured and on the structure of the continuation. In our approach, after deriving the constraints from the structure of the given continuation, constraint solving techniques (see Section 4.2.6) are used to infer admissible ranges for variables that lead to either SLA satisfaction or violation.

We require that these constraints lead to a conservative prediction of QoS fulfillment: under the assumption that our knowledge about the QoS characteristics of the basic orchestration components (i.e., atomic activities or external services) is correct,¹ we want that any prediction we make about the conformance of an execution w.r.t. the stated SLA is also correct. In this direction, we make no assumptions on the (in)dependence of behavior of individual components. I.e., if the behavior of two external services seems to be strongly linked (because of e.g. past history), we do not take this apparent correlation into account for the sake of prediction safety. Such information, if available, could be added to try to make predictions more precise: for example, given that some service took less time than expected to answer, we might assume that the same is going to happen to some other service which is apparently historically related. While this seems to help in making predictions more accurate, it also makes them potentially unsafe.

We illustrate constraint derivation with two metrics: running time and availability. For a continuation consisting of a (complex) activity $a$ representing the remainder of the execution, the total running time of the orchestration is a sum of the elapsed time since the start $T_a$ and the pending time $T(a)$. The total availability is equal to the pending availability $\lambda(a)$, as explained before. We derive $T(a)$ and $\lambda(a)$ structurally, and then constrain them against the SLA limit: $T_{\text{max}}$ for the maximal allowed execution time by and $\lambda_{\text{max}}$ for the negative logarithm of the minimal allowed availability (see Section 4.2.1). The resulting constraints:

**For SLA conformance:** $T_a + T(a) \leq T_{\text{max}}$ and $\lambda(a) \leq \lambda_{\text{max}}$.

**For SLA violation:** $T_a + T(a) > T_{\text{max}}$ and $\lambda(a) > \lambda_{\text{max}}$.

are solved to obtain the (approximate, but safe) ranges for $T(a)$ and $\lambda(a)$, and thus for the total QoS, for the two cases of conformance and violation, respectively.

¹Note that in reality this knowledge is always inexact and subject to dynamic changes. However, we are putting ourselves in the situation that this knowledge is exact, and we want to ensure that, at least in this optimistic situation, the constraints we generate meet safeness requirements.
We generate the above constraints by formulating a constraint for each simple activity contained inside a (usually relating the value of the QoS metric for the activity with its expected bounds) and combining these constraints (using disjunctions and conjunctions according to the structure of a) into a larger constraint which provides bounds for $T(a)$. The right hand side of Fig. 4.4 shows the set of constraints corresponding to the process on the left. We will now detail how constraints for simple and complex activities are generated.

**Simple activities.** For a simple activity $a$ — a simple operation, relax or stop — and simple operations (in curly braces), the assumption is that they include only elementary constructs and do not entail complex computations. A lower bound for this is always $T(a) \geq 0$, and an upper bound depends on the execution environment (computer clock, CPU, etc.). It is usually on the order of microseconds, and should be experimentally determined for each architecture. In the example we have put some reasonable limits, which do not necessarily reflect a real situation. As for the availability, since no external components are involved, in this case we have $\lambda(a) = 0$.

**Sequences.** Since we are considering cumulative metrics, the metric values are accumulated for the case of sequences: for sequence $a \equiv a_1, a_2$ we have $T(a) = T(a_1) + T(a_2)$ and $\lambda(a) = \lambda(a_1) + \lambda(a_2)$.

**Service invocations.** For an activity $a$ that is an invoke to an external service, for both the running time $T(a)$ and the availability $\lambda(a)$ the analyzer needs to rely on empirically or analytically derived estimates, which include the local message handling and network delivery. In our approach, we deal with the ranges of possible values, rather than with probable or expected values. That means that in absence of any information, we simply have $T(a) \geq 0$ and $\lambda(a) \geq 0$, but the upper bounds on $T(a)$ and $\lambda(a)$, if known, must be safe, or else the prediction will be too optimistic and fail to detect some cases of possible SLA violations.

**Parallel flows.** In the case of a parallel flow $a \equiv a_1 \land a_2$, $T(a)$ must lay somewhere between $\max(T(a_1), T(a_2))$, when $a_1$ and $a_2$ run fully in parallel, and $T(a_1) + T(a_2)$, which

---

2 Or those that can be converted into a cumulative (e.g. additive) equivalents.
is the worst, sequential case of execution. Therefore, it is safe to take
\[
\max(T(a_1), T(a_2)) \leq T(a) \leq T(a_1) + T(a_2)
\]
as a conservative approximation.

This approximation can however be too cautious and may lead to overly pessimistic estimates. If we have additional information about the semantics of the orchestration language and the implementation of the execution engine, we can refine the estimate for \(T(a)\). For instance, if the execution of local activities is single threaded, while external services invocations are ensured to run in parallel, we can use the following scheme. Consider the case where \(a_1\) and \(a_2\) are sequences ending with an invoke activity, i.e., \(a_1 \equiv a_{11}, a_{12}, \ldots, a_{1k}, a^*_1\) and we call \(a'_1 \equiv a_{11}, a_{12}, \ldots, a_{1k}\) (respectively for \(a_2\)). We will assume that \(a'_1\) and \(a'_2\) are sequences of activities to be executed locally by a single thread, even if they appear in different branches of the flow, while \(a^*_1\) and \(a^*_2\) can be executed remotely in parallel. In this case, the total estimated time for the flow is
\[
\max(T(a'_1) + T(a^*_1), T(a'_2) + T(a^*_2)) \leq T(a) \leq T(a'_1) + T(a'_2) + \max(T(a^*_1), T(a^*_2))
\]

If, say, \(a^*_1\) is not an external invoke, but \(a^*_2\) is, then \(T(a^*_1)\) is part of \(T(a'_1)\). If neither \(a^*_1\) nor \(a^*_2\) are external invokes, then simply \(T(a^*_1) = T(a^*_2) = 0\). This structural analysis can of course be easily extended to more than two parallel flows. The running time of an OR-parallel flow can be conservatively approximated using the case of AND-parallelism.

From the point of view of availability, parallel flows do not affect the total risk of failure, since the total availability depends on availability of all used components, regardless of their order of execution. Therefore, for \(a \equiv a_1 \land a_2\) or \(a \equiv a_1 \lor a_2\), we have \(\lambda(a) = \lambda(a_1) + \lambda(a_2)\).

**Conditionals.** For a conditional \(a \equiv (\langle cond \rangle \rightarrow a_1 ; a_2)\), where \(a_1\) is the then part and \(a_2\) is the else part, the metric depends on how the condition is evaluated. We introduce a Boolean variable \(b_{\text{cond}}\) to represent the result of the condition evaluation, so that we can state the following disjunctive constraint: either (1) \(b_{\text{cond}} = 1\) and \(T(a) = T(\langle cond \rangle) + T(a_1), \lambda(a) = \lambda(a_1)\), or (2) \(b_{\text{cond}} = 0\) and \(T(a) = T(\langle cond \rangle) + T(a_2), \lambda(a) = \lambda(a_2)\). The value of \(b_{\text{cond}}\) is generally unknown, but can be constrained to either 0 or 1 as the result of
constraint solving. This makes it explicit that either the then or the else part can be taken, but not both.

**Loops.** In case of a loop $a$ — while or foreach with body $a_1$ — we introduce an integer variable $k_a \geq 0$ that stands for the number of loop iterations. Then, we have $T(a) = k_a \times (T(\text{cond}) + T(a_1)) + T(\text{cond})$ and $\lambda(a) = k_a \times \lambda(a_1)$. The actual value of $k_a$ is generally unknown, but its inclusion into the constraints allows us to reason about the maximal or minimal number of loop iterations that lead to SLA compliance or violation.

### 4.2.5 Using Computational Cost Functions

To improve the precision of the predictions, the constraint-based predictor is able to use computational cost functions for service orchestrations (Chapter 3, [ICH10b]), which, in this case, express lower and upper bounds of the number of loop iterations as a function of the input data to the orchestration. These computation cost functions may be automatically inferred at the start of an orchestration, statically determined at design time, or manually asserted for known cases. The inference of the computation cost functions depends on the semantics of the workflow constructs and the (sub-)language of conditions and elementary operations in which the orchestration is expressed.

If computation cost functions are available, the default constraint for the number of iterations of loop $a$ ($0 \leq k_a$) can be strengthened to $\ell \leq k_a \leq u \land 0 \leq k_a$, where $\ell$ and $u$ are, respectively, lower and upper bounds on the number of iterations, which depend on the actual values of the input data. In the absence of one (or both) of the bounds, the corresponding constraint is simply not generated (as in Fig. 4.4, right).

### 4.2.6 Solving the Constraints

The constraints derived from the orchestration continuation relate the QoS metrics for the entire continuation with those of individual activities, component services, Boolean results of evaluating the conditions, the number of loop iterations, and the limits from the SLA. As such, they represent a constraint satisfaction problem [Dec03] that can be solved for values of the constrained variables, which, in our case, include QoS metrics, Boolean conditions and loop iteration counters. Depending on the type of problem and the particular constraint solver used, solving the CSP may involve several iterations of
constraint propagation and problem splitting [Dec03, Apt03], which are used to reduce
the equations in the original CSP to a series of simpler ones, before attempting to assign
to the constrained variables values that satisfy the constraints.

In our case, we use the interval constraints (ic) solver from the ECLIPS Constraint
Logic Programming (CLP) system [AW07, Cis06]. The underlying Prolog subsystem of
ECLIPS is used for constructing the constraints from a continuation, handling informa-
tion on QoS metrics of component services, and reporting the results. The solver han-
dles constrained variables over bound and unbound integer (discrete) and real number
dense) domains. The values of the constrained variables are represented as (possibly
unbound) real or integer intervals. Integer variables with bounded domains are han-
dled in a manner similar to finite domain solvers [Dec03]. The solver directly supports
disjunctive constraints (which we use for conditionals) and reified (Boolean valued) con-
straints.

The solver produces results given as bounds on values of the constrained variables,
obtained from propagation of arithmetic constraints, or fails if the constraints cannot be
satisfied. In our case, as mentioned before, we always solve two CSPs, one modeling SLA
conformance and another one modeling SLA violation.

The constraint solver is complete, i.e., it does not discard feasible solutions. There-
fore, upon constraint satisfaction, the answer intervals for the variables include all
admissible values, and values outside these intervals cannot possibly satisfy the con-
straints. On the other hand, it may be that some combinations of values inside the
answer intervals do not satisfy the constraints. Let us see an example: the constraint
\[ 0 \leq T(a_1) + T(a_2) \leq 100 \]
has as answer \[ T(a_1) \in [0..100] \land T(a_2) \in [0..100] \]. This contains all
feasible solutions (for example, \( T(a_1) = 0 \land T(a_2) = 100 \)) but also combinations of values
which do not satisfy the constraints (for example, \( T(a_1) = 50 \land T(a_2) = 51 \)). Of course, if
the latter values are fed into the constraint solver together with the initial constraint,
the constraint solver will determine that the system is unsolvable.

4.3 An Example of Application

Table 4.1 shows the results of running our QoS prediction framework applied to the
orchestration in Fig. 4.4 (corresponding to the workflow in Fig. 4.1) and using execution

68
### Case 1: Unconstrained iterations

#### Successive running time SLA ranges

<table>
<thead>
<tr>
<th>Variable</th>
<th>Metrics</th>
<th>success</th>
<th>violation</th>
<th>success</th>
<th>violation</th>
<th>success</th>
<th>violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>ms</td>
<td>—</td>
<td>600 .. +∞</td>
<td>600 .. 750</td>
<td>750 .. +∞</td>
<td>750 .. 1500</td>
<td>1500 .. +∞</td>
</tr>
<tr>
<td>cond(if)</td>
<td>bool</td>
<td>0 .. 1</td>
<td>1</td>
<td>0 .. 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>iter(while)</td>
<td>nat</td>
<td>—</td>
<td>0 .. +∞</td>
<td>—</td>
<td>0 .. +∞</td>
<td>0 .. +∞</td>
<td>0 .. +∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Metrics</th>
<th>ms</th>
<th>500</th>
<th>450</th>
<th>500</th>
<th>1200</th>
<th>700</th>
<th>2700</th>
</tr>
</thead>
<tbody>
<tr>
<td>% E.C.D.T</td>
<td>—</td>
<td>0%</td>
<td>66%</td>
<td>60%</td>
<td>33%</td>
<td>80%</td>
<td>23%</td>
<td>90%</td>
</tr>
<tr>
<td>Lead</td>
<td>ms</td>
<td>500</td>
<td>250</td>
<td>300</td>
<td>1000</td>
<td>300</td>
<td>2300</td>
<td>300</td>
</tr>
</tbody>
</table>

### Case 2: Between 1 and 10 iterations

#### Successive running time SLA ranges

<table>
<thead>
<tr>
<th>Variable</th>
<th>Metrics</th>
<th>success</th>
<th>violation</th>
<th>success</th>
<th>violation</th>
<th>success</th>
<th>violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>ms</td>
<td>600 .. 7820</td>
<td>600 .. 750</td>
<td>750 .. 7820</td>
<td>750 .. 1500</td>
<td>1500 .. 7820</td>
<td>1500 .. 3000</td>
</tr>
<tr>
<td>cond(if)</td>
<td>bool</td>
<td>0 .. 1</td>
<td>1</td>
<td>0 .. 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>iter(while)</td>
<td>nat</td>
<td>1 .. 10</td>
<td>1 .. 10</td>
<td>1 .. 10</td>
<td>1 .. 10</td>
<td>1 .. 10</td>
<td>1 .. 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Metrics</th>
<th>ms</th>
<th>500</th>
<th>250</th>
<th>500</th>
<th>1000</th>
<th>900</th>
<th>2500</th>
</tr>
</thead>
<tbody>
<tr>
<td>% E.C.D.T</td>
<td>—</td>
<td>0%</td>
<td>66%</td>
<td>33%</td>
<td>33%</td>
<td>66%</td>
<td>30%</td>
<td>83%</td>
</tr>
<tr>
<td>Lead</td>
<td>ms</td>
<td>500</td>
<td>250</td>
<td>500</td>
<td>1000</td>
<td>500</td>
<td>2100</td>
<td>500</td>
</tr>
</tbody>
</table>

### Component running time assumptions

<table>
<thead>
<tr>
<th>local op.</th>
<th>account_svc</th>
<th>usage_svc</th>
<th>reuse_svc</th>
<th>gen_svc</th>
<th>conf_svc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running time (ms)</td>
<td>0 ms .. 10 ms</td>
<td>500 ms .. 800 ms</td>
<td>200 ms .. 500 ms</td>
<td>100 ms .. 400 ms</td>
<td>200 ms .. 600 ms</td>
</tr>
</tbody>
</table>

Table 4.1: Sample QoS prediction results.
time as QoS metric. The assumptions on ranges for the invocations of external services are shown at the bottom. These ranges would be updated by the QoS predictor based on the observation of invoke/reply events published by the process engine. Note that we are only concerned with the range of possible running times for each component, not the probability distributions within these ranges, and therefore we only need only to adjust the boundaries of the corresponding ranges.

The top part of Table 4.1 shows the results for the case of an unbound number of while loop iterations, which is the default if no additional information is provided. A series of successive assumed running time limits (500, 750, 1500 and 3000 ms) was considered, and both the SLA compliance (success) and violation results are shown. The meaning of the rest of the rows are as follows:

**duration** shows the predicted running time ranges for the orchestration in ms.

**cond(if)** is a Boolean value showing the possible evaluations of the condition in the conditional (1 for the “then” branch and 0 for the “else” branch).

**iter(while)** shows the range of possible iteration counts in the while loop (corresponding to the repetition after testing the condition in the “else” branch).

**E.C.D.T. earliest certain detection times**: the earliest time at which a certain violation or success can be detected.

**% E.C.D.T.** percentage of the total (maximum) execution time which elapsed up to the E.C.D.T.

**lead** time between E.C.D.T. and the closest moment in which the orchestration can finish (i.e., the shortest time span to react in the worst case).

The results show that the lowest limit of 500 ms could not be met under the initial assumptions regarding execution times for atomic activities and external services. The 750 ms limit can be met, if the conditional evaluates to 1, meaning that the while loop is avoided. The 1500 ms limit can be met in both cases of the conditional, but can be violated only for the case of taking the “else” branch. Finally, for the range of running times between 1500 ms and 3000 ms, the prediction shows that, under the given assumptions, the only possible situation for both compliance and violation is taking the “else” branch,
with the number of iterations in the range 0..11 and 3..+∞, respectively. Note that for the latter limit, between 0 and 2 iterations guarantees compliance, and more than 11 iterations guarantees violation of the limit. An adaptation mechanism can, use these predictions to prepare and trigger adaptation actions that may prevent, minimize, or compensate for possible SLA violations ahead of time.

The earliest time at which a success or violation can be predicted depend on the particular execution. Let us look at an example: in Table 4.1, Case 1, columns “750 ms .. 1500 ms”, successes have been detected at 500 ms and SLA violations at 1200 ms. The reason that successes have been detected before violations is that these correspond to different executions: in the case of violation, the “else” branch (with the loop) has been taken, it is detected that there will be a violation after some iterations. On the other hand, if the “then” branch is taken, certainty of success is immediately detected, as there are no loops to be taken. With this interpretation in mind, the constraint-based predictor is able to detect SLA violation with certainty up to between 300 and 500 ms in advance, while SLA conformance can be detected as early as after 500 or 700 ms of running time. In relative terms, SLA conformance has been detected in the experiments when only between a 23% and a 66% of the maximum execution time has elapsed, and SLA violations have been detected in some cases when only a 60% of the execution has elapsed.

The middle part of Table 4.1 shows a hypothetical case where, based on input data and computational cost functions, the predictor is able to infer that the actual number of loop iterations, in case the “else” branch is taken, must fall between 1 and 10. The results follow the same pattern as in the first case, but this time the predictor is able to infer that the duration of the orchestration under the assumptions must fall between 600 and 7 820 ms. This inferred running time range for the orchestration can be used by other parts of the runtime system (including predictors themselves) to update their QoS metrics assumptions on the deployed components. Note that the guarantee of at least one loop iteration increases the lead for the earliest certain detection of violations to 500 ms.

The average net time for performing one running time limit compliance/ violation prediction depicted in Table 4.1 (not counting the time for sending and receiving data over the network), based on the average from 10 000 executions, was 0.574 ms on a small
end-user non-dedicated machine.\(^3\)

### 4.4 Implementation Notes

We have tested the approach using a prototype implementation of the architecture from Fig. 4.2, which includes the process engine, the QoS predictors, and the event bus, organized as a distributed and scalable system of components that communicate using reliable messaging. The tests included deployments on Linux and Mac OS X 32 and 64 bit platforms.

In our running prototype, the QoS predictors are implemented in \textit{ECL\(^i\)PS\(^e\)} Constraint Logic Programming system, while the process engine (that executes orchestrations) is implemented in Ciao Prolog \cite{HBC12}. Both Prolog dialects support a variety of constraint logic programming techniques, but have, at the moment, slightly different orientation and strong points. \textit{ECL\(^i\)PS\(^e\)} provides very robust, industrial-scale constraint solvers which can easily handle very complex problems involving thousands of constraints and variables, while Ciao is a flexible multi-paradigm programming environment with sophisticated support for concurrency. Fortunately the fact that they are both Prolog-based systems greatly facilitates interfacing and putting together the required architecture.

In our prototype, the language in Fig. 4.3 is used to define service orchestrations and to maintain instance control state throughout execution, so that there is no additional overhead in communicating continuations to QoS predictors, other than message transfer times. Also, any adaptation that changes the orchestration structure for a running instance can be simply implemented by replacing one continuation with another.

The messaging subsystem is implemented using ZeroMQ \cite{iMa11}, which provides fast and reliable multi-part binary message exchange primitives on top of TCP networking and IPC subsystems, including request-reply, push-pull and publish-subscribe patterns. We have developed Prolog (Ciao and \textit{ECL\(^i\)PS\(^e\)}) bindings to ZeroMQ with data (term) serialization that provide transparent higher-level data exchange primitives.

\(^3\)The tests were run on a 32-bit 2GHz Intel Core Duo notebook with 2GB of RAM, running Mac OS X 10.6.7 and \textit{ECL\(^i\)PS\(^e\)} version 6.0.167.
4.5 An Empirical Evaluation Framework

To evaluate the constraint based prediction approach more thoroughly, in the remainder of this chapter we shall look at an industrial service workflow, proposed by Leitner et al. [LHD11] in the context of fault prediction based on data-mining and cost-optimization of service composition adaptations.\footnote{More details are available at \url{http://www.infosys.tuwien.ac.at/prototypes/VRESCo/experimentation}} This service workflow uses 18 component services to handle order processing, manufacturing, billing, quality control, and shipment tasks in an on-demand production line scenario. The control structure of the orchestration contains some of the typical control constructs that can be found in real-world orchestrations: branching, looping, parallel flow control constructs, and sequential chains of activities. We look at the execution time as the QoS attribute of interest.

Quality of prediction is naturally very much dependent on the structure and logic of an orchestration, and the sample industrial service workflow we use here may be used as an illustrative case, without precluding different prediction quality for other, significantly more complex or simpler cases. Given the composition, we next study two factors that can affect the quality of prediction: correctness of the assumptions about the component QoS ranges, and the choice of the constraint solver.

