Runtime Adaptation of Scientific Service Workflows

Dissertation

zur Erlangung des

Doktorgrades der Naturwissenschaften

(Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik

der Philipps-Universität Marburg

vorgelegt von

Ernst Juhnke

geboren in Gießen

Marburg, 2014

Vom Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg als
Dissertation am 16.1.2014 angenommen.

Erstgutachter: Prof. Dr. Bernd Freisleben, Philipps-Universität Marburg
Zweitgutachter: Prof. Dr. Manfred Grauer, Universität Siegen

Tag der mündlichen Prüfung: 3.7.2014

Hochschulkennziffer: 1180

Erklärung

Hiermit erkläre ich, Ernst Juhnke, dass ich meine Dissertation mit dem Titel *Runtime Adaptation of Scientific Service Workflows* selbständig verfasst habe und ausschließlich die angegebenen Hilfsmittel und Quellen benutzt habe.

Die Dissertation wurde in der jetzigen oder einer ähnlichen Form noch bei keiner anderen Hochschule eingereicht und hat noch keinen sonstigen Prüfungszwecken gedient.

Marburg,

Zusammenfassung

Veränderung ist eine der Konstanten in der Entwicklung von Software. Je komplexer eine Softwarelandschaft wird, desto größer wird auch die Wahrscheinlichkeit, dass an der einen oder anderen Stelle eine Veränderung eintreten wird. Dies kann sowohl während des Entwurfs als auch zur Laufzeit einer solchen Architektur eintreten und die Software selbst wie auch die ausführende Hardware betreffen. In diesem Fall müssen neue Architekturen entworfen, umgesetzt und ausgerollt werden. Deren ständige Überwachung – zumindest zu Beginn – kann bis dato unentdeckte Schwachstellen aufzeigen, die dann wiederum behoben werden müssen. Die erfolgreiche Umsetzung der meisten dieser Schritte erfordern einen oder mehrere manuelle Eingriffe. Allerdings sind diese mit einer gewissen Fehlerwahrscheinlichkeit verbunden.

Die vorliegende Arbeit beschäftigt sich mit dieser Art von Laufzeit-Anpassungen. Mit einem Schwerpunkt auf Service-orientierte Architekturen werden Lösungen vorgestellt werden, die die Integration existierender Software als Webservices erlauben. Ebenso wird eine Lösung präsentiert, die das Modellieren von Workflows auch für Nicht-Workflow-Experten zugänglich macht, indem die Aufgaben und die Rollen des Domänen-Experten von der des Workflow-Experten innerhalb der Modellierungssoftware getrennt werden. Ein auf die Laufzeit ausgerichtetes Monitoring-Werkzeug überwacht die Auslastung der in einer Service-orientierten Architektur genutzten Ressourcen. In Kombination mit einer durch Plugins erweiterten Workflow-Engine wird eine automatische horizontale Skalierung durch die Nutzung von Infrastructure-as-a-Service-Anbietern erreicht. Im Bedarfsfalle werden nicht nur neue Ressourcen hinzugefügt, sondern auch im Nicht-Bedarfsfalle genauso automatisch wieder Ressourcen entfernt, z.B. durch einen Rückgang an ausgeführten Workflows durch sinkende Nachfrage. Dies geschieht aus Workflow-Sicht transparent und ermöglicht einen kostensparenden Einsatz dieser Lösung.

Verteilte Umgebungen ergeben die Notwendigkeit der Kommunikation über Netzwerke. Netzwerk-basierte Kommunikation führt aber auch eine weitere Fehlerebene in die Ausführungslogik ein. Um Anwendungen bzw. Workflows robust gegenüber solcherlei Fehlern zu gestalten, muss eine gesonderte Fehlerbehandlung durchgeführt werden. Dies bewirkt aber, dass die Prozesslogik eines Workflows mit Details der ausführenden Infrastruktur durchmischt wird und den beschreibenden Prozess deutlich komplexer macht. Eine Lösung wird in dieser Arbeit vorgestellt, die automatisch infrastrukturelle Fehler behandeln kann und damit die Trennung von Infrastruktur- und Prozesslogik aufrecht erhält.

Wenn Komponenten in einer verteilten Umgebung, wie sie typisch für eine Service-orientierte Architektur ist, verändert werden müssen, muss dies an jeder Komponente einzeln durchgeführt werden. Auch wenn Techniken wie Objekt-orientierte Programmierung oder bestimmte Entwurfsmuster diesen Vorgang erleichtern können, so muss doch jeder Service einzeln rekonfiguriert oder umprogrammiert werden. Aspekt-orientierte Programmierung wiederum erlaubt es, neue Funktionalitäten in den existierenden Programmcode zu weben, ohne den Sourcecode dafür verändern zu müssen. Allerdings muss dafür auf der Implementierungsseite eines Services eingegriffen werden. Service-orientierte Architekturen allerdings abstrahieren über die Implementierung eines Services

und kommunizieren durch den Austausch implementierungsunabhängiger Dokumente miteinander. Auf dieser Ebene setzen die in dieser Arbeit vorgestellten Request/Response Aspects an und erlauben das Einweben neuer Funktionalitäten auf Kommunikationsebene in einer SOA-kompatiblen Art.

Die wesentlichen Beiträge dieser Arbeit sind:

Integration. Die generische und erweiterbare *Legacy Code Description Language* und das dazugehörige Framework erlauben die einfache Integration existierender Software in Service-orientierte Architekturen. Die Komposition bestehender Services wird mit dem vorgestellten *SimpleBPEL*-Ansatz, der Wissen über technische Details nicht mehr dem Domänen-Experten aufbürdet, sondern die Rollen und damit die jeweiligen Entwicklungsaufgaben zwischen dem Domänen-Experten und dem Workflowexperten trennt, vereinfacht.

Laufzeit. Basierend auf der standardisierten *Business Process Execution Language* wird eine Lösung vorgestellt, die eine automatische Ressourcenallokation vornimmt. Zu diesem Zweck wird die Auslastung der verwendeten Ressourcen überwacht. Bei entsprechender Nachfrage und Auslastung werden automatisch passende Ressourcen allokiert und hinzugefügt. Sinkt die allgemeine Auslastung wieder, werden nicht mehr benötigte Ressourcen wieder freigegeben und de-allokiert. Der Allokationsalgorithmus berücksichtigt die Datenübertragung zwischen den beteiligten Partnern, um unnötige Transfers zu vermeiden, da diese zu einer Verlängerung der Gesamtausführungszeit eines Workflows führen können. Ein *mehr-dimensionaler genetischer Algorithmus* ist darüber hinaus noch in der Lage, die durch die Nutzung eines Cloud-Anbieters entstehenden Kosten zu berücksichtigen. Der Nutzer kann dabei seine persönliche Präferenz zwischen günstiger und schneller Ausführung individuell festlegen. Kommunikationsfehler werden dabei automatisch überwacht und – unter Einhaltung gegebener Rahmenbedingungen – korrigiert.

Kommunikation. Die hier vorgestellten Request/Response Aspects erlauben es, neue Funktionalitäten in die Kommunikationsebene von Webservices zu weben. Durch eine Pointcut-Sprache, die auf die ausgetauschten Dokumente ausgelegt ist, gelingt es, das Einweben von Aspekten vollkommen implementierungsunabhängig vom jeweiligen Service zu gestalten. Die Kontrolle über das Einweben von Aspekten wird wiederum selbst durch Webservices gesteuert. Dadurch wird eine natürliche Integration der Request/Response Aspects in Service-orientierte Architekturen sichergestellt.

Abstract

Software landscapes are rather subject to change than being complete after having been built. Changes may be caused by a modified customer behavior, the shift to new hardware resources, or otherwise changed requirements. In such situations, several challenges arise. New architectural models have to be designed and implemented, existing software has to be integrated, and, finally, the new software has to be deployed, monitored, and, where appropriate, optimized during runtime under realistic usage scenarios. All of these situations often demand manual intervention, which causes them to be error-prone.

This thesis addresses these types of runtime adaptation. Based on service-oriented architectures, an environment is developed that enables the integration of existing software (i. e., the wrapping of legacy software as web services). A workflow modeling tool that aims at an easy-to-use approach by separating the role of the workflow expert and the role of the domain expert. After the development of workflows, tools that observe the executing infrastructure and perform automatic scale-in and scale-out operations are presented. Infrastructure-as-a-Service providers are used to scale the infrastructure in a transparent and cost-efficient way. The deployment of necessary middleware tools is automatically done.

The use of a distributed infrastructure can lead to communication problems. In order to keep workflows robust, these exceptional cases need to treated. But, in this way, the process logic of a workflow gets mixed up and bloated with infrastructural details, which yields an increase in its complexity. In this work, a module is presented that can deal automatically with infrastructural faults and that thereby allows to keep the separation of these two layers.

When services or their components are hosted in a distributed environment, some requirements need to be addressed at each service separately. Although techniques as object-oriented programming or the usage of design patterns like the interceptor pattern ease the adaptation of service behavior or structures. Still, these methods require to modify the configuration or the implementation of each individual service. On the other side, aspect-oriented programming allows to weave functionality into existing code even without having its source. Since the functionality needs to be woven into the code, it depends on the specific implementation. In a service-oriented architecture, where the implementation of a service is unknown, this approach clearly has its limitations. The request/response aspects presented in this thesis overcome this obstacle and provide a SOA-compliant and new methods to weave functionality into the communication layer of web services.

The main contributions of this thesis are the following:

Shifting towards a service-oriented architecture. The generic and extensible *Legacy Code Description Language* and the corresponding framework allow to wrap existing software, e. g., as web services, which afterwards can be composed into a workflow by *SimpleBPEL* without over-

burdening the domain expert with technical details that are indeed handled by a workflow expert.

Runtime adaption. Based on the standardized *Business Process Execution Language* an automatic scheduling approach is presented that monitors all used resources and is able to automatically provision new machines in case a scale-out becomes necessary. If the resource's load drops, e. g., because of less workflow executions, a scale-in is also automatically performed. The scheduling algorithm takes the data transfer between the services into account in order to prevent scheduling allocations that eventually increase the workflow's makespan due to unnecessary or disadvantageous data transfers. Furthermore, a *multi-objective scheduling algorithm* that is based on a genetic algorithm is able to additionally consider cost, in a way that a user can define her own preferences rising from optimized execution times of a workflow and minimized costs. Possible communication errors are automatically detected and, according to certain constraints, corrected.

Adaptation of communication. The presented *request/response aspects* allow to weave functionality into the communication of web services. By defining a pointcut language that only relies on the exchanged documents, the implementation of services must neither be known nor be available. The weaving process itself is modeled using web services. In this way, the concept of request/response aspects is naturally embedded into a service-oriented architecture.

Acknowledgements

First of all, I wish to thank my supervisor Prof. Dr. Bernd Freisleben. I would like to thank the former as well as the actual colleagues and students of the Distributed Systems Group I worked with, too. Also, all of my friends deserve my thanks. Most of all, I thank my parents and my wife Tina.

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Research Contributions . . . 5
1.3 Publications . . . 6
1.4 Organization . . . 8

2 Problem Statement 11
2.1 Introduction . . . 11
2.2 Use Cases . . . 12
2.2.1 Semantic Annotation of Multimedia Data . . . 14
2.2.2 Medical Research . . . 18
2.2.3 Systems Biology . . . 19
2.3 Requirement Analysis . . . 22
2.3.1 Runtime Adaptation of Web Services and their Communication . . . 23
2.3.2 Orchestrating the Flow . . . 23
2.3.3 Non-functional Requirements . . . 26
2.3.4 Addressing Cross-Cutting Concerns . . . 28
2.4 Summary . . . 29

3 Related Work 31
3.1 Introduction . . . 31
3.2 Runtime Adaptation of Workflows . . . 32
3.3 Orchestrating the Flow . . . 33
3.3.1 Wrapping of Legacy Code . . . 33
3.3.2 Simplified Modeling . . . 35
3.4 Non-functional Requirements . . . 36
3.4.1 Automatic Scaling . . . 38
3.4.2 Dataflow-aware Scheduling . . . 38
3.4.3 Multi-objective Scheduling . . . 40
3.4.4 Fault Tolerance . . . 40
3.5 Addressing Cross-Cutting Concerns . . . 42
3.6 Summary . . . 46

4 An Architectural Blueprint for Runtime Adaptation of Workflows 47
4.1 Introduction . . . 47
4.2 Service-oriented Architectures for Scientific Workflows . . . 48
4.3 Development Support . . . 51
4.3.1 Integration of Legacy Code . . . 51
4.3.2 Simplified Modeling . . . 52

4.4 Runtime Support . . . 54
4.5 Handling of Cross-Cutting Concerns . . . 59
4.6 The Blueprint . . . 60
4.7 Summary . . . 61

5 Service Development 63
5.1 Introduction . . . 63
5.2 Design of the Legacy Code Description Language . . . 64
5.2.1 Framework . . . 64
5.2.2 Legacy Code Description . . . 64
5.2.3 Runtime Support . . . 67
5.3 Implementation of LCDL . . . 70
5.3.1 Modeling . . . 70
5.3.2 Model Processing . . . 70
5.3.3 Runtime . . . 72
5.3.4 Extensibility . . . 73
5.4 SimpleBPEL Design . . . 74
5.4.1 Extensions to DAVO . . . 74
5.4.2 SimpleBPEL Composer . . . 75
5.5 Implementation of SimpleBPEL . . . 79
5.5.1 Extensions to DAVO . . . 79
5.5.2 SimpleBPEL Composer . . . 81
5.6 Summary . . . 83

6 Adaptable Service Workflows 85
6.1 Introduction . . . 86
6.2 Combining BPEL and Scheduling – The Basic Approach . . . 86
6.2.1 Extensions to the BPEL Engine . . . 87
6.2.2 Load Balancer . . . 88
6.2.3 Host Analyzer . . . 90
6.2.4 Implementation . . . 91
6.3 A Cloud-enabled Data Flow Scheduling Architecture . . . 95
6.3.1 Workflow Development . . . 95
6.3.2 Workflow Execution . . . 96
6.3.3 Implementation . . . 100
6.4 Multi-Objective Scheduling . . . 103
6.4.1 Framework Components . . . 105
6.4.2 Multi-objective Scheduling Algorithm . . . 108
6.4.3 Implementation . . . 111

6.5 Reliability for Service Workflows . . . 112
6.5.1 Fault Tolerance Module . . . 113
6.5.2 Dynamic Resolver . . . 116
6.5.3 Implementation . . . 116
6.6 Reliability and Scalability of the Workflow Engine . . . 119
6.7 Summary . . . 121

7 Request/Response Aspects 125
7.1 Introduction . . . 125
7.2 Aspect-oriented Programming in General . . . 126
7.3 Request/Response Aspects . . . 126
7.3.1 Framework for Request/Response Aspects . . . 127
7.3.2 Pointcut Description . . . 128
7.4 Implementation . . . 131
7.4.1 Advice Interface . . . 131
7.4.2 Aspect Configurator . . . 132
7.4.3 Aspect Provider . . . 133
7.4.4 Security Manager . . . 134
7.4.5 Aspect Invocation Handler . . . 134
7.5 Aspects for Web Service Communication . . . 135
7.5.1 Workflows for Multimedia Analysis . . . 135
7.5.2 Data-flow and BPEL . . . 135
7.5.3 Flex-SwA . . . 136
7.5.4 Data Transmission in BPEL via Flex-SwA . . . 136
7.6 Efficient Data Transmission Using AOP . . . 137
7.7 Summary . . . 141

8 Evaluation 143
8.1 Introduction . . . 143
8.2 Service Development . . . 144
8.2.1 Legacy Code Wrapping . . . 144
8.2.2 Simplified Modeling of BPEL Workflows . . . 147
8.3 Scalability and Reliability in Workflows . . . 150
8.3.1 The Basic Approach . . . 150
8.3.2 Data Aware Scheduling . . . 157
8.3.3 Multi-objective Scheduling . . . 160
8.3.4 Fault Tolerance . . . 164
8.4 Request/Response Aspects . . . 168
8.4.1 Applicability of Request/Response Aspects . . . 168
8.4.2 Request/Response Aspects for Multimedia Data Analysis . . . 173

8.5 Summary 179

9 Conclusions 181

9.1 Summary 181
9.1.1 Service Development 182
9.1.2 Adaptable Service Workflows 182
9.1.3 Request/Response Aspects 183
9.1.4 Evaluation 184
9.2 Future Work 184
9.2.1 Service Development 184
9.2.2 Adaptable Service Workflows 185
9.2.3 Request/Response Aspects 185

Bibliography 187

1 Introduction

Contents

1.1 Motivation 1
1.2 Research Contributions 5
1.3 Publications 6
1.4 Organization 8

1.1 Motivation

The solution of scientific problems often involves the use of computational resources either to simulate, to calculate or to approximate results. A closer look shows that those computations usually consist of several steps that are combined and repeatedly performed to deliver the desired results, e. g., in the context of parameter studies or by applying them to different input data. In many cases, the repetition is done by manually executing all necessary steps of the computation, which can become costly both in terms of time and flexibility. To minimize the manual effort, computational steps that belong together are integrated into some kind of script. Nevertheless, scripts often can automate only parts of the complete process and manual interaction cannot be avoided. If such a process has to be executed several times, the overall throughput is often limited due to and as a result of manual interventions. To overcome even this limitation, a holistic workflow automation can significantly improve the process throughput.

However, the development of workflows that satisfy the specific needs in a scientific environment can be challenging: The computational complexity of the problems to be solved tend to become higher and higher as, for example, current research challenges like particle physics, climate prediction or systems biology show [187]. Though there can be different reasons for this matter, a common consequence is that powerful computational resources are required in order to address these issues: Typically, such resources need to be able to handle large amounts of data that need to be processed. Algorithms for problems that cannot be solved efficiently, i. e., for which only greedy

algorithms can be applied or which are inherently NP-hard. These algorithms should only consume a reasonable and predictable computation time even for large problem sizes.

In order to meet such requirements, different kinds of systems have to operate in a coordinated manner. The management of such distributed applications is an important challenge for different research fields. This comprises the integration of homogeneous resources that are – due to hardware and/or security constraints – distributed over the Internet or of (heterogeneous) services that are only available at specific sites. One such example is an expensive gas chromatograph that is only available once in a laboratory but needs to be integrated into several processes. Similarly, different multimedia analysis processes use input from the same storage facility that cannot be cloned or copied completely to the service location.

Typically, such processes are recurrent, because they are applied on many different input data sets or are configured with different parameters – or both. Therefore, an automation of those workflows is a reasonable approach to increase their effectiveness.

In contrast to business workflows, where partners are often determined by agreements or contracts (e. g., the otherwise often referenced loan approval workflow), scientific workflows rather incorporate a specific functionality, e. g., a specific algorithm for face recognition, then a specific partner. (This is in contrast to the loan approval workflow, which indeed interacts with specific partners in order to determine a person's creditworthiness.) Although business and scientific workflows differ in various aspects [79], in particular the former one is fairly interesting. While the description of a partner within a given workflow exactly defines what kind of service is to be expected (from a syntactical point of view), the concrete binding to a particular resource can be determined later at runtime. Yu and Buyya have provided an overview of different workflow systems (Figure 1.1, cf. [206]) that specifically show different types of workflow management realizations. It can be seen that the very first challenge a domain expert has to face is the choice of an appropriate workflow environment. The taxonomy not only lists all different combinations a workflow system can embody, but also shows the possibilities that arise for runtime adaptation of workflows.

The property of "late binding" constitutes an interesting facet of workflow development as well as of workflow execution. From an executional point of view, workflows can remain *abstract*, and thus, runtime optimizations such as load balancing, fault handling or communication improvements can be integrated. This thesis will explain how to take advantage of this and other facets of service-oriented architectures in order to realize a runtime adaptation of service workflows. The adaptation will also be amended by the use of cloud computing platforms.

Before any kind of runtime adaptation can take place, workflows need to be designed and implemented. Furthermore, the services that are incorporated in a workflow need to be developed first. This preparatory work can be very time-consuming and in addition requires extensive knowledge not only in this very specific area of distributed systems but also regarding service and workflow modeling. The overburdening of domain experts with such details can be prevented by tool support. Such tools ease the development of services and the integration of existing building blocks. The successive workflow modeling has to expose all necessary constructs so as to model control and data flow of the desired composition. But, it needs to automate – and therefore to mitigate – as

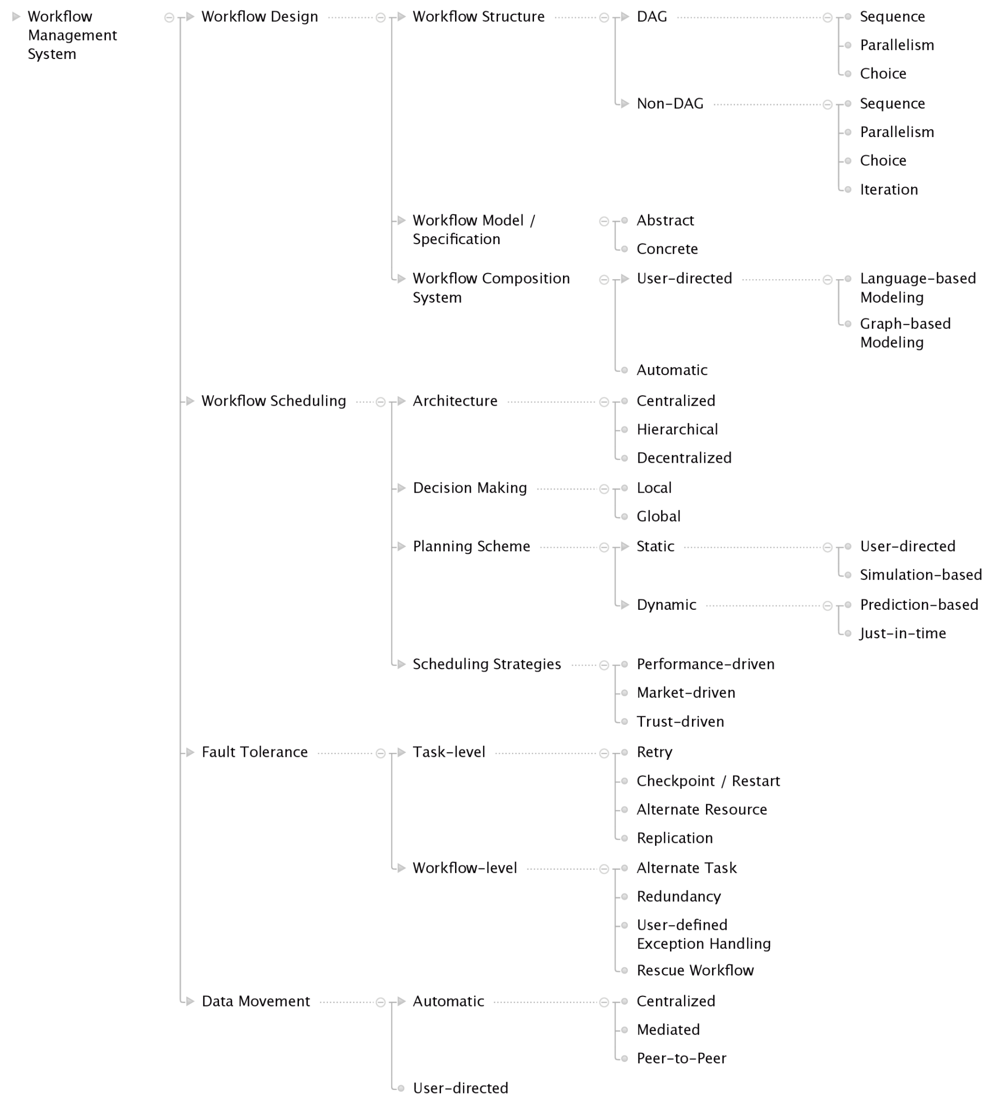


Figure 1.1: A taxonomy of workflow management systems. Based on [206].

many technical details as possible, as not to bother the user (i. e., the domain expert) with details that might distract him from his original work. For this reason, the interface for modeling workflows must offer support for the user in his work.

During runtime of a workflow, several situations can arise that typically require a manual intervention and correction, respectively. For example, if a workflow is deployed and attracts many new (or returning) users to execute it, several high-load or even overload situations can occur (such as the “slash dot effect” [59]). This can lead to a consumption of either too much network bandwidth or too much computational power. Ideally, such a situation is detected and handled automatically without any further user interaction. This task can be achieved by the application of run adaptation of workflows: tasks facing high demands are complemented with additional services to distribute the work load.

In a next step, the mechanism of load balancing can be extended further to facilitate a fault tolerance for the workflow itself: in a distributed environment that excessively uses wide area connections, communication problems are not unusual. Though from a theoretical point of view this is common knowledge, it complicates the modeling of workflows because it blows the business logic with additional infrastructural details. As an unfortunate consequence, fault handling is often slightly neglected. In cases with communication problems and without proper fault handling, the arising fault is propagated through the complete workflow. In the worst case it causes an abandonment of the entire workflow. However, communication problems do not necessarily cause such a workflow abandonment: if the task is stateless (e. g., data analysis), its call can be repeated without having any negative side effects. This fact can be exploited for an automated fault tolerance mechanism. The execution of workflow tasks is monitored and if a communication problem arises (and if the task execution is stateless), the call can be retried or even migrated to a different machine. In this way, not only the mix-up of business logic fault handling and infrastructural fault handling is avoided, but also the workflow’s reliability is increased.

In addition to communication flaws, an outage of a service can be caused by several other reasons. Besides technical faults, one reason (if not the main reason) is the change of requirements or contextual/environmental changes that imply a modification of a service or workflow implementation itself. Besides very specific changes within a single service, several partners (i. e., services) in a given software landscape are affected in a similar way. Examples are the introduction of a monitoring layer or the use of a more efficient data transmission layer concerns multiple partners. One can consider those emerging changes as *cross-cutting* in a software landscape. Instead of addressing such concerns for each service individually, they can often be addressed in a more general manner, by utilizing the techniques of aspect-oriented software design and aspect-oriented programming, respectively. Additionally, some concerns are located within the interaction of two services or a workflow and a service. By combining both aforementioned factors, the application of aspect-oriented programming at the communication level is enabled. Given an appropriate runtime environment, this offers a runtime adaptation of service behavior in the context of communication facets.

1.2 Research Contributions

This thesis presents solutions for some of the challenges of the development of scientific service workflows and their runtime adaptation.

Concerning the development time of service workflows, one of the first steps after its design is to expose (parts of) the existing software landscape as services. This task often has to be done manually and supports automation only partially. The *Legacy Code Description Language* (LCDL, see Section 5.2) presents a generic approach for wrapping existing (legacy) software for new technologies. As opposed to existing approaches, this solution is not limited to a specific mapping but instead is extensible by design. As a result, new technologies can be integrated and reused easily.

The composition of existing building blocks of a service-oriented architecture is an expensive task with regard to knowledge of service composition: e. g., the correct use of service descriptions or the correct assignment of input and output data. First and foremost, the actual composition requires the knowledge of a domain expert, who cannot be expected to be familiar with workflow composition. In this thesis, a composition framework that overcomes those prerequisites and supports a separation of infrastructural and domain specific modeling views (SimpleBPEL, see Section 5.4) is presented.

After setting up the services and composing them into a value-added workflow, the newly modeled service can be used. But at execution time the demand with respect to workflow executions may vary significantly. E. g., if new data become available, workflows are used more frequently and thereby services that are used by a specific workflow can suffer in overload situations. Therefore, transparent load balancing has to be considered as an important aspect (Scheduling in BPEL, see Section 6.2). The overall runtime, especially of scientific workflows, is also influenced by the large amount of data, which need to be exchanged between different services. The distribution of the tasks involves network-based data transfers that in turn significantly influence the workflow's runtime. To achieve a reasonable scheduling of such workflows, both the data flow and the locality of resources have to be taken into account (Data-flow enabled scheduling, see Section 6.3). As in the context of cloud computing, another dimension arises: a usage-based pricing model. For this reason, a multi-objective scheduling that also takes costs into account becomes necessary (Multi-objective Scheduling, see Chapter 6.4). In a distributed execution environment, communication errors (e. g., a infrastructural error) can occur at any time. To prevent workflow abandonment (and, in this way, the waste of CPU hours) caused by infrastructural faults, a specific fault handling that also favors the separation of infrastructural and business logic faults is essential (Reliability, see Section 6.5).

It is often necessary to address requirements (and their changes) that are scattered into different components. The building blocks (i. e., the services) of service-oriented architectures as well as the composing workflows can be affected by such changes. Several solutions exist that address this issue either at the composition site or at the implementation level of a web service. Changes that are scattered *within* different partners in a service-oriented architecture can only be addressed in a very difficult way, typically by modifying each involved service. Request/response aspects

(see Chapter 7), as presented in this thesis, apply the paradigm of aspect-oriented programming to service-oriented architectures and therefore allow the adaptation of requirements into services in a convenient way without affecting the service's availability.

To sum up, the contributions of this thesis are threefold:

- The first main contribution is a method to easily wrap existing software components for a service-oriented architecture. A comprehensive model is used to describe existing legacy code and in turn to automatically infer the desired wrapping functionality. A role-based workflow modeling tool allows to shift the modeling complexity from a domain expert to a workflow expert and thereby non-workflow experts are supported in modeling complex scientific workflows.
- The second main contribution is an approach to improve scalability and fault tolerance in workflow systems. A Cloud-based infrastructural layer allows to transparently perform scale-out and scale-in of the resource pool while respecting data flow dependencies and arising costs at the same time.
- The third main contribution is an aspect-oriented programming approach to weave new functionality into the communication between services without the need to change their implementation. A pointcut-language, specifically designed for SOAP-based communication, together with an implementation-independent description of advice and weaving processors, exposed as web services, allow the aspect-oriented programming paradigm to fit naturally into a web service environment.

1.3 Publications

The following publications (in chronologically descending order) have been published during the course of this thesis.

- T. Dalman, T. Dörnemann, E. Juhnke, M. Weitzel, W. Wiechert, K. Nöh, and B. Freisleben. Cloud MapReduce for Monte Carlo Bootstrap Applied to Metabolic Flux Analysis. *Journal of Future Generation Computer Systems (FGCS)*, 29(2):582–590, 2013
- A. Mdhaffar, R. B. Halima, E. Juhnke, M. Jmaiel, and B. Freisleben. AOP4CSM: An Aspect-Oriented Programming Approach for Cloud Service Monitoring. In *Proceedings of the 11th IEEE International Conference on Computer and Information Technology*, pages 363–370. IEEE Press, 2011
- E. Juhnke, D. Seiler, R. Ewerth, M. Smith, and B. Freisleben. Request/Response Aspects for Web Services. In H. Mouratidis and C. Rolland, editors, *Advanced Information Systems Engineering*, volume 6741 of *Lecture Notes in Computer Science*, pages 627–641. Springer-Verlag, 2011

- E. Juhnke, T. Dörnemann, D. Böck, and B. Freisleben. Multi-Objective Scheduling of BPEL Workflows in Geographically Distributed Clouds. In *Proceedings of the 4th IEEE International Conference on Cloud Computing (IEEE CLOUD)*, pages 412–419. IEEE Press, 2011

- B. Ihle, S. Kirch, E. Juhnke, T. Dörnemann, D. Seiler, and B. Freisleben. A Workflow Management Platform for Media Analysis in BPEL-based Grid Environments. In *Central Europe (CEUR) Workshop Proceedings*, page (online), 2011

- D. Seiler, E. Juhnke, R. Ewerth, M. Grauer, and B. Freisleben. Efficient Data Transmission Between Multimedia Web Services via Aspect-Oriented Programming. In *ACM International Conference on Multimedia Systems*, pages 93–104. ACM, 2011

- T. Dalman, T. Dörnemann, E. Juhnke, M. Weitzel, M. Smith, W. Wiechert, K. Nöh, and B. Freisleben. Metabolic Flux Analysis in the Cloud. In *Proceedings of IEEE eScience 2010*, pages 57–64. IEEE Press, 2010

- M. Harbach, T. Dörnemann, E. Juhnke, and B. Freisleben. Semantic Validation of BPEL Fragment Compositions. In *Proceedings of the fourth IEEE International Conference on Semantic Computing (ICSC2010)*, pages 176–183. IEEE Press, 2010

- T. Dalman, E. Juhnke, T. Dörnemann, M. Weitzel, K. Nöh, W. Wiechert, and B. Freisleben. Service Workflows and Distributed Computing Methods for ^{13}C Metabolic Flux Analysis. In *Proceedings of 7th EUROSIM Congress on Modelling and Simulation*, page (online), 2010

- E. Juhnke, T. Dörnemann, S. Kirch, D. Seiler, and B. Freisleben. SimpleBPEL: Simplified Modeling of BPEL Workflows for Scientific End Users. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 137–140. IEEE Press, 2010

- T. Dörnemann, E. Juhnke, T. Noll, D. Seiler, and B. Freisleben. Data Flow Driven Scheduling of BPEL Workflows Using Cloud Resources. In *Proceedings of 3rd IEEE International Conference on Cloud Computing (IEEE CLOUD)*, pages 196–203. IEEE Press, 2010

- E. Juhnke, T. Dörnemann, R. Schwarzkopf, and B. Freisleben. Security, Fault Tolerance and Modeling of Grid Workflows in BPEL4Grid. In *Proceedings of Software Engineering 2010, Grid Workflow Workshop (GWW-10)*, LNI, pages 193–200. GI, 2010

- C. Schridde, T. Dörnemann, E. Juhnke, M. Smith, and B. Freisleben. An Identity-Based Security Infrastructure for Cloud Environments. In *Proceedings of IEEE International Conference on Wireless Communications, Networking and Information Security (WCNIS2010)*, pages 644–649. IEEE Press, 2010

- S. Heinzl, D. Seiler, E. Juhnke, and B. Freisleben. Exposing Validity Periods of Prices for Resource Consumption to Web Service Users via Temporal Policies. In *Proceedings of the 11th*

International Conference on Information Integration and Web-based Applications & Services, pages 233–240. ACM and OCG, 2009

- E. Juhnke, D. Seiler, T. Stadelmann, T. Dörnemann, and B. Freisleben. LCDL: An Extensible Framework for Wrapping Legacy Code. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pages 648–652. ACM, 2009

- E. Juhnke, T. Dörnemann, and B. Freisleben. Fault-Tolerant BPEL Workflow Execution via Cloud-Aware Recovery Policies. In *Proceedings of 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 31–38. IEEE Press, 2009

- S. Heinzl, D. Seiler, E. Juhnke, T. Stadelmann, R. Ewerth, M. Grauer, and B. Freisleben. A Scalable Service-Oriented Architecture for Multimedia Analysis, Synthesis, and Consumption. *International Journal of Web and Grid Services*, 5(3):219–260, 2009

- T. Dörnemann, E. Juhnke, and B. Freisleben. On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '09)*, pages 140–147. IEEE Press, 2009

- T. Dörnemann, M. Mathes, R. Schwarzkopf, E. Juhnke, and B. Freisleben. DAVO: A Domain-Adaptable, Visual BPEL4WS Orchestrator. In *Proceedings of the 23rd IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 121–128. IEEE Press, 2009

- D. Seiler, S. Heinzl, E. Juhnke, R. Ewerth, M. Grauer, and B. Freisleben. Efficient Data Transmission in Service Workflows for Distributed Video Content Analysis. In *Proceedings of the 6th International Conference on Advances in Mobile Computing & Multimedia (MoMM2008)*, pages 7–14. ACM and OCG Book Series, 2008

- T. Dörnemann, M. Smith, E. Juhnke, and B. Freisleben. Secure Grid Micro-Workflows Using Virtual Workspaces. In *Proceedings of 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 119–126. IEEE Press, 2008

- T. Dörnemann, T. Friese, S. Herdt, E. Juhnke, and B. Freisleben. Grid Workflow Modelling Using Grid-Specific BPEL Extensions. In *Proceedings of German e-Science Conference (GES)*, pages 1–8, 2007

1.4 Organization

This thesis is organized into eight chapters. Chapter 2 first describes three use cases originating from different academic areas. Based on these use cases, a requirements analysis is conducted. Chapter 3 discusses existing approaches and reviews related work. In Chapter 4, the general scenario and the architectural blueprint, on which this thesis' contributions is founded, is presented. Service

development, comprising legacy code integration and simplified workflow design, is discussed in Chapter 5. The service composition and its resource allocation is addressed in Chapter 6. Aspect-oriented programming for web services is described in Chapter 7. The evaluation is presented in Chapter 8. Chapter 9 concludes this thesis and provides an outlook on further research.

2 Problem Statement

Contents

2.1 Introduction 11
2.2 Use Cases 12
2.2.1 Semantic Annotation of Multimedia Data 14
2.2.2 Medical Research 18
2.2.3 Systems Biology 19
2.3 Requirement Analysis 22
2.3.1 Runtime Adaptation of Web Services and their Communication 23
2.3.2 Orchestrating the Flow 23
2.3.3 Non-functional Requirements 26
2.3.4 Addressing Cross-Cutting Concerns 28
2.4 Summary 29

2.1 Introduction

This chapter first summarizes the needs for runtime adaptation of service workflows. Then, three use cases from different research areas describe typical challenges that arise in the composition and execution of complex and distributed applications. Based on these use cases, a requirements analysis is conducted.

The design as well as the implementation of distributed applications are often elaborative tasks. Not all requirements can be foreseen and the design involves different levels of expertise. It requires specific knowledge in developing distributed applications as well as a good competence with domain specific applications. For the integration of (existing) research code into a distributed application, a service-oriented architecture is a natural choice. It allows to combine and to reuse the building blocks in a flexible manner. Services are composed into a value-added workflow that in turn is executed and ideally completes successfully. Otherwise, the reasons can be manifold. For example, there can be errors in the application logic of the workflow itself. A workflow that has

been executed successfully beforehand, may fail if it is executed many times in parallel (e. g., due to large amounts of new input data), since single services are often not designed to handle such a high load scenario. Finally, the workflow may fail because one of its services did not response in time or fails completely.

Therefore, it is necessary to deal with infrastructural details during runtime, i. e., depending on the actual demand scale-out and scale-in operations of the available resources have to be performed. Eventually, infrastructural faults have to be handled without interfering (and therefore bloating) the application logic.

Additionally, during the development as well as the execution time of a distributed system, several issues – or concerns – arise that need to be addressed at several sites. Their re-implementation each time they occur can be prevented by applying the principles of aspect-oriented software development. Cross-cutting concerns are extracted and woven into the distributed system. In this way, maintenance as well as altered requirements can be addressed more easily.

2.2 Use Cases

Applications in computer science and information technology tend to become more and more complex: a larger number of users are using the same applications, while at the same time different and additional functionalities need to be integrated, and furthermore applications have to evolve over time. Despite the trend towards more complex applications, these applications are nevertheless formed of several basic components that in turn fulfill different functions. In many cases, a single component can and should be reused within different scenarios or configurations. Service-oriented architectures (SOA) are a paradigm that comprise – among others – the principles of reusability, abstraction, and standardized service contracts (see Figure 2.1). Besides the composability of existing services, the loose coupling of services allows for an easy integration of service of a third party into an application.

A service-oriented architecture is typically realized using web services as the implementation technology. Since their communication is based on a standardized exchange of XML documents, the integration of a heterogeneous software landscape is facilitated. This standard – the SOAP standard [199] – describes the encoding of data types and method invocations into an XML document. Although SOAP does not force a specific transport protocol, it is typically used in combination with HTTP. Therefore, web services are usually invoked by an HTTP request that contains the SOAP envelope (i. e., the SOAP XML document) in its body.

The integration of a heterogeneous software environment is easily manageable by means of service-oriented architectures: these are primarily used for providing value-added services by composing several web services. Processes are compositions of web services and can be consumed as these – thus leading to a recursive nature of software composition[1] [15]. Even though composition languages – like the *Business Process Execution Language for Web Services* (BPEL) [11] – originally

[1]Since processes expose themselves as web services and can be integrated in processes again, the emerging software landscape is frequently described as “fractal-like” [23].

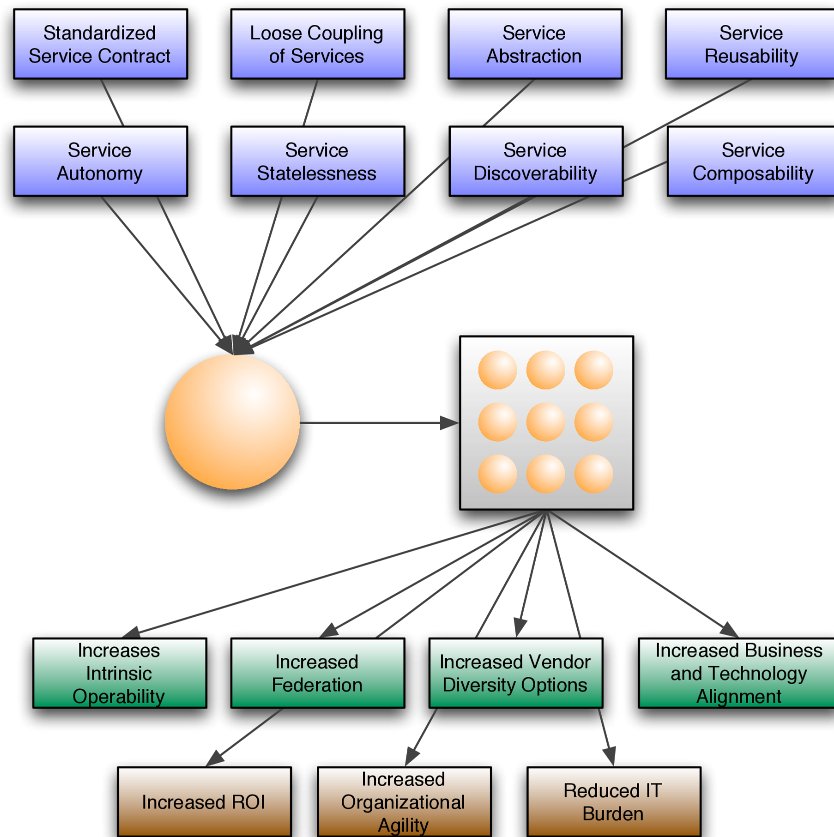


Figure 2.1: Each of the service-orientation principles contributes to the strategic goals and benefits associated with service-oriented computing. Based on [62].

emerged from a business context, the integration of heterogeneous and distributed services is also an important criterion not only for business applications but also for scientific ones.

Both types of applications are most often modeled by phrasing functional requirements not only out of the single component but also out of the overall system. Functional requirements can be considered as the *what* of the behavior of a component or of a system. For these reasons, functional behavior can be thought of as a mathematical-like definition: a web service w acts by $w(x) = y$, where x is the input and y the output of the web service.

In addition to functional specifications, a system incorporates also *non-functional requirements*. On a high level of abstraction, those requirements describe *how* a specific component has to deliver its service. Although one might have a rough understanding of what non-functional requirements really are, the term itself is still a controversial subject in the requirements engineering community [30, 81]. Nevertheless, there is a broad consensus that non-functional requirements indeed are "important and can be critical for the success of a project" [81]. SOA based applications comprise non-functional requirements on different layers. These reach from the building blocks, i. e., the web services that must model non-functional requirements like security, over the composition that has to ensure transactional behavior, up to the communication layer that needs to be efficient and reliable.

The work in this thesis in mainly motivated by three use cases stemming from different research areas. The first use case addresses the analysis of multimedia data by using different computational and data intensive algorithms for face, text, and speaker recognition. The integration of these algorithms is aimed at a concept detection approach that enables a semantic search within a given set of pictures or videos. The second use case is motivated by the D-Grid project MediGrid and performs an analysis of many patient's electrocardiograms and electroencephalograms. Based on the signals recorded overnight during a patient's sleep, enhanced signal processing is conducted in order to detect phases of respiration dropouts (apnoea syndrome). The third use case concentrates on the simulation of chemical reactions in certain microorganisms. Simulations of metabolic fluxes are important for a holistic understanding of organisms in the field of Systems Biology.

2.2.1 Semantic Annotation of Multimedia Data

This use case is motivated by a multimedia application that basically enables users to search for features in image or video databases, such as YouTube or desktop multimedia databases. The ultimate goal pursued is a search in such databases not only based on tags that were inserted manually (like in YouTube), but which indeed can detect audiovisual scene contents. This search is enabled by algorithms that analyze images and videos in order to extract and determine content-related information. These analysis algorithms (e. g., [63–65]) proceed by annotating images or video shots with information about the automatically recognized audiovisual content. In order to process a video, one component for video decoding, another one for pre-processing (e. g., Wavelet transformation), and several other ones for content analysis (e. g., face detection, other object and event recognition algorithms), and for post-processing (merging of all determined information) are utilized.

The field of multimedia data analysis has a wide range: it combines the analysis of high level attributes, like face detection, speaker recognition or text recognition, with low level attributes (like color, contrast, or frequency distribution) on different media types (e. g., audio streams, text documents, etc.). Based on previous work in multimedia data analysis (see, e. g., [63, 82, 174] and Figure 2.2), different tasks from this area are identified: face and text detection in video files and speaker recognition in audio streams. The overall aim of these algorithms is the (semi-)automatic annotation of multimedia data with metadata.

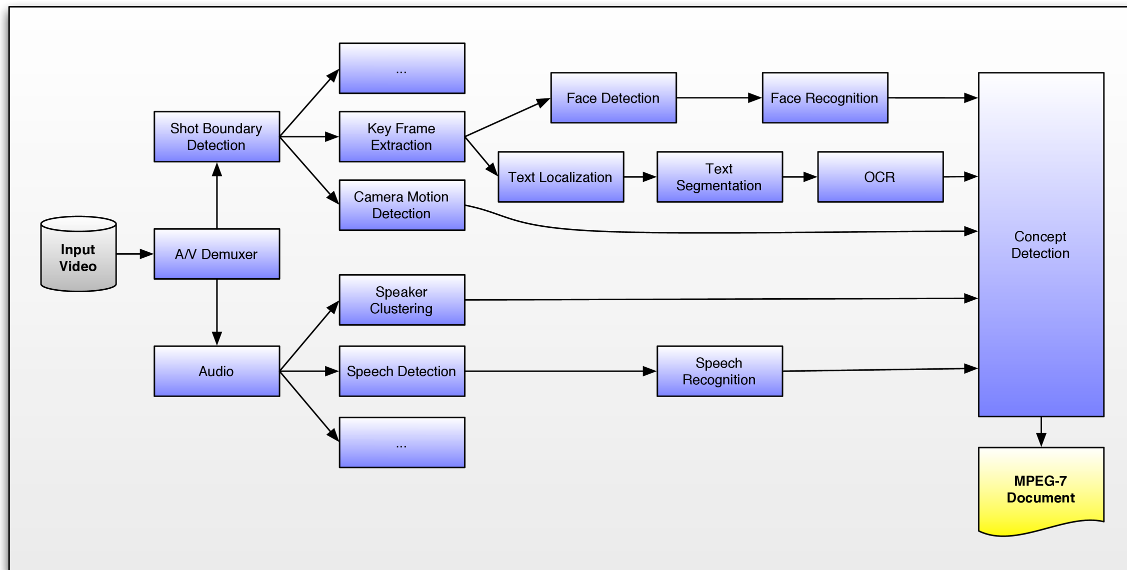


Figure 2.2: Multimedia analysis.

Face and Text Detection in Videos

The building blocks of face and text detection are:

- Decoder: an input MPEG video is split into single frames, in the first step.
- Face Detector: analyzes a single frame searching for human faces and returns the result as an MPEG-7 element.
- Text Recognizer: scans a frame for text (fragments) and returns the result as an MPEG-7 element.
- MPEG-7 Converter: merges all MPEG-7 elements provided by the Face Detector and Text Recognizer and returns an MPEG-7 document that, for instance, can be merged with the input video to mark all the faces that have been found.