To measure the quality of prediction under different experimental settings, we establish a set of indicators (also known as prediction quality metrics, discussed in more details in Section 4.5.4), such as precision, accuracy, and recall, and, besides, we look at how long before the predicted event the prediction was made, which is clearly very important for adaptation.

4.5.1 Inaccurate and Imprecise Component Assumptions

In section 4.3, we have assumed that the knowledge on the execution times of the component services is accurate, but obviously that does not need always be the case. To assess the impact of inaccurate and imprecise assumptions on the QoS of the components on the quality of the constraint-based prediction, it is useful to establish a baseline case against which the comparison is made. In the baseline case, we use the most accurate and precise component QoS assumptions, and measure the quality of prediction obtained under
such ideal conditions. The accuracy and precision of prediction is of course different from the accuracy and precision of the assumptions on which the prediction is based.

A second issue is how to define the situations where the assumptions on the QoS of some component differ from the real bounds. The bounds we work with can be tighter than the actual ones, and thus they may not contain some of the values that the actual QoS can take, but all points in the range are valid. A representation of the situation is labeled as inaccurate in Figure 4.5. It is also possible that all valid values lie inside the range, but some points in the range are not valid. This is labeled imprecise in Figure 4.5. Estimated ranges can also mix approximations in different directions in both extremes.

### 4.5.2 Choice of Constraint Solver

In our experiment, we used the Eclipse CLP system [AW07] with its interval constraint (ic) solver, which is licensed as open source and freely available. The solver supports linear and non-linear arithmetic constraints (equalities and inequalities) over real and integer variables that range over intervals of finite or infinite size (with the finite domain as the special case). These capabilities fit well with the shape of the CSP generated from the continuation [ICH11b], which may contain non-linear (min/max, multiplicative) and disjunctive constraints, and where ranges of the component QoS values are represented as intervals. Other, publicly and commercially available constraint solvers can be used as well, as long as they have can express the same kind of constraints, or in special cases where the constraints can be simplified, e.g., made linear.

Some constraint solvers are better suited for some constraint domains and classes
of constraints than the others. The results obtained from constraint solvers are sets of values for the constrained variables that are always complete (no solution is left out), but maybe not correct (they may contain values that are not part of any solution [Dec03]). More precise constraint solvers would be able to narrow down the value sets closer to the actual answers, and consequently can detect inconsistent constraints stores, which translates in more precise failure or success predictions.

For our evaluation, the prediction of SLA violation or success depends on the ability of the constraint solver to deduce unsatisfiability of the constraint model for either the case of SLA compliance, or violation, and therefore choose the other of these two possibilities as its prediction as early in the execution as possible. The results in this chapter illustrate the quality of prediction that can be achieved using a state-of-the-art, open source constraint solver, but we do not rule out a better quality of prediction with more advanced solvers or in special cases (e.g., when the constraints are linear).

### 4.5.3 Experimental Setup

A perfectly precise and accurate bounds of the component QoS (i.e., for the base case) can generally be known only *a posteriori* or in a *post mortem* analysis. For that reason, and also to ensure that all other factors remain equal when introducing inaccuracies and imprecisions in the assumptions, we based our experiments on execution logs from 100 runs of the sample industrial orchestration, which had approximately 160 events recorded per instance.

The prediction was simulated by feeding the time-stamped events from the log for each instance to the constraint-based predictor in the same manner as it would be done at run-time. At each point of prediction, the continuation was constructed from the known original structure of the orchestration and the events observed in the log. For the baseline experiment under precise and accurate assumptions, we have used the minima and maxima of the component running times recorded in the logs.

Experiments were conducted against a number of execution time limits (SLA objectives), which were chosen based on the recorded running times in the logs to ensure that a certain percentage of the 100 instances violated the limit. These percentages, the failure rates, ranged from small and moderate ones (1%, 5%, 10%) to relatively high ones (20%, 33%) and were, for completeness and observation of trends, extended to unrealis-
tically high ones (failure rates $\geq 50\%$).

The average time spent to make about 160 predictions per instance was between 283 ms to 491 ms, which corresponds to between 1% and 2% of the average instance running time.\footnote{The experiments were performed on a low-end Intel x64 machine with solid-state disk and 4GB RAM, running Mac OS X 10.7.3.} This suggests that the prototype implementation of the predictor does not impose a significant prediction overhead. The first OK (SLA met) / FAIL (SLA violated) prediction for every instance was taken as the definitive one.

### 4.5.4 Prediction Quality Metrics

Different measurements can be used to assess the quality of prediction of SLA violations. In our approach we use the evaluation metrics proposed by Salfner et al. [SLM10] in a recent study of online failure prediction methods. All of these metrics are based on counting true positives ($TP$, failure predicted and occurred), false positives ($FP$, failure predicted, but did not occur), true negatives ($TN$, non-failure predicted and no failure occurred), and false negatives ($FN$, non-failure predicted, but failure occurred). To these, we add the cases labeled $UP$ (number of successes when the method could not predict either OK or FAIL) and $UN$ (number of failures in a similar case) All of these metrics are real numbers between 0 (the worst case) and 1 (the best case). We proceed by explaining each one of them in turn.

- Precision $p = \frac{TP}{TP + FP}$: when we have a decision on a failure, how precise can we expect it to be?

- Negative predictive value $v = \frac{TN}{TN + FN}$: how many success predictions turned out correct.

- Recall $r = \frac{TP}{TP + FN + UP}$: how many failures were correctly predicted.

- Specificity $s = \frac{TN}{TN + FP + UN}$: how many successes were correctly predicted (a counterpart to recall).

- Accuracy $a = \frac{(TP + TN)}{(TP + FP + UP + TN + FN + UN)}$: how many instances were correctly predicted.
Since in most failure prediction techniques there is a trade-off between precision and recall, as pointed out by Salfner et al. [SLM10], a harmonic mean between these two metrics, known as $F = \frac{2pr}{p + r}$, can be introduced. In our case, it also tends to have a value close to 1. (We defer discussion of some additional adaptation-specific metrics to Section 4.6.4.)

4.6 Experimental Results

4.6.1 Baseline Case

The quality prediction metrics for the baseline case (i.e., using the most precise accurate assumptions about the bounds of the component QoS value ranges) are shown in Table 4.2. Columns $u$ and $m$ are treated in Section 4.6.4.

Precision $p$ tends to increase with the fault rate, because at small fault rates the SLA failures are so rare that even a single false positive diagnosis constitutes a significant proportion of the predicted failures. Recall $r$ also tends to be a value close to 1, generally higher than $p$, except for the fault rate 10%, where unpredicted positives (UP) constitute a significant fraction of true positives (TP). The $F$-measure ranges between $p$ and $r$ and it amplifies the effects of the worse among them. Specificity $s$ and negative predictive value $v$ maintain values close or equal to 1 across all failure rates. The accuracy $a$ is consistently high across failure rates, and for the low to medium failure rates (between 0% to 10%) it ranges between 97% and 99%.

4.6.2 Simulating Inaccurate and Imprecise Assumptions

To compare with the baseline case, we run a number of experiments with inaccurate and imprecise assumptions (Figure 4.5). For clarity of data presentation, the results presented in the text that follows do not mix over-approximation of one bound with under-approximation of the other bound.

Figure 4.6 shows the comparison of precision, accuracy, specificity, and recall for three groups of inaccurate and imprecise assumptions. The first group of tightly clustered thick lines in the top part of each graph, marked with empty and full circles, pluses, and asterisks, correspond to predictions that were based on inaccurate assumptions. The baseline component ranges $[a, b]$ are here replaced with tighter ranges $[a', b']$,.
Table 4.2: Comparative prediction quality indicators for the most precise and accurate component QoS assumptions.

<table>
<thead>
<tr>
<th>Failure Rate (%)</th>
<th>m</th>
<th>n</th>
<th>p</th>
<th>Aggregate Metrics</th>
<th>Basic Metrics</th>
<th>Failure Rate (%)</th>
<th>TP</th>
<th>FP</th>
<th>TN</th>
<th>FN</th>
<th>UP</th>
<th>UN</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>75</td>
<td>1</td>
<td>2</td>
<td>23</td>
<td>97</td>
<td>99</td>
<td>99</td>
<td>75</td>
<td>2</td>
<td>2</td>
<td>23</td>
<td>97</td>
<td>99</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>96</td>
<td>99</td>
<td>99</td>
<td>67</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>96</td>
<td>99</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>64</td>
<td>99</td>
<td>99</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>64</td>
<td>99</td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>49</td>
<td>99</td>
<td>99</td>
<td>34</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>49</td>
<td>99</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>24</td>
<td>99</td>
<td>99</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>24</td>
<td>99</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>17</td>
<td>99</td>
<td>99</td>
<td>20</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>17</td>
<td>99</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>99</td>
<td>99</td>
<td>99</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>
Figure 4.6: Comparative measures of prediction quality for inaccurate and imprecise assumptions.

\( a < a' < b' < b \), that cover 98%, 90%, 80%, 50% and 30% of the original range \([a, b]\).

The great degree of overlap between the lines in this group can be explained, in our experiment, by looking into the distributions of the execution times for individual service components. It turns out that the spread of these time ranges is measured in tens or hundreds of milliseconds, i.e., it is about two orders of magnitude smaller when compared to the total orchestration execution time. Therefore, under-approximating such already narrow ranges does not have a big impact on the generated constraints. In other situations, where components have larger QoS value spreads, deviations from the baseline due to under-approximation should be expected to be greater than in our experiment.

The second group of lines are dashed and correspond to the cases of accurate, but imprecise assumptions where the baseline component ranges \([a, b]\) are replaced with
wider ranges \([a/k, b \times k]\), where the lower and the upper bound are under- and over-approximated, respectively, by factor \(k\) that is 1.5 (□), 2 (◇) and 10 (■). While the first two over-approximation factors exhibit gradual, but not radical, deterioration of prediction quality, the latter is an order-of-magnitude over-approximation which leads to significantly worse prediction quality, which is hardly unexpected.

The third group are the thin solid lines on the graphs that also correspond to the cases of accurate, but imprecise assumptions, but differ from the second group of lines in that the baseline component ranges \([a, b]\) are here replaced with even wider ranges \([0, b \times k]\), where the lower bound is lost, and the upper bound is over-approximated by a factor \(k\) that is 1 (△), 1.5 (▽) and 2 (▲). (Due to overlapping, these appear as 6-spike stars on some graphs.)

It turns out that loosening the lower bound, however small, hurts prediction quality even more than an order-of-magnitude over-approximation of the upper bound. The reason for this is that an assumed lower bound of 0 prevents the constraint-based predictor from discarding the case when there is definitely no failure, and therefore the predictor is unable to predict failure in a large number of cases. The importance of the lower bounds in constraint-based prediction is therefore at odds with the usual perception that upper bounds are key predictors of failures.

4.6.3 Prediction Timing

Figure 4.7 shows the distribution of the time at which a (true or false) negative prediction was made, measuring from the orchestration start across different SLA failure rates for the baseline case. The red line above represents the time limit for each fault rate. For small and medium fault rates, the picture shows that on average the predictor was able to issue a negative prediction relatively early, approximately within one fifth of the instance execution time. Only for very large failure rates (50% and over) the prediction was made much later, closer to the execution time limit. The reason for that is that, other things being equal, a higher execution time limit makes it easier for the constraint solver to discard the possibility of an SLA violation.

The left graph of Figure 4.8 compares the average true negative prediction times between the three group of inaccurate assumption cases, and the two groups of accurate, but imprecise ones (using the same markings as before). Again, the inaccurate cases
Figure 4.7: Distribution of negative prediction times (ms).

Figure 4.8: Comparative true negative/positive prediction times for inaccurate and imprecise QoS assumptions.

behave closely to the baseline case, while a later prediction due to the loss of precision is visible in the cases of imprecision (with and without the lower bound). The worst case here is the tenfold over-approximation, which produces the most delayed true negative predictions.

Figure 4.9 shows the distribution of lead times between the (true or false) positive
predictions and the SLA running time limit, for the baseline case. The results suggest a relatively stable average lead time of about 9 sec. across the failure rates, which corresponds to about one fourth of the average instance execution time. Of course, whether such lead time is sufficient for adaptation purposes depends on the type of adaptation to be employed. Figure 4.8, right, shows the comparison between the average true positive prediction time leads under inaccurate and imprecise assumptions, which qualitatively follows the precision and accuracy pattern. The earliest true positive predictions are obtained under the accurate assumptions (with varying precision), while the imprecise assumptions that ignore the lower bounds are practically unable to predict true positive before reaching the time limit corresponding to each of the fault rates.

4.6.4 Suitability for Predictive Adaptation

The usability of a SLA failure prediction method in the context of predictive adaptation (i.e., adaptation triggered ahead of the actual failure with the goal to avoid it or to mitigate its effects) depends not only on the type of adaptation, but also on some general characteristics of the prediction methods such as examined by Sammodi et al. [SMF+11], who have identified several helpful metrics for that purpose.

The measure $u = 2ps/(p + s)$ harmonizes prediction precision and specificity, and quantifies the extent to which the prediction method manages to avoid unnecessary adaptations. The values for $u$ in Table 4.2 tend to be high and to move in harmony with precision $p$, because specificity is (in the baseline case) always equal or very close to 1. The degradation of the $u$ metrics under various types of inaccurate or imprecise
assumptions, shown on the left side of Figure 4.10, follows a similar pattern to that of precision in Figure 4.6.

The measure $m = 2rv/(r + v)$, on the other hand, harmonizes recall and negative predictive value, and quantifies the extent to which the prediction method manages to avoid missed adaptations. Again, since $v$ is always very close to 1, the values for $m$ in Table 4.2 follow those of recall $r$. Figure 4.10, right, shows that $m$ degrades under inaccurate and imprecise assumptions following a pattern similar to that for recall in Figure 4.6.

4.7 Conclusions

We have devised and implemented a method which makes it possible to predict possible situations of SLA conformance and violation in a service orchestration, and to obtain information on the internal parameters of the orchestration (branch conditions, loop iterations) that may occur in these situations. The method is based on modeling QoS metrics of a service orchestration using constraints, based on assumptions on the behavior of the orchestration components. That analysis can, in principle, be applied at each step in an orchestration based on the current continuation. This allows periodic or continuous updating of the predicted bounds for QoS metrics for the orchestration and therefore a continuous assessing of conformance to SLA, which can be useful for proactive adaptation and self-healing. This approach can be combined with automatically inferred computational cost functions for service orchestrations, which can express the
bounds of internal parameters (such as loop iterations) as functions of input data given to the orchestration instance, to provide a higher level of prediction precision. We have implemented the method in a prototype and reported some efficiency results.

Experiments using the prototype predictor based on a realistic experimental service orchestration case suggest that constraint-based SLA failure prediction, under the precise and accurate component QoS assumptions, is a very reliable prediction method that offers a good combination of precision, specificity, recall, and other quality of prediction metrics. For reasonable failure rates, the method is able to make an early prediction of the non-failure of an SLA according to execution time, as well as to predict failures significantly ahead of actual failures.

When the accuracy and precision of the component QoS assumptions deteriorates, the method’s quality of prediction generally tends to deteriorate gracefully, unless gross (order-of-magnitude) imprecisions are introduced. However, the method is very sensitive to the loss of information on the lower bounds for component QoS ranges, which tends to decrease the quality of prediction more than a significant over-approximation of the upper bounds. This requires special care when collecting the component QoS information using external sources and/or collecting events from logs and monitoring.

The prediction quality attributes, such as precision, recall, negative predictive value, and specificity, suggest that this prediction method can be expected to provide a good basis for predictive run-time adaptation with the aim of avoiding detected failures or compensating their effects.
Chapter 5

A Constraint-Based Approach to Quality Assurance in Service Choreographies

Service Level Agreements (SLAs) are commonly specified under the assumption that each interaction between the client and the service is viewed as a single session, and, accordingly, such end-to-end SLAs correspond to a request-reply message exchange pattern between the two parties. However, many business processes involve more complex message exchange patterns between two or more stateful participants, where several interactions may belong to the same session and build upon each other, and where the data that is exchanged may significantly affect the behavior of the participants in terms of the Quality of Service (QoS), including the number of messages exchanged.

For such complex, multi-participant choreographies, a coherent support for QoS assurance which includes negotiation, prediction, and QoS-driven adaptation [MBC+10] is relevant both theoretically and practically. While several types of run-time adaptation aimed at avoiding or mitigating SLA violations have been proposed [DNGM+08, HKMP08, SPJ11], these are often only applicable to the request-response message exchange pattern and/or to acyclic control structures. Several prediction and run-time adaptation approaches based on machine learning [LMRD10], online testing [SMF+11], and model checking [SM11], well suited for orchestrations with centralized control flow,
have been proposed.

Building on the results from the Chapter 4, we here propose a constraint-based approach for supporting QoS assurance for service choreographies that involve multiple, stateful participants and complex message exchanges. The proposed approach can be applied both at design time and at run time to support QoS negotiation, prediction, and QoS-driven adaptation. This work extends [ICH11b] for the case of service compositions with interconnected constraint models of stateful, interacting choreography participants, combining the derivation of QoS constraints with static analysis techniques.

We first present a motivational example (Section 5.1), then describe our approach (Section 5.2), review several examples of its application (Section 5.3), and finish with some conclusions (Section 5.4).

5.1 Motivation

Figure 5.1 shows a simplified example of a choreography for purchasing goods or services in a large organization where the procurement function is centralized. It uses the BPMN notation [Obj09] with swim lanes delimiting participants, and dashed lines showing the flow of messages between them.

Participant A is the procurement process, which starts by receiving a procurement request \(a_0\), and continues by sending the list of specifications to the agent \(a_3\) and retrieving budget line information for this purchase \(a_2\), in parallel. Participant B is the agent which receives the list of specifications \(a_{13}\) and performs a loop \(a_{14}\) for each item from the specification list. For each item, B looks into the supplier catalogs \(a_{15}\) to find out alternative purchasing options; since that can depend on the choice of earlier items, specifications are processed sequentially. If only one alternative is found, it is automatically chosen \(a_{19}\), but if two or more alternatives exist, B asks A to chose among them \(a_{17}\) and waits for the answer \(a_{18}\). The choice is added to the purchase order \(a_{20}\). After processing all specifications, agent B returns the final purchase order document to A \(a_{21}\). Whenever A is asked to choose between alternatives \(a_5\), it acts based on the budget line restrictions: if forced, it uses the cheapest option; otherwise it tries to chose the best solution. After answering all choice queries, A receives the purchase order from B \(a_{10}\) and starts the purchase \(a_{11}\), which provides the return
Figure 5.1: An example choreography for purchase ordering.
notification to the requester, to whom the purchased goods and services will be delivered directly.

Figure 5.2 abstracts away the logic of the participants from Figure 5.1, and concentrates only on the exchange of messages. The left-hand side of the figure shows a sequence diagram for message exchanges in a session involving the initiating user and the participants A and B. The right-hand side of the figure shows A and B as components with connector links (req, agent, and check for A, client and approval for B), with messages sent and received over these links. The number of message of each kind within a single session is shown in parentheses. Wiring between the connectors is shown with thick dotted lines. For each wire, both the kind and the cardinality of messages in both directions must match.

The end-to-end QoS characteristics of A (such as, e.g., its execution time) depend on several factors. Firstly, if the number of specification items \( n \) is \( > 0 \), there can be between 0 and \( n \) callbacks from B to A in the foreach loop. Secondly, the behavior of A for each callback from B depends on whether it is forced to choose the cheapest alternative, which is known at the exit of \( a_2 \). Some of these factors are controlled by the user \( (n) \), some by third parties \( (a_2 \) which sets the forced flag for \( a_6 \), \( a_{15} \) which generates alternatives), and some on the implementation of A (the logic and complexity of determining the best
choice in \( a_7 \). With respect to the quality assurance issues illustrated by this example, we are interested in tackling the following problems:

- **Automatically deriving a QoS model of the choreography for a given input request or a class of input requests.** Such a model can be used as an input for determining SLA offerings from the service provider to the users.
- **Using the QoS model of the choreography to predict SLA violations at run-time,** at different points in execution. E.g., greater accuracy of prediction can be obtained when the forced flag becomes known after \( a_2 \).
- **SLA compliance checking of choreography participants at design-time for a given class of input requests.** This is the basis for adaptive dynamic selection (binding) of service components.

## 5.2 Constraint-Based QoS Modeling for Choreographies

The proposed constraint-based approach to modeling QoS for service choreographies is implemented in two main phases. The first one focuses on the creation of QoS models for the choreography participants as Constraint Satisfaction Problems [Dec03, Apt03] (CSP). We will show how to generate a model of the QoS metrics under consideration (Section 5.2.1), capture the view of each participant regarding the effective QoS at every moment in the execution (Section 5.2.2), and how to automatically derive these models (Section 5.2.3). The model is enriched with information about the shape and size of messages, inferred using static analysis techniques, in order to increment its accuracy and the precision of the prediction (Section 5.2.4).

The second phase of the approach consists of connecting the models for the different participants and solving them as a whole (Section 5.2.5). Note that when deriving QoS models for choreographies, joining the different sub-models is done following the structure of the composition. In the present case, the overall structure may not lend itself to structural analysis and participants take a prominent role. Therefore determining the overall QoS characteristics is done by joining per-partner models (Section 5.2.5) mimicking the topology of the choreography.
5.2.1 Modeling Cumulative QoS Metrics

Execution time, availability, reputation, bandwidth consumed, and cost are some of the most common QoS attributes. In this work we focus on attributes that can be numerically quantified using some measurement scale, or QoS metric: e.g., execution time can be measured using time units. QoS metrics do not need to have a fixed origin (a “true zero” value), but one unit of distance needs to express the same variation in the attribute everywhere on the scale. This requirement excludes, for instance, ordinal voting-based reputation ranking between services, where the unit difference in ranking does not carry information about the difference in votes received.

We additionally require QoS metrics to be cumulative and non-negative: QoS values of activities in a sequence add up to give the QoS value for the sequence, and this value should never decrease by adding more activities. Some QoS metrics, such as availability (expressed in terms of probabilities), that do not use addition to calculate aggregation in a sequence, can be converted into additive metrics using a suitable transformation. For instance, the availability \( p \) of \( n \) sequential activities with availabilities \( p_i \) is \( p = \prod_{i=1}^{n} p_i \) and can be converted into \( \lambda = \sum_{i=1}^{n} \lambda_i \) with the transformation \( \lambda = -\log p \).

Cumulative QoS metrics allow us to represent the QoS of a service composition at any point in execution as a sum of two components: the previously accumulated QoS up to that point, and the pending QoS for the remainder of the execution. Non-negativity guarantees that the pending QoS can only decrease as the execution proceeds. While the accumulated QoS can be estimated empirically (by measuring elapsed time, network traffic, or accumulated monetary cost), the pending QoS for the remainder of the execution is in our approach modeled as a CSP over variables that represent QoS values for composition activities and control constructs. Solving this CSP gives a prediction of the pending QoS.

5.2.2 QoS Models of Participants and Continuations

Service choreographies provide a “global view” of a multi-participant, stateful message exchange within some logical unit of work. There are several possibilities to provide both abstract and executable descriptions of choreographies. On the more abstract side BPMN (as in Figure 5.1) or WS-CDL [Wor05], which is a high-level spe-
cialized choreography language, can be used. On the more executable side, we can use choreography extensions of standard process (orchestration) languages, such as BPEL4Chor [DKLW07]. In our approach, we assume that the implementation details of the participants are essentially private and that the participants can be viewed as communicating components that conform to the protocol (as in Figure 5.2). Conformance, compatibility, and realizability of choreographies has been studied using formal methods such as Petri Nets [vdADO+06], session types [DCD10], and state machines [BBO12].

As mentioned before, we proceed by developing a separate QoS model for each participant in the choreography. Each participant is seen as a component with a number of connector links (or channels, in WS-CDL terminology). Each link $c$ is bi-directional, and each direction (in/out) is characterized by a triplet of the form $\langle N_{in/out}(c), \bar{q}_{in/out}(c), \Delta \bar{q}_{in/out}(c) \rangle$, where $N$ is multiplicity of in/out messages, $\bar{q}$ are QoS values corresponding to the first in/out message, and $\Delta \bar{q}$ are increments of QoS values for the successive messages (for $N>1$). For example, for the case of execution time, $T_{in}(c)$ is the time when the first message was received over link $c$ and $\Delta T_{in}(c)$ is the time interval between the successive messages. $N$, $\bar{q}$ and $\Delta \bar{q}$, as well as other variables in the constraint QoS models developed in this section, are not numeric constants, but represent intervals of possible numeric values for all legal execution cases, whose upper and lower bounds are inferred from the constraint model.

We build the QoS model of a participant by looking at its current point in execution. To stay close to the executable specifications, we follow the same approach as in our previous work on run-time prediction for orchestrations (Chapter 4). We use the notion of a continuation which describes the current state of the participant and the remainder of the computation until its end [Rey93]. At the beginning, the continuation is the entire process and it is gradually reduced by eliminating the completed activities as the execution proceeds. The continuation information is always implicitly present in the state of the engine which executes the participant, and, in principle, can be obtained either by inspecting its internal state or by observing the process events from the outside. The latter is less robust since missed events or run-time composition modifications can invalidate the information inferred through external observation.

We represent continuations using an abstract language for the participant processes (Figure 5.3). It is based on a prototypical process language implementation that provides
the continuation information explicitly at each execution step (cf. Fig 4.3). The participant state is kept in variables whose types are described in Section 5.2.4. Variable values are assigned using the let construct or received over some link with recv. The standard sequential operator, if-then-else, and AND-parallel splits/joins are supported. For simplicity, we present only two foreach looping constructs: one over elements of a list and another one over messages received over some channel. The send and recv messaging constructs can be combined into an invoke; note that request-reply patterns are not enforced (this is left to the protocol). Participants use the stream construct to send a series of messages within the same session which can be received with a recv-based foreach.

5.2.3 Automatic Derivation of the QoS Constraint Model for a Participant

The constraint QoS model for a participant is inferred automatically from the continuation and the previously accumulated QoS, using the structural approach of [ICH11b], where QoS values for complex constructs are derived from their components. A separate constraint QoS model is derived for each QoS metric of interest. Due to space constraints, we will present here only on the derivation of execution time. More details and treatment of other metrics, such as availability, can be found in [ICH11b].

Figure 5.4 shows the automatically derived QoS constraint model for the execution time for participant A at its start, i.e., when the continuation consists of the entire par-
Figure 5.4: Structurally derived QoS constraint model for participant A. The code for the participant A is shown on the left-hand side in the abstract syntax, and the generated constraints appear on the corresponding lines to the right. For an activity on line $i$, we denote its starting time with $T_i^+$ and its end time with $T_i^-$, $T_i^- \geq T_i^+$. $T_A$ represents the execution time at the current execution point (here at the start), and is an input to the model. The code communicates over channels req, agent, and check (Figure 5.2), plus an additional channel budget which is used to invoke the budget line information service $a_2$ from Figure 5.1.

The execution of participant A is a sequence of commands, and the metric for the execution time is cumulative. Therefore for a sequence $S = [S_1, S_2, ..., S_n]$ we have $T^+ = T_1^+$, $T^- = T_n^-$, and for adjacent activities $S_i$ and $S_{i+1}$ we have $T_i^- = T_{i+1}^-$. For clarity of presentation, here we ignore the internal time used by the process engine between steps, which needs to be taken into account in real applications (see Section 4.2.4).

The reception of a single message with $\text{recv}(c, v)$ (lines 1 and 12) finishes at time $T_i^- = \max(T_{i-1}^+, T_{\text{in}(c)})$, where $T_{i-1}^+$ is the finish time of the previous activity, and $T_{\text{in}(c)}$ is the time at which the message arrives on the channel $c$. Since in our case messages are received over the same channel at a single place in code, the $\text{recv}$ construct also sets $N_{\text{in}(c)} = 1$. The command $\text{send}(c, E)$ (lines 3, 10, 13) delivers a message to the mailbox on the other side of the channel, for which it takes some time marked with $t_{\text{send}}$, which is also a constrained variable and considered an input to the model. $T_{\text{out}(c)}$ is equated with

1 Remember (Section 5.2.2) that these variables actually contain admissible ranges.
the finish time $T_i^-$ of the send construct. Outside a loop (lines 3 and 12), $N_{\text{out}}(c)$ is set to 1, and $\Delta T_{\text{out}}(c)$ is not constrained, because it is not applicable. The invoke construct in line 2 is treated as a send-recv sequence.

The timing for the AND-parallel flow (ending in line 4) depends on the particular process engine implementation, and can vary between real parallelism and sequential execution of the two activities. Without a more detailed knowledge of the implementation details, the duration of the parallel flow $T_4^- - T_4^+$ may vary between the maximum and the sum of durations of the two “parallel” activities.

The recv-based loop (line 5) starts when both the preceding activity has finished ($T_4^-)$ and the first message on the check channel has become available ($T_{\text{in}}(\text{check})$). The number of iterations of the loop ($k_5$) equals the number of messages arriving through the channel, $N_{\text{in}}(\text{check})$. Since every loop iteration can start only upon message reception, the effective length of a loop iteration $L_5$ is the maximum between the actual duration of the loop iteration ($T_5^- - T_5^+$) and the interval between incoming messages $\Delta T_{\text{in}}(\text{check})$. Sending a message in each iteration of the loop (line 10) equates the multiplicity of outgoing messages $N_{\text{out}}(\text{check})$ to the number of loop iterations $k_5$, and the interval between messages $\Delta T_{\text{out}}(\text{check})$ to the effective iteration length $L_5$. The if-then-else construct (line 6) introduces a binary constraint variable $c_6$ which captures the truth value of the condition, and a disjunctive constraint (line 9) which covers the then and the else cases. Finally, the internal operations, such as the expression evaluation (line 8) and a call to an internal procedure best (line 7), simply add the corresponding time intervals (resp. $\Delta t_{\text{expr}}$ and $\Delta t_{\text{best}}$).

### 5.2.4 Analysis of Message Types With Size Constraints

The constraint QoS models whose derivation we described above include a number of internal structural parameters, such as the number of loop iterations and condition truth values ($k_5$ and $c_6$ in Figure 5.4) that depend on data that is received by these services. There are several ways in which the information about shape of the data can be organized and used to further constrain the values of these structural parameters and, therefore, make the constraint models more precise. One possibility would be to apply computational cost analysis techniques from Chapter 3 to an appropriate abstraction of the participant processes in order to obtain an analytic functional relationship between

94
\[ \tau := \text{any} \mid \text{none} \quad (\text{some unspecified value and no value}) \]
\[ \mid \text{bool}(a..b) \quad (\text{Boolean between } a \text{ and } b, a, b \in \{0,1\}, a \leq b) \]
\[ \mid \text{number}(a..b) \quad (\text{number between } a \in \mathbb{R} \cup (-\infty) \text{ and } b \in \mathbb{R} \cup (+\infty), a \leq b) \]
\[ \mid \text{string}(a..b) \quad (\text{string with finite size between } a \in \mathbb{N} \text{ and } b \in \mathbb{N} \cup (+\infty), a \leq b) \]
\[ \mid \text{list}(a..b, \tau) \quad (\text{list with finite size between } a \in \mathbb{N} \text{ and } b \in \mathbb{N} \cup (+\infty), a \leq b) \]
\[ \mid \{x_1: \tau, x_2: \tau, \ldots, x_n: \tau\} \quad (\text{record with named fields } x_1, \ldots, x_n, n \geq 0) \]

Abbrev.: bool \equiv \text{bool}(0..1), \text{number} \equiv \text{number}(-\infty..+\infty), \text{string} \equiv \text{string}(0..+\infty), \text{list}(\tau) \equiv \text{list}(0..+\infty, \tau)

Figure 5.5: A simple typing system for messages with size constraints

the size of input data (number magnitudes, list lengths, etc.) and the upper and lower bounds of possible values for the structural parameters. Another possibility, which we discuss in this subsection, is to use a simple form of type analysis which is directly applicable to the abstract representations of continuations used in our approach.

Figure 5.5 shows a simple type system with size constraints which includes Booleans, numbers, strings, lists, and records with named fields. Each type \( \tau \) in this system has a denotation \( [[\tau]] \) which is the set of all values that belong to it. For instance, \( [[\text{number}(0..1)]] = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\} \). By definition, we take \( [[\text{none}]] = \emptyset \). We write \( \tau_1 \sqsubseteq \tau_2 \) as a synonym for set inclusion \( [[\tau_1]] \subseteq [[\tau_2]] \). The set of all types with size constraints together with the relation \( \sqsubseteq \) forms a complete lattice [DP02] with any as the top element, and none as the bottom element, i.e., none \( \sqsubseteq \tau \sqsubseteq \text{any} \) for arbitrary \( \tau \). We introduce the least upper bound operation \( \sqcup \) on types, where \( \tau_1 \sqcup \tau_2 = \tau \) means that \( \tau \) is the smallest type (w.r.t. \( \sqsubseteq \)) such that \( \tau_1 \sqsubseteq \tau \) and \( \tau_2 \sqsubseteq \tau \). For example, \( \text{number}(0..10) \sqcup \text{number}(8..100) = \text{number}(0..+\infty) \), \( \text{list}(1..5, \text{number}) \sqcup \text{list}(9..9, \text{bool}) = \text{list}(1..9, \text{any}) \), and none \( \sqcup \tau = \tau \sqcup \text{none} = \tau \).

The lattice structure of types from Figure 5.5 provides a domain for the application of abstract interpretation-based analysis techniques [CC77b] to obtain a combination of type and size analysis for data in the participant processes before constructing the QoS model. This kind of analysis is well suited for our case in which looping is done by iterating over list elements and streams of messages, where the size range of the list type directly translates into the range of loop iterations. We enrich the link (channel) descriptions by adding input and output message types, \( \tau_{\text{in}}(c) \) and \( \tau_{\text{out}}(c) \).

For instance, in Figure 5.4, we start with \( \tau_{\text{in}}(\text{req}) = \{\text{specs} : \text{list}(a..b, \tau_{\text{spec}}), \text{userId} : \text{any}\} \).
recv(client,specs), \( \tau_{in}(client) = \text{list}(a..b, \tau_{spec}), 1 \leq a \leq b \)

let po = [], po: list(0..0, none)

stream(approval) do

foreach(spec:specs) do [ \( a \leq k \leq b \) ]

invoke(gen,spec,alts), \( \tau_{out}(gen) = \tau_{spec}, \tau_{in}(gen) = \text{list}(1..+\infty, \tau_{alt}) \)

( if(count(alts) > 1)

-> invoke(approval, alts,choice)

; let choice=first(alts)

), choice: \( \tau_{alt} \)

let po = po + [choice]

po: list(0..0, none)

send(client,po)

\( \tau_{out}(client) = \text{list}(a..b, \tau_{alt}) \)

Figure 5.6: Analysis of types with size constraints for participant B.

Figure 5.7: Centralized (left) and distributed (right) processing of choreography QoS constraints.

number) where \( a \geq 1 \) and we derive that \( \tau_{out}(budget) = \text{number} \) and \( \tau_{in}(budget) \sqsubseteq \{ \text{forced: bool} \} \). Also, in participant B, \( \tau_{out}(agent) = \text{list}(a..b, \tau_{spec}) = \tau_{in}(client) \). The result of the analysis for B is shown in Figure 5.6. From it, we infer that \( A.N_{in}(check) = B.N_{out}(approval) \) is between 0 and \( \max(a, b) \).

5.2.5 Centralized and Distributed Processing of QoS Constraints

Solving a constraint model involves finding one (or several) set(s) of values for the constrained variables that satisfy the set of constraints, or determining that the set of constraints is unsatisfiable. Constraint solvers sometimes need to give an approximation of the actual solutions. These approximations are always complete (no solution is dis-
carded), but maybe not correct (they may contain values that are not part of any solution [Dec03]). Some constraint solvers are better suited for some classes of constraints than others. E.g., if the generated constraints are linear, a linear constraint solver is likely to detect inconsistencies and to narrow down the value sets closer to the actual answers, compared to a more general one. The constraint models generated using our approach in general involve non-linear integer and real arithmetic constraints, as well as disjunctions of constraints.

The constraint QoS models for each participant can be, in principle, derived and analyzed for the different message types separately, and the models obtained in that way can be composed together by connecting the appropriate input/output links and solving the resulting integrated model centrally. This architecture is shown on the left-hand side of Figure 5.7. Different participants may, in general, execute on different nodes (process execution engines) in a Service-Oriented System. They publish participant continuations and the related monitoring events (which can be used for establishing the previously accumulated QoS) to an event bus. An aggregated feed of continuations is read from the event bus and processed by a single component that performs the analysis, modeling, and constraint solving of the integrated participant models, and publishes the (updated) QoS metrics ranges for the entire choreography. An advantage of the centralized approach is that it offers integrated information about the behavior of the participants and QoS for the choreography. However, it may not scale well, since it requires global streaming of continuations, monitoring events and results to and from a single processing component. Besides, it can be undesirable in some settings since data regarding execution characteristics may need to be sent from their administrative domains to a central, external point.

A decentralized approach aimed at alleviating somehow these issues is shown on the right-hand side of Figure 5.7. Here, continuations and monitoring events published by process engines are processed by modules which can be close in the network topology to the engines, and (optionally) inside their administrative boundaries. These modules perform a per-participant QoS analysis that updates the ranges for \( (\tau_{out}, N_{out}, \bar{q}_{out}, \Delta\bar{q}_{out}) \) for each outgoing channel using the corresponding ranges for \( (\tau_{in}, N_{in}, \bar{q}_{in}, \Delta\bar{q}_{in}) \) that are produced by the modelers/solvers for participants at the other end. The updates are communicated to the connected participant models and the process is repeated until
a stable solution is reached. This can be achieved using distributed constraint solving algorithms [FY05], which ensure termination, completeness, and correctness.

5.3 Examples of Application

In this section, we illustrate how the proposed constraint-based approach can be of benefit in providing answers to the questions posed at the end of Section 5.1, using the motivating example. The aim of the approach is to be fully automated and supported by tools. Currently, our prototype executes processes written in the continuation language (Section 5.2.2), transmits continuations, and formulates and solves the QoS constraint models.

5.3.1 Supporting SLA Negotiation For Classes of Input Data

A constraint-based QoS model can be used at design time to help the providers of the participating processes in a choreography develop realistic SLA offers that can be used to negotiate with their users. In such a case, participant providers (e.g., the provider for participant A from Figure 5.1) can use the derived models, along with assumptions and empirical assessments of the behavior of the environment (network latency, component behavior, etc.) to develop reasonable SLA offers to the end users.

We illustrate this application with an experiment on an SLA addressing execution time. Assuming that participant A receives the request of some user at time $T_{in}(req) = 0$, we are interested in knowing which guarantees can be offered to the user with respect to $T_{out}(req)$ for a given class of input data. Besides the data, the participant QoS models for A and B depend on several internal activity parameters: $t_{send}$ is the time needed by a participant to deliver the message to a participant mailbox, $t_{budget}$ is the time needed to retrieve budget line information in activity $a_3$, and $t_{best}$ is the time required by activity $a_7$ to find the best choice among the alternatives offered.

The ranges of values for these parameters are normally empirically established by monitoring. Such empirical data is effectively a sample (or a collection of samples) of the “true population” set from which the QoS metric values are drawn and whose exact bounds are generally unknown. We can use well-known techniques of descriptive statistics on these samples to estimate the parameters of central tendency (mean, median)
Ranges for internal activity parameters

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Confidence interval 99% parameter range [ms]</th>
<th>Confidence interval 90% parameter range [ms]</th>
<th>Confidence interval 80% parameter range [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_3$: $t_{\text{budget}}$</td>
<td>500 .. 1500</td>
<td>642 .. 1167</td>
<td>673 .. 1094</td>
</tr>
<tr>
<td>$a_7$: $t_{\text{best}}$</td>
<td>100 .. 700</td>
<td>195 .. 509</td>
<td>215 .. 468</td>
</tr>
<tr>
<td>$a_{15}$: $t_{\text{gen}}$</td>
<td>200 .. 500</td>
<td>247 .. 404</td>
<td>257 .. 384</td>
</tr>
<tr>
<td>$t_{\text{send}}$</td>
<td>25 .. 150</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Case 1: Varying confidence intervals for participants A and B

<table>
<thead>
<tr>
<th>Spec. list size</th>
<th>Confidence interval 99% $T_{\text{out}(\text{req})}$ range [ms]</th>
<th>Confidence interval 90% $T_{\text{out}(\text{req})}$ range [ms]</th>
<th>Confidence interval 80% $T_{\text{out}(\text{req})}$ range [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 .. 10</td>
<td>274 .. 17100</td>
<td>322 .. 14868</td>
<td>332 .. 14376</td>
</tr>
<tr>
<td>11 .. 20</td>
<td>2274 .. 32100</td>
<td>2797 .. 27970</td>
<td>2912 .. 27057</td>
</tr>
<tr>
<td>21 .. 50</td>
<td>4274 .. 77100</td>
<td>5272 .. 67273</td>
<td>5492 .. 65103</td>
</tr>
<tr>
<td>50 .. 100</td>
<td>10074 .. 152100</td>
<td>12450 .. 132780</td>
<td>12972 .. 128512</td>
</tr>
<tr>
<td>101 .. 200</td>
<td>20274 .. 302101</td>
<td>25069 .. 263793</td>
<td>26128 .. 255330</td>
</tr>
</tbody>
</table>

Case 2: Varying confidence intervals for A and B with force=true

<table>
<thead>
<tr>
<th>Spec. list size</th>
<th>Confidence interval 99% $T_{\text{out}(\text{req})}$ range [ms]</th>
<th>Confidence interval 90% $T_{\text{out}(\text{req})}$ range [ms]</th>
<th>Confidence interval 80% $T_{\text{out}(\text{req})}$ range [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 .. 10</td>
<td>274 .. 10100</td>
<td>322 .. 8817</td>
<td>332 .. 8535</td>
</tr>
<tr>
<td>11 .. 20</td>
<td>2274 .. 18100</td>
<td>2797 .. 15867</td>
<td>2912 .. 15376</td>
</tr>
<tr>
<td>21 .. 50</td>
<td>4274 .. 42100</td>
<td>5272 .. 37017</td>
<td>5492 .. 35900</td>
</tr>
<tr>
<td>50 .. 100</td>
<td>10074 .. 82100</td>
<td>12450 .. 72268</td>
<td>12972 .. 70106</td>
</tr>
<tr>
<td>101 .. 200</td>
<td>20274 .. 162100</td>
<td>25069 .. 142768</td>
<td>26128 .. 138518</td>
</tr>
</tbody>
</table>

Table 5.1: Experimental inputs and outputs of the execution time model.
and dispersion (standard deviation) for the whole population of values. In that way, we can define intervals whose bounds include the QoS values with some level of confidence. This level will be < 100%, since, in general, total confidence is not attainable. Note that the choice of the confidence level is generally a matter of heuristics. A 99% confidence interval, for instance, is wider (and thus safer) than a 90% one, but, depending on the distribution of values, it may lead to overly conservative predictions and SLA offers to the clients that are safer, but too pessimistic, unattractive, and uncompetitive. The top part of Table 5.1 lists the ranges of the mentioned component execution time across three experimental confidence levels: 99%, 90% and 80%, with a common range for $t_{send}$.