The algorithms that perform face and text detection operate on single images rather than on the whole video. This feature makes a pre-processing step (cf., see Figure 2.3) that splits the video into single frames necessary. Since video streams are typically compressed by sophisticated mechanisms like motion prediction and quantization, so-called intra-frames that contain the essential information of a complete image are used as *key frames*.

In the next step, a face detection based on the algorithm of Viola and Jones [190], originally developed for object detection in (still) images, is executed on each of those intra-frames. This algorithm uses a sliding window technique that performs contrast-based filter operations. The latter ones use several different internal classifiers. These are configured in an earlier training phase in which weights for each internal classifier are determined and thereby influencing the final result. The generation of such training data is a computationally demanding operation that can last several days and is therefore preferably performed only once [167]. The results of the training are persisted and are reused. In the case of face detection, the object detector was trained to recognize frontal faces. If one or more faces are detected within a given frame, their corresponding bounding boxes[2] are returned. Such a bounding box – or several of them, if multiple faces are detected within one image – is produced for each analyzed image. In the final step, all bounding boxes are aggregated into a single metadata document, which is based on the MPEG-7[3] standard [137].

In addition to face detection, text recognition is performed – again in a frame-based manner. It can basically be divided into three steps. First, possible text boxes are detected within the given image. Next, a segmentation step yields to an improvement of the recognition. In contrast to scanned documents, a text box of a video frame may have a blurry or busy background. To avoid this undesirable effect, the segmentation step blanks out the background and creates a monochromatic image. Afterwards, each of these segmented text boxes is subject to an optical character recognition (OCR) that eventually results in a machine readable text fragment or phrase. The OCR is based on the Tesseract OCR engine [170]. The recognized text is combined with its surrounding text box and finally returned as an MPEG-7 element.

An MPEG-7 document that consolidates all the MPEG-7 elements generated by the algorithms can contain several metadata attributes, like structural and semantic information, and is saved as an XML document in a separate file. Thereby, the exchange of metadata becomes uncomplicated. Actually, such a file does not need to refer to a video file based on the MPEG standard, it can also refer to any other kind of multimedia data. In the first case, the MPEG-7 information can be encoded into special data chunks of the video file, in the same way the complete metadata are contained in this very video file.

As indicated above, most approaches for content analysis that are built in a monolithic fashion follow a sequential structure (cf., Figure 2.3 and Eide et al. [56]) and distinguish between two phases: classification and training. In the classification phase, preparatory tasks (e. g., retrieving images from a database, decoding frames of a video or decompressing audio signals) generate the smallest entities of interest needed to solve the problem in question. Next, preprocessing steps,

[2]A bounding box is a rectangular box that surrounds faces found.
[3]MPEG-7 is used as a short term for "Multimedia Content Description Interface".

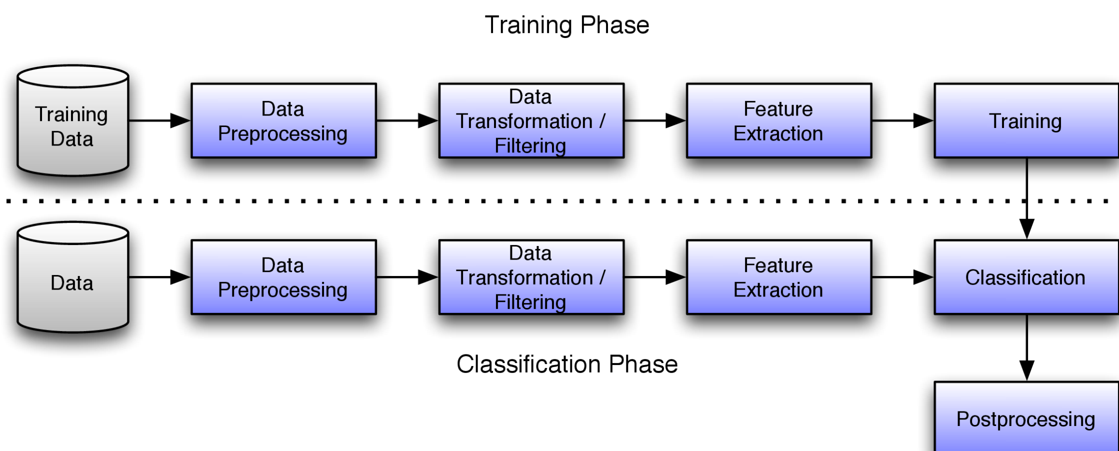


Figure 2.3: Multimedia analysis procedure.

such as transforming or filtering, take place. Afterwards, followed by the actual classification the basic step of feature extraction is performed. Sometimes, a postprocessing step is needed, such as creating a representation of the results of the analysis. The training phase is similar, but a training step substitutes the classification step: i. e., trained classifiers are not applied to actual data but their configuration is persisted. Furthermore, the classification step can be considered as a feature extraction step on a meta-level: it offers the opportunity to combine additional features in order to build more complex analyses. For example, the presence of many faces in a given picture indicates the presence of a crowd.

Speaker Clustering

Another major task in the area of multimedia data analysis is speaker recognition. Unlike the conventional speech recognition algorithms, which attempt to transform spoken language into written text, speaker recognition algorithms identify different speakers in an audio stream. Moreover, they determine exactly the specific time at which each speaker recurs. For this reason, each speaker has a unique fingerprint, which can subsequently aggregate the same speaker within a given video (by using the fingerprints as unique labels). Technically speaking, a combination of the "earth mover's distance" [160] for measuring the distance between two speaker models, with the MIX-MAX model [159] for dealing with the presence of noise, provides both fast and robust speaker recognition.

A common feature of the aforementioned tasks is the realization of their building blocks as separate binary files. For this reason, an application that utilizes these tools must be able to interact with either binary executables or binary libraries or both. The Viola-Jones algorithm is part of the Open Computer Vision library [147] that was initially developed by Intel and that is implemented in C/C++. The algorithms for speaker clustering are also written in C/C++, whereas the main applica-

tion – Videana – that integrates these algorithms and provides a user-friendly graphical front end is implemented in Java. Within Videana, all algorithms are encapsulated in separate Java classes. Special Java classes are provided in order to use the algorithms not only within the GUI of Videana but also to supply a command-line interface for the execution in (GUI-less/CLI) cluster environments.

2.2.2 Medical Research

A cooperation with researchers of the medical institute of the University of Marburg led to a use case in the area of sleep research. The studied patients are suffering from a suspension of external breathing during their sleep – the so-called *sleep apnoea syndrome*. While usually there is a continuous gas exchange between the lungs and the cellular respiration, an apnoea phase can cause a lack of oxygen in the human body because the lungs do not carry out any gas exchange. Since this can obviously lead to life-threatening situations, medical attendance is important.

As a preceding step of a sleep analysis, an examination of the patient is conducted. Besides being observed by medical staff, apnoea phases can algorithmically be detected by the analysis of a patient's electrocardiogram (ECG) and electroencephalogram (EEG). Besides other information, this data is used to treat the patient with the correct cure.

In a sleep laboratory, different signals of a patient are gathered and recorded. The studies try to identify apnoea situations during a patient's sleep and analyzes them further. Among others, the ECG and EEG of a patient are electronically recorded during his or her sleep. The ECG and EEG signals are saved using the "European Data Format" (EDF) [117]. Despite the fact that the used toolkit internally utilizes a different file format, using EDF as data format allows for using a widespread set of tools for analyzing biological and medical data. The data size of a typical overnight recording of a single patient amounts to approximately 130 MB.

Analyses of such ECG and EEG data sets are performed by means of the Physio Toolkit, a common set of open source tools in biomedical sciences [155]. Most of the tools are written in plain C/C++, some of them have special software dependencies like Perl or Matlab. The toolkit is distributed as source code and – before using it – needs to be compiled by the user for each single platform separately.

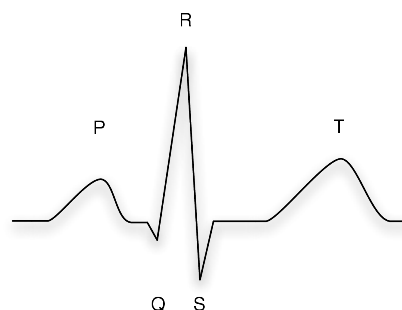


Figure 2.4: Schematic of a human's heartbeat.

The analysis of a patient's ECG and EEG consists of several sub-steps: Before a data set can be analyzed by the mentioned toolkit, a conversion from EDF into the Physio Toolkit specific data format (WaveForm DataBase) is necessary. The recorded ECG signal does not contain all peaks of the patient's heartbeat. Especially its Q-S peaks, which are important for further processing, are missing. To eradicate this flaw, a signal processing step analyzes the ECG data, identifies the R peak, and – by using numerical methods – determines

the Q and S peaks (cf., Figure 2.4). Eventually, the saved ECG signal is enhanced with this information. In order to provide researchers a more useful context for the determined apnoea phases, a heartbeat detection is also performed on the gained QRS signals. Since the apnoea detection and the heartbeat detection are independent computation tasks as sketched in Figure 2.5, they can be performed in parallel. To confirm the patient's personal rights and legal constraints, all records are anonymized beforehand.

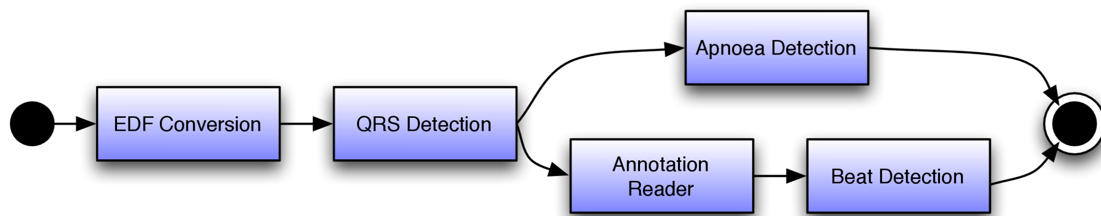


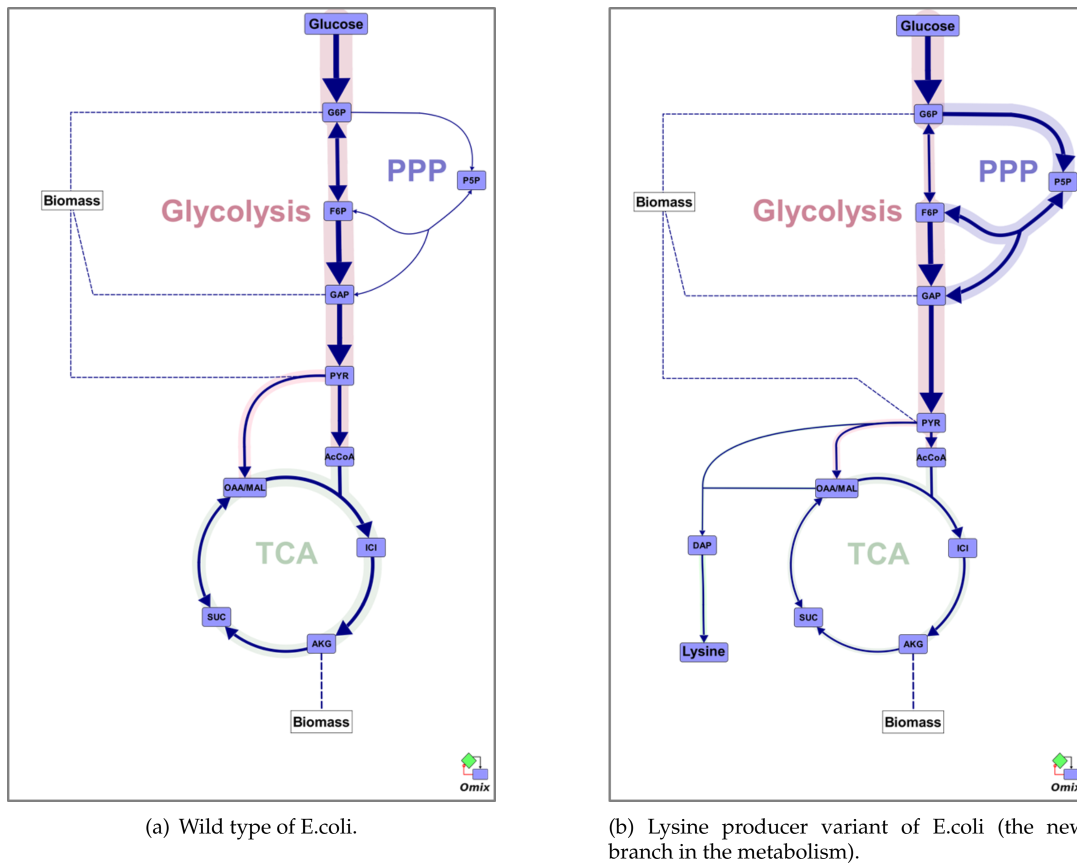
Figure 2.5: Analysis sequence of the apnoea detection.

Though the typical sequential runtime of the aforementioned analysis is only about 3.5 minutes, it is usually invoked in a – highly – parallel manner in order to analyze several records of different patients at the same time. Furthermore, each single analysis step depends not only on its direct predecessors but also on the input record as well as on the intermediary results that are produced by the different analysis steps.

2.2.3 Systems Biology

In the last decades, the interdisciplinary field of systems biology is concerned with a holistic understanding of biological systems. In the use case presented here, a model of the intracellular fluxes of microorganisms, especially of the Escherichia coli (E.coli) bacterium, is created. These bacteria are of special interest, due to their capability of synthesizing a type of insulin that is similiar to the insulin synthesized in the human body. However, before the E.coli bacteria can be used for insulin production at a larger scale, their inter-bacterial metabolism has to be optimized. The optimization manipulates the metabolism of an organism in such a way that specific substrates that beforehand are only internally used are now produced in a greater amount (cf., Figure 2.6(a) vs. Figure 2.6(b)). The excess of those substrates can either be used directly or as an educt for further chemical reactions, e. g., being a producer of insulin or lysine (Fig. 2.6(b)). Such an optimization requires (almost) all intra-cellular fluxes and reduction rates known beforehand.

Metabolic Flux Analysis with ^{13}C-isotope tracers (13C-MFA) is a modern approach in Metabolic Engineering for model-based estimation of non-measurable intracellular reaction rates [196]. To summarize, based on the model building cycle, the overall 13C-MFA application consists of several steps: starting with the definition of the model structure, the model is parameterized. Unknown model parameters have to be uniquely determined by given measurement configurations (identifi-



(a) Wild type of E.coli.

(b) Lysine producer variant of E.coli (the new branch in the metabolism).

Figure 2.6: Different metabolic fluxes of the E.coli bacterium. The thickness of an arrow represents the amount of metabolic fluxes between different substances. Source: Conference presentation of [34].

ability analysis) and have to be estimated with the best possible precision and accuracy (parameter fitting). Statistical quality measures then provide means to handle the fitted parameters' certainty. In the validation step, the model is tested with new experimental data typically unraveling a number of model deficiencies. By a planned experimental design, this process is repeated until finally the validation step is considered to be satisfactory (by using the statistical quality measures).

Methods from inferential statistics are used to measure the significance of the estimated results, which have previously been obtained from the (error-prone and thus uncertain) data. In the parameter estimation step, Monte Carlo-like sampling methods are used. Those are data-based simulation methods that generate an ensemble of simulated data sets from random numbers. From these simulation experiments, in turn, the reliability of the estimated flux parameters can be assessed.

The parametric Monte Carlo Bootstrap (MCB) algorithm draws random samples from a known (parameterized) probability distribution (e. g., a Gaussian distribution given by its mean and variance) and generates new parameter estimates by repeating the model fitting step for each sample. If statistical quality measures indicate a convergence of estimates, the last estimates are taken as the resulting distribution parameters [156]. Roughly speaking, MCB provides a conceptually simple way to derive confidence intervals (i. e., sufficiently small intervals containing the true value of the unknown model parameters) for all flux values. In addition, further statistical characteristics like skewness, bias, kurtosis, etc. can be deduced [55].

For the application of 13C-MFA, an important point to be mentioned is that the following preliminary assumptions are made:

1. Measurements are independent and identically-distributed, and systematic errors are excluded.
2. The sample size is chosen sufficiently large, and the sampler generates pseudo-random numbers.
3. The optimizer converges to the global optimum, i. e., the globally best point in the parameter space of interest.

Whereas the first issue is usually taken for granted (by evaluation of replicated measurements), the second point is practically limited due to the still acceptable computation time. The third aspect – if at all possible – can be met only by highly affordable global search approaches (without a guarantee of success). "Globalized" heuristic extensions of local search methods are used in order to mitigate locality effects of optimization. The proposed solution implements a random multistart approach that performs several local searches from different initial points, which are selected randomly from the search domain. Note that even this kind of sampling is not trivial, if the domain is high-dimensional and has some tight restrictions.

The basic building blocks of the implemented Monte Carlo Bootstrap method are the following:

- From the original experimental values according to the specified measurement noise, typically a Gaussian distribution with given mean and variance (*perturb*), n pseudo-measurements are generated.

- For the multistart approach, m initial points are sampled in the search space in order to mitigate well-known locality effects of non-linear optimization problems (*ssampler*).
- From a given pseudo-measurement, flux parameters are fitted (*fitfluxes*).

Algorithm 1 PSEUDOCODE OF THE MCB. The MCB Algorithm is split into three phases: (1) $n \times m$ models are generated and each subsequent `map` operation receives a list of files (lines 1 – 2); (2) perturbation and parameter fitting are wrapped by a `map` function, taking an index and a model file as its input (lines 3 – 14); (3) `reduce` wraps the *collectfitdata* call and processes the estimated flux vectors (line 15).

```
MFA-MCBOOTSTRAP(FML, n, m)
 1  ▷ Generate m random initial flux vectors
 2  S ← ssampler(FML, m)
 3  for i ← 1 to n
 4      do
 5         ▷ Model with perturbed measurements
 6         P ← perturb(FML)
 7         for j ← 1 to m
 8             do
 9                ▷ Insert random initial fluxes
10                M[i, j] ← setfluxes(P, S[j])
11                ▷ Parameter estimation
12                F[i, j] ← fitfluxes(M[i, j])
13
14
15  R ← collectfitdata(F)
```

The MCB (cf., Algorithm 1) method is parametrized by the number of Monte Carlo samples n and the number of initial values m. Due to the law of large numbers, the reliability of the estimated confidence intervals grows with increasing sampling number n. As suggested in the literature, $n = 1.000$ was chosen for the number of pseudo-measurement data sets to be generated [29]. Since m depends on many aspects, including the degree of non-linearity, the conditionedness of the parameter fitting problem, and on the number of parameters and the size of the search space, no general guideline can be given for the optimal value of m (and n, respectively).

Since even the most basic applications in the context of 13C-MFA are complex, the development of large-scale applications becomes a difficult task. Moreover, due to recent developments in experimental technologies an increased need for a reliable higher-throughput 13C-MFA has been created, i. e., for the automated processing of tens or even hundreds of data sets.

2.3 Requirement Analysis

This section conducts a requirements analysis for the use cases sketched in the previous section. In all use cases, functional as well as non-functional ones are identified and generalized. In addition

to specific requirements, supplemental requirements that allow a more general treatment of the specific ones are deduced.

This section roughly consists of two parts: the first one addresses the technical prerequisites that are necessary for dealing with non-functional requirements in service-oriented architectures. The second part identifies explicit non-functional requirements that emerge from the analysis of the use cases.

2.3.1 Runtime Adaptation of Web Services and their Communication

The different tasks of the aforementioned use cases are not only computationally demanding but also request – to some extent – specific (functional) requirements in hardware and also in software. In many cases, both obstacles make it fairly difficult to treat all arising requirements on a single platform. This induces the necessity to utilize different runtime environments. Moreover, the parallel execution, as described in the medical use case (cf., Section 2.2.2), may cause an overload situation on a single machine. Thus, the use of additional machines becomes necessary. In turn, this implies that the overall system must be able to deal with remote components located on different machines, i. e., it must be able to call remote components in order to transmit input data and to eventually receive the final results.

In service-oriented architectures, these objectives are attained by the usage of web services, which represent single (but nevertheless complex) functional components. Moreover, each web service has to meet various non-functional requirements. In addition to functional behavior, which is typically expressed within the implementation of a web service itself, several cross-cutting functional and non-functional requirements have to be addressed.

Despite of being a functional or non-functional requirement that is subject to change, such modifications are often accompanied with source code or interfaces changes. Eventually, the modified code has to be enrolled into the existing software landscape. In a productive environment this can cause a (short) disruption of a service's availability during the exchange – or even to a longer downtime due to unforeseen issues that appear during deployment and that need to be resolved before the software landscape can become productive again.

In order to minimize such downtimes, software architectures are required to be responsive to upcoming modifications, especially during runtime.

REQUIREMENT 1: *Web services, web service compositions as well as the communication between these parties are subject to change. Those changes needs to be addressable and solvable.*

2.3.2 Orchestrating the Flow

Though all applications mentioned above can theoretically be seen as monolithic ones, they are not in practice. While they still appear monolithic from a very high-level point of view, all applications

of the different use cases consist of several components. Besides common advantages, such as reusability, this allows for a broader view on the composition of the considered tasks. The resulting requirements are detailed in the following.

The need for a workflow system. Common to all three use cases is the consecutiveness of distinguished tasks. This can either be the MPEG decoding and the successional face detection in the multimedia use case, or the sequence of format conversion and subsequent signal processing in the medical use case. Although some of these tasks are executed in a strict sequential order, parallel execution of branches (e. g., apnoea detection) as well as conditional execution (e. g., face recognition is only necessary, if faces have been previously detected) is also present.

Each use case carries the inherent requirement that single tasks may be replaced by others. In the multimedia use case, for example, for comparison purposes it might be reasonable to replace the Viola-Jones based face detection by the wavelet-based object detection algorithm by Schneiderman and Kanade [163]. For the use case in systems biology the situation is similar: the numerical estimation of intra-cellular fluxes is replaced by an analytical one in order to gain more accurate results at the expense of longer execution times.

Thus, a flexible composition of the execution of tasks is required. Additionally, it should not be necessary to change the implementation of a specific task if it is used in another context. Since the composed workflow has to be integrated into an existing application, the control-flow must reside at the composition level in order to provide full control over the execution and must provide instant user feedback.

The need for a general-purpose and Turing-complete workflow system. Different workflow systems can roughly be grouped into two classes depending on the power of the modeling language. One can distinguish between modeling the workflow graph 1) as a directed acyclic graph (DAG) or 2) as a non-directed acyclic graph (non-DAG). As the naming indicates, a DAG does not contain any loop. While this eases the problem of assigning tasks to resources significantly, its drawback is that it is not applicable in every (use) case. For example, face detection is applied to each video frame but during the design time or even at the start time of the workflow it is unknown how many frames and iterations have to be generated.

Although it is possible to determine the frame count in a preprocessing step (before the workflow is actually executed), such a preprocessing thwarts the idea of formulating the complete analysis as a single workflow. Even worse, it is not possible to determine the iteration count of a loop in any case. If, for example, an optimization algorithm stays in a loop until an optimum has not been found, the concrete number of iterations will be known not before the actual iteration have already occurred.

Therefore, the workflow system must also be able to model non-DAGs as workflow graphs in order to achieve Turing-completeness. Several DAG-based [38, 76, 97, 208] and non-DAG workflow [10, 11, 127, 197] systems exist. Yu and Buyya provide a comprehensive taxonomy of workflow

systems in [206]. The benefit of standardized workflow languages is that they offer a widespread set of – existing – development and runtime tools.

To subsume, the use cases demand for the necessity of a Turing-complete workflow language that is capable of handling all sketched runtime demands.

The need for modularity. Although a workflow system is capable of describing the composition of single services in order to provide a value-added workflow, several issues still remain unconsidered. Some non-functional requirements are necessarily reflected on the workflow side. For example, a balanced invocation of multiple web services (each of them delivering the same functionality) in order to prevent peak load situations on the web service side is such a task. Since there exist different possibilities to solve this issue (workflow-based, engine-based, middleware-based, etc.), a concept of modularity that is applicable to all of them is needed. Moreover, the integration of non-functional requirements into the different participating parties (client, engine, service) calls for a modular interception: efficient data handling affects the communication, providing scalable workflow environments affects the scheduling strategy.

Besides modularity concerning the runtime components, also the modeling tools need to be designed so as to support modularity. First, some specifics of runtime peculiarities are necessarily reflected on the modeling side, and second, user-specific workflow development views (as described below) can be based on existing software models and thus can be modularized and – using plug-in mechanism – used to extend existing software.

Modularity stands for the integration of functional and non-functional requirements into service invocations. This integration is mostly done on the three aforementioned different levels (workflow, engine, and middleware). Therefore, each of these layers – and most important the workflow engine as the central component – has to provide mechanisms to achieve this extensibility.

The need for user support. Although Turing-completeness is an important factor for workflow languages especially during runtime, the development time of the workflow needs to be considered as well. Since Turing-complete workflow languages tend to be more complex than pure DAG languages, the modeling side must take this into account so as to obviate an overburdening of the workflow developer with language details. For this reason, a user being either a domain specialist or a workflow specialist must be supported during the conception and the modeling of the workflow. Tools should not only respect the various orientations of different workflow experts and domain specialists, but should also support the typical life cycle of a workflow.

Besides the workflow development and modeling, a typical workflow life cycle comprises the succeeding steps of 1) workflow packaging and deployment, 2) execution of the deployed workflow, and 3) its monitoring during runtime. For reasons of usability, these different peculiarities have to be handled by a single environment in order to provide an integrated view.

REQUIREMENT 2: *A standard-based, general-purpose Turing-complete workflow system for distributed components is essential. It is important that such a system can be handled not only by workflow experts but also by domain experts.*

2.3.3 Non-functional Requirements

In addition to the aforementioned – basic – requirements, several non-functional requirements, like reliability and scalability, cut across the architecture of a (scientific) workflow system. This is explained in more detail in the following.

The need for scalability. To provide users a workflow system, i. e., an infrastructure, that is able to deal with a potentially high fluctuating user demand, scalability is an important criterion. Although relatively few of the single tasks of the use cases are computationally intensive themselves, the parallel execution of several workflows may cause a high load or even an overload situation on the machine where the executing takes place.

Consider the cases that researchers share computational resources in a Grid's virtual organization (VO) or that a company offers a web-based service that utilizes computationally intensive workflows in the background. If many researchers or customers access the infrastructure simultaneously (by invoking the processes), the quality is quite likely to be reduced. A study on this so-called "Flash Crowd" effect has been conducted by Elson and Howell [59]. They emphasize the importance of highly and quickly scalable infrastructures in order to keep customers satisfied. More precisely, their use case addresses the availability of conventional web sites that must be accessible even in the momentum of being "slashdotted" (sudden surges of visitors that find an article/link on Slashdot.com and follow it). In case a site becomes unavailable, because of an unexpected high level of requests, customer get a negative impression and often tend to never visit this specific site (or never use a specific service) again.

To avoid this issue, the system should be able to react to fluctuating requests and loads, and to automatically provide an (almost) constant degree of high quality.

REQUIREMENT 3: *The workflow system must be able to scale automatically according to the demand of users and clients in order to provide an adequate degree of – high – quality.*

The need for reliability and fault tolerance. Applications may be composed of existing components that are provided by a third party over the Internet, by the application developer or by both. When using distributed components, the process of software development differs from traditional approaches: instead of tying together application components via (local) method calls in a programming language like Java or C, the components are loosely-coupled via message exchanges that are controlled by an execution engine. Furthermore, the components do not necessarily have to be

installed on the same machine or in the same network. Often, the used components even do not belong to the administrative domain of the developer, meaning that the latter does not control them in an administrative way. Due to this fact and due to the highly distributed nature of these composed applications, the failures (e. g., network failures, software failure on remote hosts) become quite likely.

In particular, when long-running or computationally intensive workflows are executed, fault-handling is very important, since the failure of just a single component can cause the abandonment of the entire workflow. This may lead to a loss of stability in the execution system; high costs caused by wasted CPU hours due to lost intermediary results of preceding process steps are another undesirable consequence. Many faults can be corrected either by simply retrying the failed operation or by substituting the failed component by an equivalent one. Such a substitution can be performed using equivalent components that are either already running somewhere or by deploying and starting the required components on-demand. The latter option is especially useful when no equivalent component is available. The required components are hosted on Cloud computing infrastructures – a user only has to pay for the additional resources when they are actually used (pay-as-you-go pricing). This guarantees the availability of fail-over/stand-by components without creating the need for additional spare hardware.

REQUIREMENT 4: *The workflow system must be able to react to infrastructural faults – and in case of non-business logic faults also to perform a recovery from such fault states entering into a well-defined and stable state again.*

The need for security. When the apnoea detection in the medical use case has finished, a physician has to make a diagnosis. In order to do so, the performed analysis has to take into account the patient's specific information. As a consequence, the data, which is sent to and exchanged with the analysis web services, must contain some patient specific information. Legal requirements in some countries (e. g., in Germany) prohibit transferring information that can be related to patients in order to protect their privacy. While a connection-based encryption would protect such information during transmission, these mechanisms cannot provide any means of security or encryption during the data processing phase of the service. This implies that (at least theoretically) analytical data and patient information can become correlated at this moment – which is not allowed.

While cryptographically ensured security is important to protect the privacy of patients, other domains also require means of security: nowadays multimedia data is valuable and their use and especially their distribution is restricted by the right owner. Therefore, eavesdropping must be prevented.

Privacy and authenticity are important concerns in the context of workflow executions: The communication between the different parties (client, workflow engine, and services) must be cryptographically secured. Furthermore, specific parts of the exchanged data need to stay encrypted

during the complete analysis process. (Since a processing of the encrypted parts is not necessary, cryptographic techniques like homomorphic encryption is not required.)

Therefore, mechanisms that allow a secure transmission of (parts of) sensitive data in combination combined with a reliable authentication of transmitter and receiver are desirable.

REQUIREMENT 5: *The workflow system must be able to communicate with all participating parties in a cryptographically secure manner. Moreover, encryption of parts of exchanged messages must be enforceable during the complete workflow.*

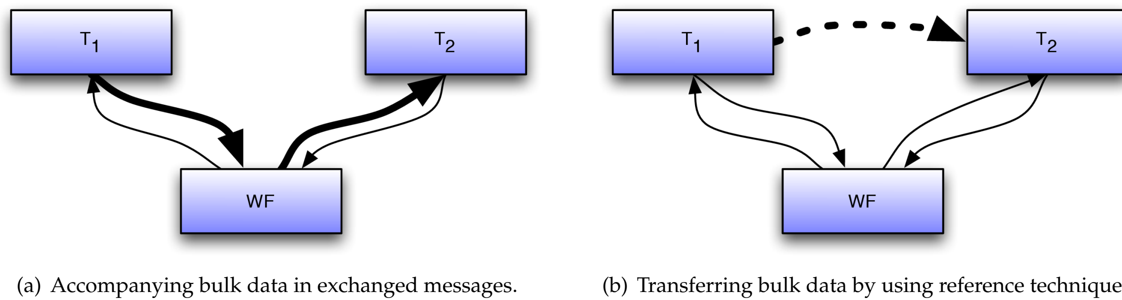
2.3.4 Addressing Cross-Cutting Concerns

In a distributed system, where tasks are orchestrated in order to compose more complex tasks, the paradigm of composing web services in a service-oriented architecture is often referred to as *Programming in the Large*. The composite task orchestrates the control flow between the original tasks, determining the order in which their methods are to be called. To keep tasks decoupled, their interaction is managed by a workflow engine, and thus, a task does not need to have any information about its successors.

While powerful in terms of compositionality, the downside of the *Programming in the Large* paradigm is that (though not always appropriate) the control flow dictates its own structure onto data-flow related concerns as sketched in the multimedia use case and also delineated by Ewerth et al. [63, 64]. Problems caused by this dominance of the control flow structure are foreseeable in a large-scale platform for content-based search in image and video databases .

Data transfer between pairs of multimedia components is forced to travel via the workflow in between. Thereby, the amount of data actually transferred over the network is doubled and obviously data transfers becomes impracticable for such data-intensive components. One would like to superimpose a structure over the component control flow, along which the data transfer cuts across the control flow. While the control flow goes from task T_1 to the workflow engine WF and then from there to task T_2, the data follows a direct shortcut from T_1 to T_2 (cf., Figure 2.7). Such a shortcut should be realized without changing neither the implementation of tasks nor their interface, and without introducing a specific coupling between tasks. Other data-related concerns such as data compression or data encryption also cut across the control flow in component compositions.

This basic example that abstracts the operation sequence of the aforementioned use cases points out that different non-functional requirements, like reliability and scalability, need to be addressed not only on the side of the workflow engine but also for each service. This requires that each service implements the same mechanisms for data handling. Since requirements may evolve over time, it might become necessary to incorporate this data handling even after the initial service deployment. A change in the service interface and a potential outtake of the specific service or an outtake of the complete middleware can be the consequence of manually incorporating such changes. Besides



(a) Accompanying bulk data in exchanged messages. (b) Transferring bulk data by using reference techniques.

Figure 2.7: Different realizations of data-flow between tasks in an orchestrated distributed system.

the potential outtakes, the same code for, e. g., data handling is scattered in the services of the distributed system and any change entails changes at each single service.

Traditionally, this is implemented for each component. in particular, the code for, e. g., data handling is scattered in the components of the distributed system.

Both, aspect-oriented software development and aspect-oriented programming, aim at an increased modularity of software by encapsulating cross-cutting concerns in aspects [118].

REQUIREMENT 6: *The system must be able to react to changing requirements and to implement new concerns without overly interfering with existing interfaces and existing implementations of the participating parties.*

2.4 Summary

The first section of this chapter described three use cases upon which this thesis is mainly built. They have their origin in the areas of multimedia analysis, medical research, and systems biology, respectively. Altogether, six different types of requirements for runtime adaptation of service workflow executions in the cloud were derived from these use cases. Driven by these requirements, web services in combination with BPEL were identified as the building blocks for the work presented in this thesis. While such a service-oriented architecture fulfills almost all functional requirements, other also important non-functional ones remain unaddressed. Since in a service-oriented architecture these arise at different parties (e. g., at the client or the service) and different at parts (e. g., within a compositing workflow), it is important to develop an environment that is capable of handling also these requirements, especially during runtime. The following chapter proposes an architectural blueprint that provides such an environment. Its design is elaborated in more detail in the succeeding chapters.

3 Related Work

Contents

3.1 Introduction 31
3.2 Runtime Adaptation of Workflows 32
3.3 Orchestrating the Flow 33
3.3.1 Wrapping of Legacy Code 33
3.3.2 Simplified Modeling 35
3.4 Non-functional Requirements 36
3.4.1 Automatic Scaling 38
3.4.2 Dataflow-aware Scheduling 38
3.4.3 Multi-objective Scheduling 40
3.4.4 Fault Tolerance 40
3.5 Addressing Cross-Cutting Concerns 42
3.6 Summary 46

3.1 Introduction

This chapter reviews related work from the field of runtime adaptation of scientific service workflows. In the scientific area, several workflow systems haven been emerged, e. g., Taverna [97], Kepler [124], or Triana [180]. These systems often concentrate on specific scientific domains and lack the support of a general purpose workflow management system [173]. Using standardized and industrial proven workflow systems, one can benefit from additional existing solutions that (may) address challenges that one has to manage manually in case of domain-specific workflow systems. Nevertheless, even when using a well-established service-oriented architecture and a standardized workflow environment remains some challenges. This thesis follows the idea of Barker and Hemert in "stick[ing] to standards" [17] by using web services as a realization of a serive-oriented architecture. In combination with industrial-proven standards like the "Business Process Execution Language" (BPEL, [11]), developer as well as user can profit from a extensive ecosystem of existing standards and implementations (e. g., the WS-* universe, cf., Figure 4.1).

First, approaches aiming at a general runtime adaptation in service-oriented architectures are discussed. Afterwards, the different facets of runtime adaptation, as stated in the requirements analysis in Section 2.3, are recalled. Related work concerning the typical life cycle of the development of a scientific application is presented. Finally, the fields of simplified modeling, runtime support, and cross-cutting concerns are detailed.

3.2 Runtime Adaptation of Workflows

Liu et al. [123] propose a framework that applies techniques from autonomic computing in order to monitor and analyze the execution of components. By using a rule-based executor component, an adaptive behavior can be achieved. The framework is based on an open source middleware platform MeDICI [84] that internally uses a service bus. The proposed systems assumes that the in workflows are stateless. Furthermore, the proposed system is not conform to any standard but rather states a new workflow description type.

An adaptation approach that improves quality-of-service (QoS) parameter of a given BPEL workflow is presented by Moser et al. [136]. By intercepting the message exchange of the BPEL process, service invocations are redirected to a service repository that is maintained by the proposed VieDAME environment. While the proposed system allows to select services based on different QoS criteria, like response time or availability, it does not perform any automatic actions like scale-out in case of an overall increase of response time and therefore an overall decrease of quality-of-service.

Aimed at a "workflow-as-a-service" solution, Zhao et al. [211] propose an integration of a workflow system called "Swift" [210] and OpenNebula [148]. The proposed solution allows to shift the different layers of workflow executions (e. g., services only, workflow engine and service) on Cloud resources and also to provision resources automatically. It remains open how a scale-in can be performed (besides a complete shutdown of the provisioned architecture) or how fault-handling (neither on the business-layer nor on the infrastructural layer).

A standardized graphical notation of adaptation patterns for workflow systems is presented by Döhring et al. [42]. They extend BPMN2 to explicitly model design- and runtime variations of a workflow. Further, they distinguish between structural and behavioral adaptation patterns. A prototypical implementation for designing and executing such extended BPMN workflows is based on JBoss Drools [105]. While explicitly modeling infrastructural adaptation patterns in a business workflow might be helpful, in every case in mixes business logic and infrastructural logic and in most cases complicates the view of the main business process. In fact, the solution proposed by this thesis features the separation of these two concerns.

Cardellini and Iannucci [25] present a model-based self-adaptation of SOA systems. The prototype (called MOSES) takes an abstract BPEL process as input, utilizes information provided by one or more UDDI registries, and for each service invocation a suitable subset of service instances is computed. Each invocation is either performed as a "1-out-of-n" invocation, where n invocations are executed in parallel and the first response is taken as the qualified result, or each single service within the computed subset is executed sequentially until one finishes successfully. While this ap-

proach can achieve a robust execution of service workflows, it lacks the ability of performing any kind of automatic scaling with respect to the actual demand. Therefore, the system can only optimize quality-of-service parameters within given resources, but is not able to ensure them by taking automatic actions.

Habib et al. [87] present a multi-objective optimization algorithm for scientific workflows. They weigh compute efficiency versus data efficiency. The first one is achieved by distributing all given tasks on different machines, while the latter one is accomplished by consolidating all tasks on a single machine. This approach does not take advantage of techniques that allow to pass references rather than actual data [91] in order to optimize the data flow. Therefore, a considerable potential of optimization is omitted. Furthermore, the presented approach does not discuss the problem of dealing with dynamic scaling infrastructures and the thereby arising questions when or where scale-out or scale-in operations are adequate.

3.3 Orchestrating the Flow

Based on a service-oriented architecture, the following two subsections review related work for using legacy code in service-oriented architectures and related work that allows non-workflow experts to model complex scientific workflows.

3.3.1 Wrapping of Legacy Code

The integration of legacy code into other applications has a long tradition and several approaches supporting this task can be found in the literature. Typically, those approaches proceed by mapping a special type of legacy code to some higher software component level, which can be a web service, a Grid service or something similar. This section discusses some of the existing approaches.

Glatard et al. [80] present a service wrapper that is able to integrate existing code into a service-based framework, independent of any application. Their service exposition approach is based on a factory pattern for the dynamic optimization of services by dynamically grouping them according to different strategies. Since a third party trying to execute the original service may be unable to find it due to the optimization, this interface of service changes can be considered rather harmful in a – contract-first – service-oriented architecture.

The "Grid Execution Management for Legacy Code Architecture" (GEMLCA) introduced by Delaitre et al. [39] also makes use of a factory pattern. In this approach, legacy code is wrapped, which is deployed on the head node of a Globus Toolkit installation and executed on the computed nodes managed by this head node. Delaitre et al. utilize a "Legacy Code Interface Description", a special XML-based format. For service invocations, a portal-based solution using Gridsphere is proposed. Since user interfaces and their generation significantly depend on the user's experience, provisioning and user interfacing can remain completely separated. The authors use the power of stateful services primarily rather for the support of a multi-user environment than for the life cycle management of native code.

Huang et al. [96] introduce a semi-automatic conversion of legacy code to Triana components. This is achieved by a two-step procedure: Firstly, C/C++ code is automatically wrapped as Java classes and secondly, these Java classes are embedded into Triana classes. The first step is performed by a tool named "JACAW". Listing 3.1 shows an example of such an automatically generated class. It should be noted that, as indicated by the "M" in the class name, JACAW generates a Java class for each single method. The same applies to enumerations, unions, and structs. In this way, a single header file, which is used as the basis for the JACAW wrapping procedure, can generate not only one but multiple (Java) class files. The call-by-reference paradigm is mapped to dedicated get methods within the generated Java class. Moreover, an invocation of a legacy method makes it necessary first to create a new Java object, then to execute the method `nativeCall` and eventually to call the corresponding get methods. The second step involves "MEDLI" (Mediation of Data and Legacy Code Interface), whose task consists of the wizard-based mapping from methods and fields of the generated JACAW wrapper and Triana classes/components. Although JACAW presents an interesting approach for the mediation of legacy C code to Java, as stated in the Java Code Conventions [150], the created classes do not conform with common Java pragmatics. The scope of JACAW as well as of MEDLI is rather limited. In particular, JACAW only generates Java wrapper and MEDLI only generates Triana components.

Zou and Kontogiannis [214] describe a framework that wraps legacy code in form of CORBA objects. Furthermore, they design a concept for a service repository, including the registration and localization of services. Their approach resembles UDDI. Though the authors claim to introduce an "extensible service description language", this extensibility indeed is limited to environmental descriptions, e. g., operating systems etc.

```
public class utilJ_bubblesortM {
  boolean needCopy = true;
  int arg0;
  int[] arg1;

  public utilJ_bubblesortM(int arg0, int[] arg1) {
    this.arg0 = arg0;
    this.arg1 = arg1;
  }

  public void nativeCall() {
    utilJ.bubblesort(this.arg0, this.arg1,
    this.needCopy);
  }

  public int getArg0() {
    return this.arg0;
  }

  public int[] getArg1() {
    return this.arg1;
  }
```

```
  public void setNeedCopy(Boolean needCopy) {
    this.needCopy = needCopy;
  }
}
```

Listing 3.1: Java interface of legacy code implemented in C/C++ [96].

Canfora et al. [24] present a framework that is capable of wrapping legacy interactive systems to service-oriented architectures. Legacy interactive systems require user-interaction in order to achieve the desired functionality. The described approach is able to deal with block-oriented and stream-oriented terminals. The single steps to be passed by a user in order to achieve a specific functionality are modeled by a finite state automaton. This automaton refers to XML-based descriptions of different screens and the corresponding actions. Although this is indeed an interesting approach for legacy interactive systems, some drawbacks exist as well. First, the solution can only wrap legacy interactive systems to web services. Second, the presented examples (wrapping of the terminal-based email program "PINE" for Unix-like operating systems) need to share a common state between different web service invocations (e. g., the modification of a user's email account), without providing any information how to manage this state by means of conventional (and stateless) web services – instead of WSRF services. Nevertheless, having the ability to interface (and to wrap) GUI-based legacy code seems an interesting area of future work.

3.3.2 Simplified Modeling

Due to the standardization of BPEL and its role as the de facto standard standard for process modeling in enterprises, many commercial and open-source development tools and execution engines exist. The majority of tools focus on the one-to-one mapping between BPEL activities and elements in the modeling tools. A couple of commercial approaches and some approaches in scientific workflow projects exist that pursue the same or similar goals as the developed approach.

In the scope of the OMII-BPEL project, the graphical BPEL modeler Sedna [194] has been developed. In addition to the modeling with plain BPEL activities, it introduces two levels of abstraction levels in order to ease the composition of workflows. The first level of abstraction is called "Scientific PEL" and offers pre-defined function blocks, like parallel loops, that are frequently utilized in scientific workflows. The second level of abstraction is "Domain PEL", which introduces the additional functionality of defining domain-specific BPEL function blocks. These functions blocks are coined macros and, like in SimpleBPEL, can contain several BPEL activities. Major design goals are reusability, increased clearness and to ease development. However, Sedna provides no instruments for distinguishing between BPEL experts and domain experts, meaning that while using macros one also has to deal with BPEL constructs. The authors of Sedna state that "Computational scientists can [...] be regarded as highly computer literate [...]. We can expect to find some programming skills." While this statement might actually be true for scientists from natural sciences, according to the author's experience it does not hold for scientists from other areas.

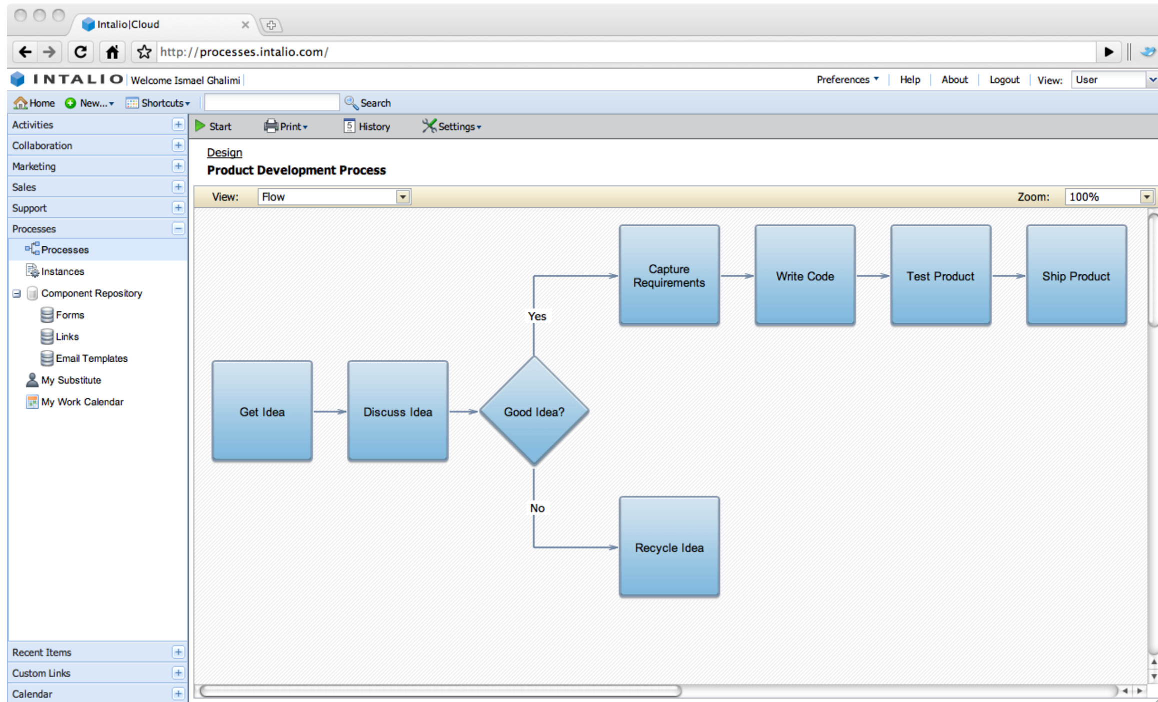
Eclipse BPEL Designer [51] is a classic BPEL modeler that uses the aforementioned one-to-one mapping. It does not offer any possibility to abstract from the complexity of BPEL modeling. Streule [176] developed a special plug-in for Eclipse BPEL Designer that provides the functionality of creating different views on BPEL processes. This feature may be used, for instance, to ease readability of complex processes or to hide confidential data when a process is handed over to business partners. As described, the plug-in only provides simplified *views* of processes rather than to allow the modification of processes. Therefore, this tool becomes quite useful for *understanding* existing BPEL processes, but does not support a non-BPEL expert to develop processes.