The central part of Table 5.1 shows the ranges for $T_{out}(req)$ obtained by solving the model for each confidence interval in the experiment. In general, for each class of input data sizes, the range of $T_{out}(req)$ contracts, and its maximum, which can be offered as an element of the SLA, decreases when using smaller confidence intervals. To further refine the SLA offer, the provider for participant A can look at the branch condition in $a_6$, and offer more attractive “fast-track” conditions (with circa 40% reduction in the upper execution time bound) when it becomes known that the force flag will be set to true, as shown in the lower part of Table 5.1.

We used the $\text{ECLiPSe}$ constraint logic programming system [AW07] which has native support for integer and real non-linear arithmetic constraints, including disjunctions of constraints, that are used in the derivation of the model. Deriving the constraint models with our pilot implementation and solving them with a centralized solver took on average around 260 ms on an i86_64 laptop computer with 4GB of RAM running Mac OS X 10.7.3.

### 5.3.2 Predicting SLA Violations at Run Time

The constraint-based QoS model can be used for predicting SLA violations at runtime. Since the participant SLA is always related to some event that happens in one of the participants (such as sending the reply in activity $a_{11}$ of our sample choreography), we can apply a variation of the constraint-based prediction method for orchestrations [ICH11b]. In that method, we make predictions at each point in execution of the participant processes for which we have the continuation and the monitoring data describing the previously accumulated QoS metrics. In the case of execution time, the imminent failure condition for participant A is predicted when the constraint $T_{out}(req) \leq T_{max}$ is proven.
unsatisfiable in the constraint QoS model, i.e., when SLA compliance cannot possibly be achieved.

Using the experimental settings from the previous subsection, we predict SLA violations for a running choreography with fixed input data size (known at run time), by taking $T_{\text{max}}$ to be the upper bound of $T_{\text{out}}(\text{req})$ for the 80% confidence interval in each input data class from Table 5.1. The thick black line in Figure 5.8 shows $T_{\text{max}}$ for input data sizes in the range 1..50. The dashed lines show the upper and lower bound of $T_{\text{out}}(\text{req})$ for a 99% confidence interval. SLA violations are possible in the gray zones that correspond to data size intervals 9..10, 17..20, and 41..50. In those intervals, imminent SLA violation can be predicted between 175 ms and 325 ms ahead of $T_{\text{max}}$. For other input data sizes (in ranges 1..8, 11..16, and 21..40), the predictor is able to predict SLA conformance at the very start. In both cases, the percentage of correctly predicted cases is typically very high, between 94% and 99% [DCH12].

5.3.3 SLA Compliance Checking, Dynamic Binding and Adaptation

We now turn to a situation where there exist several implementations for a participant role in a choreography, that are known to be compatible with the communication protocol, message data types, and message cardinalities. We now want to see how the knowledge about participant QoS models can help us rule out some combinations of participant implementations (or promote others) at design time.

For instance, let us take participant $A$ from Figure 5.1, and assume that there are two implementations that can take the role of $B$ and which differ only in the method for
generating alternatives in activity $a_{15}$: while $B_1$ can generate one or more alternatives, $B_2$ always generates at least two. Although the ranges for all participant model variables of $B_2$ are subsets of the corresponding ranges for $B_1$, the combination of $A$ with $B_2$ is illegal for some SLAs and input data sizes for which $A$ with $B_1$ may work. E.g., for $T_{\text{max}} = 18000 \text{ms}$, the constraint model predicts that the combination of $A$ and $B_2$ is guaranteed to fail for input data sizes of 50 and above when $\text{forced}=\text{false}$ in $A$. Since $A$ does not control $\text{forced}$, for such data sizes it should rule out $B_2$, and choose $B_1$ which has a chance to meet the SLA.

This kind of analysis can be performed by checking that every internal structural parameter of $A$ in the constraint QoS model for the choreography (such as the condition in $a_6$ and the number of iterations of $a_4$) augmented with condition $T_{\text{out}}(\text{req}) > T_{\text{max}}$ has at least one value for which the condition $T_{\text{out}}(\text{req}) \leq T_{\text{max}}$ is satisfiable for the given range of input data sizes. Alternatively, the same check can be used for dynamic binding at run-time to select an implementation for the role of $B$ for the known size of the particular input request. Such dynamic binding provides a finer-grained per-request selection, at the cost of additional run-time analysis.

However, selecting $B_1$ does not guarantee $T_{\text{out}}(\text{req}) \leq T_{\text{max}}$: if at run time each invocation of $a_{15}$ happens to return more than one alternative (thus behaving in the same way as $B_2$), the SLA will be violated for some input data sizes. Participant $B$ can use its QoS model to detect such a situation and to adapt by forcing $a_{15}$ to start returning single items. At the beginning of each iteration in loop $a_{14}$ from Figure 5.1, $B$ can test whether the execution of $a_{15}$, if it generates multiple alternatives, can lead to an SLA.
violation. If so, it can coerce $a_{15}$ to produce a single item and so enforce the SLA. The earliest points in time when that can happen for input data sizes in the range $17..20$ and $T_{\text{max}} = 27057$ ms (the central gray zone in Figure 5.8), are shown in Figure 5.9. $k_B$ stands for the previous number of iterations of $a_{14}$, and $k_A$ stands for the previous number of times when more than one alternative was generated in $a_{15}$.

5.4 Conclusions

The constraint-based approach to QoS assurance for service choreographies presented is based on the automatic derivation of QoS constraint models from abstract descriptions of multiple participating processes that can engage in complex, stateful conversations. The QoS attributes that can be modeled include execution time, availability, monetary cost, the quantity of data transferred, and any others that can be mapped onto cumulative, non-negative numerical metrics. For greater precision, the model derivation is augmented with an analysis of message types with size constraints, and the resulting models are data sensitive. The participant models can be derived, integrated, and solved centrally, or in a distributed fashion. The approach can be used at design-time, for classes of input data, and also at run time, with the actual data, whenever the information about the current point in execution is provided for the participants. The resulting models can be used to support SLA negotiation, SLA violation prediction, design-time SLA conformance for classes of input data, dynamic binding of participants, and SLA-driven run-time adaptation.
Chapter 6

A Sharing-Based Approach to Supporting Adaptation in Service Compositions

In this chapter we address the problem of adaptation at the level of service compositions [DNGM+08, PPPM10], from the data perspective. At design time, adaptation\(^1\) is usually performed in order to meet new functional requirements, adjust non-functional characteristics of the composition (e.g., by removing inefficiencies and bottlenecks), or to enhance interoperability with other systems. At run time, adaptation is typically performed in response to the detection (or prediction) of component failures, extraordinary situations (e.g., exceptions, communication line breakups), or new information about the particular user’s context.

In both design-time and run-time adaptation settings, the notion of *correctness*, in the sense that the adapted service composition has to comply with its specification, is crucial. This involves ensuring, among other things, that compatibility with the specified protocols is preserved during conversations with partner (or component) services [PBCT07], that the appropriate partner operations are invoked, and that the messages that are

\(^1\)By convention, and in order to differentiate actions at design time and at run time, any adaptation which impacts the initial assumptions of a SOC, such as those stemming from changes in the requirements and which require a (deep) change in the design of the system, is termed *evolution*. We will in general not use this term unless it is unclear from the context whether we are referring to design time or run time adaptation.
exchanged have the correct format and meaning [BP08]. An adaptation can alter the state of the service composition, replace its components, rearrange activities, reroute messages, fragment the composition into several parts that are executed in a distributed manner or merge several fragments into one, etc., but any adaptation action, simple or complex, must respect the conversation protocols, service interfaces, and message formats and meaning. These, of course, are necessary, but by themselves not sufficient, conditions for correctness of the adaptation.

We argue that the degree of correctness of some adaptations can be improved by taking into account the data dependencies present in the specification, i.e., by analyzing how the content of messages and the composition state depend on the received messages or behavior of other activities and other factors. In particular, we are interested in functional dependencies, which describe what pieces of information and activities determine the content of messages and the internal state, as well as in data attributes that describe the contents of data in terms of some domain-specific (i.e., user-defined) properties. Both imply additional correctness criteria that can be used for validating a set of potential adaptation actions, or as a constraint for their generation.

For instance, an adaptation action that breaks a functional dependency by (partially) blocking the flow of information between services in the composition or by replacing one service with another that uses only a subset of inputs risks incurring in information loss and should be (in the absence of more precise information) ruled out as unsafe. On the other hand, if inputs to two activities do not mutually depend on their state or results, they can in principle be executed independently or in parallel, i.e., isolated into two separate control-flow fragments. Additionally, if the data attributes of a message carry confidential information, then we have to ensure that this information will not be revealed to distrusted participants. Also, the analysis of data attributes allows reasoning about the conceptual information content of the message, which can constrain the search for service candidates by discarding those who expect (or produce) messages whose contents are conceptually incompatible.

Our focus is on supporting an automated analysis of data dependencies that can be used both at run time and as a designer’s aid for design-time adaptation. While analysis of protocols relies mostly on automata theory and process algebras [PBCT07, DCD10], and analysis of interfaces and message formats is performed mostly by reasoning about
service ontologies [ES07], we base our approach to the analysis of data dependencies on the concept of data sharing and a number of related static analysis techniques based on abstract interpretation [JL89, MH89, MH91]. The application of program analysis techniques is necessary in this case because of two main reasons. Firstly, realistic service compositions involve complex control structures, featuring non-determinism and cyclic execution paths, which may be difficult to analyze without resorting to well-established program abstraction frameworks. Secondly, in a run-time adaptation scenario, the specification is usually the original (pre-adaptation) version of a service composition (typically describing the canonical or the most general case), which is usually expressed using programming language constructs, and a correct adaptation needs to preserve some key properties of the original.

The rest of the chapter is organized as follows. Section 6.1 presents an example that motivates our approach. In Section 6.2 we discuss functional dependencies and data attributes in the context of service compositions, and show how these two aspects of data dependency can be modeled using the same logical framework and the notion of sharing. In Section 6.3 we present the details of sharing analysis, which is the underlying technique used for inferring (safe approximations of) functional dependencies and data attributes in service compositions. We discuss the inputs to the process, and the interpretation of its results. Next, we present some applications in Section 6.4. Section 6.5 offers a review of the related work, and Section 6.6 some conclusions.

6.1 Motivation

Figure 6.1 shows an example service composition written in BPMN [Obj11] that realizes a process of purchasing goods from a seller’s online site and paying for the goods by credit card, bank transfer, or a loan from a bank. Service invocations and complex activities (XOR-splits and loops) are labeled with letter $a$ and a subscript, for easier reference in the text that follows. At the start, the buyer accesses the seller’s online site either by logging in to his or her existing account (activity $a_2$), or by browsing anonymously ($a_3$). Activity $a_4$ creates a new shopping cart, and activity $a_5$ is a loop in which the buyer browses the catalog ($a_6$) for the base product and its configuration options (e.g., disk, display, memory options for a computer, or the additional equipment for a car) and...
either adds the item found to the shopping cart (a₈) or skips it (a₉). At the end, the buyer freezes the order (a₁₀) and moves to the check-out stage. At this point, if the buyer had not already logged in, he or she will need to create a profile (a₁₃), by giving the name, address, telephone, and other basic information, which can be saved for future sessions. Once the buyer is registered, the invoice for the ordered goods is generated (a₁₄).

In the second stage of the process, the buyer chooses among several forms of payment (the exclusive choice a₁₅). One of these choices is to pay using a bank card (a₁₆), which needs to be authorized by the bank (a₁₇). Alternatively, the buyer can pay by debiting his or her bank account (a₁₈). Finally, the buyer may request a loan from a bank that the buyer uses for financing his sales (a₁₉). Depending on whether or not the buyer is a client of the bank and other circumstances, the bank may request one or more documents to be presented (a₂₀) before approving the loan (a₂₅). After fixing the payment option in the exclusive choice activity a₁₅, the payment is made by the bank on behalf of the buyer.
to the seller \(a_{26}\). Finally, once the payment is verified, the seller ships the ordered products \(a_{27}\).

In this fairly standard service composition let us assume that each atomic activity is an invocation of a service (or rather of some operation of some port of a service) which accepts some input and may produce an output message, which is remembered in the state of the composition. The components of the state are shown in Table 6.1. Where input from the buyer is required, we assume that the corresponding service either takes care of presenting the buyer with a form to fill, or has an interface that the buyer’s application can use to supply the required information (and obtain results).

The purchasing composition in Figure 6.1 is rather generic and can be adapted in several ways. In particular, there are several classes of run-time adaptations, such as parallelization, fragmentation, and compliance checks, that can be triggered and applied automatically, ideally without any human intervention. The information on which these adaptation actions can be based comes from the analysis of both the control and the data dependencies in a service composition. In this chapter we argue that both the functional data dependencies and the data attributes (which describe the information content) can be analyzed by means of an (abstract) data sharing analysis. On that basis, we are motivated by the following questions relevant for adaptation:

- **Which activities in the composition do not functionally depend on each other’s output, and can therefore be started in parallel?** For instance, in Figure 6.1, a loan request activity \(a_{19}\) can be started without waiting for \(a_{14}\) to finish (although \(a_{25}\) needs to wait for \(a_{20}\)).

- **How can we automatically fragment the composition for distributed enactment, while enforcing information flow constraints?** E.g., the fragments of the composition from Figure 6.1 are executed either centrally or on the side of the buyer, the bank, and the seller. The assignment of activities to fragments can be based on data attributes that conceptually describe the information content of the data handled or produced by each activity.

- **How can we automatically choose or replace service components based on information requirements?** Functional dependencies and data attributes can be used as one of the criteria to constrain matching or generation of the replacement, by en-
ensuring that all replacements use all of the relevant data, and by ensuring that the information content of that data (described by data attributes) is adequate.

Note that in all these cases we assume that the usual adaptation constraints related to conversation protocols, message formats, and meaning are correctly preserved, and we focus on analyzing the data content, which is not normally taken into account by protocol- and ontology-based approaches.

### 6.2 Functional Dependencies and Data Attributes Through Sharing

In this section we show how the notion of data sharing, in a very general, first-order logical framework, can be used to express two classes of data dependencies: functional dependencies and sharing of the data attributes between input messages sent to the composition, intermediate data items and activities, and outgoing messages from the composition. We start by looking at functional dependencies and data attributes in service compositions, and proceed by presenting the general framework for their common
logical representation and generation by means of Horn-clause programs and their operational semantics.

6.2.1 Functional Dependencies in Service Compositions

Functional dependencies have been widely studied in the field of database systems [Arm74], where they represent (together with multi-valued dependencies) the cornerstone of the most widely applied relational database design and normalization techniques (such as Codd’s normal forms) [Ull88]. They have also been extensively studied in logic [Hin04, Vää07], and used for mining of association rules in databases [AIS93].

In the context of service compositions, we are interested in the functional dependencies between some named items, which we call variables, that represent incoming and outgoing messages in a service composition, the internal data produced and read by the composition activities, and the activities themselves. Unlike the conventional “program variables”, these variables do not designate mutable storage locations, but rather act as logical placeholders for values. We shall use the italic letters \( x, y, z \), etc., to denote these variables, and the capitals \( X, Y, Z \), etc., to denote sets of variables. In keeping with the usual short-hand notation for functional dependencies, we shall write \( XY \) to denote \( X \cup Y \) (unless stated otherwise), and wherever a set of variables is expected, we shall allow a variable \( x \) to stand for the singleton set \( \{x\} \). E.g. \( X yZ = X \cup \{y\} \cup Z \), and \( xyz = \{x\} \cup \{y\} \cup \{z\} = \{x, y, z\} \).

**Definition 6.2.1** (Functional dependency). A set \( X \) of variables is said to determine a set \( Y \) of variables (i.e., \( Y \) functionally depends on \( X \)), which we write \( X \rightarrow Y \), if for each \( y \in Y \) there exists a rule that uniquely determines the value of \( y \) from values of the variables in \( X \).

Obviously, when \( X = \emptyset \) then each \( y \in Y \) is a constant, i.e., they have a unique value that does not depend on any other variable. For \( X \neq \emptyset \) we normally speak of a function or mapping \( F \) which, when applied to the values of some or all variables from \( X \), produces a single value for each \( y \in Y \), and we write \( Y = F(X) \).

Functional dependencies are normally required to obey a standard set of axioms, known as the Armstrong dependency axioms:
Definition 6.2.2 (Armstrong dependency axioms). A binary functional dependency relation \( \rightarrow \) between subsets of variables needs to satisfy the following set of axioms:\(^2\)

\[
\begin{align*}
(AD1) & \quad X \rightarrow X \\
(AD2) & \quad (X \rightarrow Y) \land (U \rightarrow V) \rightarrow (XU \rightarrow YV) \\
(AD3) & \quad (X \rightarrow Y) \land (U \rightarrow V) \land U \subseteq Y \rightarrow (X \rightarrow V)
\end{align*}
\]

for arbitrary sets of variables \( X, Y, U \) and \( V \).

From the point of view of adaptation, reasoning about functional dependencies in service compositions is important in several adaptation settings. If, for instance, the inputs of activity \( a'' \) do not depend on the outputs of \( a' \), then \( a'' \) can proceed independently of \( a' \) (i.e., we can parallelize the two), and even the failure or a dynamic replacement of \( a' \) with a compatible activity does not affect \( a'' \). Conversely, if some activity depends on some set of variables \( X \), then, in general, it cannot be executed before all ingredients of \( X \) become available (as received messages or as outputs of other activities), and it is generally not safe to replace that activity with another (atomic or complex) one that takes as input some proper subset \( Y \subset X \), since that could lead to a loss of data.

Note that in the adaptation examples above we preserve the conversation protocols and the message formats, but, nevertheless, the actual outcomes may differ significantly.

Example 6.2.3. In the composition from Figure 6.1, activity \( a_{19} \) (Request loan) does not depend on outputs from \( a_4, a_5, a_{10}, \) or \( a_{11} \). Therefore, if at the start the buyer is determined to take a loan, and knows the approximate price range, he or she can start the loan request process in advance of, or in parallel with, the online ordering. However, \( a_{25} \) (Loan approval) needs the invoice that is produced by \( a_{14} \) (Generate invoice).

Example 6.2.4. If payment processing (\( a_{26} \)) is executed by the bank, and shipment by the seller (\( a_{27} \)), without any direct message exchange between the two, then there is an implicit data dependency on the state of the seller’s account (which may be in another bank). This is a “hidden” variable that has to be included in the functional dependencies.

Example 6.2.5. If in \( a_{25} \) we replace the commodity loan approval with a cash loan approval, which does not look at the invoice, the payment would be made to the buyer’s

\(^2\)The set of axioms presented here follows [VMG04] and has been chosen for its minimality. These axioms are equivalent to those originally proposed by Armstrong [Arm74].
account, rather than to the seller's, and the shipment would not commence. That would require an additional bank transfer step by the user to be inserted after $a_{25}$.

**Example 6.2.6.** If the user logs in to the seller's portal ($a_2$), we can replace the general catalog search service $a_6$ with a version that takes into account the buyer's identity, which functionally determines the buyer's purchase history and interests.

Figure 6.2(a) shows a conceptual model of functional dependencies arising from a single execution of some activity $a$. If $a$ reads some data $X$ and produces some outputs $Y$, we can generally assume that there exists some functional dependency $YU' = F_a(XU)$, where either $X$ or $Y$ (or both) can be empty. $U$ is a representation of the internal state used by $a$ before its execution, and $U'$ is its updated state. If $a'$ is the next activity after $a$ that uses the same internal state, its functional dependencies have the form $Y'U'' = F_{a'}(X'U')$, where $X'$ and $Y'$ are the sets of data items read and written, respectively, by $a'$.

Our ability to draw conclusions from such a generic functional dependency $XU \rightarrow YU'$ may be limited for several reasons. Firstly, we may not (and indeed in general do not) know $F_a$, which expresses the (denotational) semantics of the computation performed by $A$. Secondly, we may not know the structure of $U$ and $U'$, unlike the structure of $X$ and $Y$ which correspond to the data items in the composition. However, more precise reasoning can be obtained under specific circumstances or when we are given some additional information:

- If the internal state of $A$ is initialized to a default value at the start of a composition, then in the first use of the state $U$ is a constant, and the dependency reduces to $X \rightarrow YU'$. This, for instance, happens with the order $o$ after $a_4$ (Figure 6.1 and Table 6.1).
• If we know that A is stateless, the functional dependency reduces to $X \rightarrow Y$. That is the case with $a_9$ (skip item) and $a_{18}$ (Get bank transfer details) that produce their results directly from the inputs (if any).