ActiveVOS [2] is a commercial BPM suite (Business Process Management), which was developed by Active Endpoints. It features the creation of so-called BPELets, certain reusable BPEL components that are added to the editor's palette after the creation and can be utilized for modeling processes. Like Sedna, ActiveVOS does not offer the possibility of distinguishing between the development of BPELets and from composing them into workflows. Instead, it allows to mix BPELets with plain BPEL activities and is mainly designed for BPEL experts who aim at reusing commonly required BPEL fragments.

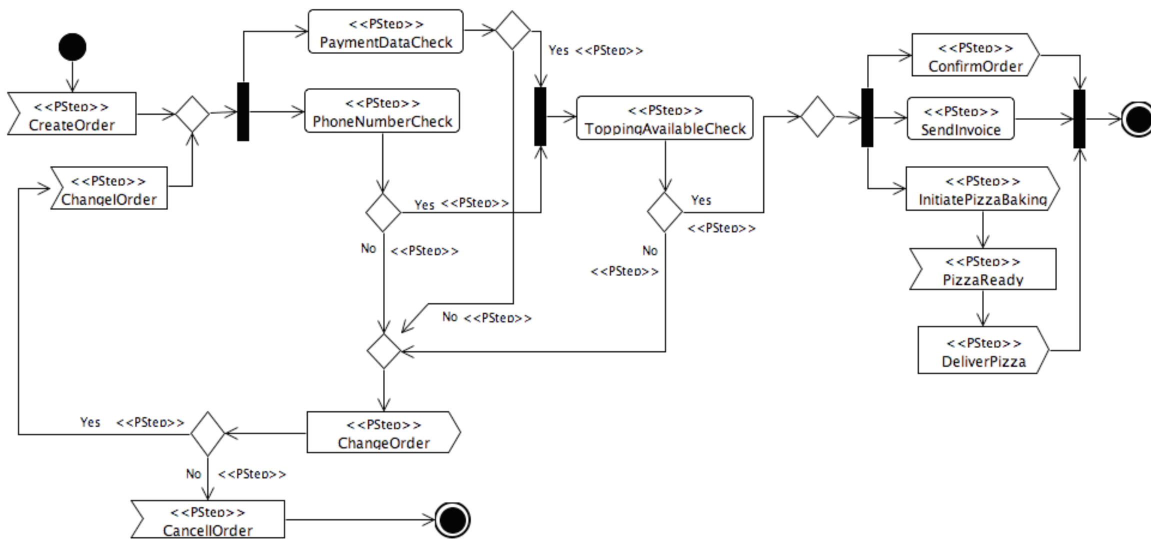
Among the variety of BPMN-based (Business Process Modeling Notation) modeling tools, Intalio|Designer is discussed exemplarily. BPMN is a graphical notation for the specification of business processes. It shows fairly many similarities to activity diagrams in UML (cf., Figure 3.3.2) and was originally designed to support business process management for not only technical but also business users. BPMN processes are abstract and cannot directly be executed by a workflow engine without any additional preparatory work by a workflow engine. However, it is possible to enrich (annotate) the process with technical information like WSDL message definitions, service endpoints and so forth and to perform a mapping to BPEL. The mapping from BPMN is not straightforward and sometimes not even possible, as discussed for instance by Ouyang et al. [153]. One of the main reasons for this is that BPMN and BPEL are fundamentally different languages: More precisely, BPMN is graph-oriented whereas BPEL is mainly block-structured. The vendors of BPMN-to-BPEL translation tools circumvent this problem by limiting the BPMN vocabulary to constructs that can be unambiguously mapped to BPEL. Since a workflow developer has to deal with WSDL messages, service port types, operations and so forth, the BPMN-based approach cannot completely hide technical details from the workflow developer.

3.4 Non-functional Requirements

Business processes as the typical domain of applications of BPEL processes make use of such static partner bindings, i. e., web service endpoints are statically defined during the development time of such a process. Nevertheless, it is beneficial to select the endpoint of a web service during runtime. Multiple options exist to achieve this aim, which are detailed below.



(a) Screenshot of Intalio|Designer [103].



(b) UML2 activity diagram of a pizza ordering process [53].

Figure 3.1: Comparison of the graphical view of the Intalio|Designer with an activity diagram of UML2.

3.4.1 Automatic Scaling

Di Penta et al. [40] have presented a dynamic binding framework called WS Binder. By means of a proxy architecture, the (re-) binding of *partnerLink*s to services during the runtime of a process becomes possible. Based on a given policy, the framework is able to schedule an invoke operation to a specific service. This binding can either occur before or during the execution of the workflow. Though the framework can support runtime recovery, it does not provide support for extending the pool of usable services, e. g., by booting a virtual machine.

By introducing a new element called *find_bind* into the BPEL language, Karastoyanova et al. try to solve the issue of runtime adaptability [116]. They develop a mechanism that is able to find services, e. g., by querying a UDDI registry. Based on policies, it selects suitable services and binds them to certain process instances. In case a service call fails, a process instance repair is guaranteed by rebinding the service to another port. Selection criteria can be modified at runtime but the dynamic provisioning of additional target hosts is not supported.

Based on BPEL, Ma et al. [125] have presented a Grid-enabled workflow management system. The system does not only allow for the interaction with stateful resources but also for dynamic service binding and moreover scalability of workflow execution. In the scenario presented in their paper, the BPEL engine – and not the target hosts of the workflow – forms a bottleneck. Scalability is achieved by first placing a load balancer in front of a cluster of BPEL engines and the subsequent scheduling of calls to the workflow engines according to the workload of the engines. Dynamic service binding at runtime is achieved in a way similar to the approach presented in this thesis. During runtime a provisioning service is called that determines a host where the requested service is installed. However, the approach neither takes the workload of target hosts into account nor does it offer any features to dynamically provide additional target hosts.

Mietzner and Leymann [135] have dealt with the problem that no current standard for a generic provisioning infrastructure exists. They argue that an architecture is needed that allows to deploy applications in different environments independently of the actual provisioning engine. The authors introduce a set of services related to provisioning and they define (several) BPEL-based workflows that make use of these services. Once widely accepted and implemented, the proposed architecture can theoretically substitute the role of the presented provisioning component. This would significantly reduce the implementation efforts, since vendor-specific backends would not have to be implemented.

3.4.2 Dataflow-aware Scheduling

Wang et al. [193] have presented a so-called look-ahead genetic algorithm (LAGA) that operates on a DAG and optimizes both the makespan (defined by the execution time) and the reliability (defined by a failure factor). For this aim, three different heuristics are employed, one with resource priority and two with task priority. Furthermore, due to the look-ahead algorithm, the GA avoids invalid solutions. Thereby, a solution better than that of other heuristics is achieved. The described approach presents sophisticated genetic algorithms that exceed a purely dataflow-aware GA by

its ability to optimize multiple objectives. Nevertheless, the approach has the common drawback that it can only operate on DAG structures and – in their presented form – is not able to deal with dynamic infrastructures. This disqualifies it from being used for web service orchestration with BPEL.

Yu and Buyya [207] have presented another approach that is based on a genetic algorithm. Their scheduling optimization problem in workflow applications takes into account two constraints, namely deadline and budget. Services are scheduled in a service-oriented architecture. Furthermore, the authors make use of advance reservations that are handled by the (Grid-) services themselves. As opposed to the approach presented in this thesis, the authors demonstrate the feasibility of their approach only by simulating it by using GridSim. On the contrary, the approach presented here does not assume services to be capable of handling reservations themselves. Furthermore, whereas the approach of Yu and Buyya primarily stresses the algorithm itself, the focus of the presented solution is on the application of the GA-based scheduling of web service invocations. It thus seems interesting to relate their results with the presented solution. In particular, the comparison and the use of different heuristics for the GA might be a reasonable enrichment of the presented approach.

De Falco et al. [67] have presented a multi-objective differential evolution algorithm (more precisely, a stochastic, non-linear optimization algorithm) for the static mapping of data-intensive Grid applications. The authors use a multi-site approach spreading application tasks among nodes, which can belong to different heterogeneous sites. Since the assign is performed statically, predictive analysis techniques for forecasting CPU load, available memory and network capacity are used. Thus, execution times (based on an instruction-count metric) and the amount of communication are incorparated into their algorithm. Due to the static assignment, increasing load and failure situations during runtime [59] cannot resolved by this approach.

Shankar and DeWitt [168] have presented several improvements to the data management capabilities of Condor, a well-known Cluster management system. The authors suggest that the disks in cluster machines should be used to cache input and output data of workflows instead of only using them for temporary storage. The cached data could then be reused by subsequent workflows, thereby avoiding (unnecessary high) data transfer times. The developed planning scheme (DAG-Condor) therefore compares the time saved by parallely running jobs on the machine that caches the data with the time spent for data transfers (in case dependent jobs are running on different machines). Once the machine with the earliest competition time has been determined, the whole workflow (DAG-based) is assigned to the machines. In the next step, DAG-Condor tries to reduce the load by the parallelization of the DAG, which again respects data dependencies. The adaptation of the presented approach to non-DAG workflows in web service environments is an interesting area of future research.

Prodan and Fahringer [157] have proposed an approach that uses a hybrid scheduling method for Grid workflows. Using existing techniques for cycle elimination, the approach is able to iteratively invoke classical optimization heuristics for DAGs and to perform task migration. Based on a genetic algorithm, the authors suggest a problem-independent search engine using different, existing prediction and performance models. Unlike the presented approach, they neither schedule web

service nor do they take dynamic resource provisioning into account. Furthermore, communication costs arising from a scheduling or a task migration remain unconsidered.

3.4.3 Multi-objective Scheduling

Yu et al. [209] have proposed a workflow execution planning approach that makes use of a multi-objective evolutionary algorithm. This approach focuses on Quality-of-Service (QoS) metrics, namely the trade-off between execution time and cost so as to meet both deadlines and budget requirements. The planner generates a set of alternative solutions if optimization criteria get into conflict with each other. Furthermore, fitness functions and penalty functions govern the minimization of the objectives. The approach has the drawback that it only operates on DAG structures and – in their presented form – do not deal with dynamic infrastructures. For this reason, they cannot be used for web service orchestration with BPEL.

A load balancing approach for BPEL workflows is presented by Ferber et al. [69]. More precisely, they introduce three scheduling algorithms (round robin, lowest counter first, and weighted random) for the dynamic invocation of web services from a BPEL workflow. This is achieved by wrapping each invoke activity within the process with a pre- and post-processing step. This means that the process is modified and the levels of infrastructural and business logic get thereby tingled. However, the authors neither take data dependencies between different invoke activities into account nor consider follow-up costs of their scheduling decision in the least. Furthermore, their system is not designed to dynamically provision new machines if a peak-load situation occurs.

Lee et al. [122] also identified the need to become aware of the underlying network infrastructure/topology in IaaS environments when data intensive workflows are used. They demonstrated this assertion by developing a simulation-based algorithm to estimate the performance of resource allocations for Hadoop MapReduce jobs. For this aim, fine-grained resource allocation policies are provided that allocate virtual machines on hosts within specific racks inside the data center of the provider. While such a fine-grained scheduling is certainly helpful for applications, like MapReduce or MPI jobs, that are communication intense web services are typically not that interlocked. Furthermore, IaaS providers do not provide the possibility to specify on which specific node a virtual machine should be provisioned; only more coarse grained possibilities like availability zones are provided.

3.4.4 Fault Tolerance

Faults in business process are typically treated by means of the business logic, i. e., exception handling methods like roll-backs are explicitly defined. In a more dynamic environment, like the proposed solution for scientific workflow computing, infrastructural faults are also fairly likely to occur. Those must be caught and handled by the business logic in order to prevent a workflow to be abandoned. A distinguished fault handling layer can undertake this task. In the following different approaches to achieve fault tolerant workflow execution are discussed.

In the approach of Di Penta et al. [40] *PartnerLink*s are bound to proxy services rather than to the original services. It is for this reason that workflows need to to be adapted to run in the environment. The occurrence of a failure causes the proxy services to be rebound to specific target services, which are determined by the discovery and the selection component of the framework. Target services are selected using a given policy that may contain QoS parameters. The introduction of a proxy service for every possible target service of a workflow leads to an increase of the complexity of the whole environment. As already stated in the discussion of automatic scaling, the framework lacks support for extending the pool of usable services, i. e., by booting a virtual machine.

TRAP/BPEL – another framework for the dynamic adaptation of service compositions presented by Ezenwoye and Sajadi [66] – makes use of a generic proxy pattern. The proxy is able to query a UDDI registry to find a suitable service. The binding is performed in an autonomic fashion, e. g., in case of a failed service call, a substitute is determined and the call is retried. The selection of such a suitable substitute can depend on a policy. Before a BPEL process can interact with the TRAP/BPEL framework, it first has to be adapted (to this one). This in particular means that all invoke operations must call the generic proxy rather than the actual service. Information about the actual target service has to be encoded in the input variable of the invoke activity for the proxy. This makes the modeling of processes much more complicated and similarly bad, common workflow modeling tools do not provide graphical assistance for this encoding. Furthermore, this is a huge source of errors: during deployment, workflow engines normally validate whether the input variables of invoke activities match the schema of the target services. If this validation is being omitted, the process deployment fails. In the approach of Ezenwoye and Sajadi, the engine can only validate whether the input variable and the schema of the proxy fit together. The validation of whether the encoded input parameters of the actual target service are compliant with its schema is not possible. Although policies can be used for service selection, it is not possible to specify post-invocation policies, e. g., for fault handling. Furthermore, the framework does not allow the dynamic provisioning of new machines that host a service equivalent to the failed one to recover from errors and and thus to prevent the failure of the entire workflow.

In addition to the formerly mentioned drawbacks, the approach of Karastoyanova et al. [116] (*find_bind*) does not distinguish between recovery strategies for different types of errors.

A self-healing approach has been pursued by Subramanian et al. [177]. The authors distinguish between different kinds of failures, e. g., functional, operational or semantic ones. Depending on the type of failure, different solutions are proposed. These cover data mediation, substitution and process reorganization. The self-healing behavior is controlled by special policies. Moreover, the authors develop a minimal prototype, SelfHeal-BPEL, that allows for the simulation of failure situations. In case of a failure, the system first suspends the running process, applies the solution that has been previously retrieved from the failure database and finally the corrected processes can be resumed.

Dobson presents a prototype tool that utilizes the implicit scope of an invoke operation to implement fault tolerance [41]. This makes it necessary to modify the BPEL process and to insert compensation and fault handler into the process. Two different fault tolerance patterns are pro-

vided. While one pattern is constantly retrying, the other one relies on an "active replication" in terms of a parallel execution of a web service invocation. Due to this active replication the latter one might yield a speed up of the process execution (by the active replication). Moreover, both patterns have infrastructural fault handling (methods) embedded into the business logic layer and mixes both layers up. Since both layers are intertwined, not only the maintainability of such a process is significantly reduced but also specific and extensive fault handling for each invoke operation becomes necessary.

3.5 Addressing Cross-Cutting Concerns

Several efforts have already been made to modularize cross-cutting concerns in a web service environment. Approaches pursuing this particular goal can roughly be grouped into three categories: (a) those that operate on the composition of services; (b) those that introduce an intermediary layer for the encapsulation of services, and (c) those that work at the remote service, i. e., on a physically not co-located service.

AO4BPEL [28] is an extension of BPEL designed for improving the modularity and for supporting dynamic adaptation of the composition logic. In this setting, every BPEL operation is allowed to be chosen as a join point. AO4BPEL operates on a different level than the work presented in this thesis, it facilitates the modularization at the level of the BPEL engine. Due to its similarity to traditional programs this seems rather natural when aspects are woven into a service-oriented architecture. However, since the services themselves need to be modified, a data transfer aspect like the one described in the presented video analysis workflow is not possible.

Courbis and Finkelstein [32] present an adaptable BPEL infrastructure that can be extended not only statically but also dynamically. Moreover, this extensibility is not limited to the engine, but can also be applied to BPEL processes. To accomplish this flexibility, the authors make use of the visitor design pattern as well as of aspect-oriented programming. Though this concept is interesting with respect to the development of a BPEL engine, – as the other considered approaches – it is fairly limited with respect to the adaptation of the BPEL engine and the BPEL process.

The modularization of cross-cutting concerns in a service-oriented environment has been identified also by other approaches, e. g., [31, 109]. In general, this objective is achieved by the scattering of concerns on the composition level. Even though this technique might help to express authentication against remote services, the mentioned approaches cannot be used anymore when cross-cutting concerns affect the functionality of a remote service.

The Web Services Management Layer (WSML) [188, 189] introduces an intermediate layer between the client and the web service, which is used for the just-in-time integration of web services into a client. Thus, the scope of WSML is limited to the client-side. In this way, limitations of a distributed application that uses web services are avoided. Since a client typically uses multiple services, emerging requirements like the dynamic selection and the integration of services have to be implemented several times. The thereby induced scattered code (by using multiple services) can be faced by utilizing Aspect Beans and Connectors of JAsCo [178]. JAsCO can work with client

requests rather than with requests on the service side. This enables WSML to apply accounting, billing, and even redirection of service invocations on the client side, but does not provide the ability of expressing the data transfer problem discussed in the previous sections.

Binder et al. [18] have introduced Service Invocation Triggers, a proxy layer based lightweight infrastructure that routes messages and thereby yields to an optimized data transfer during the orchestration of workflows. To attain this goal, the authors introduce a proxy layer that is in charge of routing messages. Thus, the exchange of messages is not controlled by the orchestration engine anymore, and it is not able to control the invocation of the services, because this task is performed by the proxy layer. The proxies themselves can be located on different machines within a distributed architecture. If the proxies are located on the same host as the services they communicate with, with respect to optimizing date transfer the (achieved) situation is comparable to Flex-SwA. However, as opposed to the approach presented in this thesis, this is accomplished by transferring the control flow to the proxy architecture rather then keeping it at the orchestration engine.

DJCutter [142] is a framework that provides remote pointcuts as a new language construct for distributed aspect-oriented programming. Remote pointcuts are used for the identification of join points in the execution of a program running on a remote machine. The architecture of DJCutter is based on a central aspect server on which the code for all remote aspects is hosted and which is notified in case a remote join point is activated. In turn, the aspect server then executes the corresponding advice. The aspect itself – except for the advice – is distributed beforehand in an automatic manner. While an aspect is distributed to remote machines, it only notifies a central server when a pointcut becomes active. In turn, this server executes the corresponding advice. However, the performance gain thus achieved is offset again since, referring back to the data transfer example, data must not only be transferred to the server but in addition must be passed back afterwards. This will certainly undo the desired performance gain.

The framework presented by Baligand and Monfort [16] is only designed to separate cross-cutting concerns within a service implementation, but cannot deal with cross-cutting concerns over multiple services or hosts. Their focus is on the weaving of aspects into the (Java) byte code of the service implementation. In particular, the presented framework is pinned to web services implemented in Java, whereas at the same time request/response aspects still operate on the message level. As a consequence, this allows request/response aspects to be independent of the concrete implementation language of a service.

AWED [139] is an aspect-oriented programming language offering explicit support for the distribution of aspects and advice. Even though both the request/response aspects presented in this thesis and AWED allow the weaving of remote aspects (as well as local aspects) and the execution of advice at remote hosts, AWED can only operate on Java. But only the XML-based WSDL-interface of a web service is known and the implementation can be done in any programming language. For this reason, it cannot be used in web service environments which have another language than Java as their implementation language.

The framework mCharm presented by Cazzola [26] reifies communication channels into their pointcut model. The fact that they use Java RMI and that their pointcut model is based on the

Java syntax prevent the use of this approach in a web service environment. If the implementation of a remote component is available, then weaving aspects into this concrete implementation is a reasonable choice. But in the SOAP context neither the implementation nor the implementation language can be assumed to be known. In general, only the exchanged XML documents for the request/response messages are available. Thus, it is necessary to define and apply aspects on the communication level.

Especially in the use case of multimedia data analysis huge amount of data has to be handled. In the following, approaches that address this topic are reviewed.

RESTful services [71] can be used for providing multimedia services. They utilize HTTP as communication protocol for the execution of remote procedures: a call simply consists of opening an URL. In this way, RESTful services represent a different architectural style of invoking services [154] than SOAP. When using RESTful services, the hosting HTTP infrastructure is utilized for the exchange of binary data. For this aim, first, the data is copied into an accessible folder of the HTTP container. Subsequently, a reference to this data is created, before the latter one is passed to the successor service or to the client. An example of such a reference from Hey!Watch is shown in Listing 3.2. In the final step, the successor can load the binary data stream via a HTTP connection.

RESTful services lack a strict communication pattern but rely upon HTTP (*GET*, *POST*, etc.) and utilize its common operation as their interaction model. Furthermore, no (standardized) interface description (like WSDL) exists for them. Their interface is described typically less formally in a human-readable manner on a web page (cf., `http://www.flickr.com/services/api/`). The same holds for the exchanged data. In contrast to SOAP-based web services, RESTful service have no constraint to encode the exchange data by using XML Schema. Nevertheless, most services use XML as the common exchange format. Bulk data are not directly included into the exchanged XML, but referenced via an HTTP link instead (as shown in Listing 3.2), which in particular supports the seamless handling of large binary data in the HTTP context.

Despite the mentioned advantages of RESTful services, there is a lack of a common agreement for the composition of different services into a workflow. Since, due to the absence of standardization, the degree of freedom is rather high, a developer has to handle (very) different interfaces and is concerned with format conversions. Thereby, in practice workflow developers have to deal not only with an uncontrolled growth of message but also of interface definitions.

As indicated above, SOAP-based web services realize communication in a different way than RESTful services. More precisely, SOAP uses a standardized format, and each message is contained in a SOAP envelope that in turn comprises the XML encoded method invocation. Moreover, based on the XML Schema [191] definition, data types are also converted into a XML representation. The bulk data typically being contained in the aforementioned data types, it is also represented as XML fragments, and, eventually, becomes part of the SOAP envelope.

```
<encoded-video>
  <title>YouTube - Mel Gibson on Grey's Anatomy</title>
  <job-id>20960</job-id>
  <specs>
```

```
    <format-name>Mobile MP4</format-name>
    <audio channels="" sample_rate="24000"
      codec="aac" stream="" bitrate="0"/>
    <format-id>15</format-id>
    <video height="144" aspect="1.22"
      length="77" container="mov" width="176"
      codec="mpeg4" fps="15.0" stream=""
      bitrate="216"/>
    <size>2037</size>
    <mime-type>video/mp4</mime-type>
    <thumb>http://static-1.heywatch.com/thumb/623207/MNmQyOlDZ4/NMZMG.↩
    ↪ jpg</thumb>
  </specs>
  <link>http: //static-1.heywatch.com/683072/zYWzTZB3kD/YouTube_-↩
  ↪ _Mel_Gibson_on_Grey_s_Anatomy.mp4</link>
  <id>508</id>
  <created-at>Fri Oct 27 01:08:17 CEST 2006</created-at>
</encoded-video>
```

Listing 3.2: Example response of Hey!Watch. Based on an example in [94].

Using SOAP, a developer does not have to deal with different interface formats anymore and, in addition, services can easily be integrated into a value-added workflow. SOAP offers several advantages, including its foundation on the WS-* protocol stack, already existing specifications and implementations of metadata, transactions, the possibility to address stateful resources and semantics [203]. One of the main benefits of SOAP is its independence of the underlying transport mechanism. Though it typically uses HTTP as its transport protocol, also the usage of message bus systems like the Java message service [106] is supported. Furthermore, SOAP permits synchronous as well as asynchronous communication, the latter one being particularly important when the tasks are long-running. In this situation, synchronous communication would lead to timeout problems due to the typical (timeout) configuration of SOAP communication. Notifications, as provided by the WS-* stack, can help to resolve this problem. Their key features, which help to resolve this issue, is that they provide distributed observer patterns and that they prevent a client from cyclic polling when it comes to long-running and/or asynchronous invocations in a situation as described above. In combination with the contribution presented in this thesis, SOAP services will offer a flexible and transparent reference mechanism for large binary data. In other words, the developed approach combines not only the advantages of RESTful services but also of SOAP-based web services such as standardized interfaces, industrial-strength tools, and the wide support by both research and industrial communities.

The CASSANDRA framework [140], a distributed multimedia content analysis system, is based on a service-oriented architecture and primarily aims at facilitating the composition of applications from distributed components within a network of cooperating devices. The individual analysis components are encapsulated into certain functional units, so-called service units. Units of the same device are managed by a local component repository that is synchronized by a master repository. A special coordination component initiates the composition of services. Each service unit has its own

control and data streaming interfaces, which are based on UPnP and TCP/IP, respectively. In order to build a SOA, this framework does not use web services for the definition of services nor for the composition of workflows. In general, the expressiveness of BPEL and the flexibility of Flex-SwA remain unused. The data transport is limited to TCP/IP only, and the adaptation to other transport protocols requires additional development efforts.

By means of a large scale content analysis engine, Gibbon et al. [78] have built a distributed system to assist content-based video retrieval and related applications. This system consists of multiple specialized servers with a centralized database. Video data is gathered through acquisition servers and delegated further to certain processing servers. The system relies on the MPEG-7 standard to store and distribute metadata. While the system itself makes use of web services to expose processing operations, it delegates the task of data transport is delegated to a subsystem. In particular, the data transfer is performed outside of the web service scope. Furthermore, the system does not rely on standardized workflow systems.

In the field of multimedia streaming, many approaches address issues like jitter and protocol-dependent APIs and foster an explicit handling of communication channels so as to address Quality-of-Service (QoS) issues (e. g., [19, 86, 138]). As opposed to this, streaming approaches do not allow to centralize the control flow and are often not intended to build a loosely-coupled analysis system.

Ecklund et al. [50] present a management protocol that addresses functional and non-functional requirements in multimedia database management systems. More precisely, the proposed protocol deals with changes of requirements during runtime. However, the authors neither address a consistency model if QoS requirements have to be enrolled on different administrative domains, nor do they consider the deployment of QoS enforcements during runtime. In other words, their middleware is only capable of mediating between QoS requirements that are already available at the affected communication partners. On the contrary, the system which evolved from this thesis is also capable of deploying new aspects during runtime.

3.6 Summary

In this chapter related work in the field of Runtime Adaptation of Scientific Service Workflows was discussed. By addressing the requirements from the previous chapter, it became clear that a holistic approach becomes necessary as it will be presented in the next chapter.

4

An Architectural Blueprint for Runtime Adaptation of Workflows

Contents

4.1 Introduction **47**
4.2 Service-oriented Architectures for Scientific Workflows **48**
4.3 Development Support **51**
4.3.1 Integration of Legacy Code 51
4.3.2 Simplified Modeling 52
4.4 Runtime Support **54**
4.5 Handling of Cross-Cutting Concerns **59**
4.6 The Blueprint **60**
4.7 Summary **61**

4.1 Introduction

This chapter presents the basic design principles, upon which the individual components presented in the following chapters will rely on. The design principles will reveal how functional as well as non-functional requirements in service-oriented architectures have to be handled. Based on the described building blocks, the detailed descriptions of the implementations are provided in the following chapters.

Conforming with the variety of existing standards and products in the context of service-oriented architectures, the developed design also complies with the design ideas of service-oriented architectures: (among others) "stick to standards"[1] [17] and focus on the integration with and to other standards.

[1] Although Barker and Hemert use the term "stick to standards" in [17] in the context of developing workflow systems, the underlying idea is the same: take advantage of existing ideas and solutions and improve them whenever and wherever necessary.

Both the described use cases and the conducted requirement analysis emphasize service-oriented architectures as the chosen infrastructural paradigm. These are able to meet the challenges of a loosely-coupled integration of distributed systems. Despite different implementation technologies, like Java RMI [152], web services are the prevalent and generally accepted implementation technology for service-oriented architectures. One of their many advantages is that they make an easy integration of heterogeneous components (i. e., services) possible because of their implementation-independent exchange of XML documents.

Though some of the challenges arising from loosely-coupled, distributed systems (as shown exemplarily in Figure 4.1) have already been addressed by different frameworks (see e. g., [195, 204] for an overview). Several issues still remain unaddressed in this context.

In the first section of this chapter, the underlying workflow system that provides the infrastructure for the service-oriented architecture is described. Then, it is sketched how different non-functional requirements can be addressed in the context of a workflow system. Finally, the overall architecture is elaborated.

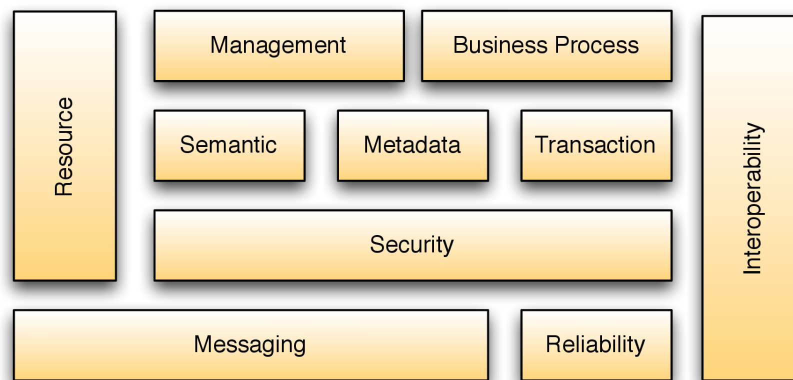


Figure 4.1: The WS-* universe: a set of standards that addresses different topics in the context of web services (and their communication). Based on [204].

4.2 Service-oriented Architectures for Scientific Workflows

All of the aforementioned use cases describe applications that consist of several basic components. During the design of non-trivial applications, an important design decision to make is the choice of granularity. This becomes even more important in distributed applications, since the granularity determines what can be provided as a service and thereby has a significant impact on reuse in different application scenarios. The applications can be built either in a monolithic fashion, i. e., the complete application/use case is represented by basically one component, or services can be more fine-grained. This allows for the reusability of services in different configurations and thus

the implementation of new applications is simplified by recomposing existing services. When using fine-grained services instead of coarse-grained or monolithic ones, a coordination of the fine-grained ones becomes necessary. Therefore, existing solutions for expressing service compositions are applied.

As shown in Figure 4.1, the WS-* universe addresses several topics concerning web services. Since services typically represent only a single functionality, the composition of services is an inherent question when it comes to service-oriented architectures. Business processes, being one topic of the WS-* universe, are used for the coordination of web service invocations and provide related functionalities: in addition to control flow and data assigning/manipulating constructs, there exist specific constructs that provide transactional behavior for a set of web service invocations. Business processes can roughly be grouped into two categories: orchestration and choreography. The first one uses a central component in order to govern the control and the data flow. On the contrary, in a choreographed composition each service has the information about its successor – without any central control. Due to the requirements stated in Section 2.3, the orchestration approach is chosen.

For the orchestration of web services the *Business Process Execution Language for Web Services* (BPEL) prevails both in industry [98, 132, 149, 161] and in academia [3, 60, 194]. It can be considered as the de-facto standard for web service compositions. It is distributed, since it is not designed to perform computations within the BPEL process itself but by the means of web services. It utilizes an ample set of standards in order to achieve extensive interoperability. Moreover, it provides a rich set of execution control constructs and tool support. From a historic point of view, BPEL has emerged from its predecessors and expresses the convergence of both powerful control constructs (XLANG) [134] and powerful graph modeling (WSFL) [99].

BPEL is standardized by OASIS[2] and therefore various implementations exist, which range from commercial-grade[3] to open-source ones[4]. One of the providers of an implementation of the BPEL standard is ActiveEndpoints. They have released their software under the terms of the GPL license. Because of this, several research projects use this implementation as their basis [49, 131, 172, 181, 185]. Besides work on the integration of different programming techniques like Java and SQL into BPEL [20, 126], there also exists prior work that enables BPEL to interact not only with – stateless – web services [61, 141] but also with – stateful – WSRF[5] services [75]. Using stateful services is beneficial if it comes to long running web services. Typical web service invocations are performed in a synchronous manner. But such a long running, synchronous invocation can cause network timeout since the established TCP connection does not transfer any data during the invocation. Therefore, an asynchronous invocation pattern is used to prevent timeouts. But using asynchronous

[2] `http://www.oasis-open.org/`.

[3] IBM WebSphere Process Server [98], Microsoft Windows Workflow Foundation [133], Oracle BPEL Process Manager [149], SAP Exchange Infrastructure [161], etc.

[4] JBoss jBPM [104], Apache ODE [13], etc.

[5] WSRF is the abbreviation for Web Services Resource Framework. Therefore, WSRF services are basically web services that are capable of maintaining information about their state, while remaining totally within the standards of web services. This is achieved by applying the factory pattern to web services and enhancing the SOAP metadata header with specific resource information that allows for a correlation. Foster and Kesselman define Grid services as "a Web service [. . .] that implements standard interfaces, behaviors, and conventions that collectively allow for services that can be transient (i. e., can be created and destroyed) and stateful (i. e., we can distinguish one service instance from another)" [75].

communication necessitates interacting with the same web service instance on subsequent calls. Since different clients can invoke and interact with the same web service, correlating an instance to the corresponding client is a difficult task. The WSRF standard [146] solves this issue: web service instances are represented as *resources* and the correlation of the web service invocation to the corresponding web service resource is achieved by the use of the meta data section of the SOAP envelop.

Nevertheless, handling necessary elements for interactions with WSRF services in plain BPEL, namely resource keys within the SOAP headers, remains difficult. It requires access to the raw messages, which are exchanged between the BPEL engine and the WSRF service, in order to manage the handling of the resource keys. Those are contained in the SOAP header, which in turn is not accessible via standard BPEL variables and access mechanisms. To solve this issue, the *BPEL4Grid* engine has been developed [43], which provides mechanisms to interact conveniently with WSRF services [47]. Activities that map the factory pattern to the BPEL language are provided as extensibility elements, such that the enhanced process remains standard compliant[6]. Besides activities that cover the interaction with the resource factory, further elements address the definition of security constraints. Those are based on the Grid Security Infrastructure that is commonly used in Grid environments [47] and therefore invocation of services becomes secure. Based on these basic interaction activities, a sophisticated pattern that models the map/reduce pattern in BPEL is provided, called *gridForEach*. It allows for a parallel execution of arbitrary BPEL containers (and thus BPEL activities) on different machines, including automatic management of the WSRF lifecycle for different threads.

In addition to a powerful workflow language and workflow enactment engine, a modeling tool that supports users during the development phase of a workflow is necessary. The *Domain Adaptable Visual Orchestrator* (DAVO) is the realization of such a tool [46]. It does not only provide a graphical user interface whose modeling capabilities covers all BPEL activities, but it also offers an extensible data and user interaction model. This allows for such an easy extension without the need of the otherwise forseeable modification of any of its internals. Using the plug-in mechanisms of Eclipse/OSGi, the editor allows for an easy extension of the graphical user interface, the data model, and export targets.

The BPEL4Grid engine along with DAVO constitutes the foundation of a service-oriented architecture (not only) for scientific workflows. Although both provide a solid platform that is able to deal with some common requirements, several issues remain unaddressed. Referring to the requirement analysis, the following issues need to be solved:

- integrating legacy code (i. e., existing binary applications) into a service-oriented architecture;
- providing a non-expert view within DAVO that is appropriate for non-BPEL experts;
- making the workflow execution more robust, i. e., scalable and reliable;

[6] Since the newly introduced elements are mapped to a namespace different from the namespace of the BPEL standard, the process is still standard compliant. However, only if the engines is able to interpret these extensibility elements, the process will be executed semantically correct.

- addressing of cross-cutting concerns in service-oriented architectures.

4.3 Development Support

Before their execution, workflows have to be modeled first. Each functionality, i. e., service, which is required within a workflow, either needs to be incorporated into a workflow in case it has been already developed, or it needs to be previously developed. In the former case, existing intellectual property in the form of legacy code has to be used. Afterwards, the single services are composed into a value-added workflow.

In the next two sections of this chapter, the integration of legacy code and a modeling approach which specifically addresses non-modeling experts is described.

4.3.1 Integration of Legacy Code

As in every software innovation cycle, the integration of existing code is an important factor for the success of a software project. In most cases, existing code represents a valuable intellectual property that has grown over a (long) period of time. Moreover, specific plattforms have been chosen to satisfy specific requirements (e. g., mathematical precision). Therefore, sometimes a redevelopment is neither reasonable nor possible.

The basic building blocks of a software landscape often consist of such existing code and are of great importance [22]. Sneed [171] emphasizes the integration of existing code into service-oriented architectures, because existing code often comprises essential business logic that cannot – easily – ported to the new platform. Redevelopment or migration to another programming language and/or framework is often not an option due to different dependencies or due to limitations in time, cost, and functionality [175].

Thus, the juncture between existing code (*legacy code* in the following) and a web service based computation platform requires the development of “glue code”. This is a kind of connection (“glue”) or wrapper code that provides a façade, i. e., a different abstraction for the underlying code. Each wrapper has to be implemented in the specific programming language of the target platform for the legacy code. This has to be repeated for all legacy code and all platforms.

Although wrappers have to be implemented for different types of legacy code, they share a common structure. This structure can be abstracted, and its functionality can be transferred into a framework that generates the common parts of the wrapper code. Binary code that is typically invoked from a command line interface interacts with the user either via command line supplied arguments via input and output files, or via standard input and standard output pipes. Invoking binary libraries is quite similar: instead of being passed by command line interactions arguments are passed by method invocations. In Listing 4.1, a manually generated wrapper code for a component from the medical use case is shown (cf., Section 2.2.2). Lines 4 – 11 represent the preparation phase, in Lines 15 – 22 the binary is actually called, and the Lines 24 – 30 fetch the result file and evaluate the return code. This code is representative for all wrapped binaries of the Physio Toolkit (although

several parts of recurring activities have already been separated into special utility classes, e. g., fetching data by using Flex-SwA [91] as it can be seen in Line 4).

The generic pattern for the framework for wrapping legacy code is the following. First, a wrapper passes all necessary arguments to the binary code, subsequently it invokes the legacy code and eventually it parses all generated results and passes them back to the initial caller. Nevertheless, the framework should not be limited to providing web services that invoke legacy code, but it should be extensible such that new data types and new bindings can easily be integrated. The extensibility should be achieved by means of a plug-in mechanism. By the implementation of new data types, the restriction of basic types like string and integer can be overcome and the framework is enabled to include complex data types like Flex-SwA references. The binding refers to the actual type of interface that is generated: providing a web service is one type of a wrapper. Thus, the binding determines which interface will be generated. The plug-in mechanism should be capable of integrating new message transport types such as Flex-SwA and new binding types like RESTful services [71].

The data model that builds the foundation for the legacy code wrapping must be able to cope with this extensibility. Despite this ample extensibility, it has to be user-centric in terms of simple usage. Moreover, it remains declarative and thus independent of any implementation details and operating systems of the wrapper code, respectively, since legacy code should be integrable into different frameworks.

4.3.2 Simplified Modeling

When used by scientists that are no experts in the IT or workflow modeling domain, the generic nature of BPEL comes with some drawbacks. As BPEL has comprehensive constructs to manage control flow and to assign and to manipulate data, modeling is quite complex. Moreover, it requires knowledge of several standards of the web service universe: this knowledge is essential for modeling the web service façade (WSDL, SOAP) [199, 200], the used data types (XML Schema) [191], and to query and assign data from and to these data types (XPath) [202]. In addition, scientific workflows for experimental data analysis and computer simulations often consist of recurring fragments. Hence, it would be highly beneficial to have a BPEL modeling tool that simplifies workflow compositions and avoids introducing the complexity of another modeling language at the same time.

An approach for simplifying workflow compositions has to be developed. It allows the modeling of *BPEL fragments*, an arbitrary but valid combination of BPEL activities. Basically, such a fragment of a BPEL process is a part of the latter one including distinguished input and output variables. These variables are exposed as connectors in order to connect fragments with each other. BPEL fragments have to be modeled using an extended version of DAVO and have to be stored in libraries (*Domain Profiles*) that are shareable and exchangeable between scientists. The new tool enables a user to build workflows by simply connecting the input connectors of BPEL fragments with the output connectors of other ones. The system automatically has to verify whether the blocks are

```
public Reference read(Reference reference) throws Exception {

  File wqrsFile = Util.acquireReference(
    UUID.randomUUID().toString(), reference);

  String workingDir = wqrsFile.getParentFile().getAbsolutePath();
  String filename = wqrsFile.getName();

  String wqrsFileWithoutExtension = filename.substring(
    0, filename.lastIndexOf("."));
  String annotatedFile = workingDir + File.separator + ↩
  ↪ wqrsFileWithoutExtension + "_beats.dat";

  boolean success = false;

  try {
    success = Util.execute("rdann", new String[] {
        "sh", "-c", "cd " + workingDir + " ; "
        + "rdann " + "-r "  + wqrsFileWithoutExtension
        + " -a wqrs > " + annotatedFile });
  } catch (Exception e) {
    throw new IOException("Invocation failed.", e)
  }

  if(! success ) {
    throw new IllegalArgumentException("rdann completed unsuccessfully↩
    ↪ .");
  }

  Reference result = ReferenceManager.create(annotatedFile);

  return result;
}
```

Listing 4.1: Manually generated wrapper code for a component of the medical use case. It wraps the annotation reader of the Physio Toolkit that interprets annotated data of the ECG/EEG signal and write the results to standard out. Those are piped to a dedicated result file. Prior to the invocation, file acquisition is performed and afterwards a reference for the newly generated result file created and returned.

compatible and can be connected or not. In the first case, the required glue code is automatically generated, such that it can be exported to a standard-compliant BPEL workflow.

The more advanced an application becomes, the more effort has to be made to model its workflow. At a certain level of complexity, a domain expert who is unfamiliar with the technical details of workflow modeling (since this is beyond of the scope of his or her research domain) would simply become overburdened by the complexity. To avoid this, the workflow development is divided into two distinct roles with clearly defined areas of responsibility. On one hand, experienced users (BPEL experts) carry out the development of BPEL fragments for the needs of the given application domain. The compliance of this task requires a standard BPEL modeling tool with additional capabilities, like storing these fragments. On the other hand, a domain expert simply combines the fragments, which are required for his or her application. Here, a tool is required that enables the domain expert to intuitively model his or her experiments, or his or her application.

In order to provide an integrated form of a graphical user interface, this simplified modeling tool has to be integrated into well-known user interfaces. The BPEL expert view certainly uses BPEL constructs, and it should thus utilize the extension points of the existing BPEL model of DAVO in order to avoid the reimplementation of a BPEL editor. For the domain expert view it might be beneficial to integrate it into the Eclipse IDE (that also serves as the basis for DAVO) but it should also be available as a stand-alone application. Therefore, the simplified modeling tool should be exportable as an RCP (rich client platform) application.

4.4 Runtime Support

While BPEL works well for modeling processes having target hosts for the execution of process steps known at runtime (called static endpoints or static processes in the following), it has some drawbacks when it comes to dynamically selecting target hosts during runtime. Static definition of target hosts is no problem for typical business processes such as the often quoted loan approval example [77, 119], but with respect to the modeling of computationally intensive processes like scientific workflows with BPEL [4, 60], it would be highly beneficial to have a BPEL implementation that automatically selects target hosts. The selection can then prefer a host with low load offering the required service in order to prevent a degradation of the provided quality-of-service.

Therefore, the process execution system should be able to react to peak loads and it should dynamically provide additional computational resources. The easy and rapid allocation of new/additional computational resources is one of the main principles of Cloud computing, and a well-established service in this area is the Elastic Compute Cloud (EC2) offered by Amazon [6]. The BPEL4Grid should be extended, by a dynamic scheduling the service calls of a BPEL process based on the target hosts' load. To handle peak loads, a provisioning component has to be integrated that dynamically launches virtual machines in Amazon's EC2 infrastructure and deploys the required middleware components (web/Grid service stack) on-the-fly.

BPEL offers a rich vocabulary and control mechanisms to express sequences of activities like *receive*, *invoke* and *reply*, parallel execution, loops, error handling as well as compensation-mechanisms

```
<assign>
  <copy>
    <from>
      <literal>
        <wsa:EndpointReference xmlns:ns="NSPACE">
          <wsa:Address>http://FQDN:PORT/ADDRESS</wsa:Address>
          <wsa:ServiceName PortName="PORT">ns:NAME</wsa:ServiceName>
          <wsa:ReferenceParameters>
            <wsa:To>...</wsa:To>
            <wsa:Action>...</wsa:Action>
          </wsa:ReferenceParameters>
        </wsa:EndpointReference>
      </literal>
    </from>
    <to variable="targetEPR"/>
  </copy>

  <copy>
    <from variable="targetEPR" />
    <to partnerLink="targetPL" />
  </copy>
</assign>
```

Listing 4.2: Manual setting of a dynamic endpoint in BPEL, truncated for the sake of readability.

to perform roll-back actions. Here, the *invoke* activity is of particular importance. It is used for the modeling of invocations of external services. The target service to be called is described via a so-called *partnerLink* that – among others – contains the following two important elements: *partnerLinkType* and *EndpointReference* (EPR). The first one is the static information that has to be known at design time. It refers to the WSDL description of the *portType* of the partner service, while the EPR refers to the concrete service which is invoked. An EPR contains a name of a service, its port (and binding mechanism) and its address given by a URI. The EPR is evaluated at runtime and may even be set at runtime, as Listing 4.2 illustrates.

This listing demonstrates that it is rather complicated to set a single service endpoint at runtime, and that it bloats the business logic with code which is related to infrastructural settings. Furthermore, prior to the setting of the EPR, the target address must be determined via the call of a scheduling or discovery service.

Current BPEL implementations do not provide features for scheduling service calls depending on the load of possible target hosts. When computationally intensive processes are running simultaneously on a dedicated infrastructure, either the response times will increase or the invoked services even might not react at all, resulting in a potential abandonment of the entire process. This may lead to a loss of stability of the workflow system. Moreover, the loss of intermediate results causes wasted CPU hours thereby resulting in higher costs. The system can be stabilized and the response times can be decreased by adding additional hosts on-demand. Since Cloud computing infrastruc-

tures allow to dynamically provide hosts with custom software installed, such an infrastructure can be used to react to peak loads or even to replace dedicated infrastructures. In this scenario, resources only have to be paid for when they are indeed used. Administration complexity is reduced to a minimum.

Cloud computing is a style of distributed computing in which resources are provided in form of services. Users may access resources on-demand (mainly computational power and storage) via simple interfaces like web services, having the knowledge or control of neither the technology nor the infrastructure supplied by the provider. For the implementation, many Cloud computing providers like Amazon make use of virtualization. Thereby, more than one virtual machine can run simultaneously on a dedicated host, user isolation is possible and provider can give the customers full (root) access to the (virtual) machines. Root access is often necessary in order to install and configure software and to open the Cloud infrastructure for a broad variety of applications. In the Amazon Cloud infrastructure, users can configure virtual machine images with customized operating systems and installed user applications that are stored in the Cloud infrastructure. On request, such a virtual machine is booted on a physical host of the infrastructure and it can be used like a dedicated physical host shortly after the request. The price per slot depends on the requested configuration of the virtual machine (number of CPUs, amount of RAM, instance storage).

Besides the aforementioned basic activities, like invoke or assign, BPEL also offers a rich vocabulary for error handling within a BPEL process. Since a BPEL workflow declaratively describes the business logic, fault and compensation handling have to focus on faults that are associated with this logic, i. e., business faults. In contrast, *infrastructural failures* like network timeouts and server outages must not be handled using the language mechanisms, since the composition logic would thereby be cluttered with non-business logic related aspects. Consequently, classes of faults have to be identified that can be handled automatically and a policy language for the automatic configuration of recovery behaviors without requiring additional explicit fault handling mechanisms of the BPEL process has to be defined. Therefore, an automatic Cloud-based redundancy of services to allow substitution of defective services should be provided.