• If A does not update its state, then we can decompose $XU \rightarrow YU'$ into $XU \rightarrow Y$ and $U' = U$. That is the case with $a_{14}$ (Generate invoice), which reads the state variable $o$ (the order), but does not change it.

• If we know that for some $Z \subset X$, $ZU \rightarrow YU'$, we can eliminate from consideration all irrelevant variables from $X \setminus Z$. E.g., loan approval $a_{25}$ may require proof of residence address $r$, but may not use its content to generate payment authorization $z$ needed by $a_{26}$.

Direct functional dependencies, such as those from Figure 6.2(a), can be given in the form of assertions or metadata attached to service activities, while indirect (transitive) dependencies are obtained by applying the Armstrong axioms from Definition 6.2.2. The general idea is that this kind of reasoning about functional dependencies becomes more precise if in the activity assertions we can narrow the left side of "\( \rightarrow \)" (replacing $X$ with $Z \subset X$, or eliminating $U$), and/or if we know more about the right hand side (e.g., $U' = U$).

If no cycles are involved, the set of all functional dependencies (over a finite set of variables) can be obtained by applying the Armstrong axioms directly, a finite number of times. However, if an activity that reads $X$ and writes $Y$ is in fact a loop $a^*$ whose body $a$ can be executed zero or more times, then we are generally interested in some common properties of the set $Y^*$ of all possible values of $Y$, rather than a $Y$ from a single iteration. This corresponds to the notion of collecting semantics in static program analysis [NNH05]. Figure 6.2(b) shows the conceptual model for this case. Starting from an empty set $\emptyset$, each iteration in $a^*$ updates both the internal state and the set of outputs $Y^*$. In some cases, $Y^*$ may stabilize after a finite set of iterations, but that is generally not guaranteed. Since we are interested in treating this general case, which may involve loops with a generally undecidable number of iterations, we resort to techniques based on abstract interpretation that will be discussed in Section 6.3.

6.2.2 Data Attributes in Service Compositions

The messages that are sent and received by participants in a service composition are typically dynamic XML documents that may have a complex structure, with nested, op-
tional, and alternative elements. While different technologies (such as DTD and XML Schemas) can be used to constrain the shape and the content of XML messages, these checks can be expensive in terms of computation time, and there is generally no assurance that a particular service infrastructure enforces all the tests for each message that is sent or received. In general, it is the responsibility of the developer to ensure that the messages conform to their XSD specifications, besides being well-formed XML documents. The same applies to the results of XPath or XQuery queries or XSLT transformations used in compositions for computing values of composition state variables.

We are interested in conceptual descriptions of data, where we use the notion of a data attribute to describe some property that holds for a data item represented with a variable (in the sense of variables from Definition 6.2.1 in the previous section). Some properties (i.e., attributes) can be verified by performing a test on the variable (i.e., by inspecting its content), while other properties can be inferred based on domain-specific inference rules from the tests.

**Example 6.2.7.** For instance, variable $d$ (Identification document) from Table 6.1 can be tested for properties such as “contains element address of type string” (property $p_1$), “contains attribute @allowsResidenceCheck with value true” (property $p_2$), or “contains element PIN” (property $p_3$), while an abstract property can be “has a known residence address” ($p_4$). The inference rules could be $p_1 \rightarrow p_4$ and $p_2 \land p_3 \rightarrow p_4$.

We start by defining the notion of structural data description, following an approach similar to that of Data Semantic Structures [BP08], that originate from semantic data descriptions and ontology matching [Shv05, ES07].

**Definition 6.2.8 (Structural Data Description).** A conceptual data description is a structure $\text{sdd}(D, \sqsubseteq_d, \subsetneq, \sim)$ where:

- $D$ is a set of concepts (conceptual data types), where each $d \in D$ denotes a set of objects (data items) conforming to $d$;
- $\sqsubseteq_d$ is a partial order on $D$, called the simulation relation, where $d \sqsubseteq_d d'$ means that an object from $[d]$ can be used whenever an object from $[d']$ is expected;
- $\subsetneq \subseteq D \times F \times D$ (where $F$ is a set of field names) is a component relation, such that $\langle d, n, d' \rangle \in \subsetneq$, written as $d \subsetneq_n d'$, meaning that each object from $[d]$ has a field (element or attribute) called $n$ that holds an object from $[d']$;

115
• \( \leadsto \) is a partial transformation functional that maps a pair \( \langle d, d' \rangle \in \mathbb{D}^2 \) (if a transformation from \( d \) to \( d' \) is defined) into a function \( (d \leadsto d'): \llbracket d \rrbracket \rightarrow \llbracket d' \rrbracket \) that transforms object \( x \in \llbracket d \rrbracket \) into its image \( x' \in \llbracket d' \rrbracket \).

Structural data descriptions can be thought of as a sort of typing system for data that is exchanged by services in the composition. The simulation relation \( d_1 \leq_d d_2 \) requires that whenever \( d_2 \sim_n d \), then also \( d_1 \sim_n d \), i.e., objects in \( \llbracket d_2 \rrbracket \) must have all components that the objects in \( \llbracket d_1 \rrbracket \) have, and possibly some more. Also, in that case \( (d_1 \leadsto d_2) \) exists and is an identity function.

**Example 6.2.9.** Figure 6.3 shows a simple example of a structural data description for identification documents that may be used for identifying a buyer in some country, i.e., as the data item \( d \) from Table 6.1 which is obtained from activity \( a_{19} \) (Loan request) in Figure 6.1. These include passports (Passport), driving licence (DL), and identity cards (ID-Cards), which are further subdivided into those for citizens of the country (CitizenID) and other residents (ResidentID). While passports and driving licenses represent the person’s name and address simply as strings, identity cards use a more structured representation via conceptual data types Name and Address. While Address \( \leadsto \) string and Name \( \leadsto \) string are straightforward, string \( \leadsto \) Address may be difficult, and string \( \leadsto \) Name may be ambiguous. CitizenID and ResidentID are discriminated by fields national and resident, respectively. Some field names have different meanings, e.g., number in Passport and in Address.
### Definition 6.2.10 (Data attributes). For an ssd(\(D, \sqsubseteq_d, <, \sim\)), a data attribute system is a structure \(da(A, T, R, \models_d)\), where:

- \(A\) is a set of data attributes, which are propositional symbols that may evaluate to true or false for each \(d \in D\).
- \(T\) is a set of deterministic and terminating Boolean tests that describe the contents of a data object. A set of objects \(X\) satisfies \(t \in T\) iff \(t(x) = \top\) for each \(x \in X\). A \(d \in D\) satisfies \(t\) iff \([d]\) satisfies \(t\).
- \(R\) is a finite set of implication rules of the form \(\land B \rightarrow a\) (\(B\) finite), where \(a \in A\) and \(B \subseteq A \cup T\). Circular implications (direct or indirect) are not permitted.
- \(\models_d\) is the attribute inference relation that associates a set of data objects \(S\) with attributes from \(A\), in such a way that for all \(a \in A\), \(S \models_d a\) holds if and only if \(a\) can be inferred from the tests that are satisfied by \(S\) and the set of rules \(R\).

### Example 6.2.11. Figure 6.4 shows a data attribute system for the sample structural data description form Figure 6.3. Tests \(t_1, \ldots, t_5\) are checked at the level of a data instance \((x)\), and here they have the simple format \(x : d\) (meaning that \(x\) complies with the structural schema for \(d\), in tests \(t_1\)–\(t_3\)), or an XPath query \((t_4\) and \(t_5\)). The attributes \(m_1, \ldots, m_5\) represent domain-specific properties related to the content, rather than to the structure of a data object. They are obtained from the tests using the given set of rules (Horn clauses). The bottom part of Figure 6.4 shows a matrix of the tests and attributes for several cases.
of data objects. The cases mainly correspond to the conceptual data types from Figure 6.3, but the case called XY Passports is a set of objects which is structurally indistinguishable from Passport, yet its content (i.e., the fact that the passport is issued to the citizen of the hypothetical country XY in question) makes it different from the point of view of the domain-specific attributes.

When considering data at the level of data attributes, we are mostly concerned with the conceptual grouping of the possible cases on the basis of the information they carry with respect to the attributes. For that, we use concepts from Formal Concept Analysis (FCA, a branch of lattice theory [DP02, GSW05]). In FCA, a context is a triplet \((G, M, I)\) where \(G\) is a set of objects (or cases), \(M\) is a set of attributes, and \(I \subseteq G \times M\) is a relation that associates the objects with the attributes. For arbitrary sets of objects \(X \subseteq G\) and attributes \(Y \subseteq M\), we define the operators:

\[
X' = \{ y \in M | (\forall x \in X)(\langle x, y \rangle \in I) \}
\]

\[
Y' = \{ x \in G | (\forall y \in Y)(\langle x, y \rangle \in I) \}
\]

which can be described as follows: \(X'\) is the set of all attributes associated (via \(I\)) to all objects of \(X\), and \(Y'\) is the set of all objects associated to all attributes from \(Y\). The pair \(\langle X, Y \rangle\) is called a concept if \(X' = Y\) and \(Y' = X'\), i.e., if \(X\) and \(Y\) completely determine each other by means of the operators \((\cdot)'\). \(X\) is called the extent of the concept, and \(Y\) its intent. The fundamental theorem of FCA asserts that concepts form a complete lattice, with the ordering \(\langle X_1, Y_1 \rangle \leq \langle X_2, Y_2 \rangle\) if \(X_1 \subseteq X_2\), or, equivalently \(Y_2 \subseteq Y_1\). Concept lattices are usually represented with a variation of Hasse diagrams with the greatest concept at the top, and the smallest concept at the bottom. Each node is a concept, and is decorated with attributes that do not appear in greater concepts, and the objects that do not appear in smaller ones.

**Example 6.2.12.** Figure 6.5 shows a concept lattice based on the attribute matrix from Figure 6.4. The concept ordering shows how informative the different cases are from the point of view of attributes \(m_1, \ldots, m_5\). Passport is more general than other identity documents (except DL), while CitizenID is more informative than other variations of ID Cards and passports. Generic IDCard and ResidentCard are conceptually the same. The data item \(d\) from Table 6.1 can be chosen from the set of objects assigned to nodes.
A complex message may consist of several parts that carry their own data attributes. In such a setting, we are interested in all the attributes present in the message. If the components of a message are represented with sets of attributes, then the attributes of the entire message can be combined on the basis of the component relation “⊆”. The input $X$ to an activity $a$ can be represented as a set of variables carrying data attributes of the component data objects, which come from the inputs or previous activities. The output $Y$ from $a$ can be represented as $Y = ZU$, where $Z \subseteq X$, whose attributes $Y$ inherits, and $U$ represents some new components added to $Y$ by $a$.

### 6.2.3 Representing Dependencies With Substitutions

In our approach we use notions from first-order logic as the underlying mechanism for both functional dependencies and data attributes. A first-order language represents objects in the universe of discourse by means of terms, which are built from variable symbols ($x, y, z, \ldots$), constants (e.g., 0, $a$), and function symbols ($f, g, \ldots$). A complex term of the form $f(t_1, t_2, \ldots, t_n)$ (where $t_1, \ldots, t_n$ are terms) is normally seen as the result of applying some function $f$ of $n$ arguments to terms $t_1, \ldots, t_n$. Therefore, an equation $z = f(x, y)$ is a common mathematical way of expressing a functional dependency of $z$ on $x$ and $y$ by means of $f$. To actually apply $f$ to its arguments, we need to be equipped with an interpretation that assigns some actual computation procedure to the function symbol $f$, but even when $f$ is left uninterpreted, $z = f(x, y)$ is a statement of the existence of a functional dependency $xy \rightarrow z$. On the other hand, from a purely syntactic point of view, $f(x, y)$ can be seen as a grouping of $x$ and $y$ under $f$, in much the same way as
fields are packed together in a record. If \( x \) and \( y \) carry their sets of data attributes, then \( z = f(x, y) \) is also a convenient way to state that \( z \) inherits the attributes of both.

We can represent the above equation with a substitution \( \sigma = \{ z \rightarrow f(x, y) \} \). When applied to a term or a statement, a substitution simultaneously replaces all occurrences of the variables on the left-hand side of “---” with the corresponding terms on the right-hand side. With \( \text{dom}(\sigma) \) we denote the set \( \{ x \mid (x \rightarrow t) \in \sigma \} \), and with \( \text{range}(\sigma) \) the set \( \cup_{(x \rightarrow t) \in \sigma} \text{vars}(t) \), where \( \text{vars}(t) \) is the set of variable symbols occurring in term \( t \). To rule out circular references, we require \( \text{dom}(\sigma) \cap \text{range}(\sigma) = \emptyset \). Also, to enforce determinism, if a substitution contains mappings \( x \rightarrow t \) and \( x \rightarrow t' \), then we require \( t \equiv t' \), where “\( \equiv \)” stands for the syntactical identity of terms as strings of symbols. In our example, \( z\sigma \equiv f(x, y) \) and \( g(x, z)\sigma \equiv g(x, f(x, y)) \).

In Definition 6.2.2 (the Armstrong dependency axioms) we have already stated the properties expected of a relation that models functional dependencies. With the following definition we provide the substitution-based functional dependency relation.

**Definition 6.2.13.** Let \( X \) and \( Y \) be two subsets of variables, and \( \sigma \) a substitution. We say that \( Y \) functionally depends on \( X \) under \( \sigma \), and write \( X \rightarrow_{\sigma} Y \), if \( \text{vars}(Z\sigma) \subseteq Y \), where \( Z = X \setminus Y \).

We establish the expressiveness of “\( \rightarrow_{\sigma} \)” with the next two lemmas and the theorem.

**Lemma 6.2.14.** For an arbitrary substitution \( \sigma \), relation \( \rightarrow_{\sigma} \) satisfies the Armstrong dependency axioms.

**Proof.**

1. From Definition 6.2.13, \( \text{vars}((X \setminus X)\sigma) = \text{vars}(\emptyset) = \emptyset \subseteq X \), thus \( X \rightarrow_{\sigma} X \).

2. From Definition 6.2.13, \( X \rightarrow_{\sigma} Y \) and \( U \rightarrow_{\sigma} V \), we have \( \text{vars}((Y \setminus X)\sigma) \subseteq X \) and \( \text{vars}((V \setminus U)\sigma) \subseteq U \), and, by union, \( \text{vars}((Y \setminus X)\sigma) \cup \text{vars}((V \setminus U)\sigma) \subseteq XV \). Now, \( \text{vars}((YV \setminus XU)\sigma) = \text{vars}((Y \setminus XU)\sigma) \cup \text{vars}((V \setminus XU)\sigma) \), and because \( Y \setminus XU \subseteq Y \setminus X \) and \( V \setminus XU \subseteq V \setminus U \), we have \( \text{vars}((Y \setminus XU)\sigma) \subseteq \text{vars}((Y \setminus X)\sigma) \) and \( \text{vars}((V \setminus XU)\sigma) \subseteq \text{vars}((V \setminus U)\sigma) \). Again, by union, we obtain \( \text{vars}((YV \setminus XU)\sigma) = \text{vars}((Y \setminus XU)\sigma) \cup \text{vars}((V \setminus XU)\sigma) \subseteq \text{vars}((Y \setminus X)\sigma) \cup \text{vars}((V \setminus U)\sigma) \subseteq XV \), which means \( XU \rightarrow_{\sigma} YV \).

3. Here again from Definition 6.2.13, \( X \rightarrow_{\sigma} Y \) and \( U \rightarrow_{\sigma} V \), we have \( \text{vars}((Y \setminus X)\sigma) \subseteq X \) and \( \text{vars}((V \setminus U)\sigma) \subseteq U \). Because \( U \subseteq X \), we also have \( V \setminus X \subseteq V \setminus U \), and therefore \( \text{vars}((V \setminus X)\sigma) \subseteq \text{vars}((V \setminus U)\sigma) \subseteq U \subseteq X \), i.e., \( X \rightarrow_{\sigma} V \).
Indeed, as the Armstrong axioms suggest, some functional dependencies may be evident directly in the substitution, while others can be deduced from them. In our example, $\sigma = \{z \rightarrow f(x,y)\}$, the dependency $xy \rightarrow_{\sigma} z$ is directly represented, while, e.g., $xyz \rightarrow_{\sigma} z$ and $xy \rightarrow_{\sigma} y$ are implicit. We formalize the notion of the directly represented relationships in the following definition.

**Definition 6.2.15.** For a given substitution $\sigma$ the functional dependency basis is defined as $[\sigma] = \{\langle X, y \rangle \mid (y \rightarrow t) \in \sigma \wedge X = \text{vars}(t)\}$.

**Lemma 6.2.16.** For an arbitrary substitution $\sigma$, no relation that is smaller than $\rightarrow_{\sigma}$ and includes $[\sigma]$ satisfies the Armstrong dependency axioms.

**Proof.** Suppose $\triangleright$ is a relation between subsets of variables from language $\mathcal{L}$ that satisfies the Armstrong dependency axioms, such that $\triangleright \subseteq \rightarrow_{\sigma}$ and $[\sigma] \subseteq \triangleright$.

(1) For arbitrary $X$ we have $\text{vars}((X \setminus X)\sigma) = \emptyset \subseteq X$, i.e., $X \rightarrow_{\sigma} X$ and also in $X \triangleright X$ from (AD1) in Definition 6.2.2.

(2) For arbitrary $X$ and $Y$ such that $Y \subseteq X$, from Definition 6.2.2 we have $\text{vars}((Y \setminus X)\sigma) = \emptyset \subseteq X$, i.e., $X \rightarrow_{\sigma} Y$. Also, from Definition 6.2.2 (AD1) we have $X \triangleright X$ and $Y \triangleright Y$, and with $Y \subseteq X$ from (AD3) we obtain $X \triangleright Y$.

(3) For arbitrary $X$ and $Y$ such that $Y \not\subseteq Y$ and $X \rightarrow_{\sigma} Y$, from Definition 6.2.2 we have $\text{vars}((Y \setminus X)\sigma) \subseteq X$. Therefore, for each $y \in Y \setminus X$, (and there has to be at least one such $y$), $\sigma$ must contain a mapping of the form $y \rightarrow t$, where $\text{vars}(t) = \text{vars}(y\sigma) \subseteq X$, or, in other words, $X \rightarrow_{\sigma} y$. Now, from (2) above, we have $X \rightarrow_{\sigma} \text{vars}(y\sigma)$ and also $X \triangleright \text{vars}(y\sigma)$. Since $[\sigma] \subseteq \triangleright$, we have $\text{vars}(y\sigma) \triangleright y$, and, by (AD3), $X \triangleright y$. By applying axiom (AD2) over all such $y$, we conclude $X \triangleright Y$.

From (1)-(3) we conclude that all elements from $\rightarrow_{\sigma}$ are also present in $\triangleright$, and therefore, no proper subset of $\rightarrow_{\sigma}$ which contains $[\sigma]$ satisfies the Armstrong dependency axioms.

**Theorem 6.2.17.** For an arbitrary substitution $\sigma$, $\rightarrow_{\sigma}$ is the smallest relation that expresses exactly those functional dependencies that are either present in the base $[\sigma]$, or can be deduced from it using the Armstrong dependency axioms.

**Proof.** Follows directly from Lemmas 6.2.14 and 6.2.16.
To prove the adequacy of using substitutions for expressing inheritance of data attributes it suffices to demonstrate that if $x_1, \ldots, x_n$ are variables from $\text{dom}(\sigma)$ that represent data objects, so that $\text{vars}(x_i \sigma)$ is the set of data attributes for $x_i$ under $\sigma$, then by extending $\sigma$ with a mapping $z \mapsto t$, where $\text{vars}(t) = \{x_1, \ldots, x_n\}$, from the definition of $\text{vars}(\cdot)$ it follows that the set of attributes of $z$ under $\sigma$, $\text{vars}(z \sigma) = \bigcup_{i=1}^n \text{vars}(x_i \sigma)$.

### 6.2.4 Variable Sharing

It can be easily seen that expressing functional dependencies and data attribute inheritance by means of substitutions does not depend on the choice of function symbols and the shape of the terms on the right-hand side of “$\mapsto$”. As long as the invariant $\text{vars}(t) = \text{vars}(t')$ holds, we can replace any mapping $x \mapsto t$ with $x \mapsto t'$ in $\sigma$ without losing any result from the previous subsection. This indicates that a substitution can be presented in a more abstract manner [JL89, MH89, MH91]. The following definitions formalize that notion.

**Definition 6.2.18** (Sharing). A non-empty set $S$ of terms is said to share if $\bigcap_{t \in S} \text{vars}(t) = X \neq \emptyset$. $X$ is the set of the variables shared in $S$.

**Definition 6.2.19** (Abstract Substitution). Let $Y$ be a set of variables of interest, and $\sigma$ a substitution. We define the abstract substitution $\alpha_Y(\sigma)$ in the following way:

$$\alpha_Y(\sigma) = \{\{y \in Y | x \in \text{vars}(y \sigma)\} | x \in Z\},$$

where $Z = (Y \setminus \text{dom}(\sigma)) \cup \text{range}(\sigma)$. 