Even the very simple example in Listing 4.3 already illustrates that the process logic is heavily bloated by fault handling and so far it only cope with the faults of a single invoke. Process that contain more than one invoke, this fault handling has to be repeated for each invoke. Faults that are related to the logic of a process (i. e., faults explicitly thrown by the invoked service, e. g., the depicted *buy:CreditCardNotApproved*) have to be handled explicitly within the process, since they clearly influence its logic. The process must be able to react to these faults – for instance, by executing a special branch.

At the same time, it should be possible to handle faults related to *infrastructural* errors (e. g., network timeouts and software faults in invoked services) without further interference with the BPEL process. Especially during the execution processes with many *invoke* operations, it is quite likely that at least one service is not available or even malfunctions. Without fault handling for every single *invoke* operation, the failure of just one service would already cause the failure of the entire process. Given a (simple) process with $n = 10$ services and a failure probability l of 0.01 (1%)

```
<faultHandlers>

  <catch faultName="buy:CreditCardNotApproved" faultVariable="Fault">
    <!-- make a callback to the client -->
    <invoke partnerLink="Client"
            inputVariable="Fault"
            portType="buy:ClientCallbackPT"
            operation="ClientFaultCallback" />

  </catch>

  <catchAll>
    <sequence>
     <assign>
       <!-- create the fault variable -->
       <copy>
         <from expression="string('Other fault')" />
         <to variable="Fault" part="error" />
       </copy>
     </assign>
     <invoke partnerLink="Client"
             inputVariable="Fault"
             portType="buy:ClientCallbackPT"
             operation="ClientFaultCallback" />

    </sequence>
  </catchAll>

</faultHandlers>
```

Listing 4.3: Manual fault handling in BPEL. Fault handler are declared per container, thus a BPEL process can contain several fault handler. In order to retry single web service invocations, each web service invocation has to be embedded into a single scope with its own fault handler.

for each service, the likelihood that this process finishes successfully only amounts to: $(1-l)^n = 0.99^{10} \approx 0.904 \approx 90\%$, i. e., on average one out of ten processes fails.

For many of these infrastructural faults, a straightforward strategy for recovery exists. For example, if a network timeout occurs, as a first step the invocation could just be retried several times (with a maximum number of attempts). If this does not solve the problem, as a next measure the service could be substituted by an equivalent one. This substitution could also be retried several times – leading to an eventual failure of the process if none of the attempts succeeds. To make a solution as flexible as possible, the recovery behavior needs to be configurable. There might be, for instance, cases in which a retry does not make sense at all or the maximal number of attempts for substitution has to be set to a specific value, cf., Figure 4.2.

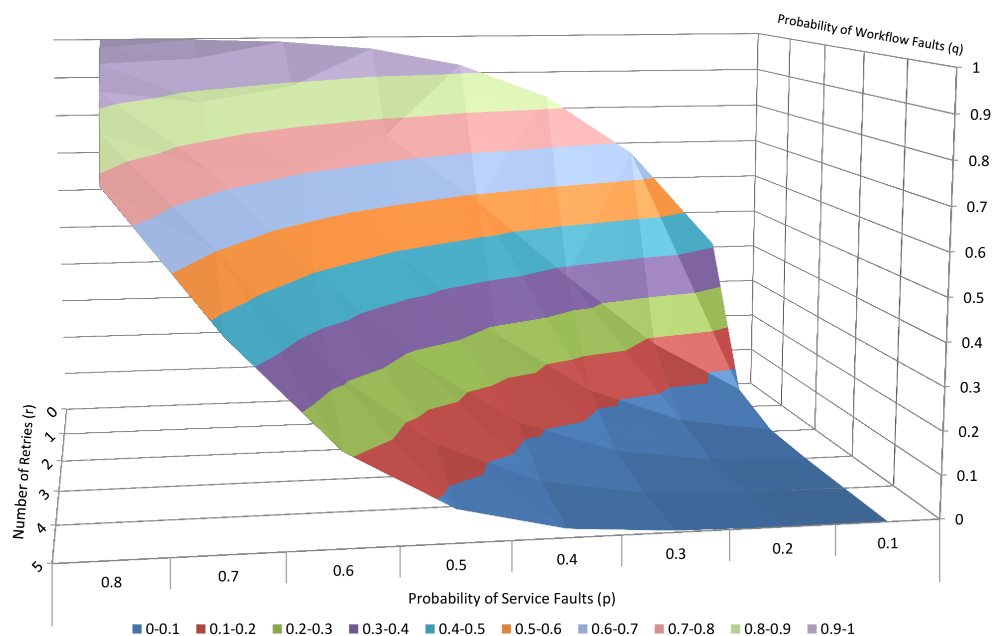


Figure 4.2: The graph shows the probability q that a workflow consisting of five service invocations fails, depending on the probability that a service invocation fails (p) and the number of retries in case of a failed service invocation (r).

The BPEL4Grid workflow engine provides a production-stable workflow system. Since this workflow system is well-documented and provides publicly accessible plug-in mechanisms for the integration of different invoke mechanisms (so-called *invoke handlers*), it is a reasonable choice for the workflow component. In combination with the open-source license of the engine, this engine is preferred for further development. The development should concentrate on avoiding any changes

of the BPEL language and its standard. Otherwise, developed workflows would be bound to a very specific – modified – engine in order to be executed. Furthermore, runtime information like the concrete binding of web service invocations to web service endpoints are not described in the BPEL process itself, but in a – separated – deployment descriptor (file). The latter one is able to utilize user-supplied invoke handlers and which is the preferred linkage between the workflow engine and the load balancing and fault handling mechanisms.

4.5 Handling of Cross-Cutting Concerns

In contrast to the aforementioned concerns that clearly affect the side of the workflow engine (e. g., load balancing or fault handling) or the service side (e. g., security and authentication), in the following, issues related to efficient data transfer in a large-scale service-oriented platform for content-based search in image and video databases [63, 64] as motivated by the multimedia data analysis use case (cf., Section 2.2.1) are elaborated.

One of the problems in the considered use case is concerned with the sequential invocation of two subsequent services in a BPEL workflow, one of them producing (a potentially large amount of) data which is then consumed by the other one. If the data flow follows the control flow, the data are transmitted via the composition client (the BPEL engine). This is neither necessary nor desirable: the available network bandwidth of the machine hosting the engine that invokes the services can very quickly become a bottleneck and the runtime performance will decrease significantly.

One way to circumvent this issue is to prevent the transfer of huge data via the BPEL engine by using Flex-SwA: instead of immense binary data, only a small reference is transferred from the data-producing service to the composition client and then passed on to the data-consuming service. The latter one resolves the reference and obtains the binary data. However, this technique requires that the involved services are prepared to consider that the references are direct arguments and the return values of the services, respectively.

If the arising bottleneck in the BPEL engine is not already considered at the design or the implementation time, it is a fairly difficult and time-consuming task to implement the needed changes for Flex-SwA later on. Both a modification of the services (explicitly using references as input/output arguments and the corresponding handling) and an adaptation of the workflow (due to changed interfaces on the service side) are required. This not only induces an overhead due to workflow modifications and redeployments, but in addition the composed workflow suffers from a downtime during the redeployment process.

As indicated by the preceding description, modularity and decoupling of web services thus are afflicted with scattered code for data handling. However, it is preferable to have a solution that can cope with efficient data transmission in data-intensive service workflows without noticeable additional development efforts and without losing modularity and decoupling of services. If other data-related concerns have to be implemented, e. g., a secure data exchange, the modifications as sketched above have to performed – for each concern and for each participant in a service-oriented architecture: service, client, and composition engine, separately.

For the given reasons, the system must be able to address cross-cutting concerns not only at the BPEL engine side but also at the web service side. In order to achieve this, the principles of aspect-oriented programming are adopted to web services in the context of WSDL and SOAP. The integration of cross-cutting concerns preferably takes place without any complex changes – ideally no changes – to any middleware are necessary in order to address cross-cutting concerns. The solution to be proposed must be powerful enough to tackle the issues touched on above.

4.6 The Blueprint

In this section, the fundamental design will be elaborated that enables a runtime adapation of service workflows in service-oriented architectures. As indicated by the use cases and the requirements analysis, addressing such requirements needs support on different levels in a service-oriented architecture. These levels include the workflow engine, the services as well as the client.

The implementation will be founded on the BPEL editor DAVO and the BPEL4Grid workflow engine. Both will be subject to different extensions in order to be able to deal with runtime adaptation of scientific workflows. In Figure 4.3 the blueprint of the design is shown. Besides showing the interdependence of the different components, the figure accentuates where the different types of non-functional requirements are handled.

The proposed blueprint comprises the development time of a service-oriented architecture. Existing software has to be prepared as services. Often, this involves legacy software that cannot be rewritten due to time- or cost-restraint and need to wrapped as (web) services. Since often multiple software parts exist that need to be ported as a web service, the *LCDL framework* provides a generic integration of legacy code into service-oriented architectures. When the building blocks needs to be combined and orchestrated in order to fulfill a more complicated and valued-added task, the modeling of workflows typically involves two roles: a workflow expert (due to the complexity of the technique) and a domain expert (due to the complexity of the target domain). To harmonize both roles, the *SimpleBPEL approach* separates the design of basic tasks that are assigned to the workflow expert and delegates to overall design of workflow (without the need to dive in technical details) to the domain expert.

At runtime, several unforeseen challenges can arise if the expected number of executions of a workflow significantly differs from the actual one. Two typical scenarios arise. 1) If the expected demand was too high, to many resources are idling and are therefore causing cost and other expenses. 2) If the expected demand was too low, the used resources may suffer from overload-situations causing service discontinuation of service executions and in consequence a negative user experience. Automatic scaling (i. e., scale-out and scale-in) will help to adjust the number of available resources according to the actual demand by using *Infrastructure-as-a-Service* by Amazon EC2 [6]. By respecting the data transfer between the orchestrating workflow engine and the participating services the principles of data locality. By establishing a infrastructural fault handling the mix up of business logic and infrastructure within a given workflow description can be prevent and the overall increase of quality-of-service parameters of a workflow can be increased.

Communication-related requirements are handled at the communication level by applying techniques from aspected-oriented software development. By using *request/response aspects*, the communication of two services and partners in a service-oriented architecture, respectively, can be addressed without the necessity of manipulation of the implementation of the communication partner. Moreover, applying a communication-level approach totally avoids the presence of the partner's source code.

4.7 Summary

In this chapter the architectural blueprint of this thesis has been elaborated and presented. Taking the deduced requirements into account, which in turn are derived from (academic) use cases and generalized, this design provides a framework that addresses runtime adaptation of service workflow executions in cloud environments. Important non-functional requirements from the area of quality-of-service (e. g., reliability and scalability) or usability (e. g., simplified modeling) are identified and their general handling is explained. A more detailed elaboration of the different components will follow in the succeeding chapters.

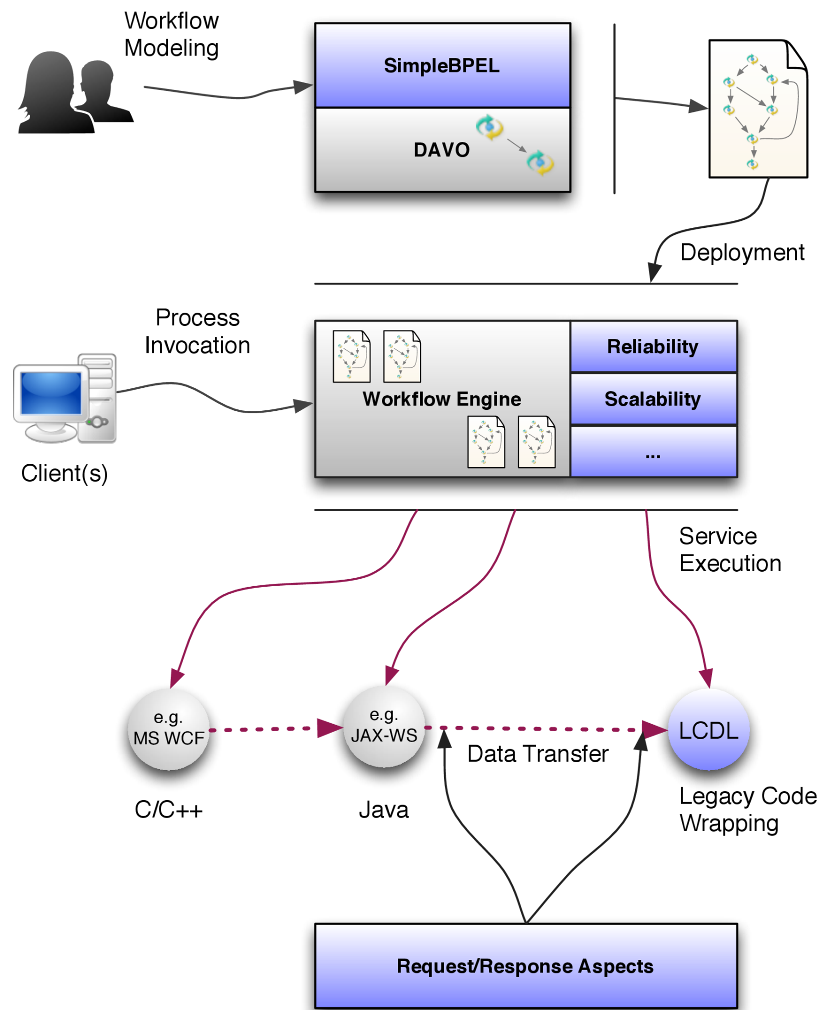


Figure 4.3: Bird's eye view on the architecture that allows to address runtime adaptation in a service-oriented architecture.

5 Service Development

Contents

5.1 Introduction **63**
5.2 Design of the Legacy Code Description Language **64**
5.2.1 Framework 64
5.2.2 Legacy Code Description 64
5.2.3 Runtime Support 67
5.3 Implementation of LCDL **70**
5.3.1 Modeling 70
5.3.2 Model Processing 70
5.3.3 Runtime 72
5.3.4 Extensibility 73
5.4 SimpleBPEL Design **74**
5.4.1 Extensions to DAVO 74
5.4.2 SimpleBPEL Composer 75
5.5 Implementation of SimpleBPEL **79**
5.5.1 Extensions to DAVO 79
5.5.2 SimpleBPEL Composer 81
5.6 Summary **83**

5.1 Introduction

Adaptation is an important topic not only during the runtime of service-oriented architectures, but also during their development time. While runtime adaptation seems natural from a technical point-of-view, adaptation during development time is of great importance for the general acceptance of users as well as developers. Such requirements that are manifested during the development time can be versatile. In service-oriented architectures, two issues typically need to be considered. First, the building blocks (i. e., the services) have to be created, and second, they need to be composed or integrated into a value-added composition or application.

This chapter addresses such requirements during the development time. Different solutions for service modeling and modeling of compositions are introduced.

Parts of this chapter have been published in [111, 115].

5.2 Design of the Legacy Code Description Language

This section presents the design of a framework that mediates legacy code to different target platforms. As it has been stated in Section 4.3.1, the invocation of legacy code generally follows a common pattern, which is used to develop a declarative description of legacy code. This description can cope with the different peculiarities of legacy code, e. g., depending on its type (binary or library/Windows or Unix) different methods for invocation are needed. Moreover, the description should be extendable by additional concepts of legacy code and data types. Based on the given description, the actual wrapper code (i. e., a façade) is generated. Therefore, the description is independent of any implementation details for the wrapper code.

5.2.1 Framework

Legacy code commonly offers multiple functionalities that are typically represented by one or several of its methods. Each of those might require different input parameters and generate a different output. All methods are wrapped within a specific wrapper that is determined by the type of the binding. At least one binding must be specified, but there can exist more than one binding in a model. If this is the case, different wrappers are generated, each one containing all specified methods.

The description is based on the model shown in Figure 5.2. An instance of this model is taken and transformed into executable wrapper code. The model shows which information is needed to – declaratively – describe legacy code, in both binary and library form. As stated before, one of the main goals of this design is its extensibility, including new types of bindings, new inputs and new output sources and types.

In order to tackle this issue, wrapping legacy code comprises two components: the legacy code description (based on the legacy code description language "LCDL") and tools that support users in creating such descriptions and runtime APIs (see Figure 5.1). The individual parts of this framework are described in more detail in the following. Without loss a generality, the usage of the different components is elaborated using the legacy code of the algorithm for speaker clustering from the multimedia data analysis use case (cf., Section 2.2.1).

5.2.2 Legacy Code Description

As depicted in Figure 5.2, the main element of a legacy code description is its `Service` element. It is the container for all `Operation`s and `Binding`s. The first ones represent methods of the legacy code. They are identified by a symbolic name which other elements can refer to and possess an `Execute` element as a generalization of the `Library` and `Binary` type of legacy code. These types

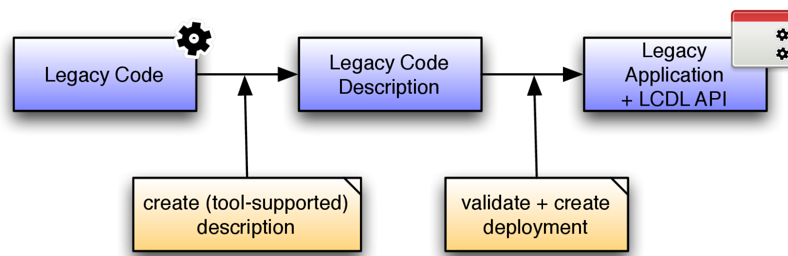


Figure 5.1: Conceptual design of the wrapping of legacy code.

require the path information of the legacy code. In some cases, it is necessary to specify additional auxiliary information for the `Binary` type.

It might be insufficient to pass input arguments directly to the binary code, since first a parameter specification like *-preEmphasizeFactor=0.5* is needed. Because several conventions for passing arguments via a command line interface exists, this variety has to be mapped to the model. Therefore, a `Parameter` element has the attributes `prefix` and `infix`. The first one maps *-preEmphasizeFactor* and the second one maps the equal sign. Depending on whether the infix attribute is a whitespace or not, the two parts of the parameter have to be passed as two separate or (together with the infix value) as one single argument. This distinction is required for `Binary` code.

`Input` arguments are determined by a type attribute, referring to an XML Schema Type [191] that in turn is identified by its qualified name (qname). To reflect the call-by-reference pattern, the mode attribute can be used to set an input parameter either to `in`, `in-out` or `out`. This task is taken by the binding has to take care of handling this correctly. If it does not succeed, a validation step must fail and report this deficiency.

As `Input`, the following specialized elements are realized. The `ElementInput` models a parameter that takes an arbitrary argument with a given type. An `OptionInput` models an argument, whose value can be optionally be set by the caller.[1] In contrast to this element, the `StaticInput` models an input argument that is always set and whose value is predefined in the model instance. `FileInput` models a file as an input parameter. The concrete treatment (i. e., signature in the façade and handling in the generated code) of this input type is determined by the `Binding`.

The `Output` consists of two elements. One is the `ReturnSource` element that declares the source of the return value. Common sources are standard out (`StdOutSource`), standard error (`StdErrSource`), the return code of a binary (`ReturnCodeSource`) or a file that has been created by the legacy code (`FileSource`). Sometimes, legacy code prints its information on standard out as well as on standard error, without using the standard semantic of standard error (e. g., FFmpeg [70]). In this situation, the `StdComposite` can be used. This one merges standard out and standard error and ignores the latter one's semantics. Besides the `ReturnSource` element, the `Output`

[1] Note that not all bindings support optional arguments. If the binding is not able to map such parameter passing correctly (e. g., the chosen binding does not support optional arguments or polymorphism of methods), the validation step has to fail and report this error.

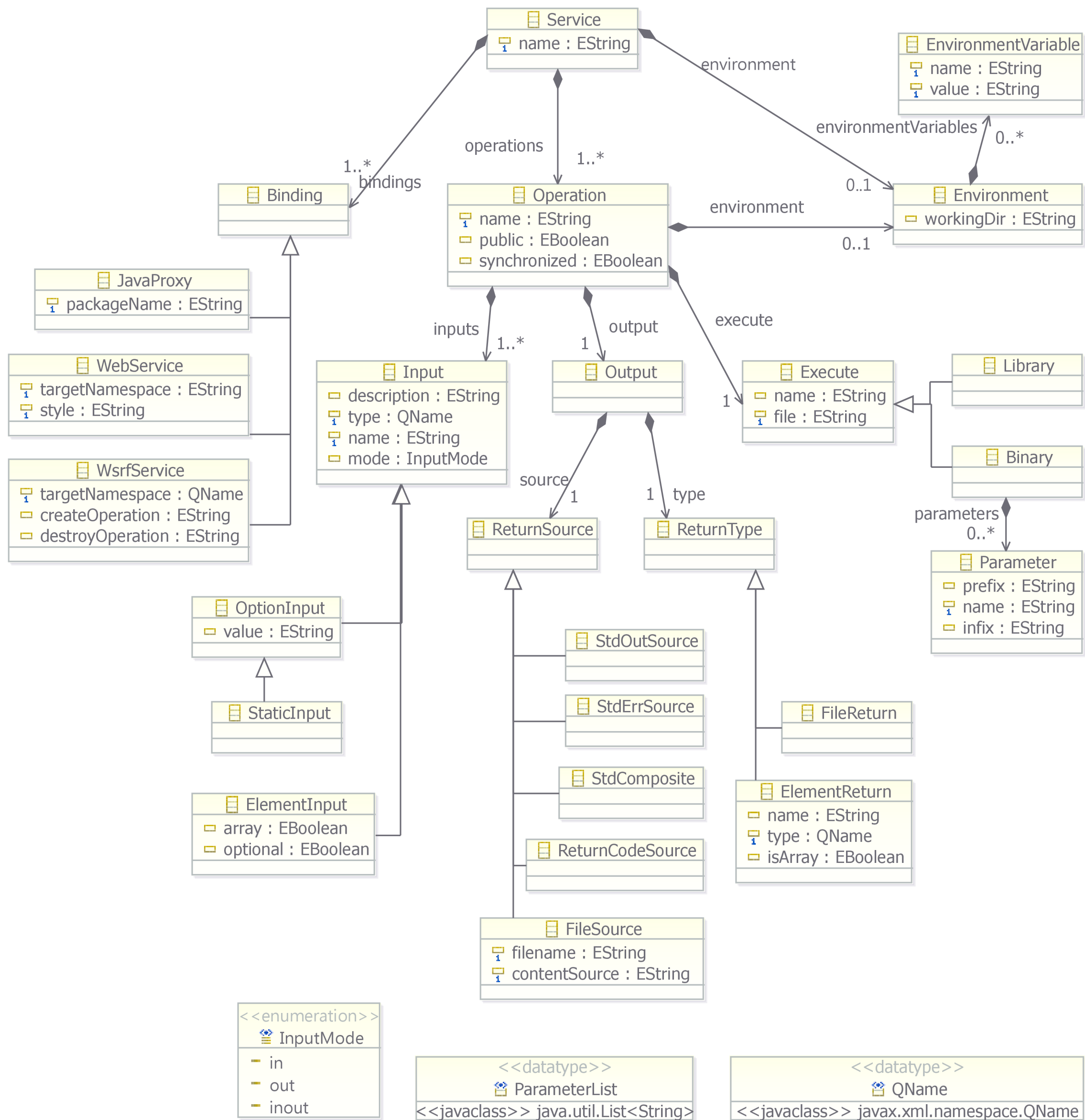


Figure 5.2: UML Diagram of the LCDL. The top entity is formed by the `Service` element that contains several `Operations`. Each one of these is mapped to a functionality provided by the given legacy code. The façade of the wrapper legacy code is determined by the `Binding` entity. Additional entities provide (optional) information about environment variables or specify which type of input and output have to be handled by the framework.

is endowed with the `ReturnType` element. It defines how the `ReturnSource` data is passed back, i. e., returned. In most cases, the `ElementReturn` is used for the `ReturnType`. For the Java binding, this is the return type of a method. Another type is the `FileReturn` that turns the return type into a file type. As in the input case, the concrete return for the `FileReturn` depends on the `Binding`. For example, the `JavaProxy` uses the *java.io.File*, whereas the `Webservice` uses a Base64 encoded string. Additionally, an `Operation` can be equipped with an `Environment` containing information about a working directory or an environment variable. The latter one might have to be set in order to execute the legacy code, like *LD_LIBRARY_PATH*.

In any field, XPath expressions can be used in order to reference attributes in other elements within the same model instance. This allows for reusing already available information within the same model instance.

The `Binding` element, which can occur multiple times within each `Service` element, defines the façade of the legacy code. So far, the LCDL model has neither any information nor any references about the type of wrapper of the legacy code. The mapping is specified by the `Binding` element. A `JavaProxy` binding generates a Java interface for the legacy code, whereas a `WebService` binding generates a web service (for a specific web service container).

As already emphasized, the extensibility of this framework is an important issue. In Figure 5.3, an extension of the `Input` is shown. As a new input type, Flex-SwA is introduced. It serves as an input type for the wrapper code in order to manage the data handling. When using Flex-SwA instead of directly passing huge binary objects, only small references have to be exchanged. In conjunction with service orchestration, this allows for speeding up the whole application, because the frequently emerging bottleneck at the orchestration node is prevented. As its only drawback, those references require a special treatment in terms of file acquisition and reference exposition. The generated wrapper has to take care of this.

Another possible extension of the binding types is shown in Figure 5.4. Besides web services and WSRF services, it creates a wrapping of legacy code as a RESTful service [71]. RESTful services provide an alternative paradigm for communication with stateful resources in a distributed system. As web services or WSRF service, their primary use is for machine-to-machine communication for which the methods of the HTTP standard (`GET`, `POST`, `DELETE`, etc.) are used. Since RESTful services match resources with URLs and defer activities to the HTTP activities, the operations of the legacy code are mapped to to the HTTP methods and the service (i. e., its name) becomes a part of the URL of the RESTful service.

All necessary information needed to generate a RESTful wrapper are gathered by the *RESTfulService* entity depicted in Figure 5.4.

5.2.3 Runtime Support

The runtime of an LCDL model consists of three phases: *modeling*, *validation/code generation*, and *execution*. A convenient editor must be provided for the modeling step. Such an editor has to incorporate the extensibility that is given by the LCDL model. In turn, the editor must provide an

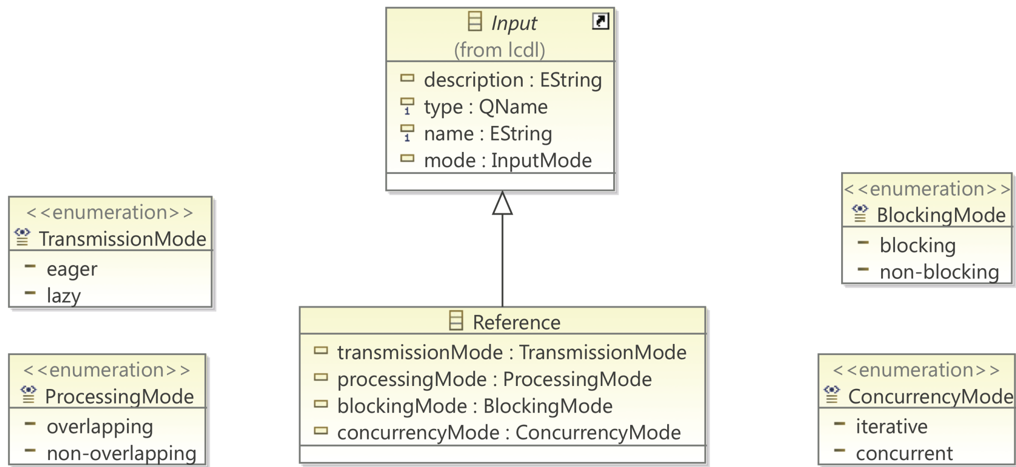


Figure 5.3: Extension of the `Input` type by a Flex-SwA element. The `Input` element is imported from the *lcdl* ecore model. All necessary configuration data for the new Flex-SwA element is also defined as enumerated elements.

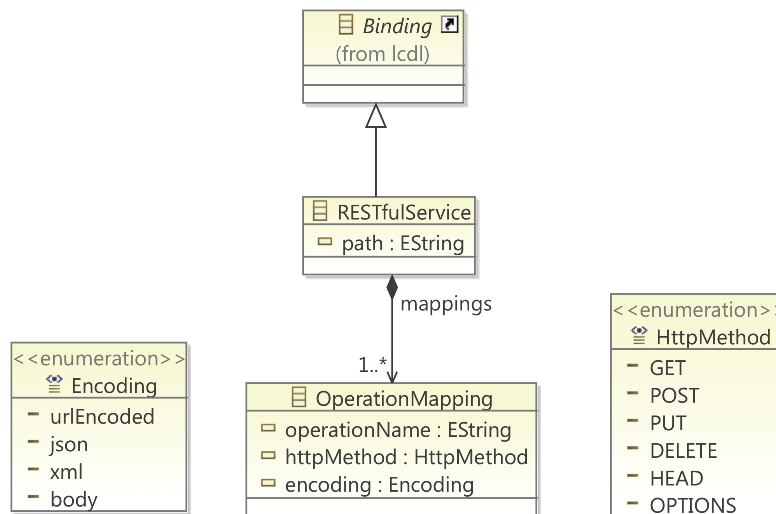


Figure 5.4: Extension of the `Binding` entity by a RESTful service element. In this variant, the legacy code becomes available via the URL of the RESTful service. The operations of the legacy code can be called via traditional HTTP methods.

extensibility mechanism offering graphical modeling support. Mechanisms to validate the modeled instance also have to be integrated into this editor. In case of a failed validation, useful information about the occurred errors have to be provided. Ideally, hints how to correct these errors are also provided. The validation can be optionally triggered by the user or it can be forced before the actual code generation process starts. The latter one is also a mechanism for triggering the code generation process form within the editor.

Again, this mechanism must be extensible in order to integrate given model extensions into the runtime deployment. Depending on the specified binding within the LCDL model, different deployments have to be created. For example, if a web service binding is chosen, a web service archive (a `war` file) has to be compiled. Such an archive file must contain almost all necessary files in order to run the legacy as a web service. Excluded are the legacy code itself and possible shared libraries needed by an LCDL runtime. The archive includes all generated wrapper classes (that are automatically created and compiled) such as all necessary meta data files. On the other hand, shared libraries should not be included in such an archive, because in the case of multiple deployments the – commonly used – library is deployed multiple times and thereby class-loading conflicts can occur.

The activity of LCDL during runtime as developers have to interact with LCDL if they want to integrate generated wrapper code into their application is depicted in Listing 5.5. A factory class is queried that resolves the concrete implementation. Afterwards, this implementation executes the generated code and perform calls of legacy code.

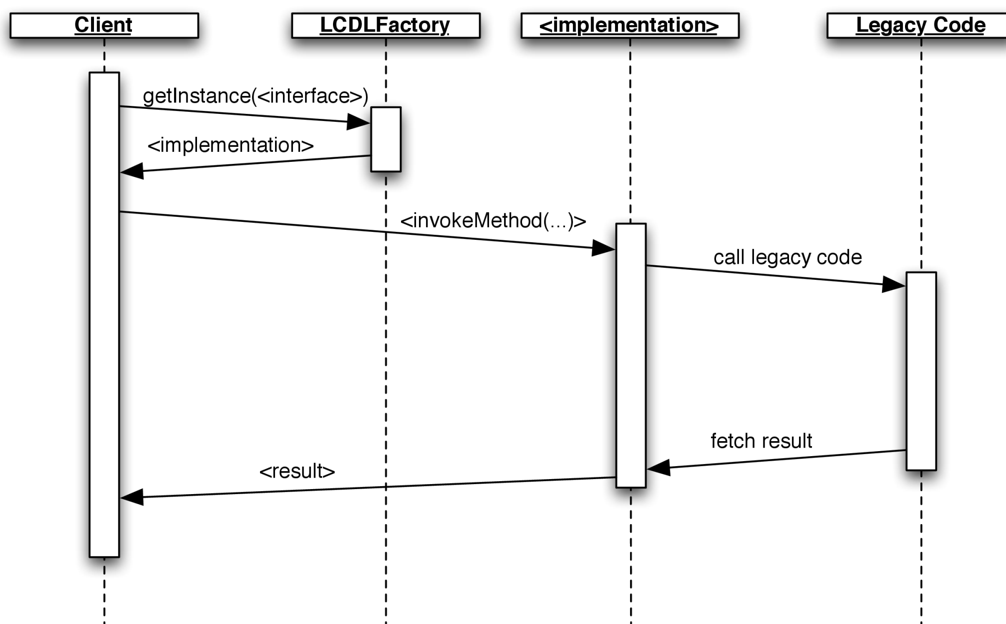


Figure 5.5: Runtime behavior of LCDL seen from a developer's perspective.

5.3 Implementation of LCDL

This section discusses the presented implementation of the LCDL framework. After the detailed description of the implementation of the LCDL model, implementation details for the runtime support are presented. Finally, the extensibility mechanisms are exemplified by the introduction of a new binding type.

5.3.1 Modeling

Modeling in the context of LCDL is twofold: the model LCDL itself and an instance of the LCDL model for each single legacy code.

The LCDL model is based on the UML model shown in Figure 5.2. The Ecore plugin of Eclipse is used to generate data model files and a corresponding editor. The generated code is extended to cope with the required extensibility, i. e., extension points are defined and queried during runtime in order to load and execute necessary extensions.

While the aforementioned concerns are only important for the development of the LCDL, the generated editor is more important for the user of the LCDL (i. e., typically for the developer). It serves as the interface for compiling models, for validating them and for performing the code generation process. Using the Ecore data model, an instance of the LCDL model can also be programmatically created. In this way, LCDL can also be embedded into other frameworks and applications.

5.3.2 Model Processing

The UML model depicted in Figure 5.2 serves as input for the Eclipse Modeling Framework (EMF) project [52]. Supported by a structured data model, like the given UML model, EMF serves as a code generator and can build the aforementioned tools. The EMF plug-in permits a graphical modeling of UML diagrams as well as the generation of the corresponding model classes and a simple table-based editor (as well as test classes). In addition to this editor, EMF creates a (basic) validator that is capable of checking the (syntactical) constraints given by the UML diagram.

Based on this approach, a prototypical implementation has been developed. EMF is used to generate model and adapter classes of the UML model. EMF provides a basic editor that enables users to graphically generate an instance of an LCDL model. Moreover, a validator is provided by EMF. It examines and enforces the constraints given by the UML model.

Operations

To generate a Java interface, all `Operation` elements are mapped to Java methods and the XML Schema types of the input and output types are mapped to their corresponding Java types, respectively. This mapping is based on the mapping defined in Apache Axis [12]. Finally, the generated interface is annotated with the XML model file, and both the model file and the compiled interface are packed into a common Java archive (jar) file.

For library calls, the *Java Native Access* (JNA) API [107] is used. It offers a platform independent way of calling library code and – in contrary to the Java Native Interface (JNI) [108] – JNA works on a given library, on top of which the adequate Java code can be generated. Anyway, JNI takes a Java class, which is prepared with special keywords (*native*), and generates a header file for C/C++, which in turn has to be implemented. Subsequently, in the extreme case, an appropriate JNI wrapper has to be implemented for each single Java wrapper. The use of JNA together with the LCDL description avoids this recurring (and costly) activity.

The generated code contains all necessary code to convert input and output arguments of the Java method so that the requirements of the legacy code are satisfied. Listing 5.1 shows the pseudo-code version of the compilation process. The compiler class uses the abstract syntax tree (AST) for Java provided by the Eclipse Java Development Tools (JDT). This AST supports the complete Java language and is used in the back-end of the Java editor and is also used by refactoring tools [54]. Moreover, the correct inclusion of imported (Java) packages or the like is simplified using the JDT's AST.

```
generateImplementation:
  createAbstractSyntaxTree();
  AST ast = createPackageStructure();
  addConstructor(ast, ...);
  foreach(operation : Service)
    Method method = createMethod(ast, operation)
    executionType.fillMethod(ast, method)

fillMethod:
  foreach(input : method)
    addToMethodSignature();
    inputType.convertToLocalType();
  setupWorkingDirectory();
  setupEnvironmentVariable();
  createExecutionCode();
  generateReturnTarget(returnSource);
```

Listing 5.1: Pseudo-code for the generation of wrapper code.

Bindings

Depending on the `Binding` elements present in a service model instance, different types of code generation are triggered. The `JavaProxy` creates a Java interface within the specified package, whereas the `Webservice` creates an Axis web service [12] with the given target namespace and the RPC or document style. If the legacy code has a state (e. g., libraries that need to be initialized with certain parameters), the usage of a `WsrfService` binding is required. Unlike web services, which represent stateless resources, WSRF services have a state, which is preserved between invocations. One of these might be an initialization and the other one(s) might constitute the actual call. The create and destroy operations of the factory pattern are reflected by the `createOperation` as well as the `destroyOperation` attribute of the `WsrfService`. In turn, they reference to the symbolic

name of the operation. The framework is thereby enabled to create a WSRF service for several plattforms, e. g., the Globus Toolkit [73] that is capable of deploying and executing WSRF services.

5.3.3 Runtime

The runtime behavior of LCDL depends on the selected binding in a given LCDL model. In the following, the behavior of the LCDL framework is exemplified by a call of the `JavaProxy`.

A first approach uses the Java Proxy technique[2]. Method invocations that are performed on the generated interface class are redirected to a single proxy instance. Subsequently, this proxy instance parses the invocation details (the name of the method called, arguments passed, etc.) and searches for a matching operation within the LCDL model. Based on the found operation, the legacy code is executed and the return values are passed back to the initial caller. Although this approach allows for a dynamic creation of mediating code, a statically compiled wrapper is more lightweight because it does not make use of Java reflection technology to mediate method invocations. Moreover, since all method invocations are known (and specified) during design time of the LCDL model (i. e., only operations specified in the LCDL model can be called), the overhead induced by the Proxy architecture – although it is comparatively small [89] – can be circumvented. For these reasons, a statically compiled implementation of the user visible interface is used.

To provide a lightweight API, the user (i. e., the programmer) has to interface only the LCDL factory, see Listing 5.2. This factory provides a `getInstance(...)` method (see Line 8), which takes an LCDL generated (and annotated) interface as its input. Based on this input, the interface's annotations are parsed (mainly the `impl` annotation, Line 11) and the corresponding LCDL model file is loaded (Line 18). The annotations contain information about the actual implementation of the legacy code calling-class, which is loaded by using Java Reflection. Moreover, this loaded class implements all methods that are specified in the previously mentioned interface. Instead of using a Java Dynamic Proxy, the matching method between the interface method and the method of the implementing class is done by standard Java mechanisms. Eventually, a method that contains the aforementioned code is called in order to execute the legacy code.

```
package de.fb12.lcdl.runtime.java;
2
import de.fb12.lcdl.annotation.LcdlAnnotation;
4 import de.fb12.lcdl.exception.LcdlRuntimeException;

6 public class LcdlFactory {

8   public static Object getInstance(Class<?> interfaceClazz) throws ↩
    ↪ LcdlRuntimeException {

10     LcdlAnnotation annotation = interfaceClazz.getAnnotation(↩
       ↪ LcdlAnnotation.class);
       String implClazzName = annotation.impl();
```

[2]Note that the term *Java Proxy* does not refer to HTTP proxies that are used to cache web sites. Instead, it refers to Proxy classes of Java that provide generic implementations for arbitrary interfaces. More details can be found in [100, 151].

```
12
        if (implClazzName == null || implClazzName.isEmpty()) {
14        throw new LcdlRuntimeException("wrapper class not specified.");
        }
16
        try {
18        Class<?> clazz = Class.forName(implClazzName);
          Object instance = clazz.newInstance();
20        return instance;
        } catch (Exception e) {
22        throw new LcdlRuntimeException("unable to instantiate wrapper", ↩
          ↪ e);
        }
24    }

26  }
```

Listing 5.2: Factory class of LCDL. This class is used by a developer who wants to integrate legacy code into her or his application. The `getInstance` method takes the generated interface class and returns the (also generated) implementation by using annotations in the interface class and by means of Java reflection.

5.3.4 Extensibility

The specified extensibility is achieved by a double-step approach. In the first step, the LCDL model is extended – as depicted in Figure 5.3 or Figure 5.4. The second step is the enrichment of the model generation process. This is achieved by the usage of specific extensions points – which rest on the extension point mechanism of the Eclipse platform. For the different extensions of the LCDL model (`Input`, `Output`, `Binding`, `Execute`, etc.) corresponding extension points exist and are queried during the code generation process. For this reason, each available extension implements the specified extension points.

For example, the JavaProxy implementation declares itself as an extension of the `Execute` entity. Referring to the code generation procedure in Listing 5.1, the generator queries the extension registry of Eclipse in Line 7 and determines the precise implementation that matches with the specified `Execute` of the LCDL model.

In order to define a new `Binding`, like the binding for RESTful services sketched in Figure 5.4, the LCDL compiler has to be extended with a specific generator for these services. In Listing 5.3 the prototypical code for such a generator is shown. The `RestfulServiceGenerator` implements the `IGenerator` interface. Via the extension point mechanism, the RESTful services plug-in extends the extension point for bindings and registers the `RestfulServiceGenerator` for the `RestfulServices` binding. The LCDL compiler queries the Eclipse extension registry and calls the `generate` method. This method uses the `JavaGenerator` that is primarily applied for the generation of the JavaProxy implementation (Line 8 in Listing 5.3).

To provide RESTful services, the Restlet framework is employed [143]. For each method of the LCDL model, the corresponding HTTP method, which should be used for access, has to be specified. Although an automatic mapping of all legacy methods to, e. g., `GET`, is possible, this would neglect the semantics of the HTTP methods and it would thereby ignore an important principle of the RESTfule service communication. Thus, each method of the service has to be annotated with the HTTP method (e. g., `GET`, `PUT`, etc.) from which it should be accessible (Line 11). Then, supporting classes that handle the interaction with multiple resources – so-called applications – are created (Lines 12–13).

```
public class RestfulServiceGenerator implements IGenerator {
2
  @Override
4  public void generate(Service service, Binding binding,
    IFile modelFile, IPath outputLocation) throws ↩
    ↪ LcdlCompilerException {
6
    RESTfulService serviceBinding = (RESTfulService) binding;
8    JavaGenerator jg = new JavaGenerator();
    jg.generate(service, javaProxyBinding, modelFile, outputLocation);
10
    prepareInterface(jg.getInterface(), serviceBinding);
12    prepareImplementation(jg.getImplementation(), serviceBinding);
    prepareRestApplication(jg.getProject(), serviceBinding);
14  }

16  ...

18 }
```

Listing 5.3: A new `Binding` for RESTful services.

5.4 SimpleBPEL Design

This section presents the design of a domain-adaptable approach that significantly simplifies the modeling of BPEL workflows. The key idea is to develop a graphical modeling tool that enables a domain expert to model his or her application without being distracted by any technical details. After the discussion of the modeling of function blocks (i. e., SimpleBPEL fragments, SBFs) within the existing workflow editor DAVO, the design of the SimpleBPEL Composer itself is described. Further implementation details can be found in Section 5.5.

5.4.1 Extensions to DAVO

Since SBFs are BPEL processes (with some constraints, as mentioned below) and can utilize the full expressiveness of BPEL, a suitable modeling tool for workflows is inevitable. It ideally should be

integrated into an existing BPEL editor such that a BPEL expert does not need to learn the handling of a new tool for modeling the SBFs.

The *Domain-Adaptable Visual Orchestrator* (DAVO) [46] is a graphical BPEL workflow editor that offers an adaptable data model and user interface. It is easily extensible to meet new requirements, and it is a suitable basis for the modeling of SBFs.

So as to assemble SBFs into a BPEL workflow, the fragments are subject to some constraints that have to be enforced by the modeling tool: (1) an SBF may not have a *receive* or *reply* activity, and (2) SBFs must have (one or more) distinguished input and output variables. Without restriction (1), an SBF would already be a complete BPEL workflow and could not be added as just a part of another workflow. The variables represent the input and output for a whole SBF and define its data exchange capabilities with other SBFs.

An SBF represents exactly one (part of a) BPEL workflow. Several SBF can be put together in one SimpleBPEL workflow. If, e. g., a media researcher models a video analysis, she or he wants to compose several fragments that belong to the same domain (such as face recognition, cut detection, etc.) into a workflow. For the aggregation of multiple SBFs of one domain into a library, so-called *Domain Profiles* are used. These basically represent a binning of SBFs into a single XML file, their syntax is shown in Listing 5.4. In this way, profiles can easily be shared among different developers/researchers. Since two different roles, typically represented by two different persons, are involved in the modeling of such profiles and their usage within a SimpleBPEL workflow, profiles must be exportable so as to be exchangeable between researchers.

```
<domainProfile name="NC_NAME">
    <process name="NC_NAME">
        <description>...</description>
        <processFilename>...</processFilename>
        <variable name="NC_NAME">
            <elementHint></elementHint>
            <type>in | out</type>
            <conditionContainer>
                <variable2MessagePartMapping />
            </conditionContainer>
            <description>...</description>
        </variable>+
    </process>+
</domainProfile>
```

Listing 5.4: Domain Profiles that are used to export SimpleBPEL fragments, informal syntax.

5.4.2 SimpleBPEL Composer

The SimpleBPEL Composer must be able to import Domain Profiles and to offer the contained SBFs to the domain expert who in turn can compose them to a SimpleBPEL workflow. Such a workflow consists of several function blocks that are internally represented by the corresponding BPEL fragments (see Figure 5.6). The graphical representation of an SBF has to reflect its input

and output variables so that the domain expert gets an intuitive view of all possible connections between the SBFs.

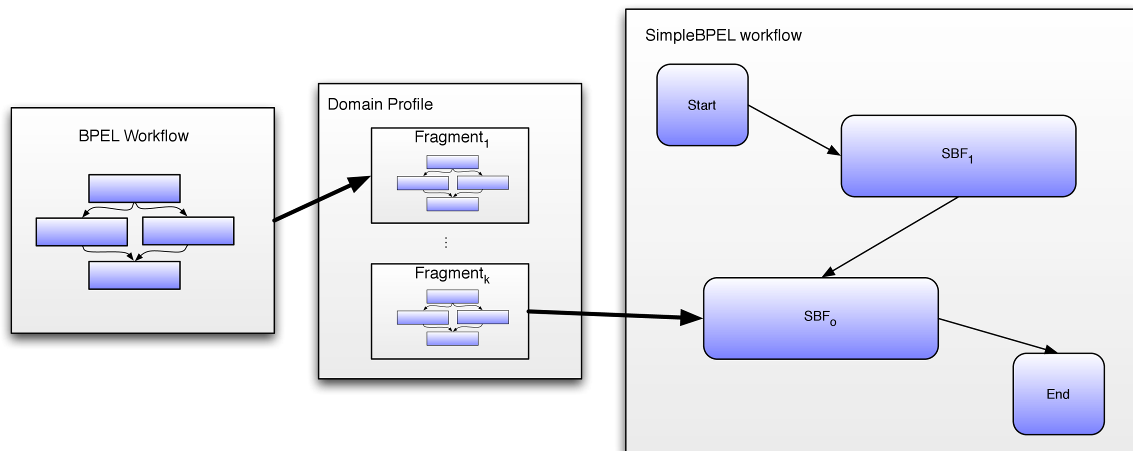


Figure 5.6: Relation between a BPEL workflow, Domain Profiles, and a SimpleBPEL workflow.

Using the Eclipse Rich Client Platform as basis, the SimpleBPEL Composer should clearly utilize standard techniques of the former one, i. e., available fragments should be arranged in a palette.

The model of the SimpleBPEL Composer essentially comprises of two types of elements. On the one hand, there are `ProcessModule`s that are represented by start, end, and function modules. On the other hand, function modules are interconnected via `VariableConnection`s. Each end of such a connection is represented by an input or an output variable of a fragment. The UML diagram of the data model of the SimpleBPEL Composer is illustrated in Figure 5.7.

Since not all connections between SBFs are reasonable, a validation has to be performed whenever the user attempts to connect SBFs. It has to be checked whether an assignment between two selected variables is reasonable and the result has to be presented in an adequate way. Since variables are based on WSDL messages that in turn are sets of parts, different validation strategies are conceivable. A part can either be a simple type (like an integer or a string) or a complex type that is comparable to a data structure in traditional programming languages. Thus, the validation can be performed on two different levels. The first one is to compare the qualified names (*qname*) of the messages themselves. Only if the *qnames* coincide, a connection is valid. The second one is to check whether all data types used within the messages conform with each other. In some special cases, where an in-order mapping is too strict, a bijective mapping between the fields of the message is more appropriate (Figure 5.8). However, since this mapping can be ambiguous (see ib.), it is only useful under special circumstances.

In order to be executable, a SimpleBPEL workflow first has to be transformed into a standardized BPEL workflow. The occurrence of equally named variables within different fragments may cause the overwriting of variables and thereby lead to conflicts. To guarantee the correctness of an exported workflow, it is essential to encapsulate the SBFs. For this reason, the BPEL code repre-

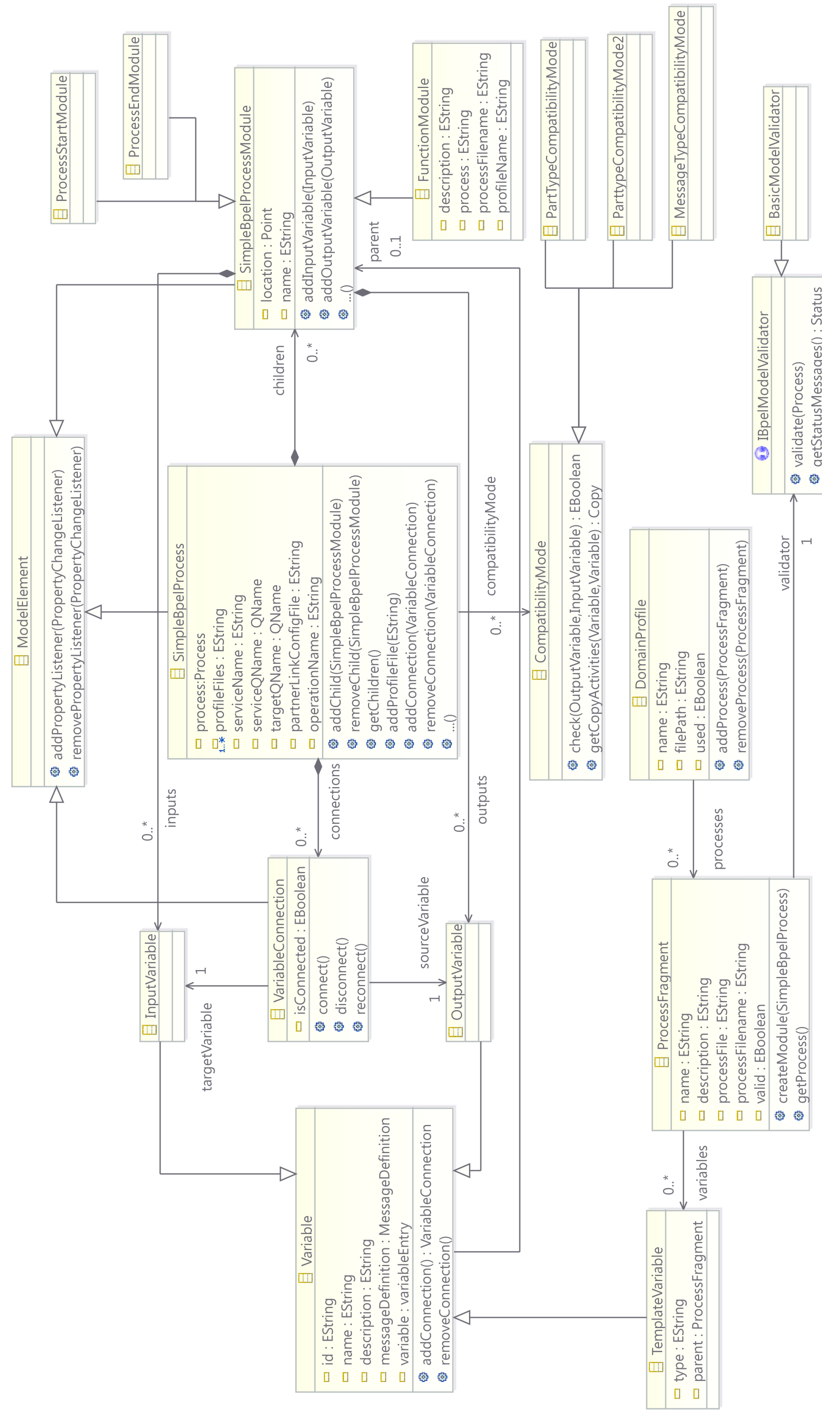


Figure 5.7: Data model of the SimpleBPEL composer.

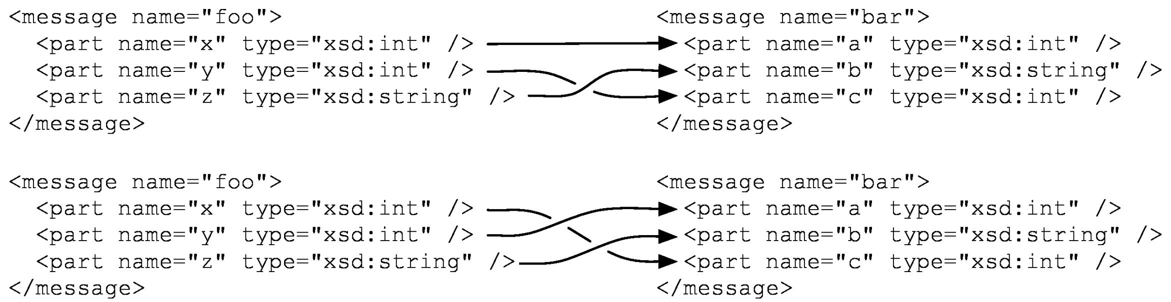


Figure 5.8: Bijective mapping between two messages. Instead of an in-order mapping between two messages (that differ in their qname), a bijective mapping is used. However, this can cause a non-deterministic, ambiguous mapping.

sented by a fragment has to be embedded into a `scope` activity. Therefore, two `assign` elements are needed, one to initialize the input variable(s) and one to copy the output into the succeeding SBF(s), as illustrated in Figure 5.9.

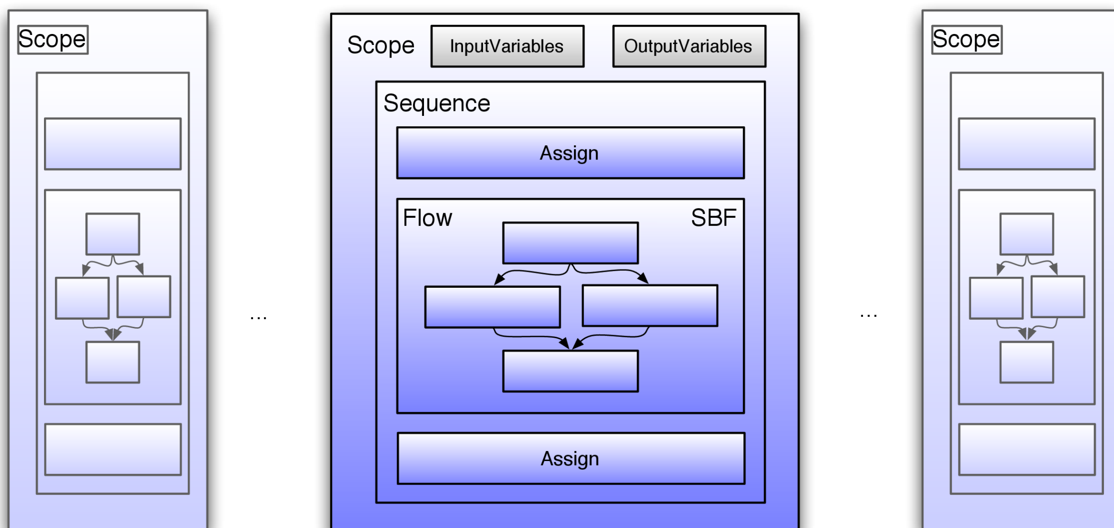


Figure 5.9: Integration of process fragments into a BPEL workflow.

Once all the fragments have been wrapped by their corresponding *scope* activities, the start and stop elements of the SimpleBPEL workflow have to be transformed as well. They are represented by a *receive* and a *reply* activity, respectively. The exact definition of the activities' variables depends on the SBFs that are connected to them. In case of the start element, the input variables of the fragments that are connected to it are used. Each part of their messages is aggregated into a new message that in turn serves as the message type for the variable which is used by the receive activity.

All the export procedures make use of DAVO's mechanisms for BPEL element creation and process modeling. This provides the possibility of reusing all the functionality available in DAVO, such as the export of the BPEL workflow into a deployable archive, which includes functionalities like generating the WSDL of the workflow and the deployment descriptor as well as hot deployment of the generated process archive.

5.5 Implementation of SimpleBPEL

In this section, some aspects of the implementation of the extensions to DAVO and of the newly developed SimpleBPEL Composer are briefly discussed. Both DAVO and the SimpleBPEL Composer have been implemented as plug-ins for the well-known Eclipse Platform. Using Eclipse has various advantages compared to implementing applications without an underlying framework. For instance, Eclipse features a powerful framework for drawing graphs and figures, the Graphical Editing Framework (GEF), that has been extensively used in this work. The implementation harnesses the extensibility mechanisms of Eclipse (extension points, OSGi bundles). For example, the user interface of DAVO has been extended by a view that allows for the storing of function blocks as a SimpleBPEL profile.

5.5.1 Extensions to DAVO

The Profile Editor has been implemented as an extension to DAVO, consisting of additions to the user interface, since the graph-based workflow editor and the data model of DAVO are suitable, because SBFs are (incomplete) BPEL processes. The user interface can be used to model SBFs, add descriptions and define for every variable if it is input or output of the SBF. When an SBF is added to a profile, the editor performs a validation following the rules defined in Section 5.4.1.

The validation process can be further extended and it is based on the interface `IBpelModelValidator`. It declares the methods `boolean validate(Process process)` and `Status[] getStatusMessage()`. The latter one not only represents the error message(s) to be displayed when the validation fails, but also returns further status messages that can also occur in a valid process (e. g., warning messages). It uses the `Status` classes of the `org.eclipse.core.runtime` bundle in order to consolidate the validation messages. Instances of the `IBpelModelValidator` are created on a per-process-basis.

An example of a validator is shown in Listing 5.5. The depicted `BasicValidator` accepts a given process as valid, if and only if it does not contain any receive and reply operations (Lines 16 ff.). If not at least one variable is tagged as input or output variable for the SBF (Lines 36 ff., a warning is issued.

```
public class ReceiveReplyModelValidator implements IBpelModelValidator↩
↪ {
2
  private ArrayList<Status> status = new ArrayList<Status>();
4
```

```
    @Override
6   /** @return returns true, if the process is valid. */
    public boolean validate(Process process) {
8     hasDedicatedVariables(process);
      return !containsReceiveOrReply(process);
10  }

12  /**
     * checks, if the process contains any receive or reply activities
14   * which are not allowed in SBF
     */
16  private boolean containsReceiveOrReply(ContainerElement container) {
      boolean flag = true;
18    for(Element element:container.getChildren()){
        if(element instanceof ContainerElement){
20        flag = flag && containsReceiveOrReply(((ContainerElement) ↩
          ↪ element));
        } else if(element instanceof Receive){
22        flag =  false;
        } else if(element instanceof Reply){
24        flag =  false;
        }
26    }

28    if (!flag) {
        status.add(new Status(Status.ERROR, LCDL.pluginid, -1, "process ↩
        ↪ contains receive or reply", null));
30    }

32    return flag;
    }
34
    /** checks, if the ProcessFragment has dedicated input/output ↩
    ↪ variables  */
36  private void hasDedicatedVariable(Process process) {
      boolean flag = false;
38
      for (VariableEntry variable : process.getVariables()) {
40      if (variable instanceof TemplateVariable) {
          TemplateVariable tv = (TemplateVariable) variable;
42        if (tv.getType().equals("in") || tv.getType().equals("out")) {
            flag |= true;
44          break;
          }
46      }
      }
48
      if (!flag) {
50      status.add(new Status(Status.WARNING, LCDL.pluginid, -1, "↩
        ↪ process does not contain any dedicated variables", null));
```

```
        }
52    }

54    @Override
      public Status[] getStatusMessages() {
56        return status.toArray(new Status[] {});
      }
58
}
```

Listing 5.5: This listing depicts the `BasicValidator` that checks whether an SBF contains any receive or reply operations and whether it has dedicated input or output variables.

Once a profile has been completed, the developer may export it to share it with other SimpleBPEL users. The profile is stored in a ZIP file that contains its XML-based description (especially the discussed input and output variables definitions), the function blocks, as well as screenshots, of the function blocks of those which allow a quick look into their intrinsics in the SimpleBPEL Composer (for documentation purposes).

5.5.2 SimpleBPEL Composer

From an user-interface perspective, the SimpleBPEL Composer can be considered as a typical Eclipse-based modeling application. The SimpleBPEL perspective consists of a navigation view on the left (see Figure 5.10) where existing SimpleBPEL workflows are listed and a graph-based editor responsible for the modeling. This editor has a palette (see ib.) where active profiles and their contained SBFs are displayed.

SBFs are added to a SimpleBPEL process by dragging them from the palette to the editing area. In turn, `SimpleBpelPaletteFactory` invokes `createModule` on the very model element (`ProcessFragment`) that represents the selected SBF. `createModule` deserializes the XML description, loads the underlying BPEL code into the data model and instantiates the corresponding GEF *edit part* and the corresponding figure class that draws the graphical representation. *Edit parts* also hold so-called *edit policies* defining the actions (*commands* in GEF terminology) that can be performed with the corresponding *edit part*. For instance, a custom command to delete function blocks is required, since this command should not only remove the element to be deleted from the data model, but also its associated variable definitions. One of the most important implemented commands is `ConnectionCreateCommand`. It is executed whenever the user wants to connect an SBF of a certain SBF with the input of another. For this aim, it uses the configured `CompatibilityMode` to check whether the selected function blocks can be tied to each other or not. In the latter case, `canExecute()` returns `false`.

The *Process Exporter* sub-component is clearly the most important and most complex part of the SimpleBPEL Composer. It is responsible for creating a standard-conform, executable BPEL process from the composed SBFs. Its pseudo-code is shown in Listing 5.6. The export procedure consists of five steps:

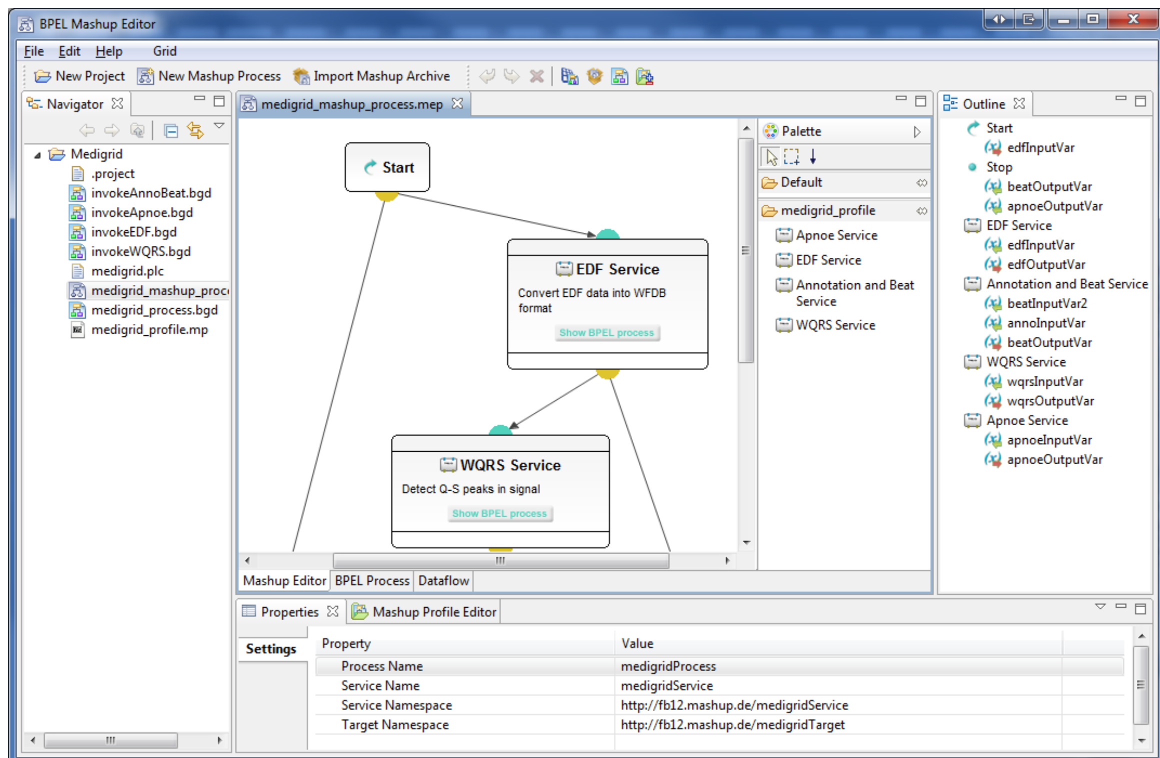


Figure 5.10: SimpleBPEL editor as a rich client application within Eclipse.

1. *initializeProcess* sets parameters, e. g., process name and namespace.

2. *addSBF* adds all SBFs using the container structure (*scope*); further information about each SBF, like variable and message definitions are extracted and attached to the process model.

3. *initReceiveReply* creates the start and end point of the process by creating input and output variables, messages and a *portType*.

4. *addAssignActivities* generates *assign* activities that convert output data of SBFs to the required input data format of dependent SBFs. Behind the scenes, XPath expressions that copy data from the output message parts of a single function block to the input message parts of succeeding function blocks are generated.

5. *finalize* wraps the created process elements in a *flow* element. The created process is written to disk using the export mechanisms of DAVO, since at this point a complete BPEL process has been created in its internal data model.

```
public void export(Process simpleBpelProcess) {
  BpelProcess process = new BpelProcess();
  initalizeProcess(process, simpleBpelProcess);
  addFunctionBlocks(process, simpleBpelProcess),
  initializeReceiveReply(process);
  addAssignActivites(process, simpleBpelProcess);
  finalize(process);
}
```

Listing 5.6: Pseudo-code of the SimpleBPEL workflow export.

5.6 Summary

This chapter presented solutions for moving software towards a service-oriented architecture. The *Legacy Code Description Language* provides a generic and extensible way to wrap existing software, e. g., as a web service. To ease the use of composition frameworks, the *SimpleBPEL* approach distinguishes between workflow experts and domain experts and provides the latter ones with a composition environment that – from a technical point of view – is distraction-free.

6

Adaptable Service Workflows

Contents

6.1 Introduction **86**
6.2 Combining BPEL and Scheduling – The Basic Approach **86**
6.2.1 Extensions to the BPEL Engine 87
6.2.2 Load Balancer 88
6.2.3 Host Analyzer 90
6.2.4 Implementation 91
6.3 A Cloud-enabled Data Flow Scheduling Architecture **95**
6.3.1 Workflow Development 95
6.3.2 Workflow Execution 96
6.3.3 Implementation 100
6.4 Multi-Objective Scheduling **103**
6.4.1 Framework Components 105
6.4.2 Multi-objective Scheduling Algorithm 108
6.4.3 Implementation 111
6.5 Reliability for Service Workflows **112**
6.5.1 Fault Tolerance Module 113
6.5.2 Dynamic Resolver 116
6.5.3 Implementation 116
6.6 Reliability and Scalability of the Workflow Engine **119**
6.7 Summary **121**

6.1 Introduction

In service-oriented architectures, service compositions are first class citizens for addressing runtime adaptation. Workflows compose several existing services into a single, new, value-added service. Besides the aimed functionality, several new (non-functional) requirements emerge for such a workflow. Some techniques allow to address these requirements on the business layer of services of those workflows. Nevertheless, there are still requirements that arise on the non-business layer, typically the infrastructure layer. The following sections elaborate a framework for runtime adaptation of service workflow executions with respect to a cloud-based infrastructure. This will be exemplified by the topics of scalability and reliability.

Parts of this chapter have been published in [44, 45, 110, 113].

6.2 Combining BPEL and Scheduling – The Basic Approach

In this section, the design and the implementation for a basic scheduling framework for BPEL workflows are presented. This framework is designed to meet the requirements postulated in Section 2.3.2 and Section 2.3.3.

The proposed solution combines the versatile and powerful composition and control mechanisms of BPEL with an adaptive runtime environment without breaking with existing standards. The system seamlessly integrates dedicated resources and on-demand resources where the latter ones are provided by infrastructures like Amazon EC2. When employed in conjunction with previously developed extensions to BPEL [43], the proposed system even allows to invoke stateful web services. These are implemented according to the WSRF standard and that are widely used in a Grid middleware like the Globus Toolkit 4 (GT4) [73, 83].

The architecture consists of three components (see Figure 6.1):

1. The Dynamic Resolver (DR) extends the invocation mechanism of the BPEL engine.

2. The Load Balancer (LB) manages existing hosts, schedules service calls and provides new hosts.

3. The Host Analyzer (HA) collects general information about hosts, e. g., system load and available services.

The interplay between the components is as follows: whenever, during the execution of a workflow, a service is called, the workflow system checks whether the target host has already been selected or not. In the latter case, the DR determines the best-matching target host. After that, it contacts the LB and passes the service description to the LB, which in turn queries an internal registry for hosts that have the requested service installed. The resulting list is enriched with information about the performance and the current load of the hosts that are acquired using the HA. The list is then passed to a scheduling component that determines the best-matching host or provides a new

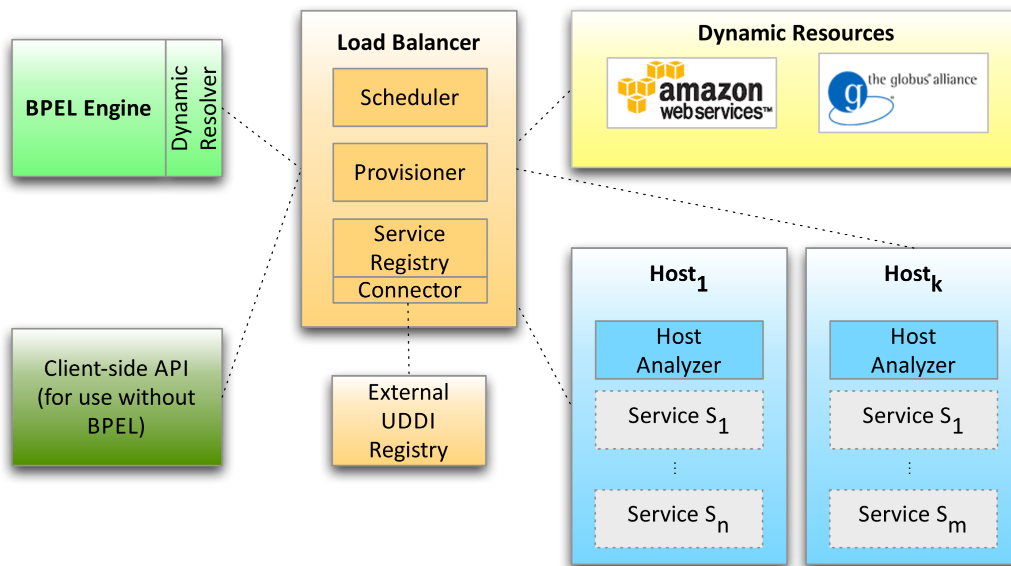


Figure 6.1: Three-tier architecture of the load balancing solution.

host in case all hosts have high load. Finally, the address of the selected target host is passed back to the workflow system to finally execute the call.

In the following, the design of the components and their interplay is discussed in more detail. A detailed view of the relations can be found in the UML diagram in Figure 6.2

6.2.1 Extensions to the BPEL Engine

The DR is designed to avoid any changes of the code of the BPEL engine. As a result, the portability to new versions is eased considerably. It is invoked whenever, during the execution of a process, a *partnerLink* is set to *dynamic* with the attribute *invokeHandler* pointing to the implementing class of the DR. It then checks whether the corresponding BPEL *partnerLink* already has an endpoint address or not. In the latter case, the DR invokes the LB component so as to determine a concrete endpoint. To achieve this, the *qname* of the *portType* and other parameters are passed to the LB component. The LB component (see Section 6.2.2) in turn returns the endpoint address of the service that matches best. In this way, the DR constitutes the interface between the BPEL engine and the LB component.

Since different scheduling strategies and infrastructural backends can be implemented, arbitrary parameters for dynamic endpoints must be definable. These parameters are passed to the LB which then in turn delegates the parameters to the scheduling and provisioning components. For instance, on one hand the workflow designer might want to define a minimum load threshold to start new virtual machines and on the other hand authorization information for infrastructures like Amazon EC2 are required.

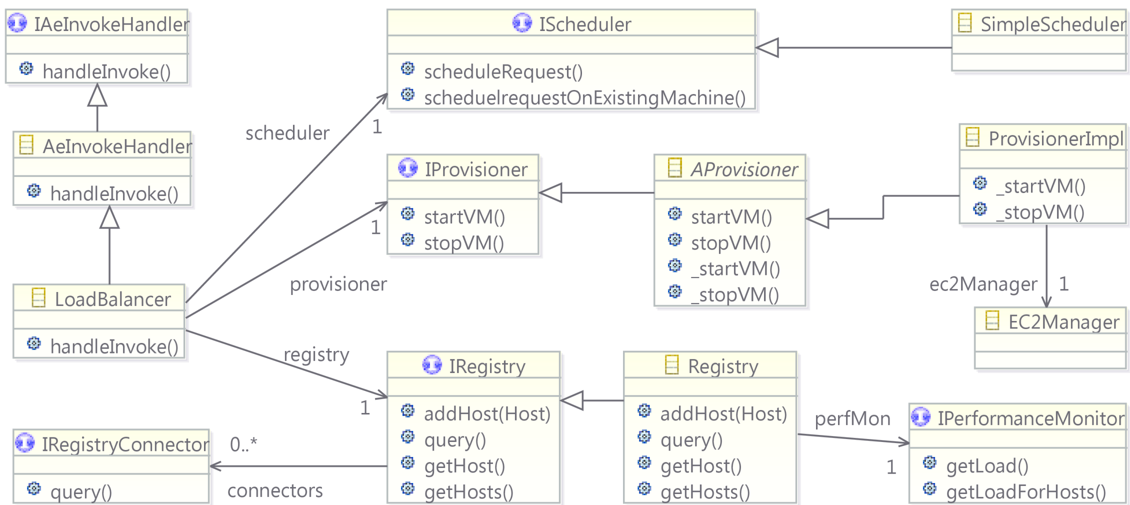


Figure 6.2: Detailed UML diagram of the architecture for (basic) load balancing in BPEL workflows.

6.2.2 Load Balancer

The load balancer (LB) is the main component in the presented architecture. It has three components: scheduler, provisioner, and registry. It manages dedicated target hosts, continuously monitors their system load and makes scheduling decisions to avoid the overloading of hosts. Whenever required, it provisions new machines to keep the workflow system stable and reactive. To make the infrastructure as cost-efficient as possible, Cloud resources are released if the overall load has returned to normal. When invoking the LB, the qname of the target service's port, a threshold value that describes the maximal load of the target systems, and additional parameters, like authorization data for the Cloud infrastructure, are passed to the LB.

The DR contacts the LB whenever an endpoint for a target service has to be resolved. Subsequently, the registry is queried for hosts having the requested service installed. Using the resulting list, the registry then collects the current system load for all qualified hosts. This information is then passed to the scheduler, which makes a scheduling decision according to the configured scheduling algorithm. This decision might involve starting new virtual machines by calling the provisioner. Eventually, the scheduler returns the target endpoint reference to the LB that in turn replies to the DR.

Details about all involved subcomponents are presented below.

Registry

The registry manages all information about hosts and their services. In particular, the endpoint of services must be accessible via the qname of their porttype. Since the start and shutdown of virtual machines are essential parts of the system, the registry must also handle the appearance and disap-

pearance of hosts and services[1]. As queries may be performed during the startup and the shutdown of of virtual machines, it must be assured that those virtual machines does not appear in the resulting set of the query. To guarantee that virtual machines are not shut down while they are being used, a construct that locks the shutdown procedure for virtual machines has been implemented in this approach.

When virtual machines are started, information about them has to be saved persistently. This makes the registry resistant to failures and ensures that running virtual machines may still be managed after a restart of the LB.

Furthermore, the registry can query other (UDDI) registries, such as Grimoires [85] and thereby extend its scope. In addition, a set of "static" hosts can be included in the registry by using a configuration file – for instance, a company's dedicated hosts.

To collect load values, the HA is queried by the registry every time it is queried. To reduce communication costs, a cache keeps the load information in memory; it is invalidated after a configurable interval. Thus, frequent queries do not affect the performance of the registry.

Provisioner

The provisioner encapsulates interfaces to manage virtual machines in on-demand infrastructures like Amazon EC2, Eucalyptus [144] and Globus Virtual Workspaces [74] or its successor, the Nimbus platform [95]. It provides general abstractions, such as starting and stopping virtual machines, and preparatory methods, such as starting middleware components.

With this abstraction, it is possible to develop drivers that interact with providers that use different implementation technologies and communication patterns for their management interfaces. These drivers may be plugged in to the provisioner by means of a simple configuration file. Therefore, the presented approach allows to integrate new infrastructures without a re-design of the component itself. For example, Amazon EC2 provides web service based access to their management interface without delivering notifications when machines have finished booting – a polling-mechanism needs to be implemented in this case. On the other hand, Globus Virtual Workspaces makes use of WSRF-based services and implements WS-Notification, meaning that a caller gets notified when certain events happen. The provisioner therefore needs to be generic and must hide implementation intrinsics.

It is important to note, that the *startVM* method must not return before the service hosting environment (web/Grid service stack) has been started. If it would do so, the services would not be available immediately, which would result in errors on the BPEL side when trying to invoke the services.

The provisioner registers and de-registers virtual machines in the registry after their booting and shutdown. In the sequel, the registry adds or removes all services installed on the virtual machine.

[1] In decentralized peer-to-peer system, this phenomena is ubiquitous and referred to "churning".

Scheduler

The scheduler gets information about the infrastructure, makes scheduling decisions and invokes the provisioner to manage dynamic resources. Since different scheduling strategies are required for different application scenarios, the scheduler plugin allows to realize different scheduling algorithms. A sample scheduling strategy and its implementation are described in Section 6.2.4.

Like the provisioner the scheduler may be configured for different scheduling algorithms. Every scheduling algorithm must implement two methods:

1. the scheduling on existing hosts,

2. the scheduling with the possibility to start additional machines.

The first method is required if problems occur during the startup of a virtual machine. For example, it might be impossible to start a new machine because the user's quota has been exceeded. A software or a configuration problem could also prevent the correct startup of the machine. The scheduler memorizes the starting time of virtual machines, since services like Amazon EC2 are billed per used time, meaning that shutting down a virtual machine before the expiration of an accounting period saves money. Therefore, it must tell the provisioner to shut down virtual machines in due time if the overall load is low. The system has to take into account that there must not be any running service calls on the machine to be shut down. Though this may sound rather trivial, it is not straightforward to determine whether a service call (or the induced computation) has been finished or not. In Section 6.2.4, this problem and a possible solution is discussed in depth.

6.2.3 Host Analyzer

Many scheduling algorithms make scheduling decisions based on the optimization of the throughput. This means that the scheduling of jobs or calls to hosts with low load is favored, since presumingly these hosts finish tasks faster than a host with higher utilization. Therefore, the system needs to collect load information for all target hosts and to provide the scheduling algorithms with this information. In the proposed solution, a web service that is installed on every host analyzes the load. It is queried by the registry in order to update system load values. In the currently implemented prototype, "system load values" represent the following tuple: the number of CPU cores, the load of the host and an estimated measure of the system performance. Moreover, the HA searches the local web service stack (i. e., the Axis engine) for available web services, which can be queried by the LB. This functionality is important because not all hosting platform necessarily host the same set of services.

On systems where a single (head) node represents a number of hosts (e. g., a cluster site), the number of cores and the load are calculated by the following simple formula. Let n be the number of nodes in the system, $c(\cdot)$ be a function that returns the number of CPU cores for a given node, and $l(\cdot)$ be a function that returns the load for a given CPU core. Then:

$$\#\,\text{cores} := \sum_{i=0}^{n} c(i) \qquad\qquad load := \frac{\sum_{i=0}^{n}\sum_{j=0}^{c(i)} l_i(j)}{\#\,\text{cores}}$$

If a cluster has heterogeneous nodes, the above values have to be calculated for each subgroup of homogeneous nodes.

Furthermore, the HA provides a method to generate a list of all services being exposed by the application container to enable the LB to automatically determine the WSDL documents of the services.

6.2.4 Implementation

This implementation is based on the BPEL4Grid engine, the workflow enactment engine originally developed by ActiveEndpoints. It is a stable and well-documented software that is published under GNU Public License (GPL).

Extensions to the BPEL engine

For the main goal of minimizing changes to the workflow engine, the DR makes use of certain interfaces and observers provided by the workflow engine. It implements the `IAeInvokeHandler` interface provided by the workflow engine. The DR is used when a call is executed on a *partnerLink* marked with the newly developed *invokeHandler* (see Listing 6.1). The latter one is defined in the process deployment descriptor, which is the file that binds a *partnerLink* to a specific endpoint. Due to this *invokeHandler*, the binding stays dynamic. Since the *invokeHandler* is specified via an URI, necessary arguments may be passed to the DR via an URL encoding (lines 4 and 5 in Listing 6.1).

```
<partnerLink name="decoderPL">
 <partnerRole endpointReference="dynamic"
  invokeHandler="java:de.fb12.mage.bpel.lb.invokeHandler.LoadBalancer
  ?threshold=1.0;accessID=***;secretKey=***;imageID=ami-95cc28fc;↩
  ↪ availZone=us-east-1c"/>
```

Listing 6.1: Definition of a custom invoke handler.

During the execution of the DR, the *partnerLink* associated with the invoke activity is checked to determine whether its endpoint reference has already been set or not. In the latter case, the LB component is executed by calling the method `getEndpointForPortType(QName portType, double threshold, Map queryMap)`. `queryMap` contains all parameters encoded in the URL in a key/value manner. The LB returns an endpoint reference (bean); its information is set as the partner reference of the used *partnerLink* before the actual call is made.

Load Balancer

The LB is the only direct connection between the BPEL engine (i. e., the DR) and the load balancing solution. No direct access to the scheduler, the provisioner, or to other components is necessary.

This leads to exchangeability and extensibility, since the components do not heavily depend on each other. Furthermore, all described components implement interfaces (`IPerformanceMonitor`, `IProvisioner`, `IScheduler`) to ease exchangeability.

The LB is implemented as a singleton per BPEL engine; it is instantiated via a factory, and so are the other components like the registry and the provisioner. A configuration file is used to define the actual implementations, which are then loaded using Java reflection.

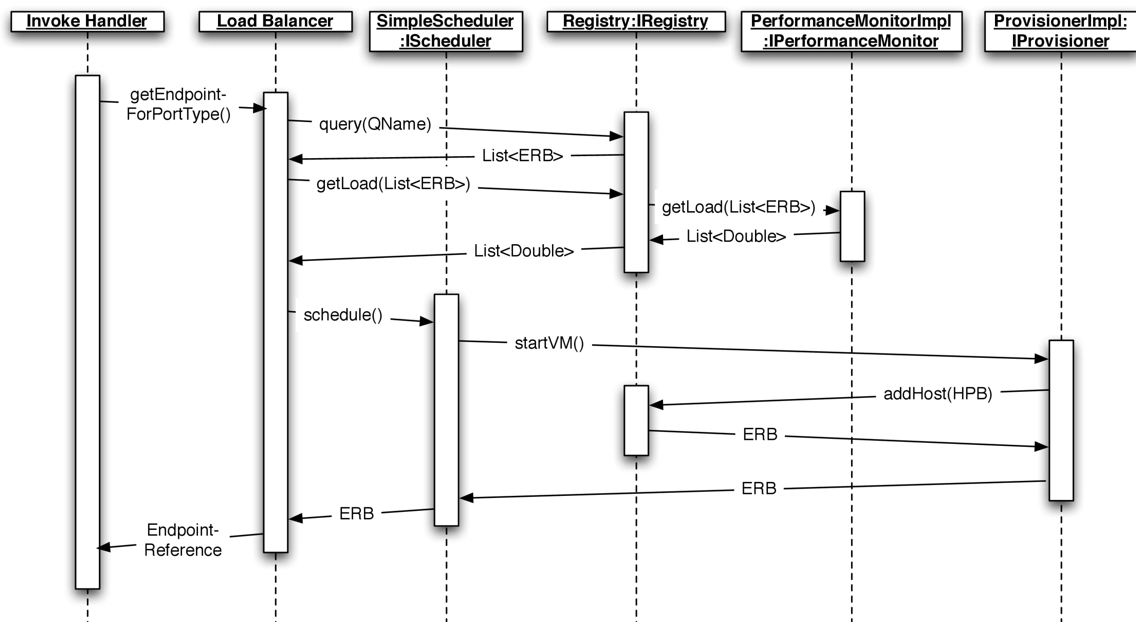


Figure 6.3: Sequence of calls to determine a dynamic endpoint. Starting at the invoke handler, the load balancer queries the registry for adequate hosts (i. e., hosts that provide the required service). If none are available, the provisioner is utilized to boot up new machines. In both cases (i. e., whether new machines are provisioned or not), the scheduler determines in a next step the most suitable machine of the given host set. Finally, this machine is passed back to the InvokeHandler and afterwards used for the actual service invocation.

When the LB is invoked (see Figure 6.3), it first queries the registry for services matching the given port type's qname. A list of endpoints, encapsulated in `EndpointReferenceBean`s (ERBs), is returned and passed to the performance monitor. This, in turn, hands back a list of `double` values representing the load of the target hosts. The LB then passes both lists and parameters, such as the mentioned threshold, to the scheduler. The configured scheduling algorithm is employed afterwards, and the endpoint of the computed target host is returned to the DR. If the scheduler decides to start additional virtual machines, the provisioner is invoked. After starting a new virtual machine, it is registered and the corresponding endpoint is returned.

To assure cost-effectiveness, the scheduler monitors virtual machines and, if the overall load is low, tells the provisioner to shut them down before the accounting period expires. The system has to consider that there must not be any running service calls on the machine to be shut down. This

is not trivial to determine. For example, consider an asynchronous call of a service that results in an asynchronous callback when the service has finished the computation. In this case a completed invoke does not necessarily mean that no response is expected anymore. To solve this issue, for each host in the registry the DR increments a counter that represents the number of active calls of services on that host. When a process has finished, the process graph is examined for invoke operations. The target host of every invoke operation is determined and the counter value is decremented. The scheduling algorithms simply have to check whether the counter of a virtual machine is zero, before telling the provisioner to shut down this machine. For this reason, the implementation needs to be notified about the termination of processes in order to decrement the corresponding per-host counter that prevents the virtual machines from being shut down. For this purpose, the LB registers itself as an event-listener in the BPEL engine (`IAeEngineListener`). At process termination, `handleEngineEvent` is invoked; the host names of the invocation targets are extracted and the registry is called to decrement the lock counter.

Registry

The registry manages available information on hosts and services and keeps them persistent. It is configurable via an XML document that is used to determined a set of permanently available hosts. Hosts are managed by so-called `HostPropertyBean`s containing information about, e. g., the startup time (relevant for virtual machines), the virtual machine type, and the type of middleware (*serverType*).

When a host is added – either by loading the static hosts or by the provisioner – the LA is queried to acquire a list of all services offered by this host. Therefore, the `serverType`, which allows to distinguish between different middlewares – and access patterns – is provided. Depending on the type of middleware (*serverType*) of the target host, different clients are used for querying the service list and the system load. Currently, two clients for plain web services and WSRF-based services exist.

Searching the registry for a specific *portType*, the list of available services is scanned. Furthermore, external registries are queried, if those are configured. The access to them is realized via an interface `IRegistryConnector`. A list of all services matching the given qname is finally returned.

Provisioner

Since the provisioner has been designed to support different backends, it is divided into two parts: (1) an abstract class provides two methods, `start` and `stop`, that handle backend-independent tasks like registering and deregistering of hosts. Prior to that, they call `_startVM` and `_stopVM` in the implementing sub-class (2). In this class the start operation returns an ERB with information needed for the registration of the machine. In addition, the abstract class is responsible for the persistence of running virtual machines so as to avoid information loss due to a restart of the LB component.

The `_startVM` method must not return the ERB before the virtual machine and the middleware have booted. Otherwise, the DR or the workflow engine might try to invoke a service on the new virtual machine before it is available. The method returns as soon as the service can be successfully invoked or aborts after several retries.

As an implementation example, this prototype implements a backend for Amazon EC2 [6]. It makes use of "Typica" [183], a Java-based open source library developed by Xerox. The implementation is able to start and stop virtual machines, to configure the firewall of the virtual machines (some ports have to be open for SOAP communication and all other ports should be closed) and to execute user scripts after the booting of a virtual machine. Since after the creation of a Amazon Machine Images (AMI) it cannot be altered anymore, a script that is executed at boot time downloads the middleware from an Amazon Simple Storage Service Bucket (S3) [7]. This assures that the middleware can easily be upgraded (or new services may be installed) without creating a new AMI. In combination with Amazon CloudFront [5], a fast download of the middleware can be achieved. The middleware is downloaded from S3/CloudFront, since in contrast to external traffic (like a *wget* from an external server) Amazon-internal traffic is not billed. Although incoming traffic is also not billed, the achieved bandwidth using S3 is typically higher than the bandwidth that can be achieved using external servers.

Scheduler

In the following, the implementation of a simple scheduling strategy is discussed as an example.

The input values for the scheduler are the names of the matching hosts (H), the load values (load) and a threshold value t. The scheduler first computes all qualified hosts H_q, where $H_q := \{h \in H \mid \text{load}_h < t\}$. If H_q is non-empty, for all hosts in H_q, the free capacity

$$c_h = \frac{WIPS_h}{\max(\text{load}_h, 0.01)}$$

is computed. *WIPS* stands for Whetstone Instructions Per Second and is computed using the Whetstone benchmark that performs integer, floating point and array operations to determine the performance of a host. The value 0.01 is required to avoid a division by zero if a host has a load of 0. Afterwards, the host with the highest value for c_h is chosen, meaning that the host, which is utilized the least, is selected. If H_q is empty, the scheduler requests a new/additional host from the provisioner. After the startup of the new virtual machine, the waiting call is immediately scheduled to the host.

Host Analyzer

The HA has been realized for Linux. It inspects `/proc/loadavg` and `/proc/cpuinfo` to determine the *Load Average* and the core count. In virtualized environments, the computed percentage of CPU usage cannot be used as an indicator for the CPU load. The machine may only have been assigned a part of the physical computing power – in this case, the percentage of CPU usage would

always be below 100%. The *WIPS* benchmark is executed in order to get an approximation of the performance of a system. For cluster environments, only a prototypical implementation is available so far. It queries the Monitoring and Discovery System of the GT4 to acquire information about available and used cores.

The HA has also a subcomponent, which determines the installed services in the hosting middleware. Those services – reduced to their service name – are returned as a list. This subcomponent is configurable, so that either all names of the installed services are returned or only a subset of them is.

6.3 A Cloud-enabled Data Flow Scheduling Architecture

In this section, the design and implementation of a data flow aware scheduling architecture for service compositions is described. In principal, it is an alternative to the scheduler of the basic framework of the previous section. Therefore, most components can be reused. Implementation details about the scheduling itself and some necessary extensions are described after development time aspects of the workflow runtime system were elaborated. Figure 6.4 sketches the complete development and execution.

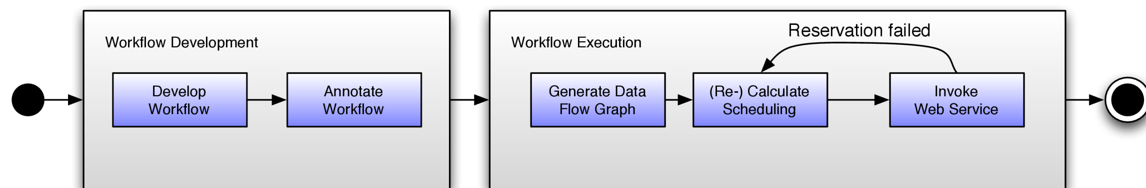


Figure 6.4: Development and execution life cycle of a workflow when data flow aware scheduling is used.

6.3.1 Workflow Development

For the assignment of workflow steps to available resources, the system needs information about the amount of data to be transferred between different workflow steps and about the expected runtime of each step. There are two methods conceivable so as to acquire this information: (1) to monitor workflow executions and to derive expected runtimes and data transfer times from these observations and (2) to let the workflow and/or the service developer annotate the workflow with this specific information. In the described approach, the latter method is used for simplicity.

As described above, BPEL is control flow driven and which in particular means that the data flow between workflow activities is not visible (and not necessarily the same, see Figure 6.5 for the control and data flow of the sample workflow of the medical use case). Therefore, a data flow view on the BPEL process has to be generated. The data flow is represented by a graph whose vertices correspond to the *invoke* activities. The edges between these vertices represent the *assign* operations

of the BPEL process. More precisely, two vertices are connected by an edge if there exists a *copy* operation within an *assign* activity that copies the output of the first *invoke* into the input of the second one. Furthermore, this view enables the workflow designer to annotate the vertices with the expected runtime of each service when it is executed on a typical resource. The edges are labeled with the amount of data transferred between the two connected *invoke* operations.

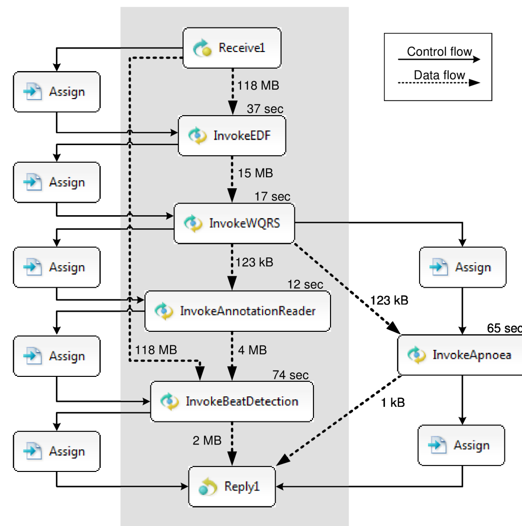


Figure 6.5: Control and data flow of the sample workflow of the medical use case.

The whole information carried by the graph is encoded and saved within the BPEL process in such a way that this information is available during runtime. For each *invoke* activity, its predecessors as well as its successors are saved and enriched with the annotations. This is done without a violation of the BPEL standard.

6.3.2 Workflow Execution

After development, a BPEL workflow must be deployed in a BPEL engine before it is exposed as a web service. At the first call of a workflow, an internal graph representation containing the data

flow information mentioned above is constructed. Using this representation, the developed system performs the scheduling. This procedure is detailed in the following.

To perform a scheduling of BPEL workflows in heterogeneous environments, several conditions have to be met. One of the main differences compared to the scheduling in cluster computing environments is that different resources offer neither the same set of services nor equal computing power. On the contrary, the resources are inherently heterogeneous with respect to the services they offer and the computing power or the network bandwidth they are interconnected with. These circumstances have to be taken into account when the invocation of a web service is scheduled to a pool of resources.