**Example 6.2.20.** Let $Y = xyzu$ and $\sigma = \{x \mapsto f(u,v,w), y \mapsto g(u,v), z \mapsto h(w)\}$. Then, $Z = uvw$ and $\alpha_Y(\sigma) = \{xyz, xyu, xz\}$. After applying $\sigma$ to $Y$ we get $\{f(u,v,w), g(u,v), h(w), u\}$, respecting the order in which the set $xyzu$ is written. The (singleton) set of variables $u$ appears in terms $f(u,v,w)$, $g(u,v)$, $u$ which correspond to the initial variable set $xyzu$ after applying $\sigma$. The set of variables $\{v,w\}$ appears in terms $f(u,v,w)$ and $g(u,v)$, coming from the initial set of variables $xyz$. The set of variables $\{w\}$ appears in terms $f(u,v,w)$ and $h(w)$, which come from the set of variables $xz$. If $Y = xyz$, we get $\alpha_Y(\sigma) = \{xy, xz\}$. If $Y = xyu$, we get $\alpha_Y(\sigma) = \{xyu, xy, x\}$. 122
Each member of $a_Y(\sigma)$ is called a sharing group, and each sharing group represents a set of variables shared between the members of the group.

**Lemma 6.2.21.** For each sharing group $S \in a_Y(\sigma)$, there exists a set $X \neq \emptyset$ of variables shared between all members of $S$, and not shared by any other sharing group.

**Proof.** First, let us note that from Definition 6.2.19, $S \neq \emptyset$. Let $X$ be the set of variables from $Z = (Y \setminus \text{dom}(\sigma)) \cup \text{range}(\sigma)$, for which $x \in X$ implies $\{y \in Y \mid x \in \text{vars}(y\sigma)\} = S$. $X$ cannot be empty, because otherwise $S$ would have to be empty. Let $S' \neq S$ be another sharing group from $a_Y(\sigma)$, and $X'$ such that $x' \in X'$ implies $\{y \in Y \mid x \in \text{vars}(y\sigma)\} = S'$. If $x' \in X$, then $S = S'$, which is a contradiction. Therefore, $X$ and $X'$ must be disjoint. □

It is easy to see that there exists an infinite number of substitutions $\theta$ such that $a_Y(\theta) = a_Y(\sigma)$ – for example, one for each unique renaming of variables in range($\sigma$). We now consider how expressive the sharing information contained in the abstract substitution $a_Y(\sigma)$ is compared to the (concrete) substitution $\sigma$.

**Definition 6.2.22** (Sharing Ordering). For two variables $y, y' \in Y$ and an abstract substitution $a_Y(\sigma)$, we write $y \sqsubseteq_s y'$ if for all $S \in a_Y(\sigma)$, $y \in S$ implies $y' \in S$.

**Theorem 6.2.23.** For a set of variables of interest $Y$ and a substitution $\sigma$:

1. For arbitrary $y, y' \in Y$, $y, y' \notin X$, if $y \sqsubseteq_s y'$ then $X \rightarrow_s y'$ implies $X \rightarrow_s y$.
2. For arbitrary $y \in Y$, $X \rightarrow_s y$, $y \notin X$, implies $\{x \in Y \mid x \sqsubseteq_s y \wedge x \notin \text{dom}(\sigma)\} \subseteq X$.
3. If range($\sigma$) $\subseteq Y$, then $[\sigma] = \{(X(y), y) \mid y \in Y \cap \text{dom}(\sigma)\}$, where $X(y) = \{x \in Y \mid x \sqsubseteq_s y \wedge x \notin \text{dom}(\sigma)\}$.

**Proof.** (1) Assume $y \sqsubseteq_s y'$ and $X \rightarrow_s y'$, and let $S_1, S_2, \ldots, S_n$ ($n \geq 0$) be all sharing settings containing $y$. From Lemma 6.2.21, there are $n$ non-empty and pairwise disjoint sets $V_1, V_2, \ldots, V_n$ such that $\cup_{i=1}^{n} V_i = \text{vars}(y\sigma)$. Since $y' \in S_i$, for each $i = 1..n$, we have $\text{vars}(y\sigma) = \cup_{i=1}^{n} V_i \subseteq \text{vars}(y'\sigma) \subseteq X$, i.e., $X \rightarrow_s y'$.

(2) Assume $X \rightarrow_s y$, i.e., $\text{vars}(y\sigma) \subseteq X$. For an arbitrary $x \in Y$, $x \notin \text{dom}(\sigma)$, $\text{vars}(x\sigma) = \{x\}$. If $x \sqsubseteq_s y$, then $x$ shares with $y$ in at least one sharing group. Therefore, $x \in \text{vars}(y\sigma)$, i.e., $\{x\} \subseteq \text{vars}(y\sigma)$. By disjointness over all such $x$, we have $\{x \in Y \mid x \sqsubseteq_s y \wedge x \notin \text{dom}(\sigma)\} \subseteq \text{vars}(y\sigma) \subseteq X$.

(3) From Definition 6.2.15, we know that $[\sigma] = \{(X, y) \mid (y \rightarrow t) \in \sigma \wedge X = \text{vars}(t)\}$. Now we need to prove that for each mapping $(y \rightarrow t) \in \sigma$, $\text{vars}(t) = X(y)$.
(3.a) Assume \( x \in \text{vars}(t) \). Therefore, \( x \not\in \text{dom}(\sigma) \), because circular substitutions in \( \sigma \) are forbidden. Since \( \text{range}(\sigma) \subseteq Y \), we have \( x \in Y \). Finally, let \( S \in \alpha_Y(\sigma) \) be a sharing group containing \( x \). By definition, \( S = \{ w \in Y | z \in \text{vars}(w\sigma) \} \) for some \( z \). Because \( x \in S \) this implies \( z \in \text{vars}(x\sigma) = \{x\} \). Therefore, \( z \) must be the same as \( x \). And, therefore, \( y \) must belong to \( S \). In other words \( x \subseteq_s y \). This completes all conditions needed for \( x \in X(y) \).

(3.b) Assume \( x \in X(y) \). Therefore, \( x \in Y \), \( x \subseteq_s y \) and \( x \not\in \text{dom}(\sigma) \). Since \( x \in Y \) and \( x \not\in \text{dom}(\sigma) \), \( x \) has to appear in at least one sharing setting \( S \in \alpha_Y(\sigma) \). We also know \( y \in S \). Following the same argument from (3.a), \( S = \{ w \in Y | x \in \text{vars}(w\sigma) \} \), and from \( y \in S \), we conclude \( x \in \text{vars}(y\sigma) = \text{vars}(t) \).

Theorem 6.2.23 tells us what we can infer about functional dependencies from \( \alpha(\sigma) \), without knowing \( \sigma \) directly, but with the knowledge of \( \alpha(\sigma) \). In the most basic case, from \( y \subseteq_s y' \) we conclude that whatever functionally determines \( y' \) also determines \( y \). We can draw more informative conclusions if we are equipped with what is usually called freeness information [MH91]: whether some \( x \in Y \) belongs to \( \text{dom}(\sigma) \) or not. With freeness information we can (at least partially) reconstruct the left side of “\( \to_{\sigma} \)”. And, by both having the freeness information and extending \( Y \) to include all variables from \( \text{range}(\sigma) \), we can in fact reconstruct the sharing basis \( \sigma \). This tells us that, under these conditions, the abstract substitution \( \alpha_Y(\sigma) \) is as expressive as \( \sigma \) when it comes to functional dependencies.

### 6.3 Sharing Analysis

From the previous sections, we conclude that the abstract substitution \( \alpha_Y(\sigma) \) is enough to derive the functional dependencies we need. The question now is how to infer this abstract substitution. The way we do it is by using a sharing analysis, applied to a Horn clause (logic program) version of the service composition.

In this section, we present the steps necessary for preparing inputs for the sharing analysis, describe briefly the analysis itself, and discuss the interpretation of the analysis outputs. Sharing analysis is an instance of program analysis, and therefore we start by describing the process of transforming a definition of a service composition into a Horn-clause form program as appropriate for the analysis. We use the approach from our previous work on fragmentation analysis [ICH10c].

124
6.3.1 Derivation of Control and Data Dependencies

As a first step towards creating a Horn clause representation of a service composition, we find out a feasible order of activities which is coherent with their dependencies and which allows the composition to finish successfully. How to do this obviously depends on the palette of allowed relationships between activities, with respect to which we opted for a notable freedom by adopting a relatively general abstract composition model. To find such an order we first establish a partial order between workflow activities which respects their dependencies; in doing this we also detect whether there are dependency conflicts that may result in deadlocks. While there is ample work in deadlock detection [BlZ04, AP08], we think that the technique we propose is clean, can be used for arbitrarily complex dependencies between activities, and uses well-proven, existing technology, which simplifies its implementation.

**Definition 6.3.1** (Abstract Composition Model). An abstract composition model is a tuple \( ac(A, C, D) \), where:

- **A** is a finite set of activities;
- **C** is a set of formulas expressing control flow preconditions for each \( a \in A \), of the form \( \text{pre-}a \equiv \phi \), where \( \phi \) is a propositional formula built from the usual logical connectives (\( \lor, \land, \neg, \rightarrow \), and \( \leftrightarrow \)) and the propositional symbols \( \text{done-}a' \) and \( \text{succ-}a' \) for \( a' \in A \), with the following meaning:
  - \( \text{done-}a' \) holds if \( a_j \) has completed;
  - \( \text{succ-}a' \) (when \( \text{done-}a' \) holds) indicates that the outgoing condition from \( a' \) has evaluated to true;
  - The combination \( \text{done-}a' = 0 \) and \( \text{succ-}a' = 1 \) is illegal.
- **D** is a finite set of data items in the compositions (giving the core data dependencies), consisting of tuples of the form \( (x, W, R) \), where \( x \) is a data item, \( W \subseteq A \) is the set of activities that write \( x \), and \( R \subseteq A \) is the set of activities that read \( x \).

This abstract composition model is able to express some of the most frequently used composition workflow patterns, such as AND/OR/XOR splits and joins. However, thanks to the flexibility of the encoding we will use for the sharing analysis, it introduces two significant extensions compared to other workflow models:

- In our approach, the activities inside a workflow can be simple or structured.
The latter include branching (if-then-else) and looping (while and repeat-until) constructs, arbitrarily nested. The body of a branch or a loop is a sub-workflow, and activities in the main workflow cannot directly depend on activities inside that sub-workflow. Of course, any activity in such a sub-workflow is subject to the same treatment as activities in the parent workflow.

- Second, we allow an expressive repertoire of control dependencies between activities besides structured sequencing: AND split-join, OR split-join and XOR split-join. We express dependencies similarly to the link dependencies in BPEL but with fewer restrictions, thereby supporting OR- and XOR-join.

Commonly, the preconditions in \( C \) use “done” symbols, whereas “succ” symbols are added to distinguish mutually exclusive execution paths. We do not specify here how the “succ” indicators are exactly computed. Note that each activity in the workflow is executed at most once; repetitions are represented with the structured looping constructs (yet, within each iteration, an activity in the loop body sub-workflow can also be executed at most once).

Example 6.3.2. Figure 6.6 shows an example. The activities are drawn as nodes, and control dependencies indicated by arrows. Data dependencies are textually shown in a “fraction” or “production rule” format next to the activities: items above the bar are used (read) by the activity, and items below are produced. Note that only items \( y_1 \) and \( y_2 \) are data dependencies; others either come from the input message \( (x_1, x_2) \), or are the result of the workflow \( (z_1, z_2) \). Item \( y_1 \) is produced by \( a_1 \) and used by \( a_3 \), and \( y_2 \) is produced by \( a_2 \) and used by \( a_3 \) and \( a_4 \).

Many workflow patterns can be expressed in terms of such logical link dependencies. For instance, a sequence “\( a_j \) after \( a_i \)” boils down to \( \text{pre-}a_j \equiv \text{done-}a_i \). An AND-join after \( a_i \) and \( a_j \) into \( a_k \) becomes \( \text{pre-}a_k \equiv \text{done-}a_i \land \text{done-}a_j \). An (X)OR-join of \( a_i \) and \( a_j \) into \( a_k \) is
6.3.2 Validity of Control Dependencies

The relative freedom allowed in the use of logic formulas to specify control dependencies comes at the cost of possible anomalies that may lead to deadlocks and other undesirable effects. These need to be detected beforehand, i.e., at design / compile time using some sort of static analysis. Here, we are primarily concerned with deadlock-freeness, i.e., elimination of the cases when activities can never start because they wait on events that cannot happen.

**Example 6.3.3.** Whether a deadlock can happen or not depends on both the topology and the logic of control dependencies. Topological information is in general not enough to determine deadlock freeness, unless there are no loops in the graph. Figure 6.7 shows a simple example where the dependency arrows are drawn from $a_i$ and $a_j$ whenever $\text{pre-} a_j$ depends on $a_i$ finishing. If the connective marked with $\bullet$ in $\text{pre-} a_j$ is $\lor$, there is no deadlock: indeed, there is a possible execution sequence, $a_1 - a_2 - a_3 - a_4$. If, however, $\bullet$ denotes $\land$, there is a deadlock between $a_3$ and $a_4$.

Therefore, in general, checking for deadlock-freeness requires looking at the formulas. We present one approach that relies on simple proofs of propositional formulas. We start by forming a propositional logical theory $\Gamma$ from the workflow by including all preconditions from $C$ and adding axioms of the form $\text{done-} a_i \rightarrow \text{pre-} a_i$ for each $a_i \in A$. These additional axioms simply state that an activity $a_i$ cannot finish if its preconditions were

![Diagram](image-url)

**Figure 6.7:** An example of deadlock dependency on logic formula: $\bullet$ can be either $\land$ or $\lor$. encoded as $\text{pre-} a_k \equiv \text{done-} a_i \lor \text{done-} a_j$. And an XOR split of $a_i$ into $a_j$ and $a_k$ (based on the business outcome of $a_i$) becomes $\text{pre-} a_j \equiv \text{done-} a_i \land \text{succ-} a_i$, $\text{pre-} a_k \equiv \text{done-} a_i \land \neg \text{succ-} a_i$. In terms of execution scheduling, we take the assumption that a workflow activity $a_i$ may start executing as soon as its precondition is met.
not met. On that basis, we introduce the following definition to help us detect deadlocks and infer a task order which respects the data and control dependencies:

**Definition 6.3.4 (Dependency Matrix).** For a given composition model \( ac(A, C, D) \), the dependency matrix \( \Delta \) is a square Boolean matrix such that its element \( \delta_{ij} \), corresponding to \( a_i, a_j \in A \), is defined as:

\[
\delta_{ij} = \begin{cases} 
1, & \text{if } \Gamma, \text{pre-}a_i \vdash \text{done-}a_j \\
0, & \text{otherwise}
\end{cases}
\]

For every data dependency \( \langle x, R, W \rangle \in D \), and for each \( a \in R \), we wish to ensure that \( a \) cannot start unless at least one of \( b \in W \) has completed, since otherwise the data item \( x \) would not be ready. Expressed with a logic formula, that condition is \( \text{pre-}a \rightarrow \bigvee_{b \in W \setminus \{a\}} \text{done-}b \).

The computation of \( \Delta \) involves proving propositional formulas, which is best achieved using some form of SAT solver. Such solvers are nowadays very mature and widely available either as libraries or standalone programs. It follows from the definition that \( \delta_{ij} = 1 \) if and only if the end of \( a_j \) is a necessary condition for the start of \( a_i \). It can be easily shown that \( \Delta \) is a transitive closure of \( C \), and that is important for the ordering of activities in a logic program representation. However, the most important property can be summarized as follows.

**Proposition 6.3.5 (Freedom from deadlocks).** The given workflow \( ac(A, C, D) \) with dependency matrix \( \Delta \) is deadlock-free if and only if \( \forall a_i \in A, \delta_{ii} = 0 \).

**Proposition 6.3.6 (Partial ordering).** In a deadlock-free workflow \( ac(A, C, D) \), dependency matrix \( \Delta \) induces a strict partial ordering \( \prec \) such that for any two distinct \( a_i, a_j \in A \), \( a_j \prec a_i \) iff \( \delta_{ij} = 1 \).

### 6.3.3 Generating Horn Clause Representations

The Horn clause representation of a service composition introduces in this section is essentially a logic program with semantics corresponding to that of a subset of standard Prolog with operational semantics based on SLD resolution [Llo87]. The goal of that program is to represent the computation of all the substitutions that express functional...
dependencies and data attribute sharing of the composition, as described in Section 6.2. In other words, the purpose of such program is not to operationally mimic the scheduling of workflow activities, but to express and convey relevant data and control dependency information to the sharing analysis stage.

Based on the strict partial ordering \( \prec \) induced by the dependency matrix \( \Delta \), in the deadlock-free case it is always possible to totally order the activities so that \( \prec \) is respected. The choice of a particular order has no impact on our analysis, because we assume that the control dependencies, from which the partial ordering derives, include the data dependencies. From this point on we will assume that activities are renumbered to follow the chosen total order. The workflow can then be translated into a Horn clause of the form:

\[
w(V) \leftarrow T(a_1), T(a_2), \ldots, T(a_n)
\]

(6.3)

where \( V \) is the set of all logic variables used in the clause, and \( T(a_i) \) stands for the translation of activity \( a_i \) into a logic (Prolog) goal. For simple activities (such as a service invocation or an assignment), \( T(a_i) \) gives a sequence of equations (explained below) that relate its inputs and outputs. The complex activities, such as branches or loops, along with their constituent parts (e.g., loop body and then/else parts) are recursively translated into separate clauses following the scheme (6.3) above, and \( T(a_i) \) is a call to such generated clause.

Logic variables in \( V \) are used to represent input and output messages, variables comprising the composition state, internal state of component services (as in Table 6.1), and the data sets read by individual activities. For each activity \( a_i \in A \) we designate a set \( X_i \) of logic variables that represent data items read by \( a_i \), a set \( Y_i \) of logic variables that stand for data items produced by \( a_i \), as well as the sets \( U_i \) and \( U'_i \) that represent the state of a component service to which \( a \) belongs before and after its execution. We designate a variable set \( A_i \subseteq X_i U_i \) that represents the total inflow of data into \( a_i \). The task of the translation is to connect \( X_i, Y_i, A_i, U_i, \) and \( U'_i \) correctly.

The translation scheme is mechanical. We first present the scheme for the simple activities. Using Prolog notation, where variable names start in upper case, we use \( A_i \) to denote all inputs to activity \( a_i \), and \( Y_i \) to denote its output. We use \( X_i a \) to denote the part of \( A_i \) used in computing \( Y_i \). It is always safe to assume that all data an activity accesses is also used in producing its output, i.e., \( A_i = X_i a \), but if we can obtain sufficient guarantees
to safely exclude some item in $A_i$ from $X_{ia}$, we can draw more precise conclusions about the functional dependencies, as discussed in Section 6.2.1. E.g., if $a_i$ internally contains a loop, data used in evaluating the loop condition (included in $A_i$) may not be used in computation of the loop output $Y_i$. Next, if $a_i$ is an invocation of a stateful component service, we use $W_{i_0}$ to symbolically denote its previous internal state, and $W_i$ its state after executing $a_i$. The part of $A_i$ used for updating the internal state is denoted by $X_{ib}$. Again, a safe assumption is $A_i=X_{ib}$, while more precision can be obtained by safely restricting $X_{ib}$ when possible. We use $X_i=[X_{i1}, X_{i2}, \ldots, X_{im}] \ (m \geq 0)$ to denote the union of $X_{ia}$ and $X_{ib}$. For analyzing data attributes that may be injected by an external activity, we optionally introduce $M_i$ to represent those attributes injected into the output $Y_i$, and $N_i$ for those injected into the internal state $W_i$.

To round up our translation scheme for simple activities, we need to take into account that they can be placed inside a loop, and in that case we are interested in the most general sharing that includes all potential loop iterations. For that reason, we include in $A_i$ and $Y_i$ their previously collected values $A_{i_0}$ and $Y_{i_0}$, respectively (which are ground on first iteration and outside the loops). Thus, $T(a_i)$ can be put in the shape of a sequence:

$$
A_i = [X_i, W_{i_0} | A_{i_0}],
Y_i = [X_{ia}, W_{i_0}, M_i | Y_{i_0}],
W_i = [X_{ib}, N_i | W_{i_0}]
$$

where the Prolog notation $[A,B|C]$ means a list that starts with $A$ and $B$, and continues with the elements of list $C$. Note that for convenience we are here using lists, since the shape of the data structure is not significant for abstract substitutions presented in Section 6.2.4 on which the sharing analysis is based. Each of the equations of the form $X=t$ at the point of execution in a Prolog program where the currently computed substitution is $\sigma$, provided that $X \notin \text{dom}(\sigma)$ (which is ensured by construction in our scheme), extends $\sigma$ with a new mapping ($X \rightarrow t$). The above translation scheme is rather generic and can be simplified in several ways, depending on the activity:

- If $a_i$ is stateless, we remove the third equation from the scheme and replace $W_{i_0}$ with $[]$ (the empty list).
- If $a_i$ has a state, but does not update it, we replace $X_{ib}$ and $N_i$ in the third equation with $[]$. With respect to the abstract substitutions, $W_{i_0}$ and $W_i= [[]] | W_{i_0}$ are
Figure 6.8: Fragments of translation to Horn clause form of the composition from Fig. 6.1.

indistinguishable.