Furthermore, since BPEL is a non-DAG and Turing-complete language, none of the existing workflow based scheduling algorithms can be applied without any constraints. Some algorithms transform the cyclic structure of a BPEL process into a DAG. For this, they either make use of an upper barrier for the iteration of *while* loops or they defer this decision to the runtime of the process [57, 158]. Even if a BPEL workflow can be converted into a DAG structure, scheduling in DAG structures remains NP-complete [184] and is computable in polynomial time only for some simple cases [121]. The usage of a heuristic algorithm ensures an acceptable runtime of the scheduling algorithm itself. This provides a compromise between scheduling runtime and optimality of the assignment.

A scheduling algorithm can operate on the entire workflow graph (*workflow based*) but also on just a single task (*task based*) of the workflow. Both solutions come with different drawbacks. While task based scheduling does not allow to consider the constraints mentioned above, the workflow based approach significantly increases the complexity. To circumvent these disadvantages, the presented approach uses a heuristic algorithm that works on *critical paths* instead of the entire workflow. After calculating the schedule for a critical path, the requested resources are allocated via a *reservation* mechanism.

Critical Paths. To reduce the assignment complexity, a data flow graph is partitioned into so-called critical paths. These are linear parts of the data flow graph that – except the start and end vertices – are disjunct from other critical paths. In the following, the term critical path means both the vertices and the edges connecting them. Critical paths are computed using a Breadth First Search starting from the *receive* activity of a workflow that instantiates a workflow. Workflows typically consist of several critical paths that cover the complete process. Figure 6.6 illustrates the critical paths of the workflow shown in Figure 6.5.

All critical paths are sorted according to descending runtimes. The runtime is determined by summing up the annotated service runtimes within each path and by adding the time needed for the data transfer. Taking the average bandwidth over all available resources, the annotated data amounts are converted into transfer times. The average value is feasible here, since the value is only used to compute the path lengths. For actual scheduling decisions, the network bandwidths between the scheduling candidates are used.

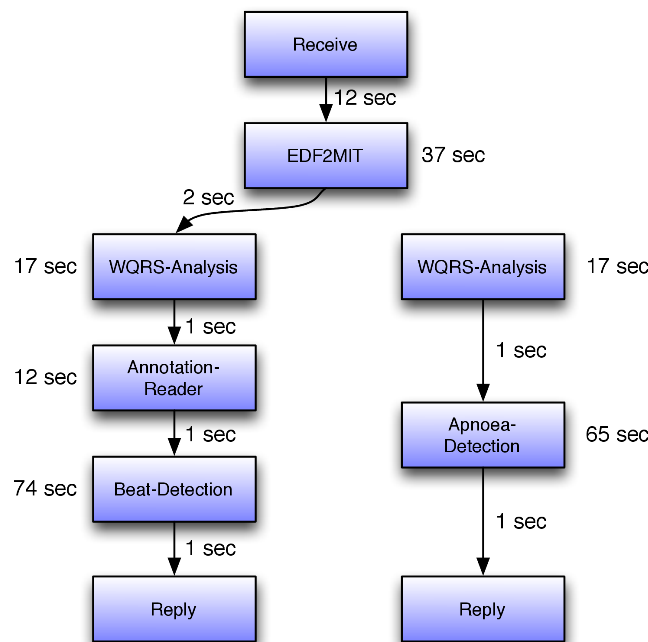


Figure 6.6: Critical paths of the sample workflow of the medical use case.

When the scheduling algorithm starts, the first critical path of the sorted set is selected. Since this is the path with the highest runtime, it completely determines the overall runtime (cf., *critical path method* when business processes are analyzed [158]). Then, a resource allocation is computed (and resources are reserved according to the schedule) via the algorithm described below. The same procedure is applied to the remaining critical paths (resource reservations are respected) resulting in a resource allocation for all activities.

Heuristic Algorithm. To handle the complexity of the graph due to the expressiveness of BPEL, a genetic algorithm (GA, see Listing 6.2) is employed. Such a GA operates on a random population and uses strategies such as selection, cross-over and mutation inspired by evolutionary theory for computing an approximate solution of an optimization problem [205]. The size of the population is calculated as follows: $size_{pop} = \lfloor 100(\cdot \ln(pc) + 1) \rfloor$, where pc (path complexity) is the product over the numbers all target resources for all workflow steps (vertices in the critical path). In this way, the population size is increasing in the problem size. From a mathematical point of view, the population (corresponding to an actual path) is modeled as an integer vector. The components of the vector are in a one-to-one correspondence with the vertices of the critical path and encode the resources that are offered by the service which the corresponding *invoke* operation requires. Whereas cross-over and mutation rely on stochastic methods that are used for the optimization of a concrete vector via randomization, the selection operator is a special filtering process within the population.

```
for( path : criticalPaths ) {
  population = generateStartPopulation( path );

  while( evolutionNotFinished() ) {
    newPopulation  = survivalOfTheFittest( population );
    newPopulationC = crossover( newPopulation );
    newPopulationM = mutate( newPopulation );
    newPopulation.add( newPopulationC , newPopulationM );
    population = newPopulation;
  }
}
```

Listing 6.2: Pseudo-code for genetic scheduling algorithm.

In the above listing, `survivalOfTheFittest(population)` calculates the *fitness* of each vector in the population. The fitness is the sum over the runtimes of all activities (including the data transfer times and waiting times for the earliest slot in the reservation schedule of the chosen machine) for all machines encoded by the vector. Ergo, the lower the value is, the shorter the execution time for the critical path is. For this reason, the vectors are sorted according to descending fitness value, and only the upper $X\%$ are kept (X is a configurable value, here $X = 35$ is chosen). The remaining $N\%(N = 100 - X)$ for the next iteration are determined using `crossover(newPopulation)` and `mutate(newPopulation)`.

The new population is completed by adding the samples generated by `crossover(newPopulation)`. In each iteration, two samples S_1 and S_2, from `newPopulation` are randomly chosen and a crossover position c is randomly assigned to them. A new sample is then constructed using the elements $[0, c]$ and $[c, S_2.length]$ from S_1 and S_2, respectively. The random selection is not based on a uniform distribution, but rather on a weighted distribution that favors candidates with higher fitness (Roulette Wheel Selection Scheme [192]). `mutate(newPopulation)` randomly selects $\frac{N}{2}$ % of the population. A random set of entries of each vector is subject to change, where the actual number of modified elements depends on the size of the vector; the alteration of the entries is controlled stochastically. This process iterates until the size of the new population has reached the size of the old one.

All operations have in common that they are subject to two conditions: (1) the randomly selected resources must offer the requested service and (2) the machine must not be in use at the calculated point in time when the service is executed. To assure the second restriction, reservations for certain machines are respected (as explained below).

Obviously, the quality of the interim results of the algorithm cannot decrease during a iteration, since the worst results are removed in every step – in the worst case, the quality of the results stays the same between two iteration steps. The iteration stops (`evolutionNotFinished()`) if (1) $fitness_{best} - fitness_{worst}$ in `newPopulation` has been below 2% for three successive iterations, or (2) if the fitness value of the best element stagnates, or (3) the maximum number of iterations (it_{max}) has been exceeded. (3) is used to bound the runtime in case the algorithm does not converge

within a reasonable number of iterations. it_{max} depends on the number of resources and the number of vertices in the critical path. More precisely $it_{max} = \lfloor 10 \cdot \ln pc + 50 \rfloor$. If a workflow is built up of 10 steps (6 in one critical path, 4 in the other one) and 20 available resources per step, then $pc = 20^6$ and $it_{max} = \lfloor 10 \cdot \ln 20^6 + 50 \rfloor = 229$ for the longest critical path. After the last iteration, the vector with the best fitness is selected from the population. This vector encodes the resource allocations with the shortest execution time.

Reservation. To avoid the overloading of resources and to optimize the overall performance of all managed resources, it is advisable to use reservations. These are made per service call and reserve a CPU core of a machine for a certain time. The period of validity for a reservation r_i (for element i in the vector) is $[endTime_{r_{i-1}}; endTime_{r_{i-1}} + execTime_{r_i}]$. Reservations are removed from the system as soon as the corresponding *invoke* has finished. To prevent a resource from being overloaded (in case a service invocation has not finished within time), it has to be checked whether there exist any reservations in the system for this specific machine. If so, a re-scheduling of the subgraph beginning with the actual *invoke* element is initiated. By using reservations, the system is not only able to allocate resources for an actual scheduling step (without overloading a resource), but it also provides a control mechanism to start new virtual machines. If the difference between the calculated start time of element i and the reservation r_i exceeds δ, a *placeholder* machine offering the required service is added to the pool of available machines. If the result is thereby improved, reservations will be made according to the schedule and the system replaces the placeholder by newly provisioned Cloud resource.

6.3.3 Implementation

One of the main requirements for the design – as well as for the implementation – is their non-invasive realization. This means that neither existing standards nor implementations should be changed but rather be extended by the use of yet available extension mechanisms.

Workflow Development

The graphical BPEL modeling tool DAVO has been extended in this approach by a data flow view that is automatically generated by analyzing *assign* operations in the control flow of a workflow. The graphical representation of the data flow can be annotated with expected service runtimes as well as approximate data amounts. The extension has been developed as a separate Eclipse bundle that integrates into the main editor using standard OSGi techniques.

In Listing 6.3 the encoding the data flow graph is described, m_i for $1 \leq i \leq n$ refers to the *bpelLocations* (XPath) of *invoke* operations. $s_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq l_i$ is formatted as follows:

$$s_{i,j} = \text{bpelLocation}[\text{dataAmount} = x_{i,j}, \text{compTime} = y_{i,j}],$$

```
me = m_1 : m_2 : ... : m_n;
        src : 1 = s_{1,1} : ... : s_{1,l_1}; dest : 1 = d_{1,1} : ... : d_{1,k_1};
        ⋮
        src : n = s_{n,1} : ... : s_{2,l_n}; dest : n = d_{n,1} : ... : d_{2,k_n};
```

Listing 6.3: Description of the format of Invoke Handler String for data flow graphs.

where $x_{i,j}$ is the amount of data transferred between $s_{i,j}$ and m_i ($y_{i,j}$ is the execution time of service $s_{i,j}$ on a typical resource). $d_{i,j}$ is defined analogously, but refers to the outgoing edge of the data flow graph.

To achieve compatibility with the BPEL standard, data flow annotations are stored within the deployment descriptor of a workflow. The information is passed from the used custom invoke handler (*DFS-InvokeHandler*, see Figure 6.7) to the Model Generator that in turn generates a graph representation to be further processed by the scheduler.

Workflow Execution

The implementation of the execution environment is based on the previously described framework (see Section 6.2), which in turn is based on ActiveBPEL [1], the workflow enactment engine developed by ActiveEndpoints.

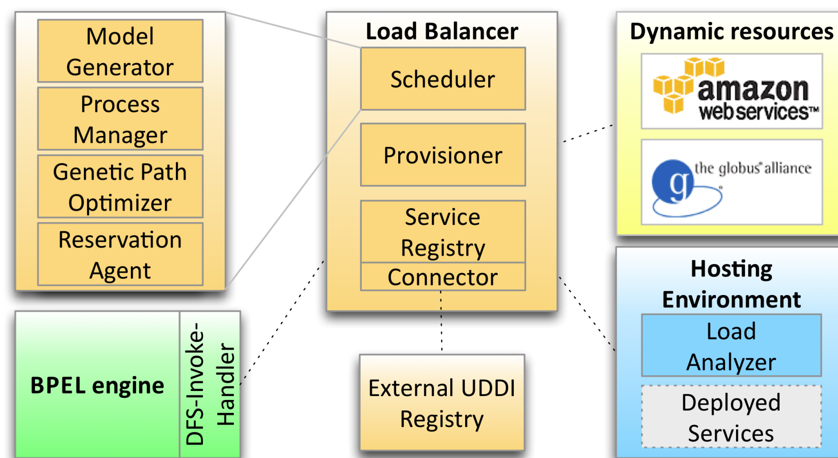


Figure 6.7: Architecture of the implemented system for data flow aware scheduling. Note that, compared to the basic scheduling framework only the Invoke Handler needed additional functionality (data flow graph encoding) and the scheduling component was replaced. All other components stayed the same.

To realize the main goal of minimizing changes to the workflow engine, interfaces and observers provided by the workflow engine are used. The *DFS-InvokeHandler* (IH) implements the `IAeInvokeHandler` interface provided by the workflow engine. The IH is embedded into workflows

using the aforementioned deployment descriptor and uses the format described in Listing 6.1 and Listing 6.3 to encode annotated data. It is therefore executed by the BPEL engine instead of the standard IH.

The sequence of calls starting from the invocation of a workflow and its execution using a computed invoke-to-resource mapping is sketched in Figure 6.8 and described below.

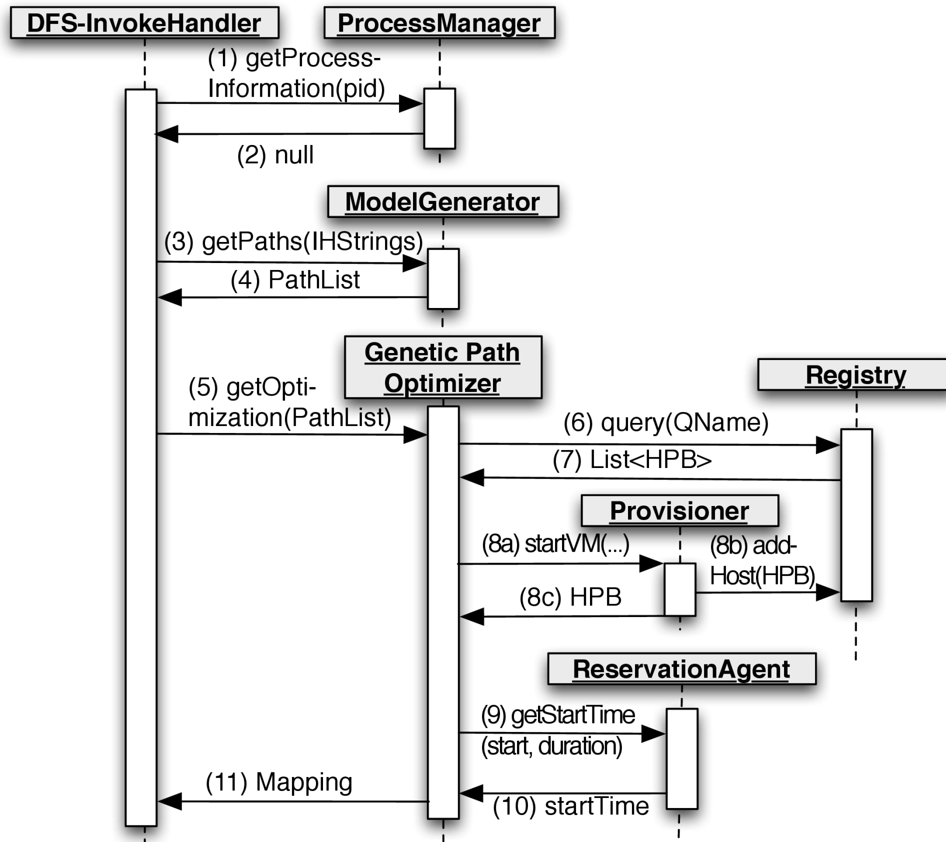


Figure 6.8: Sequence diagram of the data flow scheduler.

(1) The IH generates a unique process identifier (pid) that is used in subsequent calls to retrieve information computed by the scheduler. Since at this stage no information is present yet, `null` is returned (2). The information encoded in the IH string is passed to the *Model Generator* that in turn generates a graph representation of the data flow, computes critical paths (3) and returns a list of the paths (4). (5) Using the generated list, the genetic path optimizer performs the following steps for each path separately: (6, 7) For every node of the path, the *Registry* component is queried for the *portType* of the required services and returns all available resources offering the service. (8) If the returned list is empty, obviously no machine is available is that offers the requested service. In this case, the *Registry* is queried again and returns a *VM Descriptor* including information about a *VM Image* that contains the service. Using this, the *Provisioner* component carries out the following

steps: (8a) Provisioning of a new virtual machine according to the *VM Descriptor*. (8b, c) The provisioned VM is registered in the *Registry* and thus can be used by the system in subsequent service invocations. At this point, it has been assured that – for every node in the path – at least one machine offering the required service exists. Next, the GA is executed. First, it randomly generates a population, i. e., a set of random assignments of nodes to resources for each critical path. The following steps are iterated (see Section 6.3.2). Using the *fitness function*, for every element in the population, the total runtime of a path is calculated. To do so, the *Reservation Agent* of a resource is queried for the earliest slot in the reservation schedule. The total runtime is the sum of the waiting time for the reserved slot and the actual runtime (9, 10). Successively, the steps *selection*, *crossover* and *mutation* are performed as described in Section 6.3.2.

Finally, the GA terminates and the mapping with the lowest total runtime is chosen (11). If the waiting time for a time slot within the critical path exceeds the upper barrier δ, it is conceivable that – by adding further machines with the required service to the pool – the overall runtime can be improved.

To detect this possible improvement, this situation is simulated by adding a placeholder machine and by re-executing step (9) of the GA. If the result is better, the placeholder is replaced by a newly provisioned Cloud resource. This procedure is repeated until the waiting time for all resources is below δ.

Once all paths have been optimized, the assignments are stored in the *Process Manager* such that the schedule does not have to be recomputed for subsequent *invoke* operations of the same workflow instance. In this case, step (2) does not return `null`, but the previously computed mapping.

Workflows that contain while activities require a special handling. Indeed, *while* induces a cycle in the control flow of a workflow and might also cause a loop in the corresponding data flow graph. Data flow loops occur when the output of a workflow step is used as the input for a preceding workflow step within the same *while* loop. As described earlier, loops in the data flow graph have been removed in order to guarantee that the Breadth First Search terminates. At runtime, cycles can be detected via the reservation mechanism, since reservations are removed after the execution of the *invoke* activity. Therefore, after the first iteration of a *while* loop, no reservations for enclosed activities exist anymore. When the next iteration is executed, the system does not find any reservations and has to perform a re-scheduling on the corresponding subgraph. The subgraph has to be transformed in such a way that the node whose outgoing edge(s) (nodes 4 and 5 in Figure 6.9) induce(s) the cycle(s) becomes the root node of the transformed graph. If more than one of these nodes exist, the node with the highest outgoing amount of data is chosen (node 5, amount of data is represented by line width) to be the new root node.

6.4 Multi-Objective Scheduling

This section presents the design of the developed framework called CaDaS (Cost And Data flow-Aware Scheduler) that uses a multi-objective heuristic algorithm to determine scheduling decisions of invoke activities of BPEL workflows. The framework basically consists of two logical units: one

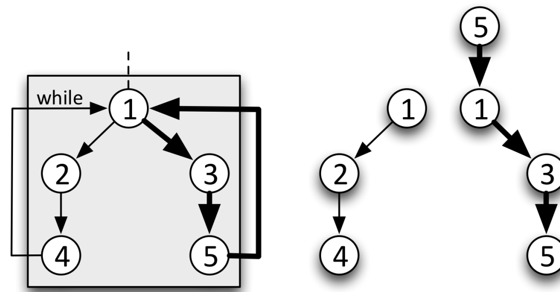


Figure 6.9: Subgraph within *while* and its decomposition into critical paths.

represents the infrastructure and provides functionalities to manage virtual machines, while the other consists of the algorithm itself (see Figure 6.7). The infrastructural model provides information about network and (virtual) machine characteristics that can be incorporated by the algorithm.

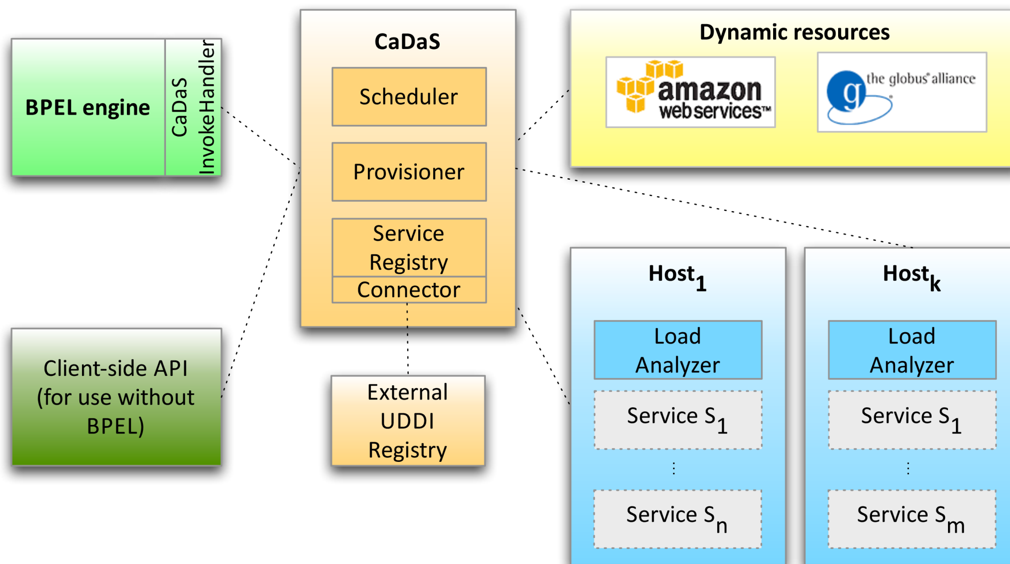


Figure 6.10: Components of the developed architecture from a bird's-eye view. In differs from the previous solutions, only in the replacement of the invoke handler and the scheduling component.

When only the current load of the target machines is used for the assignment (as in the first presented approach), it is fairly likely that the workflow system performs poorly due to the lagging nature of load monitoring and frequent data transfers between different machines (which takes place, e. g., when services A and B run on different machines). This is particularly important when the system is employed in a hybrid Cloud infrastructure or in a federation of Clouds. In this case, one has

to deal with fast local-area/intranet connections within the resource pools (nowadays 100 MBit/sec up to 10 GBit/sec) and a potentially much slower wide-area/Internet connection *between* the pools.

The previous approaches were tailored towards a reduction of the workflow runtime. Cost was not considered at all, although they might play an important role for users. For instance, a user might want to run a non-time critical workflow as cheaply as possible. Since Cloud providers typically bill both, data transfer (mostly, traffic within a data center/region is free) and consumed CPU time, the scheduling decisions must be mindful of both of these factors. Another user might prefer a solution inbetween the two extremes, best performance and lowest cost. Therefore, the scheduling decisions should be based on the user's preferences and enable her/him to rate the importance of both factors.

Practical experiments with Amazon's Cloud infrastructure during the last three years have revealed that the connection speed *within* Amazon's data centers at any time met the promised 250 MBit/sec. However, the connection speed from our institution (connected to the Internet at 400 MBit/sec) *to and between* Amazon's data centers/regions heavily varies – we have not yet found a thoroughly explanatory pattern for this phenomenon (time of day seems to play a role). Figure 6.11 shows the available connection speeds between different regions of Amazon. A high variance between different connection attempts can be detected. This issue calls for an infrastructure that continuously monitors the available bandwidths and immediately makes the measurement results available to the scheduler.

6.4.1 Framework Components

As stated before, CaDaS contains three main building blocks: (1) scheduler, (2) provisioner, and (3) service registry. The scheduler determines mappings between workflow tasks and resources. Thereby, it creates an internal representation of the workflow that is subsequently used by the *Genetic Algorithm* on which the scheduler is based (see Section 6.4.2 for a detailed description). The provisioner is responsible for managing Cloud resources, i. e., for the provisioning of new resources when needed and for the de-provisioning of unused resources.

The registry contains all infrastructural data, such as locally available resources, already started virtual machines, connection speeds between resources and resource reservations that result from computed mappings. The model used by the registry as the basic representation is hierarchical. Cloud providers typically have different data centers located in different countries or regions. These geographically distributed data centers are modeled by *zone*s. Each data center might offer different virtual machine types that can be started by a user. Each one of these types is represented by *vmType* element. Furthermore, one specific virtual machine is able to boot different images, i. e., different operating systems – represented by *image* elements in the model; see Figure 6.12 for a simplified view.

Based on this coarse grained model, the need of several other attributes arises. First, each zone must be annotated with cost attributes (*feeIn* and *feeOut* in Figure 6.12) to model data transfer costs. A distinction between internal transfers within a zone and external transfers between dif-

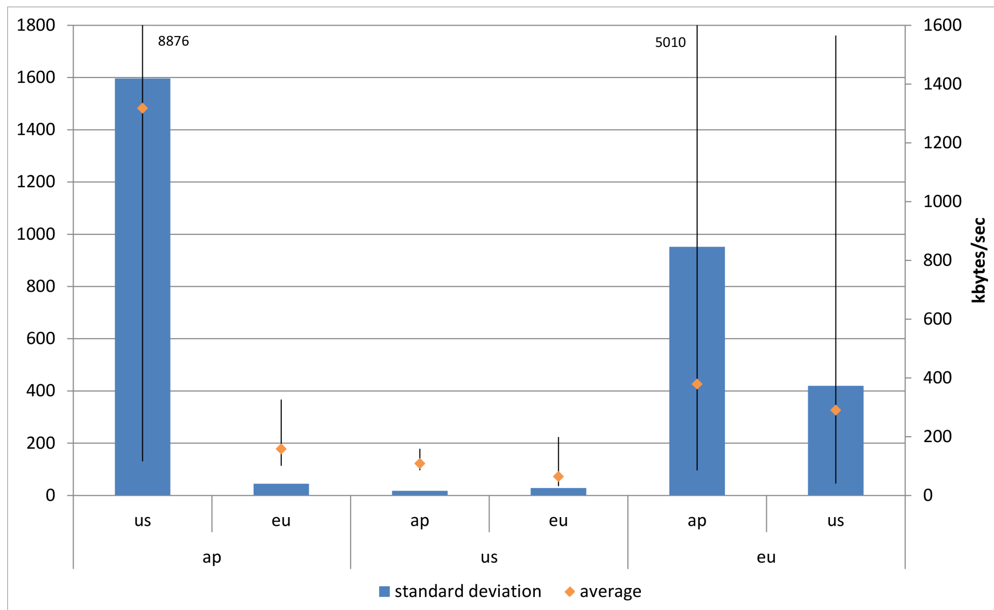


Figure 6.11: Real-life measurements of data transfer speeds between different data centers of the Cloud provider Amazon during a work day in January 2011. The left axis of the diagram depicts the standard deviation, which is shown with the blue bars. The lines show minimum and maximum connection speeds in kbytes/sec (right axis). The orange diamond marks the average connection speed. The lower part of the description of the x-axis denotes the connection source and the upper part denotes the destination of the connection. Besides the high degree of fluctuation (especially connections to and from the data center in Asia), it is noteworthy, that connection speeds are not symmetric in most cases (possibly due to different packet routes for the different connection directions). On a next day the measurements were repeated and most connection speeds were much higher (about 5 Mbyte/sec), but nevertheless the standard deviation was still immense. Therefore, continuous monitoring of connection speeds is mandatory. The locations of the cloud data centers are: ap = Singapore, eu = Ireland, us = Virginia.

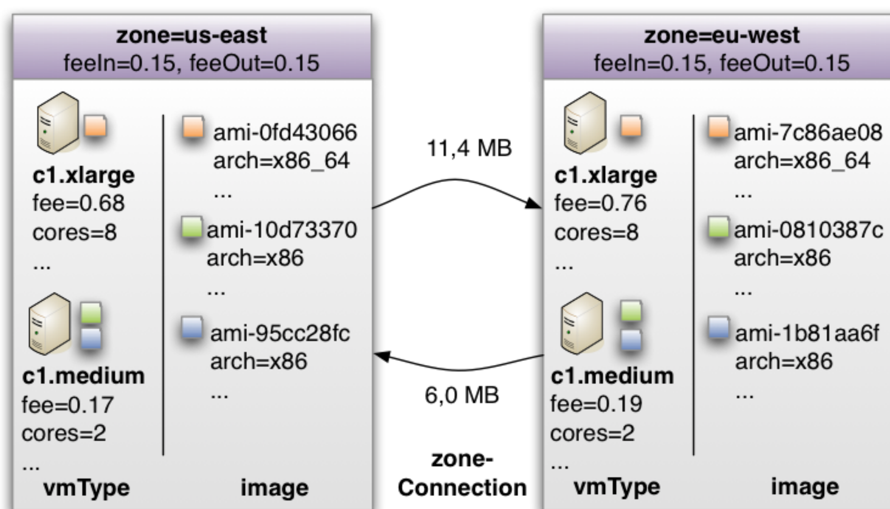


Figure 6.12: Internal representation of a Cloud topology and its resources.

ferent zones is important. Furthermore, connections and corresponding link speeds between the different zones must be determined. This distinction enables the algorithm to take communication speeds (and the induced cost) into account. Using this information and the user's preference values, the algorithm can decide whether it is a better choice to wait for a local resource to become available, or to move the data to a different zone and execute the task there (which may induce additional cost and an execution delay). A *zoneConnection* describes such a (one-way) connection including their connection speed. Since only the theoretical value can be annotated beforehand, the zoneConnections need to be updated continuously during runtime in order to provide reasonable and up-to-date information about the network.

To prevent a network overload caused by monitoring, the monitoring is performed passively by the *SystemAnalyzer*, an enhancement of the previously introduced host analyzer. Passive monitoring means that no packets are transmitted to measure the transfer rate, but the connection speeds are determined through performed data transfers between different zones. Furthermore, this information (obtained from distinct machines in different zones) is used representatively for all machines in a certain zone – thereby, a newly provisioned machine is equipped with the measured value, rather than a pre-set theoretical one. Without this mechanism, the machine would have a (possibly highly inaccurate) value until it has transmitted data for the first time. Even worse, an inaccurate value might have the effect that the machine is never used – resulting in a never-updated zoneConnection. Since measured values are only meaningful for a particular moment in time, exponential smoothing is used to profile the connection speed over time.

Each image must contain information about the zones it is available in and about which virtual machine types are compatible with this specific image (e. g., a 64-bit Debian Linux cannot be hosted by a 32-bit virtual machine). Each vmType must contain annotations with the computing costs per

hour and some hardware related attributes, such as the number of (virtual) CPU cores, and the speed of each core (measured in EC2 Compute Units, ECU).

The model described above can treat static structures like data centers and machine types, but to operate efficiently, the scheduler needs to model runtime information for the coordination of invoke activities on different machines. The fulfillment of this task is supported by a reservation system. This system operates on the granularity of a CPU core and prevents peak-load situations (per host) by exclusively assigning a core to a single invoke operation. Reservations have a specific period of validity that is derived from service runtimes and necessary data transfers (both are annotated to the workflow, see the previous sections for more details). The beginning of a web service invocation (t_{data}) and the start of the period of validity of a reservation (t_{start}) differ. The reason for this is that the data transfer has to take place first – and meanwhile does not consume mentionable computational resources. The offset between these two and t_{data} itself are automatically determined by the estimated data transfer time which is computed as $\min(kbpsOut_{src}, kbpsIn_{dest})$. As depicted in Figure 6.13, the system calculates reservations to avoid resource usage overlaps. This includes that the stage out of a workflow step $i-1$ does not interfere with the stage in of workflow step i, if executed on the same machine.

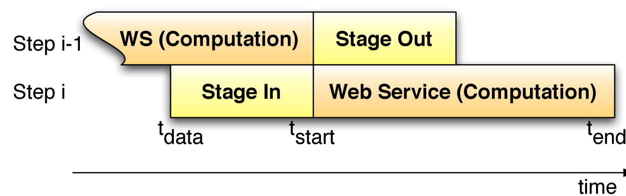


Figure 6.13: Reservations and preceding data transfer.

Despite the fact that virtualization provides a homogeneous environment to host the middleware, it does not abstract from all the characteristics of the underlying machine. Since Amazon uses different host machines with different CPUs inside (e. g., Intel Xeon and AMD Opteron) [162], virtual machines differ in their performance – although they might be in the same instance type group. Among other aspects like changing connection speeds of networks, these performance deviations have a considerable impact on service runtimes. Therefore, it cannot be expected that these reservations can be satisfied exactly and their duration limits are interpreted with $+10\%$ of the period of validity of the reservation before a reservation eventually becomes invalid. If the service finishes earlier than expected, a *backfilling* of subsequent invoke activities should be attempted by utilizing gaps between existing reservations. If this is not possible, a rescheduling and therefore a recomputation of existing reservations is triggered.

6.4.2 Multi-objective Scheduling Algorithm

As already mentioned, none of the existing workflow based scheduling algorithms can directly be applied without introducing additional constraints. The reason for this deficiency is that BPEL

is a non-DAG and Turing-complete language (cf., Section 6.3.2). In addition, the time necessary for computing a schedule should ideally not cause a significant increase of the workflow runtime, meaning that calculating all possible task-resource mappings – and selecting the best one – is not feasible, since the complexity would grow exponentially in the number of tasks and resources.

The usage of a heuristic algorithm ensures an acceptable runtime of the scheduling algorithm itself. This provides a compromise between the scheduling runtime and the optimality of the assignment. Within heuristic approaches, a genetic algorithm is a good trade-off between the runtime and the quality of the result. A genetic algorithm adopts evolutionary principles to find a solution for an optimization problem. In particular, it uses operations such as mutation, selection and elitism to calculate the mappings between workflow tasks and resources [21, 72].

Since in Cloud environments resources are billed in a pay-per-use manner, the objective function differs of the algorithm from those of typical scheduling algorithms. In addition to workflow runtime, workflow execution cost has to be considered. Hence, the solution candidates are not totally, but only partially, ordered and a selection function that computes a single solution based on both objectives is required. The following types of genetic algorithms are capable of dealing with multi-objective optimization problems: Non-dominated Sorting Genetic Algorithm (NSGA II) [37], Strength Pareto Evolutionary Algorithm (SPEA2) [212, 213] and Pareto Archived Evolution Strategy (PAES) [120]. Yu et al. argue [209] that SPEA2 has good convergence properties (much better than PEAS has) and – due to the high degree of parametrization – is adaptable to different application areas. For this reason, SPEA2 has been chosen as the basis for this work. The pseudo-code in Listing 6.4 illustrates the order of the operations.

```
generateInitialPopulation();

while ( !abortCriteria ) {
  doSelection();
  doCrossOver();
  doMutation();
  paretoSet = calcParetoSet();
}
return paretoSet;
```

Listing 6.4: Pseudo-code for a multi-objective genetic scheduling algorithm.

A scheduling algorithm can operate on the entire workflow graph (*workflow based*) or on just a single task (*task based*) of the workflow. In the latter case, (data) dependencies are not taken into account, which makes such an algorithm infeasible for the presented approach. Instead, the chosen algorithm operates again on an entire workflow (as the dataflow based described in Section 6.3 does) and calculates the task-resource mapping.

At first, the algorithm generates an initial population, consisting of a set of *individuals*. An individual in turn represents a complete (randomly chosen) mapping between all workflow tasks and certain candidate resources. Candidate resources may either be already running resources or virtual machines (to be provisioned later). Since Cloud providers offer different virtual machine types and

operate data centers at different geographical locations, representatives for all these combinations are added to individuals of the initial population. The size of the population depends on the number of existing resources (both physical and virtual), the number of virtual machine types per zone, and the workflow graph size.

After that, the algorithm iterates in order to achieve an improvement of the fitnesss of the individuals in the population. Fitness is defined as the quality of mapping with respect to cost and runtime. Thereby, both runtime and cost are aggregated over all workflow steps. The cost and runtime of a each step in turn depend on the underlying resource (since resource prices vary between different zones) and on the geographical location of the resource from which the required input data is staged. The calculation of both values, the system uses the values from zones and zoneConnections (see above).

Within *doSelection*, the algorithm selects the fittest candidates, i. e., those who dominate others with respect to the objective function. Afterwards, using *doCrossOver* and *doMutation*, new individuals are created – in contrast to the generation of the initial population, the new individuals are based on properties of the fittest existing ones. *calcParetoSet* collects the set of individuals that dominate the other ones. An example is given in Figure 6.14, in which the fitness value of one individual is defined as the number of individuals it is dominated by. Thus, a fitness of 0 means that no other individual dominates this specific one.

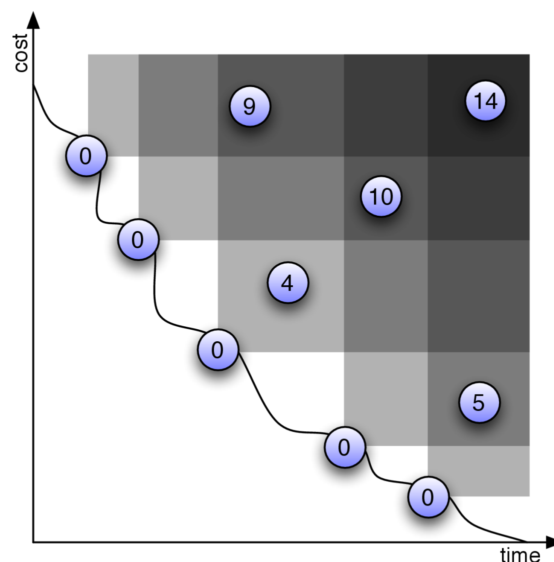


Figure 6.14: Fitness values of individuals and dominating ones (pareto front). Each circle represents an individual (only a few were selected for the sake of clarity). The encircled number (the fitness value), counts how many individuals dominate a specific one. Thus, the lower the value is, the higher the fitness of the individual is.

When the algorithm terminates (abort criteria are detailed below), based on the calculated pareto front and a user-supplied weighting function, which defines the relevance of the two dimensions

(cost and runtime) to the user, a single individual is determined. The solution represents the best mapping with regard to the user's preferences.

Clearly, the quality of the interim results of the algorithm cannot decrease while iterating. Indeed, since the worst results are removed in every step – in the worst case, the results just do not improve between iteration steps. If the pareto set remains unchanged in a series of subsequent iterations, the algorithm terminates. Furthermore, to provide an upper bound for the runtime of the scheduler, there is a hard limit of iterations, which depends on the population size.

6.4.3 Implementation

The solution combines the versatile and powerful composition and control mechanisms of BPEL with an adaptive runtime environment without breaking the compatibility with existing standards. The system seamlessly integrates on-premises and on-demand resources provided by IaaS infrastructures like Amazon EC2. The actual implementation is tailored towards EC2 and uses the Typica library to interface with the infrastructure. Since the offers of popular IaaS vendors show many similarities (such as comparable cost models, usage of virtualization/virtual machine images), the system could – with minor changes – be adapted to support those infrastructures, e. g., Eucalyptus [144].

To realize the goal of minimizing changes to the workflow engine, we make use of interfaces and observers provided by the workflow engine. For example, the *CaDaS-InvokeHandler* (see Figure 6.10) implements the `IAeInvokeHandler` interface. The custom invoke handler (IH) is embedded into workflows and enriched with annotated data (cf., Listing 6.3). It is therefore used by the BPEL engine instead of the standard IH. The IH is executed by the BPEL engine whenever an *invoke* operation in a workflow is executed. It checks whether a resource mapping for the *invoke* exists or not. If not, the scheduling algorithm is utilized to compute the mapping.

One important detail of the implementation of the provisioner is its tight integration with the reservation mechanism. It has to be taken into account that virtual machines have a startup delay due to boot time and the installation of the required web service stack. Assume that a reservation for a (not yet provisioned) virtual machine for time t exists. Then, the provisioner would begin the provisioning at time $t - d$, where d is a configurable delay (set to 90 sec to be on the safe side). Furthermore, the provisioner keeps track of already running virtual machines and deprovisions them shortly before an accounting period (one hour for EC2) expires if there are no pending reservations and currently running computations.

One interesting implementation detail concerning the *doSelection* step in the scheduler is that random selection is not based on a uniform distribution, but rather uses a weighted distribution that favors individuals with better fitness (Roulette Wheel Selection Scheme [192]). As a consequence, the quality of the population increases quickly, which in turn results in a faster convergence towards the optimal solution and in a shorter runtime of the algorithm. Moreover, the algorithm uses a special variant of the crossover operation. Normally, the crossover simply exchanges two randomly chosen mappings of individuals without considering topological information of the un-

derlying problem, the workflow graph. The implemented variant, *branch crossover*, uses topological information and exchanges whole branches (more accurately, mappings for complete branches) between individuals. The underlying assumption is that – due to the preceding selection operation – the individuals quite likely contain good mappings for subgraphs, i. e., branches. Like the optimizations in the selection operation, the branch crossover reduces the number of required iterations of the algorithm.

6.5 Reliability for Service Workflows

In this section, the design of the solution that meets all the requirements stated above is described. Implementation details will be given in Section 6.5.3.

The proposed solution adds a policy-based fault handling mechanism to BPEL without making any changes to the language standard. Using policies, it allows to enable and disable retry and also substitute actions. To keep the number of required policies low, a *Fault Classifier* (FC) divides faults into groups. Instead of defining a policy for every fault separately, policies only have be specified for each of these groups instead. Furthermore, the policies permit to set parameters such as the maximum number of retries, what kind of resources (e. g., dedicated hosts, Cloud resources, etc.) may be used for substitutions and so on. The overall architecture is sketched in Figure 6.15. The components are described below.

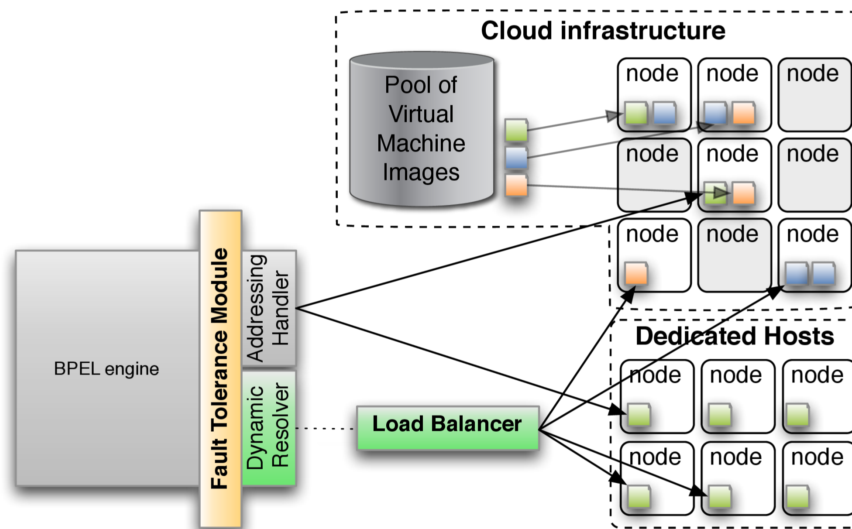


Figure 6.15: Bird's-eye view on the proposed architecture.

The proposed solution replaces the default invocation mechanism of the BPEL engine with the *Fault Tolerant Invoke Handler* (FTIH) that executes and monitors every single invoke operation. If an invocation fails, its answer is analyzed (step 3a in Figure 6.16) and classified by the FC. In this case, the *Policy Processor* (PP) applies all policies that have previously been defined for the classified fault

(step 4) to determine the next action to be under taken. Depending on the applied policy either the call may be retried or the service may be substituted (using the *Dynamic Resolver*, step 5) by an existing one or by starting a machine in the Cloud, or the process fails, if both mechanisms have been disabled in the applied policy. If the invocation was successful, the response is passed back to the engine (step 3b).

This process is illustrated in Figure 6.16. The components and their interplay, as well as the policy schema, are discussed in detail in the following sections.

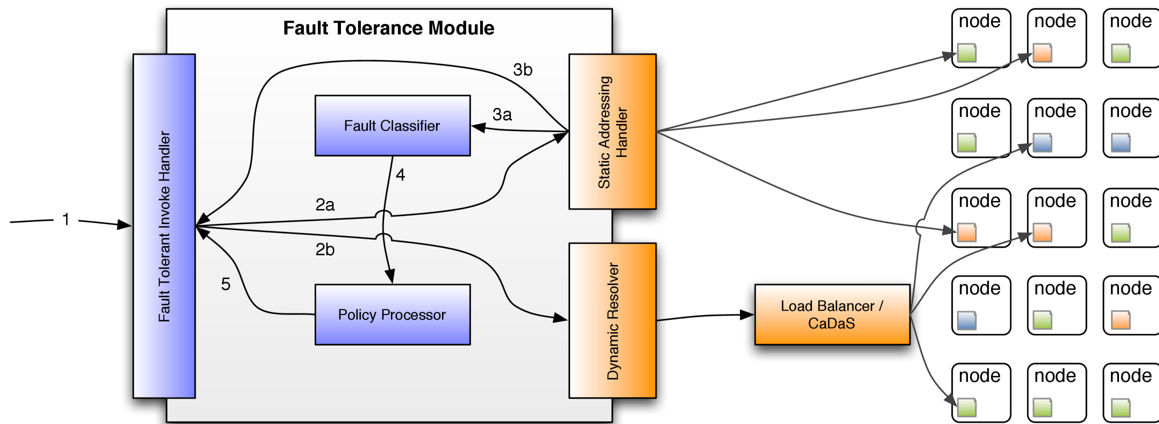


Figure 6.16: Sub-components of the Fault Tolerance Module.

6.5.1 Fault Tolerance Module

The Fault Tolerance Module encapsulates the whole functionality sketched above. The different functions (fault classification, processing of policies, service invocation) are realized as sub-components and described in the following.

Fault Tolerant Invoke Handler

The *Fault Tolerant Invoke Handler* (FTIH) is the only component that interacts with the workflow engine. One of the major design goals is to realize this interaction without the modification of the source code of the engine. Thereby, portability towards new versions of the engine shall be eased. Basically, the invoke handler is woven into a process using the deployment descriptor of the process. The engine then executes the custom invoke handler instead of the default one.

BPEL allows two different addressing mechanisms for target services: namely *static* and *dynamic* addressing. In the first case, *FTIH* uses the standard BPEL *Addressing Handler* (step 2a in Figure 6.16) to execute the call. In the latter case, the actual target address must be resolved before the execution of the *invoke* operation (step 2b in Figure 6.16). This is accomplished using the previously introduced *Dynamic Resolver* (see Figure 6.15).

When the response from the invoked service arrives, its content is checked for fault messages. If there is such a message, it is passed to the *FC* (see Section 6.5.1). The result of the classifier is then passed to the *PP* that applies all configured policies. The result contains zero or more recovery strategies annotated with priorities (*Retry* is to be performed before *Substitution*) and is again inspected by the *FTIH*. No recovery strategy might be available, if *Retry* and *Substitution* have been explicitly disabled or if the number of retries has exceeded the configured value. If a recovery strategy is available, it is used to repeat the invocation. A counter value representing the number of retries in the strategy is then incremented for each repetition.

If the *Retry* strategy is applied, the previously invoked service is invoked again. If not, the *Dynamic Resolver* is executed to perform a dynamic scheduling on those resources, which are declared in the *Substitution* strategy.

Fault Classifier

Basically, the *FC* categorizes fine-grained exceptions and invocation faults into more coarse-grained fault categories. This reduces the number of polices that need to be defined, since policies are now applied to whole groups of faults instead of individual faults. It is noteworthy that the *FC* must be able to classify two types of faults: (1) invocation faults that may occur during service operations, and (2) faults that are induced by the fault handling framework itself (*VMFault* and *SchedulerFault* classes in Table 6.1). For the second kind of fault, special handling policies are required – for instance, if the startup of virtual machines (i. e., of Cloud resources) fails, the *Substitution* strategy has to be adapted to keep the framework from further trying to acquire virtual machines.

Due to the huge spectrum of possible faults that may occur at runtime, the *FC* has to be able to react to new failures in an adaptive manner. This is to be considered in future work.