• If \( a_i \) updates its state without first reading it (e.g., by issuing an UPDATE SQL command), we replace \( Wi_0 \) in the first equation with [].

• If \( a_i \) is not inside a loop, we replace \( Ai_0 \) and \( Yi_0 \) with [].

• If we are not interested in data attributes, we replace \( Mi \) and \( Ni \) with [].

• When \( Mi \) or \( Ni \) need to be represented, but their content is not important, we can replace them with the underscore symbol “_” that represents an anonymous variable in Prolog.

• The scheme is easily extended to the activities that have several pieces of state and/or several outputs.

Example 6.3.7. Figure 6.8 shows several fragments of the Horn clause representation of the composition from Figure 6.1, in Prolog notation (comments start with “%”, and “←” is written as “:-”). Lines 16-22 show a clause for predicate \( a26c \) that models data dependencies from \( a26 \) to the finish, i.e., for \( a26 \) and \( a27 \). We use the same labels for data
items and service state from Table 6.1, but in uppercase. Lines 18-19 model activity a_{26} (Payment processing). Line 18 indicates that the activity reads the transfer order p, and line 19 indicates that the state of the seller’s account w_{4} now depends on p and its earlier state (unknown to a_{26} and thus represented with the underscore). Note that a_{26} does not have a direct output, so the second equation is missing. However, w_{4} is accessed by a_{27} (Shipment, modeled by lines 21-22), which checks that the payment has settled, and serves as the input for the shipment notice n. Since a_{27} does not modify state w_{4}, the third equation is missing. All named variables from lines 17-22 are also found in the list of arguments to a_{26}c in line 16, to propagate substitutions.

**Example 6.3.8.** Lines 1-14 in Figure 6.8 show the clauses for predicates a_{16}c and a_{25}c that model data dependencies from a_{16} and a_{25} to the finish, respectively. a_{16}c models the data dependencies of a_{16} and a_{17}, and calls a_{26}c at its end. Likewise, a_{25}c models the dependencies of a_{25} and calls a_{26}c.

For the complex constructs, such as loops and XOR-splits, the translation generates additional Prolog clauses depending on the type of construct. These are illustrated in the examples that follow.

**Example 6.3.9.** Lines 23-36 in Figure 6.8 show the translation of the loop construct a_{5} as predicate a_{5}_. The first clause (lines 23-29) models the case of exit from the loop where the initial values from a previous iteration (with suffix “_0”) are propagated to the exit. The second clause (lines 30-36) models a loop iteration. Its body consists of the translation of the loop body: activity a_{6} (Search catalog) in lines 32-33, and a_{7} which is a XOR-split implemented in line 34 as a call to a specially generated predicate a_{7}c (see the next example). The variables in the iteration wear suffix “_0” if they come from a previous iteration, and suffix “_1” if they result from the current one. The final line 36 recursively calls a_{5} and passes to it the “_1” versions as the new initial ones.

**Example 6.3.10.** Lines 38-51 in Figure 6.8 show the translation for the XOR-split (if-then-else) construct a_{7}. Each of the two branches is translated as a separate clause of a_{7}c. Lines 40-42 in the first clause model activity a_{8} (Add to cart), which uses the previous values of q and w_{2} (line 40, see Table 6.1) to update q and w_{2}, respectively (lines 41-42). Line 47 in the second clause shows the translation for a_{9} (Skip option), which does not alter the state, nor produces any outputs. Since both clauses need to have the same
interface to the calling code (in line 34), they enumerate the union of all variables from translations of both a8 and a9. The variables that are not used in a clause are propagated from the previous values with the suffix “_0”. This is the case with a9 in the first clause (line 44), and with a8, Q and W2 in the second clause (lines 49-51).

6.3.4 Sharing Analysis Proper

In this subsection we give a brief overview of the actual sharing analysis employed in our approach, whose background has been presented in Section 2.4.

We use a combined, abstract interpretation-based sharing, freeness, and groundness analysis for logic programs [MH91], which computes abstract substitutions and freeness information as described in Section 6.2.4. For a deterministic Horn clause program (i.e., with a single possible execution path) without loops, the sharing analysis computes abstract substitution \( \Theta = a_Y(\sigma) \), where \( \sigma \) is the substitution computed by that program, and \( Y \) are argument variables to a predicate that is called (e.g., arguments of a26c in Figure 6.8). For programs with non-determinism (i.e., where several control flows are possible, as in the case of the two clauses of a7c), the sharing analysis computes \( \Theta = \bigcup_{i=1}^{n} a_Y(\sigma_i) \), where \( \sigma_1, \ldots, \sigma_n \) are the substitutions computed by the alternatives. And, for the cases of looping (such as a5_), the sharing analysis computes a fixed point \( \Theta \) that is either equal or a superset of any \( a_Y(\sigma) \) where \( \sigma \) can be computed by the loop. Therefore, the result \( \Theta \) is a safe approximation, in the sense that it includes all possible sharing groups. The sharing analysis is combined with a freeness analysis, which infers which variables are unbound, i.e., have not been substituted with a non-variable term. The sharing analysis also infers groundness information, determining with variables are bound to terms that do not contain any variables (note that those variables can be excluded from any sharing group in which they may appear). Some logic program analysis tools, like CiaoPP [HBC10], have been developed which give users the possibility of running different analysis algorithms on input programs. We build on one of these analysis available in CiaoPP: shfr [MH91, BGH99].

When analyzing attributes, the inputs to the Horn clause program that represent incoming messages and internal activity attributes are normally initialized to contain a configuration of variables that represents conceptually the content in terms of data attributes of such messages. For instance, input D in predicate a25c from Figure 6.8
(lines 10-14) corresponds to one of the cases of identification documents from Figure 6.5, which are characterized by the attributes $m_1, \ldots, m_5$ from Figure 6.4. If $D$ represents a passport, we can add $D=[M_1, M_2, M_3]$ before calling a25c. This is not necessary if $D$ is the only input that uses these attributes.

6.3.5 Interpretation of Sharing Results

As mentioned above, the result of sharing analysis is an abstract substitution $\Theta$ such that $\alpha_Y(\theta) \subseteq \Theta$ for all concrete substitutions $\theta$ that can be computed by the Horn clause program. In other words, no potential sharing group is left from $\Theta$. Therefore we can construct a relation $\leq_s$ from $\Theta$ in the same way as $\sqsubseteq_s$ is constructed from $\alpha_Y(\theta)$ in Definition 6.2.22. However, it can be easily verified that such $\leq_s$ must be a subset of $\sqsubseteq_s$ for each $\theta$ for which $\alpha_Y(\theta) \subseteq \Theta$.

**Proposition 6.3.11.** For an abstract substitution $\Theta$ and its relation $(\leq_s) \subseteq Y^2$ between variables of interest from $Y$, and for any concrete substitution $\theta$ such that $\alpha(\theta) \subseteq \Theta$ and the relation $(\sqsubseteq_s) \subseteq Y^2$ induced by it, it holds that $(\leq_s) \subseteq (\sqsubseteq_s)$.

**Proof.** First, note that $x \sqsubseteq_s x$ and $x \leq_s x$ hold trivially for each $x \in Y$. Next, suppose that for arbitrary distinct $x, y \in Y$ we have $x \leq_s y$, but not $x \sqsubseteq_s y$. That is only possible if $\alpha_Y(\theta)$ contains some sharing group $S$ such that $x \in S$, but $y \not\in S$. But since $\alpha_Y(\theta) \subseteq \Theta$, then also $S \in \Theta$, which conflicts with the assumption $x \leq_s y$. Therefore, no such $S$ can exist, i.e., it follows that $x \sqsubseteq_s y$ must hold.

**Example 6.3.12.** Suppose that $Y = \{x, y\}$ and two possible concrete substitutions are $\theta_1 = \{x \leftarrow f(u), y \leftarrow g(u, v)\}$ and $\theta_2 = \{x \leftarrow g(u, v), y \leftarrow f(v)\}$. Then, $\alpha_Y(\theta_1) = \{xy, y\}$, i.e., $(\sqsubseteq_s)_1 = \{(x, y)\}$, and $\alpha_Y(\theta_2) = \{x, xy\}$, i.e., $(\sqsubseteq_s)_2 = \{(y, x)\}$. However, for $\Theta = \alpha_Y(\theta_1) \cup \alpha_Y(\theta_2) = \{x, xy, y\}$, we have $(\leq_s) = \emptyset$.

This means that $\leq_s$ derived from $\Theta$ can be used as a lower approximation for $\sqsubseteq_s$ induced by any concrete $\theta$. A natural way to compute the upper bound for $\sqsubseteq_s$ is given using the following definition:

**Definition 6.3.13** (Maximal Sharing Ordering). Let $\Theta$ be an abstract substitution, and $Y$ a set of variables of interest. For arbitrary $x, y \in Y$, we say that $x \geq_s y$ if either no $S \in \Theta$ contains $x$, or there exists some $S \in \Theta$ such that $x \in S$ and $y \in S$. 

134
**Proposition 6.3.14.** For an abstract substitution \( \Theta \) and its relation \((\lessgeq_s) \subseteq Y^2\) between variables of interest from \( Y \), and for any concrete substitution \( \theta \) such that \( a_Y(\theta) \subseteq \Theta \) and all variables from \( Y \) appear in \( a(\theta) \), with the relation \((\lessgeq_s) \subseteq Y^2\) induced by it, \((\lessgeq_s) \subseteq (\lessgeq_s)\).

**Proof.** Suppose that for arbitrary \( x, y \in Y, x \lessgeq_s y \). Since all variables from \( Y \) must appear in at least one sharing group in \( a_Y(\theta) \), and \( x \lessgeq_s y \), then there has to exist \( S \in a_Y(\theta) \) such that \( x \in S \) and \( y \in S \). From \( a_Y(\theta) \subseteq \Theta \), we conclude that \( S \in \Theta \), and from Definition 6.3.13, we obtain \( x \lessgeq_s y \).

To summarize, \( x \lessgeq_s y \) implies that all components of \( x \) are necessarily components of \( y \), while \( x \lessgeq_s y \) implies that all components \( x \) may possibly be components of \( y \). Both \( \lessgeq_s \) and \( \lessgeq_s \) are directly obtained from the result of sharing analysis \( \Theta \).

### 6.3.6 Complexity And Precision of the Sharing Analysis

The \texttt{SHFR} sharing and freeness analysis for logic programs is known to produce the most precise sharing results (i.e., the least over-approximation of \( \Theta \)), but its computational cost may grow, in the worst case, exponentially with the number of variables. Other, more efficient, but less precise sharing analysis techniques have been proposed, such as the clique sharing analysis [NBH06], or the pair-sharing analysis [LS02]. An abstract substitution \( \Theta' \) obtained from such a less precise sharing analysis technique is generally a superset of \( \Theta \) obtained from \texttt{SHFR}. It can be easily verified that in that case:

\[
(\lessgeq'_s) \subseteq (\lessgeq_s) \subseteq (\lessgeq_s) \subseteq (\lessgeq'_s)
\]

where \( \lessgeq'_s \) and \( \lessgeq'_s \) correspond to \( \Theta' \).

On the other hand, one way to increase precision of the approximation with \( \lessgeq_s \) and \( \lessgeq_s \) is to remove some of the alternative clauses from the Horn clause program whose effect on \( \Theta \) is to inflate it due to the union of abstract substitutions for each alternative. That can be done, for instance, when a (partial) trail of the execution of the composition is known.

**Example 6.3.15.** Let us take a look at the clauses of predicate \( a7c \) from Figure 6.8, lines (38-51). The abstract substitution from the first clause \( \Theta_1 \) is (in Prolog syntax)
[\[A_8, A_8\], \[A_9, A_9\], \[Q_0, A_8, Q, W_2\], \[W_2, A_8, Q, W_2\]], while \(\Theta_2\) from the second clause is \([A_8, A_8], [A_9, A_9], [Q_0, Q], [W_2, W_2]\). Their union \(\Theta = \Theta_1 \cup \Theta_2\) is more general, but less precise than both \(\Theta_1\) and \(\Theta_2\). Learning which branch was taken from the traces, or predicting which branch will necessarily be taken eliminates either \(\Theta_1\) or \(\Theta_2\) and gives a more precise result.

### 6.4 Examples of Application

In this section we show how the framework for functional dependencies from Section 6.2 and the analysis method from Section 6.3 can be used to address the problems mentioned in Section 6.1 as motivation for this work.

The steps described in Section 6.3 are currently almost completely automated and we have developed prototype tools centered around the CiaoPP program analysis and transformation system [HBC+10] that accept a description of a service composition in an abstract composition form (introduced in Section 6.3.1), prepare the Horn clause representation of the composition which is subjected to the sharing analysis, and extract the sharing results from the analysis outputs. We are working on a set of pre-processing tools that accept composition definitions written in (fragments of) the widely accepted composition languages, such as BPMN, WS-CDL [Wor05], and BPEL [Jea07a]. The prototype tools also manages export of the sharing results into a form suitable for FCA-based concept lattice visualization using external tools.

#### 6.4.1 Parallelization

The general control structure of a service composition can often be adapted to provide more flexibility while not violating the control and data dependencies. One example is parallelization, which allows composition activities to start as soon as their control and data dependencies allow. Automatic parallelization can be performed by interpreting the results of the sharing analysis over variables that represent data items and or activities (Section 6.3.5). Our criterion for parallelization will be based on the following: an activity can start as soon as all the necessary data (including the internal state of the component services) is available. Note that the necessary control and data dependencies are already encoded in the ordering of activities in the composition in the translation to the Horn
Let us look at a class of use cases of the composition in Figure 6.1 where the user always logs in (instead of starting an anonymous session), and where payments are always made by credit card. In that class of use cases, from the composition model in Figure 6.1 we prune branches with activities $a_{3}$, $a_{13}$, $a_{18}$, and $a_{19}$. The sharing analysis of the pruned composition returns an abstract substitution $\Theta$ with five sharing groups $S_{1}, S_{2}, S_{3}, S_{4}$, and $S_{5}$, over the set $Y$ of variables of interest which represent activity inputs. The membership of the variables from $Y$ in the sharing groups is shown schematically in Table 6.2.

The necessary condition for parallelization of two activities is that both draw all of their inputs from the same set of previously computed data items and component states. We use the relation $\leq_{s}$ derived from $\Theta$, because we are interested only in functional dependencies that hold under any concrete substitution. For two variables $x, y \in Y$, $x \leq_{s} y$ guarantees that all data needed to compute $x$ is included in the data needed to compute $y$. Or, equivalently, if there is not enough data to compute $x$, then $y$ cannot be computed either. If we recall the notion of FCA concept lattices from Section 6.2.2, with $Y$ as the set of objects, $\Theta$ as the set of attributes, and the membership of variables from $Y$ in the sharing groups from $\Theta$, then we can easily verify that for each $x \in Y$, $\{x\}'' = \{y \in Y \mid x \leq_{s} y\}$. Therefore, we can represent the conceptual hierarchy of activities, induced by the
guaranteed sharing ordering $\leq_s$, in the form of a concept lattice, such as in Figure 6.9 for our parallelization example.

The relationship between this conceptual hierarchy and the functional relationships modeled with "→" is the following: if the inputs of a conceptually higher set of activities $A_1$ depend on some set of inputs $X$ and component states $U$ (i.e., $XU \rightarrow A_1$), then for a conceptually “lesser” set of activities $A_2$, we have $XUV \rightarrow A_2$, where $V$ is some non-empty set of additional inputs or updated states of component services. This property holds even in a general case, when the lattice does not have a linear form as in Figure 6.9. Within a group of activities at the same level in this conceptual hierarchy, control dependencies can be freely rearranged as long as they do not clash with data dependencies: if an activity $a$ receives the output of $a'$, then $a$ must come after $a'$, but otherwise $a$ and $a'$ can be parallelized.

In Figure 6.9, activity $a_0$ at node $\top$ waits for the input $u$, without needing any previous information. Activity $a_2$ receives $u$ and produces $w_4$ (the next concept node after $\top$). In the second concept node from $\top$, activities $a_4$, $a_{16}$, and $a_{12}$ can all be executed in parallel, while for the fourth node, $a_5$, $a_{10}$, and $a_{14}$ must be executed sequentially in that order because of the data dependencies shown in Table 6.1. Note that in the latter case, the sequence $a_5$-$a_{10}$-$a_{14}$ represents a sub-workflow for an iterative browse-select-iterate-exit process typical of e-commerce Web portals, but can in principle be replaced by another set of activities that implement a different procedure for creating an invoice, e.g., based on a list of items supplied by the buyer, without the activities that come before or later observing any difference.

When evaluating usability of the sharing approach for automatic parallelization of activities, we need to take into account the complexity and accuracy of the sharing analysis in one hand, and the quality of information about data dependencies in the workflow in the other. Using our prototype tools, translation of a service workflow encoded as an abstract composition model (Section 6.3.1) is linear in the number of activities and data items in the composition, and does not present a major computational overhead. Also, interpretation of the results from the sharing analyzer is straightforward, since the structure of the resulting abstract substitution is directly transposed into the shape used by Table 6.2 (unless there is a need for visualization which involves more complex lattice construction). The greatest part of the computational complexity when applying
the proposed approach is consumed by the sharing analysis proper using CiaoPP. On a low-end personal computer running Mac OS X v.1.7.5, this stage consumes approximately between 1100 ms and 1800 ms, depending on the run.

When it comes to the quality of information on data dependencies in the composition, it should be noted that we may not normally have full information about how each invocation of a component service affects its internal state, which was in our motivation example explicitly represented in the lower part of Table 6.1. In case of several operations on the same service (unless we know it is stateless), to ensure correctness we need to make a safe assumption that each operation may modify its internal state. That may introduce additional data dependencies which tend to reduce the level of parallelization, by loosing opportunities to parallelize activities when that is not safe.

The left-hand side of Figure 6.10 shows a simple sequence of six service invocations, which refer to operations on three distinct services (1), two services (2), and a single service. If the operation $s_1$ updates the state of Service A, this in case (1) creates an additional data dependency between $s_1$ and $s_2$, in case (2) between $s_1$ and $s_2$ and between $s_1$ and $s_3$, in case (3) between $s_1$ and each $s_i$, $i = 2..6$, etc. The graph on the right-hand side of Figure 6.10 shows the proportion of possible parallelization opportunities used, depending on how good are the assumptions about the impact of the invocations on the state of the respective services. The values in the graph are averages across

Figure 6.10: Effect on the information on service statefulness on the level of parallelization.
all possible combinations of state impact for the six activities and the corresponding safe assumptions, crossed with all possible combinations of forward data dependencies between the operations on different services.

The rightmost point on the graph unsurprisingly shows that the completely correct assumptions lead to utilization of all opportunities for parallelization in the sequence in all three cases. However, as the quality of the assumptions decreases, by assuming more impact on state than necessary, more and more parallelization opportunities are lost, depending on how many actual services are involved. In case (3), where all operations belong to the same service, assuming that they all affect its state (when that is not the case) leads to a loss of about 83% of all parallelization opportunities (counting parallelization of each $s_1$-$s_6$ separately), while in case (1) the loss is smaller (about 50% on average) because the additional dependencies caused by the wrong assumptions play a smaller role compared to all other potential data dependencies.

### 6.4.2 Fragmentation

It is often of interest to take a service composition that is designed and represented as an orchestration, i.e., with a centralized control flow, and to break it into parts (called fragments) that can be executed in a distributed manner, possibly on servers that belong to different organizational domains. That process is called fragmentation, and it is a form of adaptation that can be applied at design time or at run time [ML09]. We can use the sharing approach to support fragmentation by assigning activities to organizational-domain-based fragments based on the content of data they handle. This time we model data attributes that describe the content of data, in the sense discussed in Section 6.2.2. We extend on our earlier work on automatic attribute inference and fragment identification based on sharing [ICH10c, ICH11a].

To illustrate the approach, we look at the part of the service composition from Figure 6.1 which starts with the activity $a_{15}$ (an XOR-split), and look only at the branches that correspond to credit card payment (activities $a_{16}, a_{17}$) and bank transfer ($a_{18}$). We modify slightly the generation of Horn clauses to expose the component state $w_4$ and the credit card information $c$ as input variables of interest for the analysis, along with the invoice $i$ and the user info $e$ that are inputs to that part of the composition from Figure 6.1.
Data object | Sharing groups from $\Theta$
---|---
$e$ | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
c | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
i | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
w4 | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
a16 | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
a17 | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
a18 | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
a26 | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
a27 | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
n | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
p | ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓

Table 6.3: The sharing result $\Theta$ for a case of fragmentation based on data sharing.

(a) The summary of $\succeq_s$ for attribute carriers.

For $x \in Y$, we write $(x \succeq_s y)$ to denote the set of all $y \in Y$ such that $x \succeq_s y$. Figure 6.11(a) shows the summary of $\succeq_s$ for variables that carry data attribute. Part (b) of the figure shows that summary turned into an attribute inheritance relation where the “✓” marks

(b) The attribute inheritance relation.