Policy Processor

In general, policies are attached to individual target services via the BPEL *partnerLink*s. However, in certain situations the definition of global policies that are automatically applied to all *partnerLink*s might be advantageous. One such case is the aforementioned *VMFault*. For instance, the occurrence of a *VMFault* should disable the provisioning of Cloud resources (until the cause has been rectified). For these reasons, the presented architecture incorporates not only local but also global policies. In general, local policies should be able to overwrite global policies. But to obviate undesired overwriting (which is reasonable for *VMFault*), policies can also be declared as final. The schema-like definition of a policy is described in Listing 6.5. A policy container can include one or more *Fault* elements. The global and local policies have to be kept in separate containers. The corresponding files are stored in the deployment archive used by the engine. To be able to identify the policies for each partnerlink, the filenames should be stated in the invoke handler string.

```
<Faults>
<Fault name="NCName" final="true | false">
  <byCause name="FaultCategory"/>
```

```
  <OriginHost retry="true | false">
    <MaxTries value="int"/>
  </OriginHost>?
  <Substitute resources="NONE | PHYSICAL_ONLY | EXISTING
                         | NEW | DIFFERENT | ALL">
    <MaxTries value="int"/>
  </Substitute>?
</Fault>+
<Faults>
```

Listing 6.5: Schema-like definition of a policy.

Each policy determines a behavior for a particular fault category. The fault category itself is defined by the `byCause` element. An *OriginHost* or a *Substitute* strategy (or both) must be specified that influence the behavior of the *FTIH*.

When the PP is invoked, a fault category, which has previously been determined by the *Fault Classifier*, is taken as input. The PP uses a map-like structure to perform a look up for a policy of a given fault. When performing look up operations for policies, not only the ordering of global and local policies but also the declaration of final policies have to be considered.

As Chan et al. state in [27], it is not reasonable to try to recover all kinds of faults. Recovery mechanisms have to interfere, if, for instance, a network timeout or a socket reset occurs. Therefore, the type of faults is analyzed in order to decide whether a rescheduling is sensible. Faults thrown by the service explicitly (e. g., Java exceptions like *IllegalArgumentException*) should not lead to a rescheduling, since after rescheduling the error is most likely to occur on other machines as well. If an invoke is performed on a stateful WSRF service (except the create operation), a rescheduling does not make sense, due to of the state of the resource. One would have to repeat all previous operations which led to the creation of the resource. Furthermore, only a configurable amount of retries is performed so as to assure to prevent the system from being destabilized by the repeated invocation of faulty services. Finally, only a single virtual machine may be started. Figure 6.16 illustrates the basic algorithm.

The Fault Tolerance Module handles service invocations and their faults. It is important that the process designer is able to decide, whether the recovery mechanisms are enabled or not. Basically, two types of failures may occur: (1) failures introduced by the presented infrastructure itself and (2) failures during service invocations. For the first type the strategy is fairly simple: if the error occurs during the start up procedure of a virtual machine, only existing machines are used for a rescheduling, since it is quite likely that another start of a virtual machine would cause the same error. In this case, the algorithm, which remedies invocation failures, is applied (see Figure 6.16), see the description below for further details.

6.5.2 Dynamic Resolver

The *Dynamic Resolver* is able to dynamically schedule web service calls to underutilized machines. Available machines and the services they offer are stored in the *Registry* component (cf., Section 6.2.2). The DR is able to perform load balancing between machines all of whom host the required services (cf., Section 6.2–6.4). Different scheduling algorithms may be plugged into the *Scheduler* thereby allowing the use of different load balancing approaches. Furthermore, the scheduler handles the provisioning and the deprovisioning of Cloud resources (*Provisioner* in Section 6.2.2). The integration of fault handling requires an extension of this component, so that it can cope with the restrictions set by policies. For instance in case of a *VMFault*, the *Scheduler* must be able to reschedule calls on existing machines instead of providing a new Cloud resource.

The resource types mentioned in Listing 6.5 need to be understood by the scheduler in order to guarantee a correct scheduling in compliance with the applied policies. For example, for the CaDaS scheduler, this means that reservations have to be removed and the schedule for the remaining graph has to be recomputed.

6.5.3 Implementation

The implementation is based on the BPEL4Grid engine and enhances it with a custom invoke handler.

Before the implementation of the components is discussed, first the embedding of the solution is described into the BPEL engine without changing neither the implementation of the engine nor the BPEL standard. Since the fault tolerance mechanism operates per service, it is a natural choice to use the BPEL *partnerLink*s for integrating the framework into process descriptions. *PartnerLink*s connect *invoke* activities of processes with web services; those links – besides others – comprise two important elements: (1) *partnerLinkType* and (2) *partnerRole*; the latter one defining the type of the *endpointReference* (EPR) and the *invokeHandler*. (1) is static information and is integrated into the BPEL process at design time. While it refers to the WSDL description of a *portType* of a partner service, (2) refers to runtime information. EPRs may be either *static* or *dynamic*. They are configured using the unstandardized process deployment descriptors, meaning that additional information can be specified without breaking compatibility with the standard. Listing 6.6 shows the integration of the implemented custom invoke handler, as well as the URL-encoded definition of policies.

```
<partnerLink name="ecgAnalyzerPL">
  <partnerRole endpointReference="dynamic"
     invokeHandler="java:FaultTolerantIH?
       GlobalPolicy="global.xml";LocalPolicy="local.xml" />
</partnerLink>
```

Listing 6.6: Integration of a custom invoke handler into the BPEL4Grid engine.

Fault Tolerance Module

The Fault Tolerance Module encapsulates all functionalities described before. The different functions (fault classification, processing of policies, service invocation) form sub-components that are explained in the following.

Fault Tolerant Invoke Handler. The *FTIH* handles all invoke operations whose *invokeHandler*s point to it. For carrying out calls, the standard invoke handler `AeInvokeHandler` is used. If the address of the target service has not been resolved yet (`endpointReference="dynamic"`), it first contacts the *Dynamic Resolver* (see Section 6.5.3). As soon as the call is finished, the answer is examined. If it consists of a fault, the code in Lines 12 to 18 of Listing 6.7 is invoked. The code performs a fault classification, applies matching policies, increments the fault counter (needed to check whether the maximum number of retries has been exceeded) and tries to repeat the invocation (note the `while`-loop at Line 8). It does not stop until `state.getActualStrategy` is `null`. The `state` object automatically returns the appropriate strategy that matches the applied policy (line 8), if any strategy is applicable. If *Substitution* is performed, restrictions concerning the type of the target system, such as `PHYSICAL_ONLY`, are passed on to the *Dynamic Resolver*.

```
ILoadBalancerState state = new FTState();                    1
PolicySet pS = new PolicySet();
// load policies configured in invoke handler                 3
pS.load(queryMap, invoke.getProcessName());
// ...                                                        5
// Repeat
while(state.getActualStrategy() != null) {                    7
  // perform invoke ...
  IAeWebServiceResponse response = super.handleInvoke(...);   9
  if (response.isFaultResponse()) {
    // ... fault occurred                                     11
    FaultClassifier fc = getFaultClassifier();
    IFault fault = fc.classify(response);                     13
    APolicy policy = pS.getPolicy(fault);
    policy.apply(state);                                      15
    state.incrementFaultCounter();
  }                                                           17
}
```

Listing 6.7: Excerpt of the fault handling mechanism in the fault tolerance module.

Fault Classifier. The fault classifier has to handle two kinds of exceptions: (1) local exceptions that arise within the framework itself (for instance, exceptions during scheduling), and (2) remote exceptions that occur when *invoke* operations fail. In the first case, plain Java exceptions have to be analyzed, whereas in the latter case SOAP fault messages containing the precise error must be analyzed.

Because of this distinction, the *Fault Classifier* defines two different entry points:

- `IFault classify(Throwable)` for local exceptions and
- `IFault classify(IAeWebServiceResponse)` for remote exceptions.

The classification of Java `Exception`s and `Throwable`s is performed by establishing a map-like structure that contains the mapping between the `Exception` and the defined `IFault` implementations. Assigning a SOAP fault to an `IFault` is more difficult, because the structure of a SOAP fault depends on the framework used for the implementation of the service (e. g., Apache Axis [12], gSOAP [186] or the Microsoft Internet Information Service [102]). Each single framework generates its own type of error messages. For this reason, the classifier first tries to identify the framework before introspecting further error details. Subsequently, the available error information is mapped to an `IFault` implementation (see Table 6.1).

<table>
<tr><th>Fault Category</th><th>(Java) Exception</th></tr>
<tr><td>VMFault</td><td>VMException.class</td></tr>
<tr><td>SchedulerFault</td><td>SchedulerException.class</td></tr>
<tr><td>MiddlewareFault</td><td>HeapException.class</td></tr>
<tr><td>VMFault</td><td>VMException.class</td></tr>
<tr><td rowspan="3">TransportLayerFault</td><td>ConnectException.class</td></tr>
<tr><td>SocketException.class</td></tr>
<tr><td>SocketTimeoutException.class</td></tr>
<tr><td rowspan="2">PermanentTransportLayerFault</td><td>NoRouteToHostException.class</td></tr>
<tr><td>UnknownHostException.class</td></tr>
</table>

Table 6.1: Sample mappings from individual faults to groups of faults.

`VMFault`s and `SchedulerFault`s can occur during the dynamic scheduling stage. It is assumed that `PermanentTransportLayerFault`s cannot be corrected by retries of the invocation. For example, if a `NoRouteToHostException` is caught, the same exception is very likely to be thrown again in a retry. A socket timeout error is considered as a `TransportLayerFault` that can be overcome by a retry. A typical `MiddlewareFault` is a `HeapException`. Therefore, global policy definitions that respect these implications exist. However, if needed, those can be overwritten by local policies. By default, `ServiceFault`s are not handled by policies. Since they are very likely to be business logic faults that should be handled at the process level and therefore need to be propagated to the engine instead of catching them.

Policy Processor. The PP is responsible for loading and managing policies. Loading and parsing of policies are lazily executed when the *PP* is invoked, i. e., policies are loaded if a fault message is received for a specific invoke activity. At load time, the precedences and immutableness of policies (attribute *final* in Listing 6.5) are considered. All classes representing the XML policies implement a `void apply(...)` method that is applied on the `state` object in the invoke handler (see line 17

in Listing 6.7). Each policy knows how the actual state of the Message Monitor can be manipulated. As it can be seen in Listing 6.5, the type of retrying (e. g., on the same host or by substitution) and the number of retries are modifiable by the user.

Dynamic Resolver.

The DR is only briefly discussed here; details can be found in previous sections. It has been extended to allow the transfer of policies to the Load Balancer. The load balancing component was enhanced so as to be able to handle restrictions induced by policies. For instance, the scheduling algorithm has to respect that even in situations with high load, it may not be permitted to provide new Cloud resources (`Substitute resources="PHYSICAL_ONLY"`).

The Load Balancer is based on the chain-of-responsibility design pattern. A new processing element has been added to this chain in order to able to apply the restrictions. The new processing element (called *Filter* in Figure 6.17) examines the setting of the restriction type. If necessary, it removes faulty (e. g., for the setting `resources="DIFFERENT"`) and inappropriate machines (`PHYSICAL_ONLY`), or biases the scheduling and provisioning components (`EXISTING`, `NEW`).

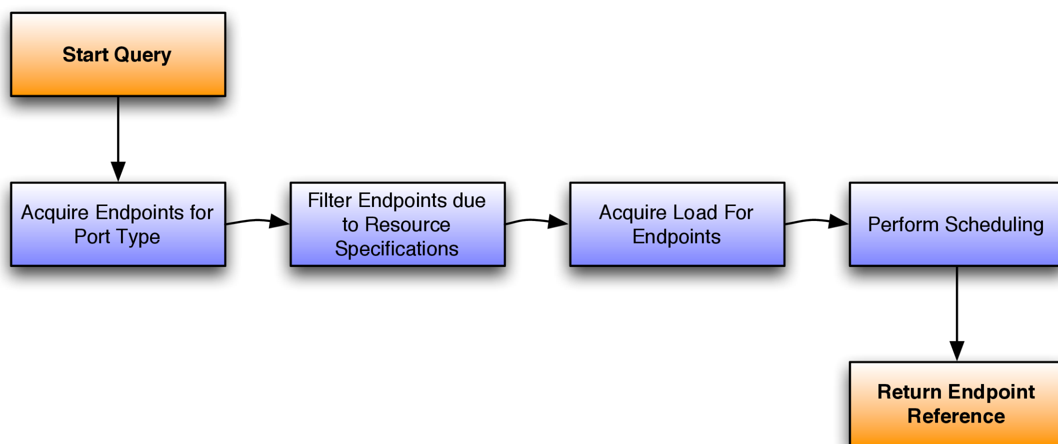


Figure 6.17: Chain of responsibility of the Load Balancer.

6.6 Reliability and Scalability of the Workflow Engine

Not only service invocations (regarding the client and the workflow engine) have to be sclabale and reliable, but also the workflow engine has to meet these requirements. Since in most cases, it is the only counterpart to a user and a client, respectively, the engine becomes a single point of failure due to overload situations and outages.

A workflow engine typically runs within an application container like JBoss or Apache Tomcat. These two containers, like most others do, provide mechanisms for clustering them in order to

improve reliability and to cope with high load situations. For such solutions, most often a set of cluster nodes is set up and configured so that it can be addressed by a round-robin DNS load balancer. Each cluster node replicates its states to the other nodes either by means of a database or by direct network connections.

Even though this solution allows for handling high load situations and for performing fail-overs, it has a major drawback regarding flash-crowd effects since the set of machines able to handle those requests is static and thus limited. Therefore, high load situations can only be handled as long as the number of requests does not exceed a certain threshold.

To overcome this limitation, virtual machines in Cloud Computing environments are used. Amazons Elastic Load Balancer (ELB) [9] in combination with Amazons Auto Scaling (AS) [8] offer an automatic scaling (i. e., provisioning and deprovisioning of virtual machines) for a specific application. In Figure 6.18 the overall setting is shown.

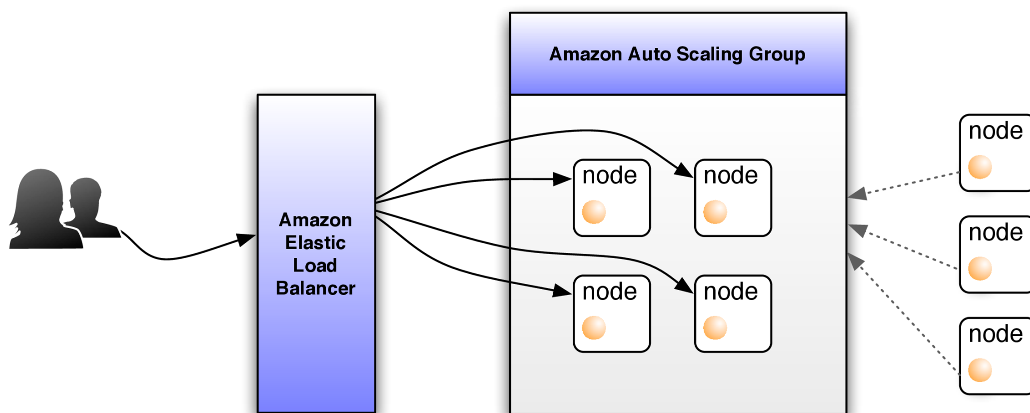


Figure 6.18: Overall architecture for a fault-tolerant hosting of the workflow engine.

The implementation comprises the configuration of ELB and of AS, since they can be integrated without any additional code. The setup of both via the command line API of Amazon is described in the following. Listing 6.8 depicts all steps necessary for the configuration of ELB and of AS. First, a load balancer for port 8080 (i. e., the standard port, a Tomcat instance is listening on) is created, upon which an auto scaling group is defined afterwards. This group determines the maximal number of hosts that Amazon is allowed to provision (in this example from Listing 6.8: 5 machines). Thereafter, policies are defined that change the pool of machines and the alarms by whom they are applied if the overall load of the machines is either higher than 70% or lower than 30%. In the first case, a new machine is added and the latter case one machine is deprovisioned.

Figure 6.19 shows that Amazon Web GUI after all commands have been executed. One machine (the desired capacity) is booted in order to serve requests.

```
$ elb-create-lb BPELEngine \
  --availability-zones us-east-1a,us-east-1b,us-east-1c,us-east-1d \
  --listener "protocol=http,lb-port=8080,instance-port=8080"
```

```
$ as-create-launch-config ASBPELEngineConfig --image-id ami-8c1fece5 \
  --instance-type m1.small --key ejuhnke --user-data-file startAS.sh
$ as-create-auto-scaling-group ASBPELEngineGroup --grace-period 90 \
  --min-size 1 --desired-capacity 1 --max-size 5 -l ASBPELEngineConfig↩
  ↪  \
  --availability-zones us-east-1a, us-east-1b, us-east-1c, us-east-1d ↩
  ↪ \
  --load-balancers BPELEngine

$ as-put-scaling-policy ASBPELEngineScaleOutPolicy \
  --adjustment=1 -g ASBPELEngineGroup -t ChangeInCapacity
$ as-put-scaling-policy ASBPELEngineScaleInPolicy \
  --adjustment=-1 -g ASBPELEngineGroup -t ChangeInCapacity

# if load > 70% scale out; if load < 30 scale in
$ mon-put-metric-alarm --alarm-name CPUHi --metric-name CPUUtilization↩
↪  \
  --namespace AWS/EC2 --statistic Average --period 60 --threshold 70 \
  --comparison-operator GreaterThanThreshold \
  --dimensions AutoScalingGroupName=ASBPELEngineGroup \
  --evaluation-periods 2 --unit Percent \
  --alarm-actions arn:aws:autoscaling:us-east-1:458761905469:↩
  ↪ scalingPolicy:c40dd3ca-5e9e-4b43-ba08-8261c0ae9618:↩
  ↪ autoScalingGroupName/ASBPELEngineGroup:policyName/↩
  ↪ ASBPELEngineScaleOutPolicy

$ mon-put-metric-alarm --alarm-name CPULow --metric-name ↩
↪ CPUUtilization \
  --namespace AWS/EC2 --statistic Average --period 60 --threshold 30 \
  --comparison-operator LessThanThreshold \
  --dimensions AutoScalingGroupName=ASBPELEngineGroup \
  --evaluation-periods 2 --unit Percent \
  --alarm-actions arn:aws:autoscaling:us-east-1:458761905469:↩
  ↪ scalingPolicy:4d219620-0b88-4ff9-a5c3-de9f63c7fb0b:↩
  ↪ autoScalingGroupName/ASBPELEngineGroup:policyName/↩
  ↪ ASBPELEngineScaleInPolicy
```

Listing 6.8: Setup of Amazons Elastic Load Balancer and Amazons Auto Scaling.

6.7 Summary

In this chapter, runtime adaptation of workflows in service-oriented architectures was addressed. The first three approaches in particular emphasize the scalability challenge and provide an automatic and workflow-based mechanism for scale-out and scale-in of a Cloud-based infrastructure. By performing not only a demand-based scale-out but also a scale-in, unnecessary costs are avoided.

Figure 6.19: Web GUI of Amazon showing the configured Load Balancer in combination with an auto scaling group. At the shown moment, only one machine is running being the minimum capacity in Listing 6.8. Clients can connect to the Load Balancer using a dedicated host name.

The three approaches mainly differ in metering the infrastructure load and in the precise dimensions that are considered as criteria during the scheduling process. As a next step, a fault tolerance module is developed that takes advantage of the statelessness of web service invocations and introduces a configurable but transparent fault handling in web service compositions.

7

Request/Response Aspects

Contents

7.1 Introduction . . . **125**
7.2 Aspect-oriented Programming in General . . . **126**
7.3 Request/Response Aspects . . . **126**
7.3.1 Framework for Request/Response Aspects . . . 127
7.3.2 Pointcut Description . . . 128
7.4 Implementation . . . **131**
7.4.1 Advice Interface . . . 131
7.4.2 Aspect Configurator . . . 132
7.4.3 Aspect Provider . . . 133
7.4.4 Security Manager . . . 134
7.4.5 Aspect Invocation Handler . . . 134
7.5 Aspects for Web Service Communication . . . **135**
7.5.1 Workflows for Multimedia Analysis . . . 135
7.5.2 Data-flow and BPEL . . . 135
7.5.3 Flex-SwA . . . 136
7.5.4 Data Transmission in BPEL via Flex-SwA . . . 136
7.6 Efficient Data Transmission Using AOP . . . **137**
7.7 Summary . . . **141**

7.1 Introduction

The previous chapters discussed (non-functional) requirements that emerge during the development and the invocation of (web) services. In this chapter, the handling of such requirements affecting the interaction of two (or more) services is addressed. This is achieved by the development of a framework that applies the paradigm of aspect-oriented programming to the communication of web services by introducing *request/response aspects*. Afterwards, the framework is used to develop concrete aspects for the multimedia data analysis use case (cf., Section 2.2.1).

Parts of this chapter have already been published [114, 166].

7.2 Aspect-oriented Programming in General

This section gives a short introduction to aspect-oriented programming in order to better understand its concepts and its application to the domain of service-oriented architectures. More detailed information can be found in, e. g., [58, 118].

Aspect-oriented programming (AOP) provides a high benefit for software developers when modularizing cross-cutting concerns [128]. In aspect-oriented software development (AOSD), a cross-cutting concern is identified and realized as an *advice*. The *pointcut language* defines a set of *join points* where these advice are integrated. In this manner, AOP eases the development of reusable and maintainable code.

According to Kiczales et al. [118], AOP has the same significance for cross-cutting concerns as object-oriented programming (OOP) has for encapsulation and inheritance. They argue that cross-cutting concerns can be programmed modularly and that one can profit by this modularity through "simpler code that is easier to develop and maintain, and that has greater potential for reuse" [118]. Modularized crosscutting concerns are called *aspects*. AOP distinguishes between two approaches. While *static cross-cutting* affects the static type signature of a program, *dynamic cross-cutting* allows for intercepting a program at well-defined points in its execution trace. This work focuses on the latter approach.

Dynamic cross-cutting consists of several parts. Join points are well-defined points within the execution trace of a program. A set of those join points constitutes a pointcut, and their behavior is defined by advice, which are certain method-like constructs.

7.3 Request/Response Aspects

In this section, the pointcut description for request/response aspects in web service environments including a supporting framework is presented. This description is not dependent on any implementation language, whereas the framework itself is located at the web service middleware (i. e., at a software stack that provides support for SOAP communication, like Apache Axis). It provides the capabilities for interpreting pointcuts and for applying the corresponding advice. The framework consists of the following components:

- an extension of the web service middleware for the weaving of aspects into web services without changing neither their implementation nor their interface. An aspect configurator service, a request/response aspect weaver, and a security manager are included;

- tools that operate on the client side, and which in particular allow to seamlessly integrate in BPEL workflows.

This framework is an essential prerequisite for writing and weaving request/response aspects. Since it is independent of the service implementation, the framework respects the black-box idea of service-oriented architectures.

7.3.1 Framework for Request/Response Aspects

The execution of web services is typically performed by an installed middleware, such as a web service container and a SOAP processor. The principal idea of the proposed framework is the weaving of aspects into the SOAP message processing chain of the SOAP processor. Since the processing chain usually operates on XML documents and since weaving into it does not interfere with the service implementation, the framework remains independent of the concrete service implementation. Moreover, an aspect configurator enhances the existing middleware by not only allowing to weave aspects but also, e. g., to check whether based on the aspect definition and the enforced security constraints an aspect can be woven into a service. The framework for supporting request/response aspects is based on the following design considerations:

Dynamic weaving. Since not only the implementation language of a web service is typically unidentifiable but also the implementation is typically unavailable, and furthermore the client's and the service's administrative domain differ, aspect weaving must be performed dynamically. As mentioned before, the middleware container hosting web services should expose web service methods for adding and also for removing aspects during runtime.

SOAP message chain. Due to the contract-first conception of web services, their previously negotiated interface (i. e., the WSDL description) must not be changed by aspects. Ideally, aspects have to be placed in the SOAP processing chain to avoid a modification of the interface. They must be able to deal with SOAP request and response messages in such a way that their modification leaves their syntactical representation unchanged.

Caller/callee interaction. On the client side, the framework must be able to distinguish between different web service invocations and communication partners who are possibly affected by an aspect. An aspect involving two subsequent services can only be added to the first service if it can also be applied to the subsequent one. In case of an error, it has to be removed. Consequently, an atomic behavior of aspect weaving is required. This includes the possibility for a caller to check whether an aspect is available, and to determine which aspects are currently active for this particular caller. WS-Policy [201] can be used to expose this information.

Scope. Aspects are required to have a certain scope [179]. A scope specifies whether the aspect is valid only for the client who has woven it or whether it is valid for all callers.

Security. One the one hand, a security manager must ensure that only authorized clients and aspects use the framework. The security manager enforces that only aspects meeting certain

security requirements can be used or deployed. Due to established authentication mechanisms (e. g., SSL/TLS with client verification [182]), only authorized users are permitted to invoke the aspect configurator and to deploy signed aspects.

On the other hand, in order to prevent the weaving of malicious aspects, a security token must be integrated into the aspect, e.g., a signed hash of the aspect. Only aspects with a valid signature are allowed to be woven into the system. For example, aspects could be enabled to adapt only the non-functional and not the functional part of a service when being applied to the exchanged messages.

State. Since web services are stateless, aspects and advice should not have any internal state, besides the information stored in the external key value store (see Section 7.3.2) or in the message itself.

In Figure 7.1, the general design of the proposed framework for request/response aspects is shown. The framework operates on the server side as well as on the client side. The server side includes a configuration interface (to be called by the client), a configuration manager and a component responsible for the actual weaving of aspects. On the client side, an API offers methods for interacting with the server component in order to weave and validate aspects.

The interaction between a client and the framework is as follows. First, the Aspect-API of the client queries the aspect configurators at the Services S_1 and S_2 to find out if the needed aspect (Figure 7.1: steps a and b). If both sites do so and respond accordingly, then the aspect is added by the client (steps c and d) into the response of S_1 and the request of S_2. The actual SOAP request sent to Service S_1 (step 1) is answered by the response message (step 2). Afterwards, the client forwards the request (step 3) to Service S_2 that due to its request aspect can process it and eventually gives back a respond (step 4). In the use case for data transfer, an aspect that supports direct and efficient data transmission from Service S_1 to Service S_2 is used. The response message of Service S_1 does not include the data D in this scenario, since the utilized request/response aspects enable direct data transmission of data D to Service S_2.

7.3.2 Pointcut Description

Although in most cases pointcut descriptions and aspect definitions are given in an (extended) Backus-Naur Form, request/response aspects are defined in a more general way. First, a tuple definition is given, which is then mapped into a XML schema. This is due to the language independent nature of web services. Thus, a request/response aspect for web services is defined by the following tuple:

$$\mathcal{A} = \{\mathcal{P}, \mathcal{O}, \mathcal{F}, \mathcal{M}, \mathcal{I}, \mathcal{K}, \mathcal{D}, \mathcal{C}\}.$$

$\mathcal{P}$ ***(porttype)*** defines the qualified name (QName) of the web service porttype to which the aspect should be applied. This element is required to cope with a wildcard operator, meaning that all porttypes within a service are relevant for this particular aspect.

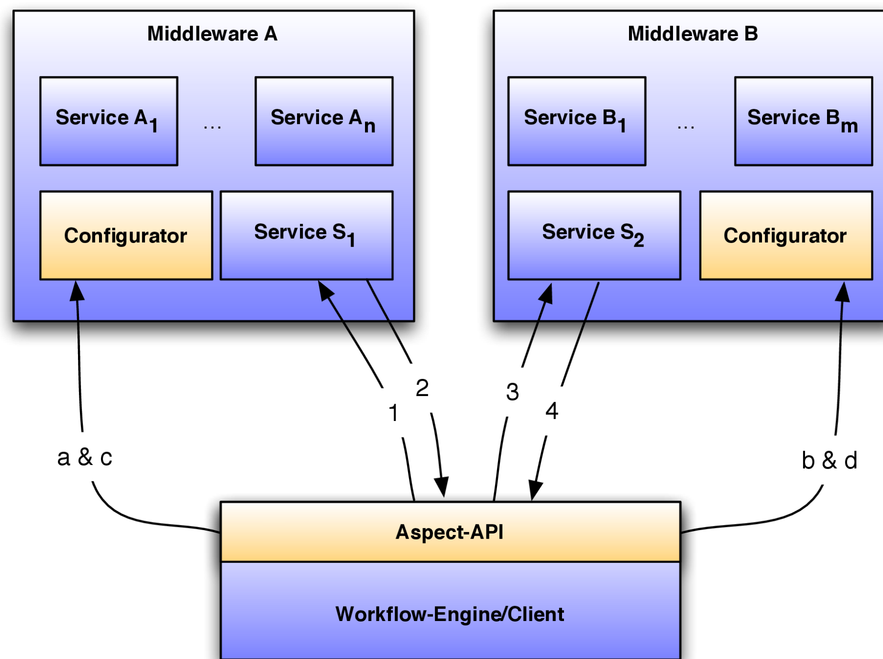


Figure 7.1: Web service invocations using request/response aspects. First, the aspect API located at the client side contacts the aspect configurator at both middlewares and validates the presence of the advice to be woven (step a and b). In case of a positive response, the aspects are woven into the communication chain (steps c and d). Afterwards, the services S_1 and S_2 can be conventionally called (steps 1 – 4) and the woven aspects are applied on the communication in a transparent manner.

$\mathcal{O}$ (***operation***) specifies the name of the operation of the given porttype. A support for a wildcard operator is also needed.

$\mathcal{F}$ (***field***) determines the actual part of the WSDL message the aspect should be applied to. Since XML schema elements can have nested complex data types [191], this field is realized as an XPath expression [202], in case the aspect should be applied only to a particular part of a complex type. The XPath language also allows the use of wildcard operators and thereby enables the application of the aspect to multiple elements or to (multiple parts of) multiple elements.

$\mathcal{M}$ (***mode***) is an element of the set {`request`, `response`, `both`}. The first and the second one means that the aspect is applied to the request message. When the mode is set to `both`, the aspect is applied to both messages.

$\mathcal{I}$ (***ID***) determines a reference to the actual advice that has to be applied on the elements specified above. Each advice has a (unique) ID by which the framework can identify, instantiate, and apply it on certain fields of the SOAP message.

In addition to the preceding elements that are mandatory, there exist the following optional elements:

$\mathcal{K}$ (***key value store***) provides configuration data in the form of key-value pairs optionally needed by the aspect.

$\mathcal{D}$ (***depends***) contains a list of aspects that have to be present and active for the join points to which the described aspect will be applied. If this dependency is not fulfilled, the adding of this aspect to the target service must fail.

$\mathcal{C}$ (***conflicts***) contains all IDs, which are in conflict with the defined aspect. If one or more of the aspects mentioned in this list are present in the target platform and active for the same join points, the weaving of this aspect must fail.

```
<complexType name="Aspect">
  <sequence>
    <element name="portType"      type="xsd:QName" />
    <element name="operationName" type="xsd:string" />
    <element name="field"         type="xsd:string" />
    <element name="mode"          type="xsd:string" />
    <element name="adviceID"      type="xsd:string" />
    <element name="adviceData"    type="tns1:HashMap" />
  </sequence>
</complexType>
```

Listing 7.1: XML schema type of a request/response aspect.

The pointcut description is represented by the XML schema type shown in Listing 7.1. The first six elements of the tuple, namely $\mathcal{P}$, $\mathcal{O}$, $\mathcal{F}$, $\mathcal{M}$, $\mathcal{I}$, $\mathcal{K}$, are mapped to the schema type. The fields $\mathcal{C}$ (conflicts) and $\mathcal{D}$ (depends) are not present in the schema type, because they are expressed by the implementation of the advice. Hence, they do not appear in the schema declaration.

7.4 Implementation

The presented Java-based, prototypical implementation of the aspect framework rests on Apache Tomcat 6 as the application container in combination with Apache Axis 1.4 as the SOAP processing engine and also as the web service execution environment. On the client side, the Axis client libraries are used. The orchestration engine for composing web services is again the BPEL4Grid engine.

The adding and removal of request/response aspects and listing of available advice are performed by a single web service that offers the corresponding methods. Further methods provided by this aspect configuration service are methods to check whether an advice of an aspect is supported and whether an aspect is woven into a service. These two methods allow the client-side component to ensure the atomic behavior of the overall framework.

```
public interface IRequestAdvice {
  public Object handleRequest(Object value) throws Exception;
  public void init();
  public void setAspectData(Map<String, String> data);
}
```

Listing 7.2: Java interface for an advice of a request/response aspect that can be applied on request messages.

7.4.1 Advice Interface

Each advice of a request/response aspect is referenced by its ID (element $\mathcal{I}$ of the pointcut description). In the implementation presented, this ID represents the base name of the Java class that effectively implements the corresponding advice. Such a class has to implement specific Java interfaces, a request advice has to implement `IRequestAdvice`, a response advice implements analogously `IResponseAdvice`. Either way, the `IAdvice` has to be implemented to provide general information about the advice to the framework. The first two interfaces require the methods `handleRequest()` and `handleResponse()`, respectively, to be present (cf., Listing 7.2 for request handling and Listing 7.3 for the general interface for advice).

These methods are called by the aspect framework when a join point shadow becomes a join point (i. e., a defined join point is triggered) for the specific advice. The argument passed to these two methods is resolved by the expression in the field element $\mathcal{F}$ (see pointcut description) of the aspect. The original value is substituted by their return values. If the field element $\mathcal{F}$ contains a wildcard operator, multiple attributes needs to be handled by the advice. Multiple attributes

represent multiple fields of a complex data type. The corresponding advice is applied iteratively to each of these attributes.

The concrete value of the scope of an aspect is either `local` or `global` with respect to communication. The value `local` means that the join point of the aspect respects the client, meaning that it is only active for the client who has woven it in. On the contrary, `global` implies that an aspect is active for all clients. Moreover, the `global` scope implicates that no other service is affected by this aspect, since the effect depends on the actual client in order to determine the succeeding service. Because web services are loosely-coupled, it is not possible to identify such a succeeding service in general.

```
public interface IAdvice {
  public enum SCOPE {LOCAL, GLOBAL};
  public enum AFFECTED {NONE, DIRECT, DATAFLOW, CONTROLFLOW};
  public SCOPE getScope();
  public AFFECTED getAffected();
  public List<String> getConflicts();
  public List<String> getDepends();
}
```

Listing 7.3: Java interface for advice meta-information.

An advice must furthermore define whether there exists a subsequent service that is affected by it. If such a service exists, the aspect must be woven into it, too. Furthermore, the value of the affect field (`none`, `direct`, `dataflow`, or `controlflow`) controls the way in which such a service is detected. For example, an aspect for profiling the communication always returns `none`, because no other partner exists. The value `direct` is used for reliable messaging since it affects both partners that are communicating directly. For a file transfer aspect, `dataflow` is chosen because the subsequent service must also support it. The value `controlflow` – as well as the last two mentioned values – is examined by the client-side library, and the service matching the returned enumeration is exposed to the corresponding aspect.

7.4.2 Aspect Configurator

The task of the *aspect configurator* is the validation and the weaving of aspects. It is realized as a dedicated web service. To decouple the aspect framework from the conventional web services, it is implemented as a distinct service, i. e., a remote interface for weaving aspects. This functionality should not be provided by the normal services hosted by the middleware, and the concerns of weaving aspects and providing business logic have to be kept separate. Thus, services need to neither be aware of the presence of the aspect framework nor provide any related functionality – this concerns the actual process of weaving request/response aspects and the actual functionality that should be woven in. The aspect configurator operates on a registry that contains all necessary information about woven request/response aspects. Since the registry is realized as a hashtable, the number of deployed aspects influences the overall runtime only marginally. The actual weaving of request/response aspects into the web service communication takes place by means of AspectJ

[14, 118]. Thereby, the functionality of weaving request/response aspects into the message processing chain of Axis without the need of modifying either its configuration or its implementation is achieved.

7.4.3 Aspect Provider

The activation of (web service) join points is realized as an AspectJ aspect, namely the *Aspect Provider*. It is woven into the global handler chain of Apache Axis. If a web service is called or if it sends its response, the around advice is executed. It checks the registry to figure out whether a request/response aspect has been woven into this concrete web service. If so, the scope and the caller are identified afterwards, and in case of a complete match, the advice of the request/response aspect also gets executed. For this purpose, the pointcut of the AspectJ aspect matches with the signature of the `invokeMethod()` method of the `RPCHandler` of Apache Axis, which in turn is in charge of invoking the concrete implementation of the web service.

In Listing 7.4, the pointcut (Lines 1 – 5) and the advice (Lines 7 – 13) of the AspectProvider are shown. The AspectProvider is integrated into the handler chain of Apache Axis. Request/response aspects are applied in the order in which they were woven. A more elaborate management strategy will be a subject of future work. Due to the implementation as an AspectJ aspect the configuration of Axis (e. g., deploying a specific handler into each service that handles aspect weaving) does not have to be modified. This potentially allows for reusing this code within other middlewares supporting web services, such as JBoss or Spring.

```
pointcut invokeMethod(MessageContext msgC,Method method,
2  Object obj,Object[] args) : call(
    protected Object RPCProvider.invokeMethod(
4     MessageContext, Method, Object, Object[]) throws Exception)
       && args(msgContext, method, obj, argValues);
6
Object around(MessageContext msgContext, Method method, Object obj,
8  Object[] args) throws Exception :
  invokeMethod(msgContext, method, obj, args) {
10   handleRequest(argValues, method, msgContext);
    Object response = (method.invoke(obj, args));
12   return handleResponse(response, method, msgContext);
  }
```

Listing 7.4: AspectProvider, implemented in AspectJ. This allows for an non-invasive interception of SOAP messages.

The request handling of aspects is illustrated in Listing 7.5. Each time a request handling is triggered, the pointcut of the AspectProvider is activated (cf., Listing 7.4) and the method `handleRequest` is called. First, all relevant aspects are looked up in the registry (Line 4). Then, the corresponding advice is loaded. Finally, all matching parts of the incoming message are identified and passed on to the `handleRequest` method of the advice (Lines 6 ff).

```
1 private void handleRequest(Object[] objs, Method method,
```

```
  MessageContext msgContext) {
3   Aspect requestAspect = getAspect(method, msgContext,
      Aspect.AOP_REQUEST_MODE);
5   ...
    Object os[] = getFields(requestAspect.getField(), objs);
7   ...
    for (Object o : os) {
9     Object container = getContainer(o);
        PropertyUtils.setProperty(
11        container, nextField,
          requestAspect.handleRequest(o));
13  }
}
```

Listing 7.5: Excerpt of the request handling of the Aspect Provider.

7.4.4 Security Manager

The security manager fulfills two tasks: it guarantees user security and aspect security. The first one ensures that only justified users are able to weave aspects. While the second one allows only certified aspects to be woven into the communication.

If an authorized user tries to deploy an aspect, its signature is validated, and in case of successful validation the aspect is woven into the system. Otherwise, the call fails and a fault message is returned. The security manager mentioned in Section 7.3.1 uses a public key infrastructure based on the X.509 [33] standard in order to authenticate users who call the aspect configurator. The same architecture is employed for the signing of aspects. More precisely, an aspect has a signature field that contains an encrypted hash sum of the whole aspect. Since the encryption is based on the same public key infrastructure, the validity of aspects can be reduced to the one of the attached hash sum.

7.4.5 Aspect Invocation Handler

To facilitate the use of request/response aspects during service orchestration, a custom invocation handler of the BPEL4Grid engine is called each time a web service is called within a BPEL process. The weaving of aspects is initiated by the Aspect Invoke Handler (AIH). The AIH provides all the client-side functionality described in Section 7.3 and is called every time the workflow engine performs a web service invocation. In order to register and to use it, an extension mechanism provided by the workflow engine is utilized. Consequently, a modification of the implementation of the workflow engine is not necessary. When the AIH is called, it checks whether the actual web service has to use an aspect. If this is the case, the AIH deploys the aspect to the actual web service and – depending on the defined relevance of the (request/response) advice – deploys it also to subsequent web services. The information on whether an aspect should be woven into the web service communication is provided by the developer during the design of the workflow.

7.5 Aspects for Web Service Communication

In this and the following section, first a typical workflow for distributed multimedia analysis and the modeling procedure of data flows in BPEL processes are described. Subsequently, it is explained how the Flex-SwA framework supports efficient data transmission between web services and its interaction with BPEL workflows. In addition, general implementation efforts for realizing efficient data transmission using Flex-SwA are discussed. To achieve this aim, the framework for request/response aspects mentioned in the previous section is used and specific aspects are developed.

7.5.1 Workflows for Multimedia Analysis

Most approaches for multimedia content analysis are built in a monolithic fashion following a sequential structure [56]. In the special case of supervised learning algorithms, this structure can be divided into two phases: training and classification (see Figure 2.3). In general, preparatory tasks are required to first generate the smallest entities of interest needed to solve the problem in question, e. g., retrieving images from a database, decoding frames of a video or decompressing audio signals. Next, some pre-processing such as transforming or filtering of image data is performed. Then, feature extraction is performed, followed by the actual classification. Sometimes, an additional post-processing step is needed, e. g., generating a representation of the analysis results. The training phase is similar, but a learning step substitutes the classification step. Another difference is that the classification step is rather a feature extraction step offering the opportunity to combine it with other features for building more complex analysis approaches.

The abstract workflow mentioned above performs a generalized analysis in a supervised learning environment. In the sequel, a concrete workflow that performs face detection based on the Viola-Jones algorithm [190] is used as an example. More precisely, a BPEL workflow has been modeled that calls web services each representing a different task of the analysis. Since BPEL is an orchestration approach, users can benefit from the centralized control flow, not only by easy control and sophisticated runtime decisions, but at the same time also by preserving the loosely-coupledness of services.

7.5.2 Data-flow and BPEL

BPEL as an orchestration language for business processes defines a workflow in form of a set of activities. These activities comprise two groups: basic and structured activities. The former ones describe single steps for the interaction with a service and/or the manipulation of data passing the engine. With the help of the second class of activities, namely the structured ones, the order of activity execution is determined. BPEL offers a rich vocabulary of control mechanisms to express sequences of activities like *receive*, *invoke* and *reply*, parallel execution, loops, error handling as well as compensation mechanisms for roll-back actions. The *invoke* activity (which is the most error-prone BPEL activity) is used to model the invocation of external services. Fault handlers may be described either on the process level or for each scope separately. Scopes are hierarchically organized parts a

process can be divided into. The *faultHandlers* element contains an arbitrary number of *catch* (and optionally a single *catchAll*) elements that define fault types to react to. This feature is comparable to a *switch-case* construct in other programming languages.

To summarize, BPEL provides a rich set of elements to explicitly model the control flow, whereas the data flow is only modeled implicitly by the manipulation and assignment of data encapsulated in request/response messages. For example, the normal way of modeling the data exchange between two web services is as follows: invoke the first service with a request message, wait for the corresponding response from the service, assign all needed data to the input variable of the second service, invoke it with a request message encapsulating the data, and finally wait for the response. It has to be noticed that for highly frequent data exchanges and transfers of large amounts of data, this way of modeling the data flow is not efficient and does not support the requirements of workflows for multimedia content analysis.

7.5.3 Flex-SwA

The Flex-SwA architecture provides a flexible way for the handling of bulk data in service-oriented environments. It differs from SOAP Messages with Attachments by offering message forwarding as well as demand-driven evaluation and transmission of binary data. By means of a reference builder one creates an XML description that refers to the actual location of a file used to transmit the binary data objects. Instead of transferring data from a remote server to the client and from the client to the next service provider, the service provider uses a reference to retrieve the data directly from a remote server (on a client computer or anywhere else in the web). References do not need to be handled by a certain service provider directly but can be forwarded to other service providers with negligible additional communication cost.

From an application developer's point of view, service invocation and data transmission remain coupled in a single service invocation operation. A service developer can choose between different communication patterns to configure how the Flex-SwA platform handles the referenced data, e. g., whether the data has to be completely transferred before the service is executed or whether the transmission has to overlap with the service execution.

7.5.4 Data Transmission in BPEL via Flex-SwA

Seiler et al. [165] have proposed to avoid the mentioned load at the orchestration engine, i. e., the BPEL engine, by the usage of the Flex-SwA framework. A reference points to a resource location in memory or to a file. The service can then directly acquire the data from the resource location. Instead of the data, only references have to be sent to the BPEL engine. Since references are very small regarding their transmission size, the load at the engine is thereby kept low.

Services to which data are sent repetitively (e. g., video frames) can be modeled in the following way: For each resulting processing unit (video part, frame, audio sample, ...), a reference passes to the BPEL engine and in turn is transferred to the subsequent service. In this way, the BPEL engine keeps full control over the data flow, while at the same time the control flow resides at the

engine. This is necessary since web services are by definition loosely-coupled and do not have any information about their environment, in particular about predecessors and successors. By locating the control flow at the workflow engine, it is possible not only to model conditional elements in reaction to the responses sent by the services, but also to realize failure and compensation handling in the workflow itself.

Figure 2.7(a) shows a typical sequence of service invocations and the corresponding – schematic – data flow. For example, just after the workflow engine has invoked service 1, this one sends its results back to the workflow engine that further delegates them to the subsequent consumer service 2. It is obvious that the data are transferred twice in this scheme. In Figure 2.7(b), both instances, the workflow and the services, make use of the reference technique. Service 1 responds with a reference to the workflow engine, which in turn sends it to service 2. Once having received this reference, service 2 resolves it via the Flex-SwA framework that transfers the data from service 1 directly to service 2. This scheme can lead to a significant speedup if the data payload from service 1 to service 2 exceeds the data volume of the messages that have to be exchanged for the delegation of the references.

To take advantage of the Flex-SwA framework, the software developer has to consider several preparatory tasks. First of all, the Flex-SwA framework must be available within the context of the application container. This means that first a special handler component has to be plugged into the handler chain of the web service stack (a step which has to be performed once per container). Afterwards, the software developer has to ensure that each method of each service that is supposed to exchange data by using Flex-SwA exposes this usage via its signature, i. e., the signature of the method contains a reference rather than the actual data type. Furthermore, the developer must make sure that the implementation resolves the reference and retrieves the actual data, when a method of a service is called. After retrieving them, the data have to be cast into the actual format used within the method. If the data should also be returned as a reference, the same procedure is necessary. This procedure has to be performed for each argument of the method to be replaced by a reference.

7.6 Efficient Data Transmission using Aspect-oriented Programming

Optimized data transmission as described above and in Section 2.3 requires several code modifications on the client and on the service side. This holds for Flex-SwA as well as for SOAP with attachments (or any other transport optimization). Apart from the needed middleware, clients and web services must be able to explicitly deal with references in request messages and to build new references for response messages. Hence, the efficient data transmission by existing web services using Flex-SwA calls for additional software development efforts. Instead, aspect-oriented programming is a suitable alternative for efficient data transmission between web services since the arising requirements can be viewed as cross-cutting concerns.

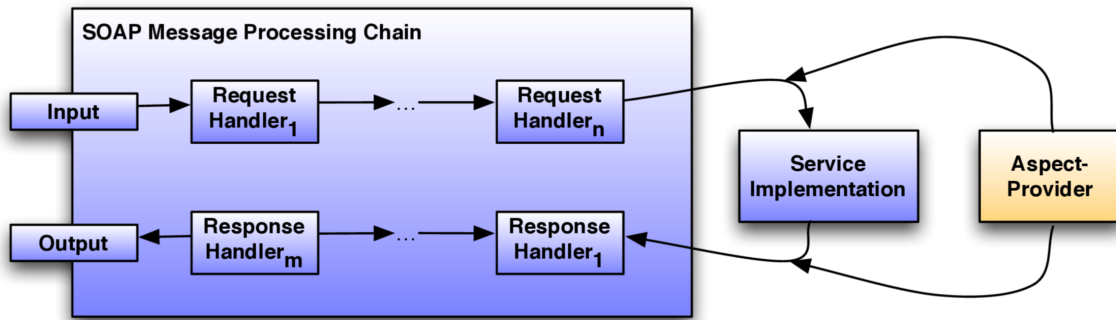


Figure 7.2: Web service invocations with aspects.

This section presents a solution for efficient data transmission for multimedia web services using *request/response aspects*. Thereby the development efforts can be reduced significantly, while at the same time the web service implementation are left unchanged. The proposed aspect-oriented framework is able to weave non-functional aspects, such as efficient data transmission, into web services at the message exchange level.