Figure 6.11: Attribute inheritance arising from by $\Theta$ in Table 6.3.

The result of the sharing analysis in this case is an abstract substitution $\Theta$ with twelve sharing groups, shown schematically in table 6.3. A horizontal line visually separates the variables that are the source of data attributes ($e$, $c$, $i$ and $w_4$) from others whose attributes we wish to infer. In this case, $\Theta$ is a union of abstract substitutions arising from two alternative paths of execution ($a_{16}$-$a_{17}$-$a_{26}$-$a_{27}$ and $a_{18}$-$a_{26}$-$a_{27}$), and includes sharing groups from both. We wish to assign data attributes in a safe way, i.e., without leaving any potential attribute out. Therefore, this time we use the relation $\succeq_s$, which is an upper approximation of the actual sharing ordering $\sqsubseteq_s$.

For $x \in Y$, we write $(x \succeq_s y)$ to denote the set of all $y \in Y$ such that $x \succeq_s y$. Figure 6.11(a) shows the summary of $\succeq_s$ for variables that carry data attribute. Part (b) of the figure shows that summary turned into an attribute inheritance relation where the “✓” marks...
Figure 6.12: A sample fragmentation scheme between four domains.

come from part (a), and the “×” mark come from the fact that \( c \) is an output of \( a_{16} \) in Table 6.1 (note that a variable \( a_i \) in Table 6.3 and Fig. 6.11(a) represents only the inputs to the activity with the same name). The attribute inheritance relation tells us that the activities can be grouped into four groups:

- \( a_{16} \) handles only the user profile \((e)\) and his or her credit card details \((c)\), without needing to know about the goods purchased \((i)\) or the state of the seller’s bank account \((w_4)\). This job can be outsourced to any freely or commercially available B2C online payment portal.
- \( a_{18} \) handles the information from the invoice \( i \) to collect the buyer’s bank account information and produce the bank transfer order \( p \). This can, in principle, be done by the buyer.
- \( a_{17} \) and \( a_{26} \) handle the information on the buyer’s credit card \((c)\) and the invoice \((i)\) to issue and process the payment order \((p)\), and to realize the order \((a_{26})\). This job is best suited for the buyer’s bank. Note that \( a_{26} \) does not always see \( c \), but only when the \( a_{16}-a_{17} \) branch is taken. However, our goal is to assign attributes for the most general case.
- \( a_{27} \) handles the buyer’s credit card details \((c)\), the invoice \((i)\), and the state of the seller’s account \((w_4)\) to produce the shipment notice \((n)\). This job is best handled by the seller. Note again that the credit card information is only potentially present.

Figure 6.12 shows a sample fragmentation scheme where part of the original com-
position is remodeled as a choreography that involves four communicating fragments placed in domain/role swimlanes: Seller, Buyer, Buyer’s Bank, and B2C, following the above classification of activities based on inheritance of data attributes.

Note that in principle the fragmentation based on data attributes can be combined with the parallelization approach from the preceding subsection, to obtain finer-grained fragments that reflect both data attribute inheritance and functional dependencies. E.g., using the lattice from Figure 6.9, we can further subdivide the fragment in the Buyer’s bank swimlane into two, \(a_{17}\) and \(a_{26}\). A basis of fine-grained composition fragments can be useful for automatic, on-demand merging of fragments based on the desired functionality or Quality of Service (QoS) constraints [ZBC10].

Using our prototype implementation, in our previous work [ICH11a] we have evaluated the sharing-based approach to fragmentation using a E-Health case study collected

---

Figure 6.13: A drug prescription workflow in BPMN from E-Health scenario.

Figure 6.14: BPMN diagram of the composite activity \(a_4\) from Fig. 6.13.
Figure 6.15: The resulting context for the drug prescription workflow analysis.

within the S-Cube project [DiN09]. Figure 6.13 depicts a drug prescription workflow in BPMN notation, annotated with data dependencies. The process is initiated by the arrival of a patient with an appropriate identification (labeled as $x$ in the figure). Next, two parallel activities ($a_1$ and $a_2$) 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 $a_1$ informs about the stability of the health of the patient. Depending on it, either the last prescription is continued (activity $a_3$) or new medication is selected (activity $a_4$). Finally, the treatment of the patient is logged (activity $a_5$). Activity $a_4$ is in itself a service composition, shown in Figure 6.14. It contains a loop that iteratively refines the prescription based on medical tests.

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. Registry and Archive cannot look into the patient’s symptoms, tests, or insurance coverage data. Medical examiners can at most see the symptoms and tests, without reference to the coverage information. Medication providers can only take care of symptoms and coverage, without reference to the medical tests. All tasks that cannot be assigned to the partners according to these rules are kept by the central Health organization.

Figure 6.15 shows the sharing analysis results for the drug prescription workflow. The upper part of the table (above the line) shows inputs, where the columns represent data attributes, $x$ stands for PatientID in Figure 6.13, while $d$ and $e$ represent the contents of the databases used by activities $a_1$ and $a_2$ to produce (on the basis of $x$) data items $y$ (Medical history) and $z$ (Medication record), respectively. The lower part of the
Table shows the inferred attributes of all intermediate data items and activities from the main workflow and the sub-workflow for activity $a_4$. Figure 6.16 shows the assignment of activities to fragments that correspond to the health organization and its partners.

The table in the upper part of Figure 6.15 shows an alternative assignment of the attributes to the inputs to the drug description workflow, where insurance coverage information also appears in the medical history database, and forms a part of the medical history record for the patient. In this case, the sharing analysis results give rise to a
reassignment of activities to the swimlanes as shown in the lower part of Figure 6.17. In this case, the health organization keeps to itself the activities that were delegated to Medical Examiners in Figure 6.16, since it is not safe entrusting such external entity with the insurance coverage details.

6.4.3 Constraining Component Search and Validation

As mentioned in Section 6.2.2, the compliance of a message with a structural data description (a semantic data type accepted by the service) in itself does not guarantee that an invoked service will be able to perform its task successfully. The reason for that is that XML messages may have many optional or alternative parts that may be present or absent in a structurally compliant message, yet whose presence or absence may cause the service to fail. Or, the message may contain references (foreign keys) to non-existent or wrong entities. In our approach, we use data attributes to represent the content, rather than the structure of a message, which, on top of the structural matching, may help us in reasoning about whether some service implementation is suitable for the given task.

Let us look again at the example composition from Figure 6.1, where activity $a_{25}$ performs loan approval based on the documents collected in $a_{19}$ and $a_{20}$. Let us suppose that, as it usually happens in reality, these documents are not passed repeatedly to $a_{25}$, but are rather stored in a “loan request file” for the buyer (a “logical” file, not a file in the O.S. sense), represented with $w_3$ in Table 6.1, which is created by $a_{19}$, updated by $a_{20}$, and consumed by $a_{25}$. For the sake of argument here, we shall also assume that the seller’s invoice $i$ is inserted into $w_3$, so that $a_{25}$ accesses $w_3$ as an integrated super-document with all pieces of information placed inside.

Figure 6.18 presents a hypothetical concept lattice of eight credit approval candidate services, for “small,” “medium,” and “big” consumer loans. The small loans are up to 3000 monetary units, the medium ones are between 3000 and 10 000, and the big loans are 10 000 monetary units or more. The candidate services are characterized by means of the required data attributes of the loan request file $w_3$ at their inputs. Small loans can be approved in cash by SMALLCASH. It requires name, pin, address, and cont (besides $v<3000$), while all other loan approval services pay directly to the seller, and therefore require SACC. All attributes required by a candidate can be collected from the lattice diagram by following all lines that go from it to the top. E.g., form med2, that is the set
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Buyer’s name given.</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>Buyer’s address given.</td>
</tr>
<tr>
<td>SSN</td>
<td>Buyer’s SSN given.</td>
</tr>
<tr>
<td>PIN</td>
<td>Buyer’s PIN given.</td>
</tr>
<tr>
<td>SACC</td>
<td>Seller’s account number given.</td>
</tr>
<tr>
<td>V&lt;3000</td>
<td>Value less than $3000.</td>
</tr>
<tr>
<td>V≥3000</td>
<td>Value $3000 or more.</td>
</tr>
<tr>
<td>V≥10000</td>
<td>Value $10 000 or more.</td>
</tr>
<tr>
<td>DEP</td>
<td>Security deposit secured.</td>
</tr>
<tr>
<td>TAX</td>
<td>Tax declaration presented.</td>
</tr>
<tr>
<td>CREC</td>
<td>Credit record presented.</td>
</tr>
<tr>
<td>CONT</td>
<td>Work contract presented.</td>
</tr>
</tbody>
</table>

Figure 6.18: A hypothetical concept lattice of credit approval procedures.

{ssn, tax, sacc, name, dep, v ≥ 3000}.

Let us now suppose that, as a result of a previous data attribute analysis, we conclude that the identity document $d$ supplied by $a_{19}$ at the start has some set of attributes $D \subseteq \{\text{name, address, ssn, pin}\}$, and that the set $I$ of attributes for the invoice $i$ is one of $\{\text{sacc, v < 3000}\}$, $\{\text{sacc, v ≥ 3000}\}$, or $\{\text{sacc, v ≥ 3000, v ≥ 10000}\}$. The question we ask is what loan service candidate we need to chose to ensure that enough information is provided to the service to perform its function.

Figure 6.19 shows an adaptive search for service candidates for $D = \{\text{name, address, ssn}\}$, depending on a combination of $a_{22}$, $a_{23}$, and $a_{24}$ executed within $a_{20}$. Each of these activities contributes some additional information, represented in the lower table in the figure. In this case, $a_{22}$ must always be executed, and for a “big” loan, also at least one of $a_{23}$ or $a_{24}$. If $D$ was computed based on the minimal sharing (relation $\preceq_s$), we obtain candidates that certainly comply, and if it was computed based on $\succeq_s$, we obtain the set of all potential candidates.

### 6.5 Related Work

This chapter builds on the previous publications by the authors that were concerned with fragmentation and attribute inference for service compositions [ICH10c, ICH11a].
<table>
<thead>
<tr>
<th>$a_{22}$</th>
<th>$a_{23}$</th>
<th>$a_{24}$</th>
<th>Validated candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(none)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>small, med1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(none)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>small, med1, med2, big1, big3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(none)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>smallCash, small, med1, med3, big2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(none)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>smallCash, small, med1, med2, med3, big1, big2, big3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activity</th>
<th>Contributes attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{22}$</td>
<td>pin, tax, crec</td>
</tr>
<tr>
<td>$a_{23}$</td>
<td>dep</td>
</tr>
<tr>
<td>$a_{24}$</td>
<td>cont, ssn</td>
</tr>
</tbody>
</table>

Figure 6.19: Adaptive search for loan approval service candidates.

In this chapter we expand the approach by providing a common, logic-based foundation for both representing and reasoning about functional dependencies as well as data attributes in service compositions, with the goal of supporting adaptation. We also introduce and formalize the upper and lower approximation of the sharing ordering which are, e.g., used for parallelization, fragmentation, and component search and validation.

Service adaptation has been widely studied, and [DNGM+08] gives a good overview of the methodological framework and some of the techniques proposed. The problem of automatic adaptation of service interfaces at the level of protocols in terms of protocol realizability and compatibility has been extensively studied, for instance by Ponge et al. [PBCT07], and other authors. In our approach, we start from the assumption that the adapted composition, or its fragments, need to preserve the original protocol present before the adaptation. The difference is that we do not use transition systems and automata to reason about the protocol invariants, but instead introduce data dependencies on the shared component state between the invocations. While well suited for analyzing functional dependencies and data attributes, more advanced interactions involving, e.g., timed conversations and transactional behavior, would require combining our approach with the protocol-based techniques.

Another interesting area in service adaptation research relates to automatic conversion of operations and message formats based on the semantic descriptions. Describing
the semantics of Web services, message types, and operations, as well as Web service search and matching has been well studied, and a good overview and the guide through the current state of the art can be found in Euzenat et al. [ES07]. In this chapter we deal with the issue of semantic matching indirectly, by using Boolean tests to detect properties of data (Definition 6.2.8 in Section 6.2.2), while not making a direct use of more powerful data abstraction techniques based on XQuery and XSLT. However, our discussion of component search and compliance testing (Section 6.4.3 relies on the assumption that adequate semantics descriptions and registries of prospective component candidates are readily available.

Automatic service composition, based on adaptive planning has been studied by Beauche and Poizat [BP08]. It represents an alternative to our parallelization approach (Section 6.4.1), in that it starts from a set of service components and their pre- and post-condition and data, and tries to combine them into a composition (with parallelization when possible). Our sharing-based analysis, in comparison, starts from an already existing composition, which is then analyzed and decomposed into subsets of activities that can be parallelized. Therefore, in our approach, the pre- and post-conditions of the basic blocks and the properties of their data are inferred rather than given in advance, which is well suited for finer-grain parallelization based on ad hoc artifacts.

Service fragmentation, as a form of adaptation, has been surveyed by Mancioppi et al. [MDKL11]. In this chapter, we are dealing with the problem of deciding which activities in the original composition should be assigned to which fragments, and we choose the information content described by data attributes to be the criterion. Other authors have proposed different criteria for fragmentation. Tan and Fan [TF07] proposed a technique for workflow fragmentation, in a way that maximizes the distribution of process activities among nodes of a grid or cluster of process execution (enactment) engines. The proposed approach is a form of run-time fragmentation that is transparent to the user/designer and is meant to be automatically applied by the nodes of the distributed process execution engine. The work by Yildiz, Fdhila and Godart [YG07, FYG09] on concentrates on fragmenting workflows between services in different business domains. The data items that are passed between the external services have different security or confidentiality levels, and therefore need to be protected from unauthorized lookup in different domains. Similarly to ours, their approach is also motivated by information
flow control, but is restricted to acyclic workflows. There are also other approaches to
fragmentation, such as that by Zaplata et al. [ZKML09], which instead of partitioning
the composition, assign different execution paths to nodes in the distributed enactment
environment. Of course, after assigning activities to fragments using some criteria, the
actual work of creating and deploying executable fragments involves many technical de-
tails and is very dependent on the composition language. For BPEL, a detailed discus-
can be found in the work by Khalaf and Leymann [Kha07, KL12].

The technical foundations of variable sharing analyses for logic programs have been
proposed by Jacobs and Langen [JL89] and by Muthukumar et al. [MH89, MH91, MBdlBH99]
and have been used effectively in program parallelization [MBdlBH99, BGH99]. These
analyses are instances for logic programs of the framework of abstract interpretation,
a general approach to program analysis that was originally proposed by Cousot and
Cousot [CC77b] and has been since applied to a great variety of languages and proper-
ties. An overview of its general application to several analysis problems can be found in
the book on program analysis by Nielson and Hankin [NNH05]. A clique sharing analy-
sis that offers interesting cost trade-offs has been described by Navas et al. [NBH06].

Using Formal Concept Analysis (FCA) for representing and reasoning about concep-
tual properties of objects is described in the standard texts by Ganther and Stumme
[GSW05] and Davey and Priestley [DP02].

6.6 Conclusions

Sharing analysis can be used as an underlying technique for ensuring correctness of
adaptation actions in service compositions, by taking into account and analyzing both the
control and data dependencies. Two important classes of data dependencies – functional
dependencies and data attributes – can be captured using a single representation frame-
work that centers on the notions of logic variables, substitutions and Horn clauses, for
which well-developed sharing analysis techniques and tools exist. The technique is well
suited for compositions involving complex control structures, including loops, branches
and parallel flows. Parallelization, fragmentation, and component selection and compli-
ance checking are some forms of adaptation whose correctness depends on respecting –
and can be informed by – the data sharing invariants inferred by means of sharing.
In this chapter we have presented a sharing-based framework for supporting adaptation of service compositions by means of analysis of functional dependencies and data attributes pertaining to data objects, component states, and activities inside a composition. The results of the analysis can be used for several adaptation related tasks: e.g., for rearranging or parallelizing activities, fragmenting a composition based on the attributes of data handled by its activities, or constraining the search for the replacement components.

The logical basis of representation allows us to derive pure Horn clause programs, a subset of standard Prolog programs, to capture both data and control dependencies, in the presence of complex control structures, such as branches and loops. On such programs we apply an analysis of sharing of logic variables which produces results that aggregate all possible groups of variables (representing data objects, component states, and activities) for all possible control paths in the composition. We introduce the notions of minimal and maximal sharing ordering to approximate sharing in any particular run, and use these approximations to reason about functional dependencies and data attribute inheritance. The precision of the approximation can be improved by eliminating unused control paths based on process traces or behavior prediction.
Chapter 7

Conclusions and Future Work

In this thesis we have demonstrated how some of the static program analysis techniques, such as the complexity and sharing analyses, can be effectively applied in the context of SOC to support QoS analysis and prediction, and several types of service adaptation. The approaches proposed on this basis can improve precision and information richness of QoS monitoring by simultaneously taking into account the control structure of a service composition, and the data structures and operations present in it.

The structure of control and data dependencies of a service orchestration can be translated into a set of logical statements (Horn clauses) that model the aspects of the actual executable artifacts that are relevant for the desired analysis type (e.g., complexity or sharing), while abstracting from other aspects of their operational semantics (such as the scheduling and message passing mechanisms and protocols). This logical representation is then fed to the static analyzers for logic (Prolog) programs, and the results of the analyses are mapped to the properties of the composition activities and data, combining them, when necessary, with the empirically collected monitoring data on the infrastructure level.

Complexity analysis for service compositions can improve precision of QoS analysis and prediction in two dimensions. Firstly, it expresses the computation cost of a service composition (in terms of the number of activities and the external service invocations) as a function of the content or size of its input messages, thus adding a dimension of data-sensitivity. Secondly, it expresses the computation cost (and the QoS based on that cost) using lower and upper bounds, thus giving a better idea about the dispersion of
the potential values, and reducing reliance on descriptive statistical parameters which may be difficult to collect or compute. Data-sensitive complexity bounds can be used in proactive adaptation to effectively decrease the overall computation cost by selecting service candidates that with the lowest cost for a given invocation message. The QoS bounds based on the cost bounds can be effectively used for predicting potential and imminent SLA violations at the point of invocation based on the value and size of the input request.

In the context of mission-critical SBAs that require continuous monitoring, an efficient and accurate continuous prediction of SLA violations can be achieved through structural constraint modeling from an orchestration continuation at the point of prediction. When available, the complexity analysis results can be used to increase precision of the constraint model. The constraint-based prediction for service orchestrations can be extended to the case of choreographies, with several stateful participants using more complex message exchange patterns than the simple reply-response.

Finally, several adaptation approaches, such as parallelization, fragmentation, and component service search and validation, can be based on the sharing analysis (of logic programs). A simple first-order logical representation of a composition is able to express both the functional dependencies between data and activities, as well as their information content (modeled with user-defined attributes), using the notion of (logic) variable substitution. This representation, and the corresponding analysis, can handle complex control flow patterns, such as branches, loops, and parallel flows.

**An Outline of Future Work**

Considering the breadth and the dynamism of service-oriented computing, which continuously opens new areas of application and poses new problems and challenges, there is no doubt that new interesting research questions are likely to continue to be raised in relation to the topics addressed in this thesis. With respect to the research results on which this thesis is based, we point out the following directions for the future work:

- **Further development and integration of the supporting tools.**

  The prototypes of tools developed in the scope of this work need to be developed into robust systems, and integrated with the relevant components of SOA. This ap-
plies both to the design tools, and the elements of the runtime environment, such as the service buses and composition execution engines. This is closely related to the issue of the service composition and description language support. The usability of the analysis techniques can be increased by supporting larger subsets of BPEL and other the composition languages, as well as by a better support for the more complex and dynamic features of the common expression languages, such as XPath and XSLT. Some features, such as obtaining continuations, full process mobility, and supporting complex message exchange patterns, which may be difficult to support from the existing systems, warrant the development of a constraint and logic programming based execution engine that supports them directly.

• *Thorough experimental evaluation and validation.*

Considering the complexity of SOC technologies and execution environments, a full experimental evaluation and validation of the approaches described in this thesis is a highly non-trivial issue, because the results may be influenced by many semi-independent parts of the system. More results in this direction can help further identify and/or disambiguate the key factors affecting the quality of the proposed approaches in real-world applications.

• *Moving towards richer QoS models.*

To increase precision, the QoS analysis approaches presented in this thesis bring data-sensitivity and the complexity/QoS bounds. One way to further increase their precision and information richness is to treat the probability distributions for service QoS explicitly, while another would be to incorporate the causality relationships (e.g., based on the Bayesian probabilistic reasoning). Other directions towards enriching QoS analysis and prediction include a systematic treatment of uncertainty and incomplete knowledge, as well as of strategic behavior of the services participating in a composition.

• *QoS planning, prediction and adaptation in a cloud.*

While not described in this thesis, our work [ITCD10] has explored the QoS analysis from the point of view of a service provider, and not one, but potentially many concurrently executing service (composition) instances, that use the common pro-
vision infrastructure, and are thus constrained by its capacity. This aspect of QoS planning from the point of the provider naturally complements the instance-based (i.e., user-centric) perspective on QoS prediction and adaptation. Considering that the cloud technologies presently provide standard platforms for scalable and highly responsive SBAs, research into cloud QoS planning and behavior may contribute to more informed predictions and adaptations.
Bibliography


[DNGM+08] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-