A request/response aspect based on Flex-SwA substitutes a field within a SOAP message. In case of a response message, the request/response aspect based on Flex-SwA takes the original data, creates a reference to it and eventually encodes this reference into the actual data field. Subsequently, on the receiving side the request/response aspect the reference is decoded again and the resolved data is written into that corresponding data field.

The combination of a web service and an invocation handler supports the scenario shown in Figure 7.3. With respect to the previously mentioned use case, request/response aspects are used in the video analysis workflow as follows: Right after calling the video decoder, the face detection is performed. If no aspects are available, each video frame is passed through the BPEL engine which in turn can become a bottleneck due to high network traffic. A combination of request/response aspect and Flex-SwA can avoid this issue (see Section 7.5.4). They are woven into the services (steps 1 and 2 in Figure 7.3) and realize the optimization of the data transfer (more details will be given in Section 8.4.2). Next, the actual services are called (step 3 and 4). The request/response aspect encodes a reference into the returned value rather than the binary data of the decoded frame. Subsequently, this reference is decoded again at the face detection service in a transparent manner.

The weaving of aspects is initiated by the AIH, see Figure 7.3. The AIH is called whenever the workflow engine performs a web service invocation. To register and to use the AIH, extension mechanisms provided by the workflow engine are used. In this way, the implementation of the workflow engine can remain unmodified. When called, the AIH checks whether the actual web service should use an aspect. If so, the AIH deploys the aspect to the actual web service and – depending on the type of the aspect – also to subsequent web services. In order to call to Aspect Configurator, the AIH constructs the aspects based on supplied information (an example can be seen in Listing 7.6). An aspect includes the qualified name of the service, the operation to be called

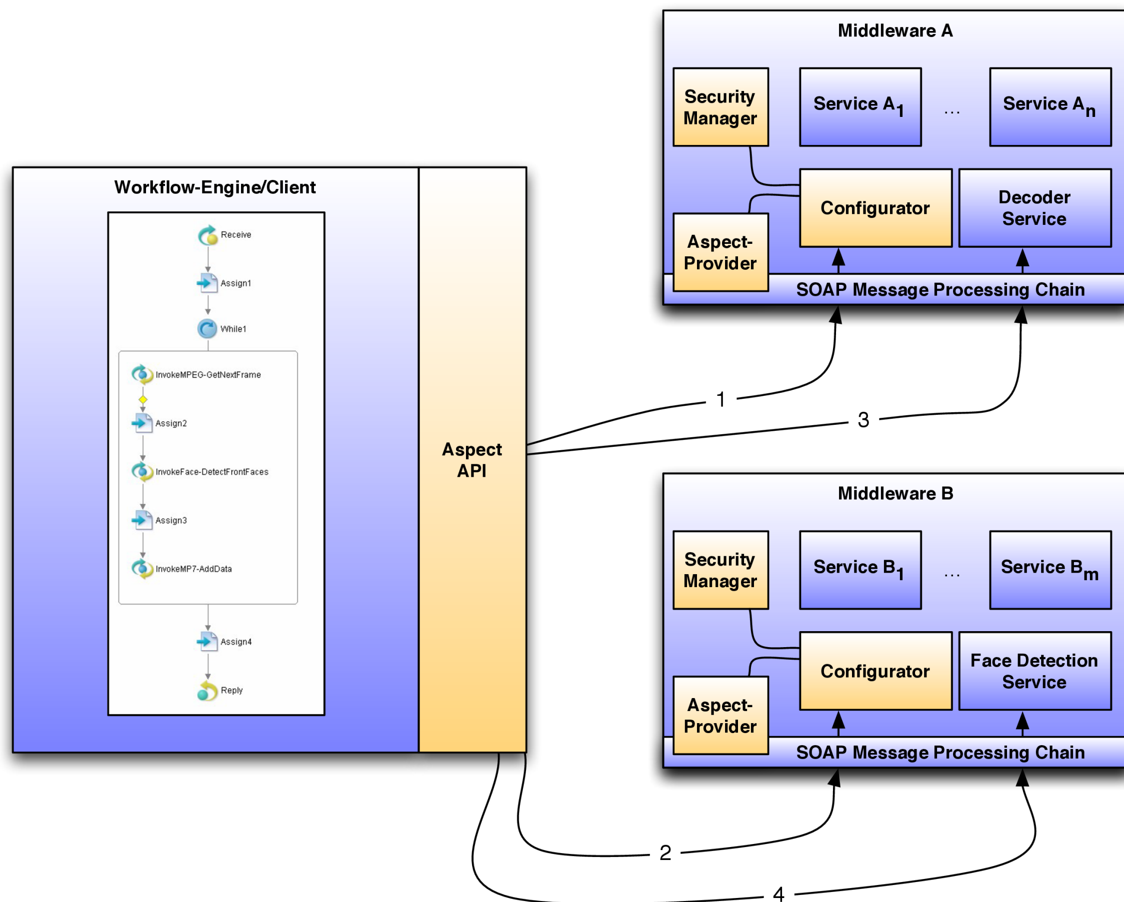


Figure 7.3: Excerpt of a face detection workflow using the presented aspect framework. The workflow engine (depicted on the left) executes a workflow and utilizes the aspect API each time a web service has to be called. In case, the called web service is indicated to use request/response aspects (as annotated within the workflow), first the aspect configuration of middleware A and the on of middleware B are called in order to weave the request/response aspects in the communication (step 1 and 2). If the weaving is successfully completed, the workflow invokes the Decoder service and a request/response modifies its response such that a reference is passed back to the engine instead of the actual frame data (step 3). Since this happens transparently for the workflow engine, the response of the decoder is forwarded to the face detection service. A request/response aspect woven into the request message decodes the reference of the incoming message and replaces it by the actual data that is transparently acquired using Flex-SwA (step 4).

and a XPath expression that describes the particular part of the message the advice has to be applied to.

```
Aspect serviceaAspect = new Aspect(
   new QName("http://www.fb12.de/videana/MpegDecoderService", "↩
   ↪ MpegDecoder"),
   "getNextFrame",
   "/0/imageData",
   Aspect.AOP_RESPONSE_MODE,
   "FlexSwAPlugIn");
```

Listing 7.6: Java bean constructor of an aspect.

After the AIH has successfully woven an aspect into a service (e. g., into the decoder service) and this service is called, in the next step the AspectProvider extracts part of the message specified by the aspect. This is shown in Listing 7.6. This listing further describes that the `FlexSwAPlugin` has to be applied on the response of the MpegDecoder, i. e., on the imageData element. The implementation of the Flex-SwA advice (see Listing 7.7) first creates a reference for the return object (Lines 12 – 17) that in the next step is further converted into the same syntactical data structure which the service actually returns (Lines 31 – 48). The aspect on the side of the receiver operates accordingly by extracting the encoded reference. Then, it acquires the actual data, converts it into the expected data format and eventually passes it back to the service implementation.

```
@Override
public Object handleResponse(Object value) throws Exception {

  if (value == null) {
    return value;
  }

  Reference reference = new Reference();

  int bufferSize = ...

  OutputStream os = reference.requestMemoryOutputStream(bufferSize, ↩
  ↪ false);
  ObjectOutputStream oos = new ObjectOutputStream(os);
  oos.writeObject(value);
  oos.close();

  String referenceString = JSONSerializer.toJSON(reference);

  Object result = null;
  if (value.getClass().isArray()) {
    result = convertReferenceToReturnValue(value, referenceString);
  } else if (value instanceof String) {
    result = referenceString;
  } ...
```

```
26   return result;

28 }

30 private Object convertReferenceToReturnValue(Object value,
     String refString) {
32
     Class<?> arrayClass = value.getClass();
34   Class<?> componentClass = arrayClass.getComponentType();

36   Object result = null;

38   if (componentClass.isPrimitive()) {
       if (componentClass.equals(byte.class)) {
40       byte[] temp=new byte[refString.length()];
         for (int i = 0; i < refString.length(); i++) {
42         temp[i] = (byte) refString.charAt(i);
         }
44       result = temp;
       } ...
46
     return result;
48 }
```

Listing 7.7: Response handling of the efficient data transmission aspect.

7.7 Summary

This chapter presented the application of the paradigm of aspect-oriented programming to service-oriented architectures. *Request/response aspects* allow to bring new functionalities into the communication between web services. Using a SOAP compliant pointcut language and administrative web services to weave advice, request/response aspects integrate well into service-oriented architectures.

8 Evaluation

Contents

8.1 Introduction **143**
8.2 Service Development **144**
8.2.1 Legacy Code Wrapping 144
8.2.2 Simplified Modeling of BPEL Workflows 147
8.3 Scalability and Reliability in Workflows **150**
8.3.1 The Basic Approach 150
8.3.2 Data Aware Scheduling 157
8.3.3 Multi-objective Scheduling 160
8.3.4 Fault Tolerance 164
8.4 Request/Response Aspects **168**
8.4.1 Applicability of Request/Response Aspects 168
8.4.2 Request/Response Aspects for Multimedia Data Analysis 173
8.5 Summary **179**

8.1 Introduction

In this chapter, an evaluation of the formerly presented components is conducted. Based on the considered use cases (cf., Section 2.2), first, workflows are developed during the following sections. The presented modeling tools are used to design the workflows. Next, those workflows are used to evaluate the different runtime components. The following sections are ordered according to the presentation of the corresponding components in the previous chapters.

Parts of this chapter have already been published [44, 45, 110, 111, 113–115, 166].

8.2 Service Development

8.2.1 Legacy Code Wrapping

In this section, the wrapping of legacy code of the multimedia data analysis (cf., Section 2.2.1) and also of the systems biology use case (cf., Section 2.2.3) is evaluated. Both applications make heavily use of native, i. e., legacy, code. For a more convenient and flexible usage, the developed algorithms have to be embedded into a service-oriented architecture [36, 93]. This task is achieved by means of the framework for legacy code wrapping. In the following, the speech processing example in multimedia and a specific example from system biology are elaborated.

Exposing Speech Processing Algorithms as Web Services

Previous publications in the field of speech processing focused on the signal processing aspects of the speech recognition problem as well as on the way of accompanying the code, exposed as a web service, with an easy-to-use client for prospective users. The described approach focuses on the prior step of wrapping the speech processing application as a Java application. The used software has originally been implemented as a Microsoft Windows binary called `SCrec` written in C++ and contains different methods related to speaker recognition. Despite the existence of an automatically generated human-to-machine interface [174], the integration of the speech processing software into more powerful and therefore value-added compositions still requires much manual effort.

As described in Section 5.2, wrapping legacy code mainly consists of two steps: after describing the legacy code by means of the LCDL framework and generating the bindings, these bindings are incorporated into another application.

More precisely, the first step is the modeling of the LCDL information. This can be achieved using a basic LCDL editor where all information is entered (see Figure 8.1). In addition, a basic validation can also be performed, in order to check whether all needed elements and attributes are set. The corresponding XML file of the LCDL model, which is the serialized form of the internal model representation of EMF, is shown in Listing 8.1.

```
<lcdl:Service xmlns:lcdl="http://fb12.de/lcdl/1.1" name="SCrec" xmi:↩
↪ version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://↩
↪ www.w3.org/2001/XMLSchema-instance">
  <operations name="screc">
    <output>
      <source xsi:type="lcdl:StdOutSource" />
      <type xsi:type="lcdl:ElementReturn" name="returnValue" type="{↩
      ↪ http://www.w3.org/2001/XMLSchema}string" />
    </output>
    <inputs xsi:type="lcdl:FileInput" name="audioFile" mode="in" type=↩
    ↪ "xsi:any" />
    <inputs xsi:type="lcdl:ElementInput" name="preEmphasizeFactor" ↩
    ↪ mode="in" type="xsi:double" />
```

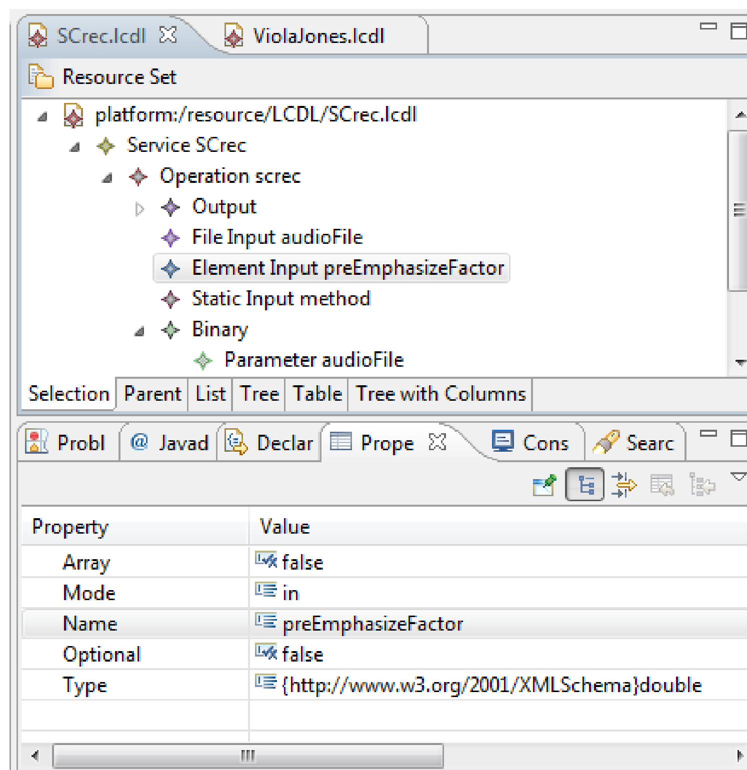


Figure 8.1: Basic LCDL editor. Graphical view of the model instance depicted in Listing 8.1.

```
    <inputs xsi:type="lcdl:StaticInput" name="method" mode="in" value=↩
    ↪ "28" type="xsi:int" />
    <execute xsi:type="lcdl:Binary" file="SCrec">
      <parameters name="audioFile" />
      <parameters name="method" />
      <parameters prefix="-featureMfcc.preEmphasizeFactor" name="↩
      ↪ preEmphasizeFactor" infix="=" />
    </execute>
  </operations>
  <bindings xsi:type="lcdl:JavaProxy" packageName="de.fb12.sclib" />
</lcdl:Service>
```

Listing 8.1: Instance of an LCDL model (cf., Figure 5.2) for a speech processing binary. It takes two modifiable input arguments and one static argument. The return value of type `string` is acquired from the standard output of the legacy code.

The considered method (`method=28`, see Listing 8.1, Line 9; different speech analysis algorithms are encoded by consecutive numbers) takes as an additional parameter the filename of an audio file for which standard speech features are extracted. The corresponding LCDL model instance is depicted in Listing 8.1. It is possible to control each parameter of the feature extraction procedure. Exemplarily, the `featureMfcc.preEmphasis` parameter controls the high frequency boost prior to further signal processing, an effect that is clearly audible in the result: the produced audio file contains the resynthesized features. The method defined above is reflected in the `StaticInput` element in Line 9. The file name is written to standard out by the program. The LCDL framework captures this information and returns it as a string.

```
package de.fb12.sclib;
import de.fb12.lcdl.runtime.java.LcdlAnnotation;

@LcdlAnnotation(model="SCrec")
public interface ISCrec {
  public java.lang.String screc(
    java.io.File audioFile,
    java.lang.Double preEmphasizeFactor);
}
```

Listing 8.2: Generated Java interface. The annotation in Line 4 enables the LCDL framework to locate the corresponding model file (if necessary). The signature of the generated method contains only two arguments. The third argument (`method`) that is required by the legacy code is passed by the LCDL framework and thus not present in the JAVA signature.

```
// ...
File audioFile = new File("input.wav");
double preEmphasize = 0.97;

ISCrec screcService = (ISCrec) LcdlFactory.getInstance(ISCrec.class);

String filename = screcService.screc(audioFile, preEmphasize);
// ...
```

Listing 8.3: Usage of the generated interface.

The LCDL generator creates an interface and an underlying implementation. For the use of the legacy code in a Java program, the code snippets shown in Listing 8.2 and 8.3 are crucial. Listing 8.3 shows how the LCDL factory is utilized to get an instance of the SCrec service (Line 5). The invocation of a method of the generated interface, depicted in Listing 8.2, only requires the call of a simple Java method (Line 7).

System Biology

As described in Section 2.2.3, Monte Carlo Bootstrap methods are data parallel and therefore the Monte Carlo workflow is expected to give scalable results. More precisely, the outer loop of this workflow iterates over n perturbations of the input model. Within each such iteration, these models are independent of each other. Despite the workflow invocation and the final data merge step of Monte Carlo results (*collectfitdata*), no communication is necessary. It follows that the execution of the Monte Carlo Bootstrap workflow in a parallel manner is easily possible.

In terms of service-orientation, the files *fml* and *R* are represented as string paths to certain file locations, whereas n and m are integer arguments of the web service. In the current implementation, files are exchanged via a NFS network share that is accessible by all 13C-MFA web services. The LCDL definition of the Monte Carlo Bootstrap service is depicted in Figure 8.2.

The exposition of this component as a web service via the invocation of a workflow script allows for an easy use and adoption of its functionalities by scientists without having to compile and to install specific tools.

8.2.2 Simplified Modeling of BPEL Workflows

This section evaluates the simplified modeling component that enables non-workflow experts to model complex (BPEL) workflows. For this aim, two workflows, originating from the multimedia data analysis and the medical research use case, respectively, are assembled using the SimpleBPEL editor.

SimpleBPEL for Multimedia Data Analysis

The sample application is a real-life video analysis workflow that performs face detection, text recognition, and camera motion estimation in MPEG videos as described in Section 2.2.1 or [165].

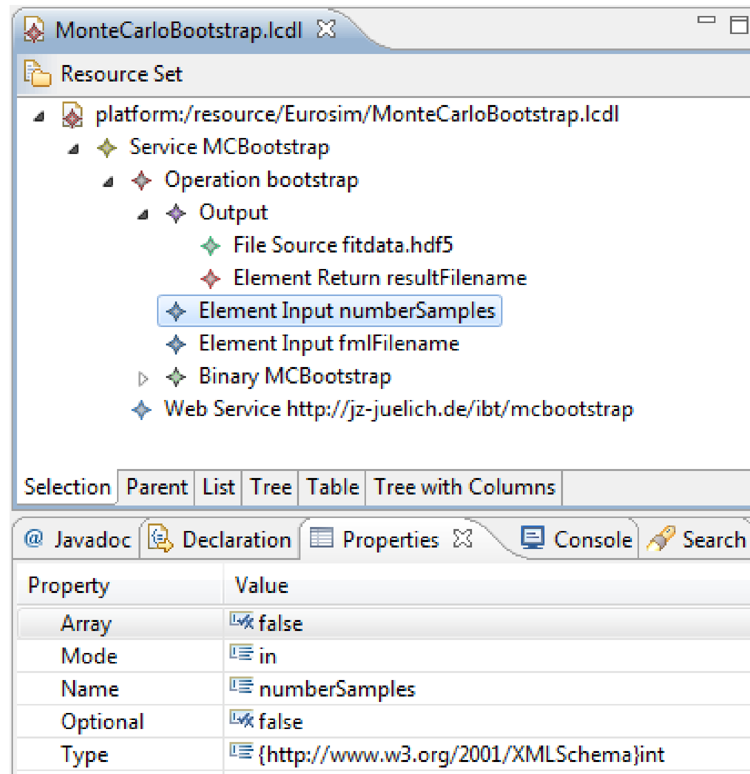


Figure 8.2: LCDL editor showing the description of the Monte Carlo Bootstrap binary for its exposition as a web service.

The fragments consist of several services, including, e. g., a Decoder service that splits a video into single video frames, services that perform face, text, or motion detection.

Based on Apache Axis, the services are realized as web services. The data exchanged between services contains video frames. To make the data transfer efficient, Flex-SwA is used. The overall process comprises several steps, the first one being the development of the SBFs by the BPEL expert. Each of those, namely text recognition, face detection, and camera motion estimation, is modeled as a BPEL workflow (without *receive* and *reply* activities, see Section 5.4.1) and saved as an SBF within a Domain Profile called “Video-Analysis”. Subsequently, input and output variables are defined, and finally, the created profile is exported.

A media researcher then imports this profile and can now model his or her own SimpleBPEL workflow. This modeling is as easy as dragging the SBFs into the workflow and connecting them. The compatibility checker mentioned in Section 5.4.2 assures that only reasonable connections are drawn. For example, the output of face detection (a MPEG-7 document) cannot be used as the input for text detection (that operates on a video) and the compatibility checker denies to connect the SBFs in this case. The complete SimpleBPEL workflow can be seen in Figure 8.3.

Eventually, the SimpleBPEL workflow can be exported as a deployable BPEL archive. This is done by means of the *deploy* function of SimpleBPEL Composer, which in turn uses DAVO to transform the internal model into BPEL code that can be deployed to a BPEL engine afterwards. Thereafter, the (Simple)BPEL workflow is available as a web service and accessible by clients.

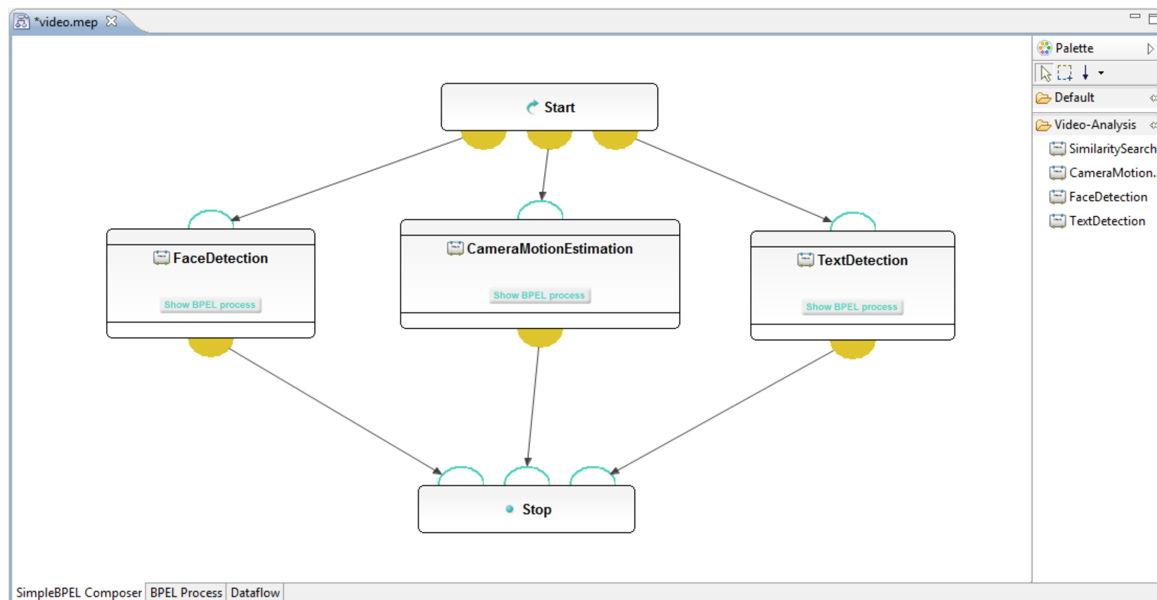
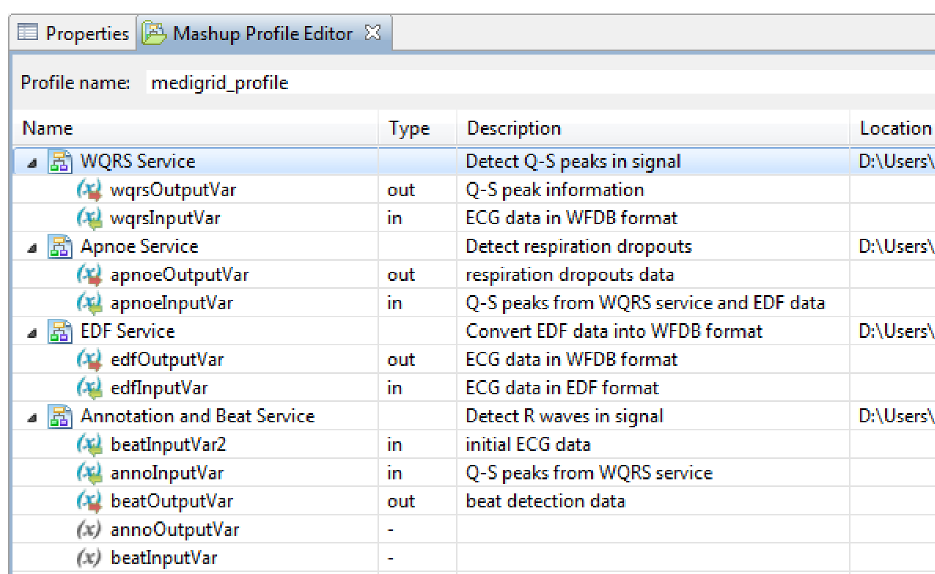


Figure 8.3: View on the video analysis workflow in the SimpleBPEL Composer.

Apnoea Data Analysis

Using SimpleBPEL for the medical use case is as straightforward as for the multimedia data analysis use case. BPEL fragments are designed by a BPEL expert and exported into a domain profile (see Figure 8.4). Subsequently, a domain expert composes these function blocks into a single SimpleBPEL workflow (see Figure 8.5) and at the same time, a corresponding BPEL process is built in the background in a transparent manner (see Figure 8.6). After having been exported, the standard-complaint process can be deployed into a BPEL engine and executed like any other workflow (see Figure 8.7).



Properties | Mashup Profile Editor

Profile name: medigrid_profile

Name	Type	Description	Location
WQRS Service		Detect Q-S peaks in signal	D:\Users\
wqrsOutputVar	out	Q-S peak information	
wqrsInputVar	in	ECG data in WFDB format	
Apnoe Service		Detect respiration dropouts	D:\Users\
apnoeOutputVar	out	respiration dropouts data	
apnoeInputVar	in	Q-S peaks from WQRS service and EDF data	
EDF Service		Convert EDF data into WFDB format	D:\Users\
edfOutputVar	out	ECG data in WFDB format	
edfInputVar	in	ECG data in EDF format	
Annotation and Beat Service		Detect R waves in signal	D:\Users\
beatInputVar2	in	initial ECG data	
annoInputVar	in	Q-S peaks from WQRS service	
beatOutputVar	out	beat detection data	
annoOutputVar	-		
beatInputVar	-		

Figure 8.4: Property view of the SimpleBPEL editor when domain profiles are exported. Variables of the SimpleBPEL fragment (SBF) are marked as either input or output variables, such that connections between SBFs are established using these variables.

8.3 Scalability and Reliability in Workflows

This section focuses on the evaluation of the handling of infrastructural non-functional requirements that affect (BPEL) workflows. Before demonstrating the applicability for more scenarios, the feasibility is elaborated. This discussion concludes with the examination of the fault-tolerance module.

8.3.1 The Basic Approach

In the following, experimental results concerning the performance of the basic load balancing approach are stated. The sample application is a real-life video analysis workflow that performs face

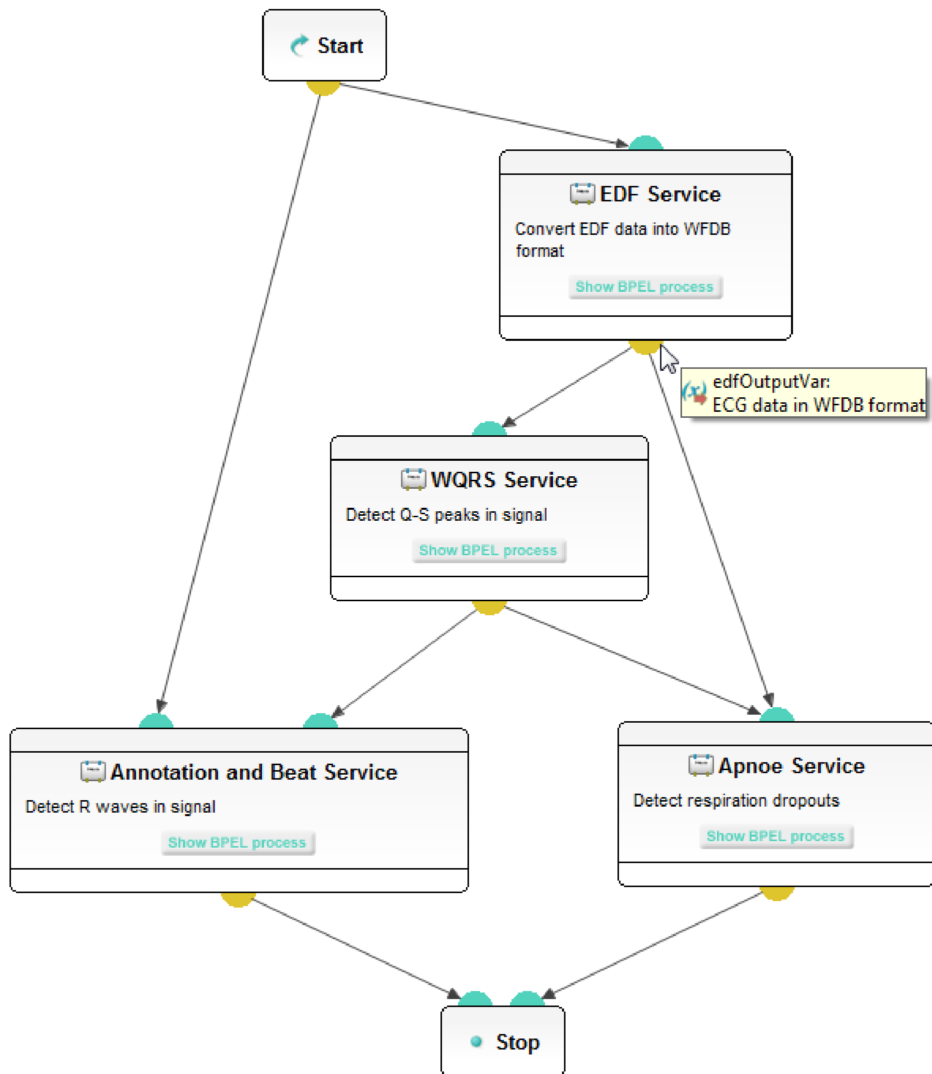


Figure 8.5: A SimpleBPEL workflow using the SBF of Figure 8.4. Hints are provided to the domain expert, when a connection between two SBFs is about to be created.

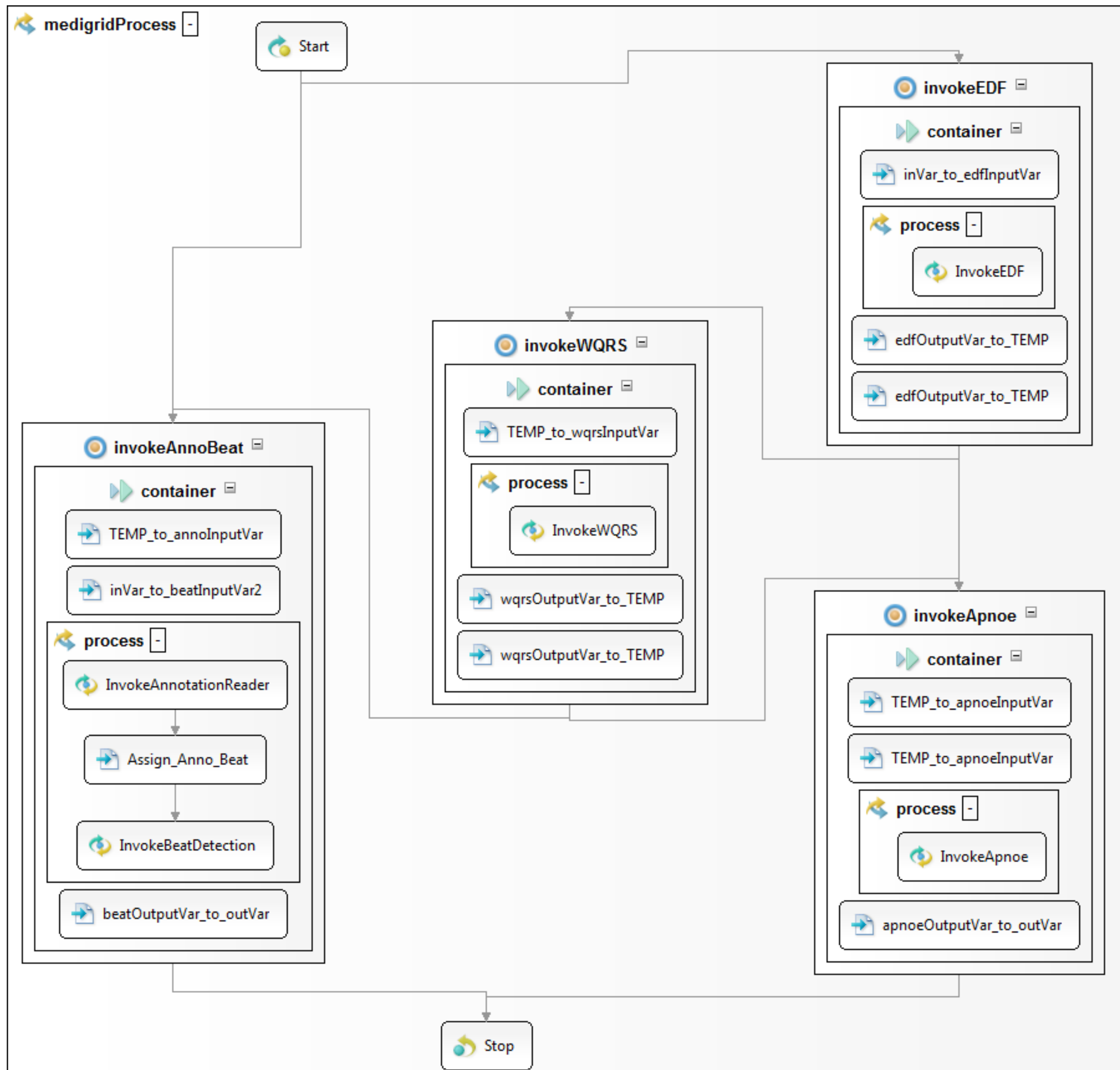


Figure 8.6: BPEL view of the SimpleBPEL workflow shown in Figure 8.5. The SimpleBPEL editor maintains a fully standard-complaint BPEL workflow of each SimpleBPEL workflow. Since this process is performed in the background, this is transparent for the user of the SimpleBPEL editor. Technically, the BPEL model of DAVO is used to create and manage the BPEL process.

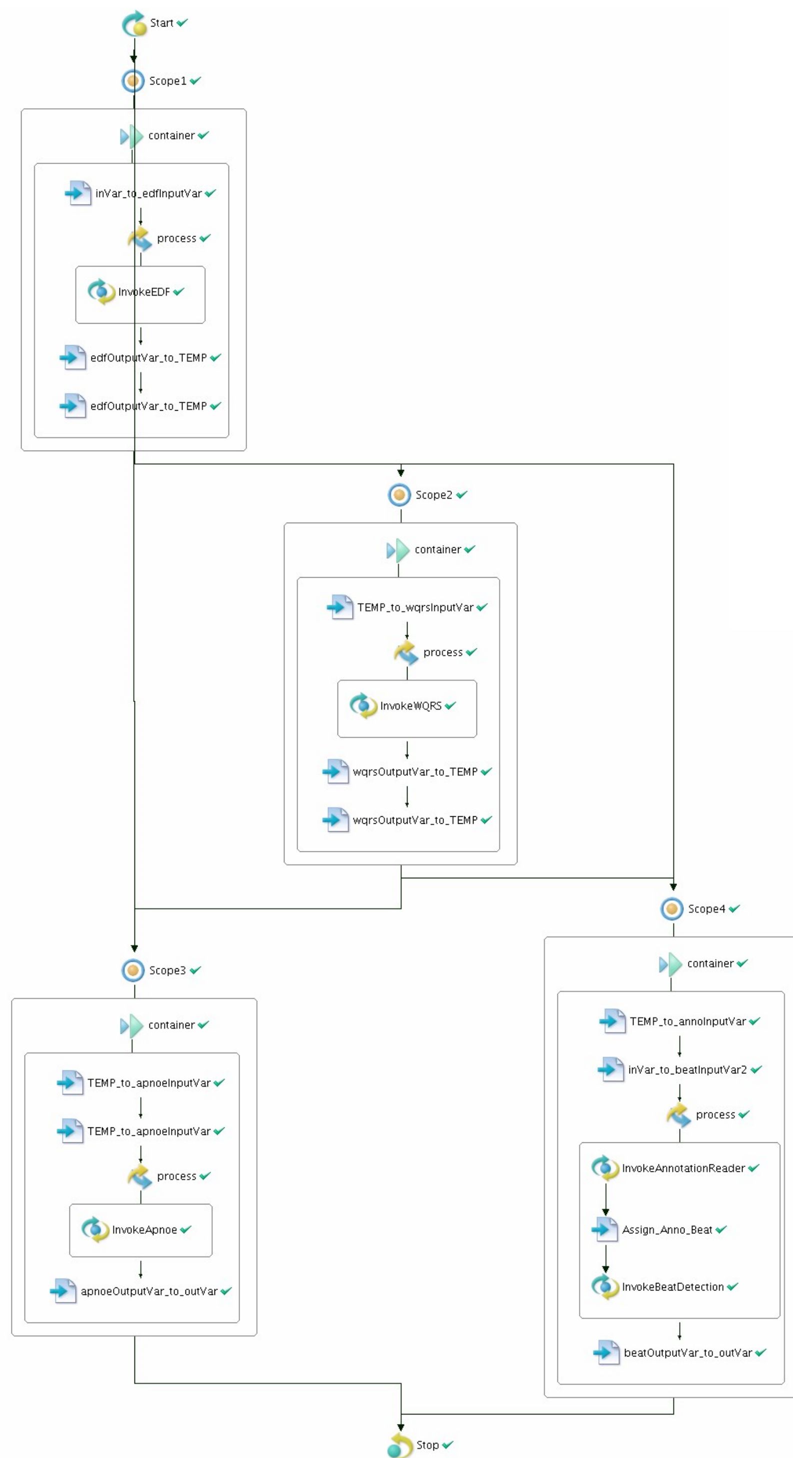


Figure 8.7: A SimpleBPEL workflow does only have higher level of nested activities compared to "typical" workflows, but being a standard-compliant workflow, any special treatment by the workflow engine is unnecessary.

detection and text recognition in MPEG videos. It is a simplified version of the SimpleBPEL workflow, which has been introduced in Section 8.2.2 and which relies on the specification(s) in Section 2.2.1.

Based on the Globus Toolkit 4, the services are realized as WSRF services. As already stated, the data exchanged between the services consists of video frames. Due to the use of Flex-SwA, the data transmission is efficient.

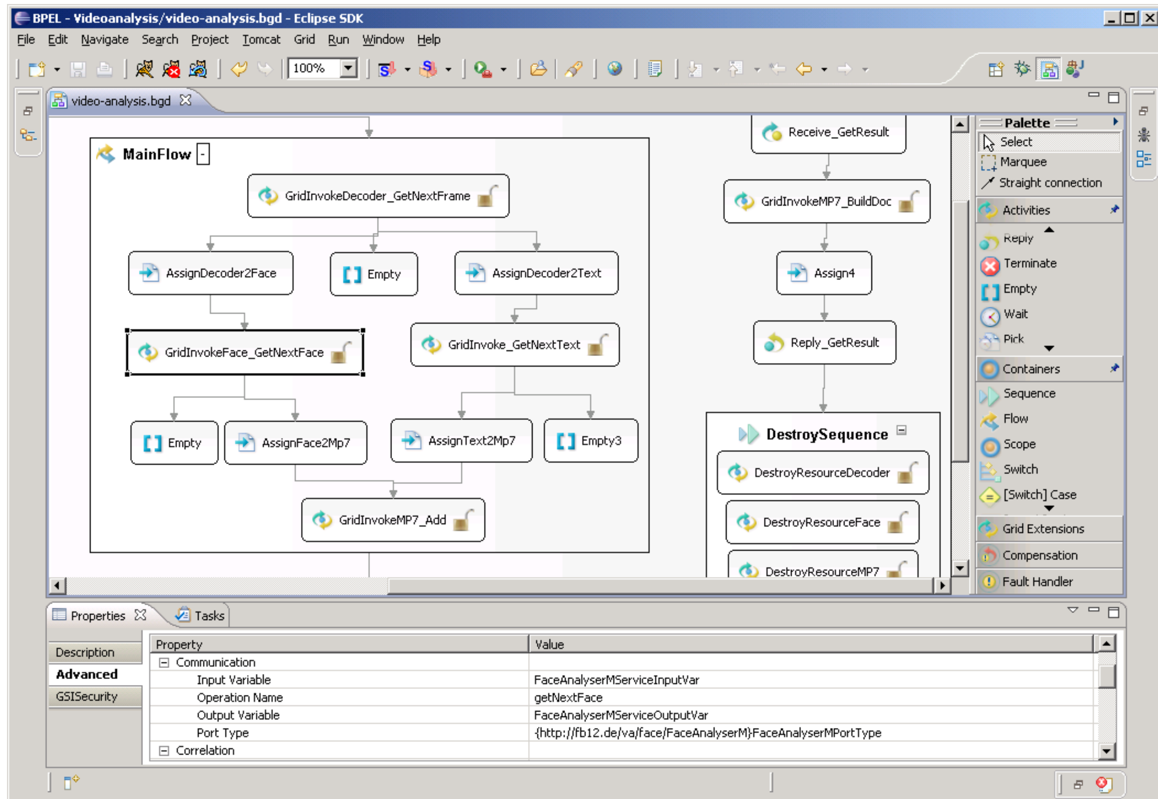


Figure 8.8: Designing the video analysis with the Visual Grid Orchestrator.

To orchestrate the Grid services, WSRF extensions for BPEL [43] are used. For the integration of the WSRF extension as well as the load balancer (LB), the *invokeHandler* of the factory link points to the LB. The resource link is kept dynamic and therefore is determined by the *createResource* operation. Figure 8.8 shows the design of the workflow using DAVO. It supports the user in several ways, e. g. by providing wizards to create an invoke operation for WSRF services.

The workflow engine and all virtual machines are hosted by Amazons EC2. This allows us to perform a computationally intensive video analysis workflow without the need of an in-house infrastructure and, more importantly, network latencies between Europe and the USA in the measurements are eliminated.

Some of the video analysis services make use of native C/C++ code. For instance, the decoder utilizes the OpenCV library [147]. Deploying the service on different platforms (e. g., 32-bit Windows

platform vs. 64-bit Linux) is a fairly complicated task, since the native code has to be recompiled for each target platform separately. Virtual machines make this task much easier. Indeed, for the video analysis workflow, a virtual machine image is created that is booted by all hosts. In this way, it is assured that all hosts can execute the native code.

In experiments, the workflow has been run more than 150 times; in more than 40 runs new virtual machines were booted due to a high load on existing machines. For the first measurements, the least powerful host type available at Amazon EC2 was chosen. Such a "small instance" has 1.7 GB of RAM, 160 GB hard disk, runs a 32 Bit Linux and costs $0.10 per hour[1]. The computing power is specified in "EC2 Compute Units" (ECU). Small instances have a computing power of 1 ECU, where ECU stands for "EC2 Compute Unit". This corresponds to the computing power of a 1.2 GHz Xeon 2007 processor [6].

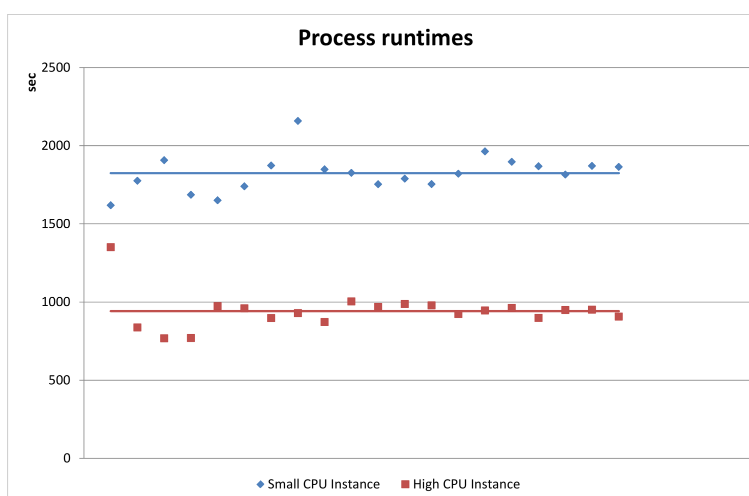


Figure 8.9: Process runtimes for different host types.

In the conducted experiments, the average runtime including the startup of new virtual machines amounts to 1821.49 sec with a standard deviation of 95.9 sec (see Figure 8.9). Contrary to arbitrary BPEL workflows, the runtime does not only comprise the computation time, but also the boot time of virtual machines and the time needed for deploying the middleware. The boot time of a virtual machine running on small instances is 37.57 sec. Another study conducted in [68] has also measured a typical boot time of the small instance machine to last 35.6 sec (with a standard deviation of about 17.2 sec). The provisioning and the start of the middleware take another 51.17 sec. Compared to the overall runtime of the video analysis itself (1732.75 sec with a standard deviation of 101.4 sec), the

[1]Since this evaluation has been performed, Amazon has constantly reduced the price for EC2 instances. As of November 2013 the price for a small instance machine is at $0.06 per hour.

total average startup time of 88.74 sec is negligible. Once a virtual machine has been started, it can be used by several workflows, which further puts this overhead into perspective.

For the comparison of the different host types offered by Amazon, all computations were also performed on a "High-CPU Medium Instance" (2×2.5 ECU, more storage, \$0.20 per hour[2]). The average process runtime on high-CPU instances amounts to 941.29 sec (standard deviation: 113.58 sec). The average boot time of such a virtual machine is 33.36 sec and the container needs 31.37 sec to start (see Figure 8.10). Although it was conceivable to expect a speedup factor between 4 to 5, a speedup factor of just 2 has been measured. This is caused by the fact that the used libraries are single-threaded and therefore do not use the additional CPU core. Moreover, the speedup is less than linear (i. e., $\lesssim 2$) due to network latency.

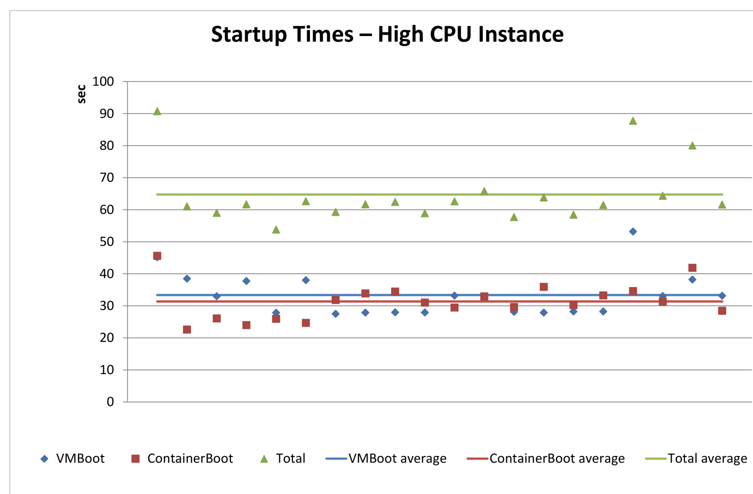


Figure 8.10: Virtual machine and middleware boot times.

In addition, a separate analysis of the runtimes of each component has been conducted. A registry query takes around 2.5 msec; determining the load takes around 100 msec per host. If the load is cached, the time for a query is reduced to about 1 msec, and becomes negligible. The scheduling decision takes about 1 msec, except when a new virtual machine has to be booted.

To summarize, the presented approach provides efficient load balancing capabilities. If, without the LB, multiple workflow instances are run in parallel, the runtime increases linearly with the number of instances. However, using the proposed solution, the runtime remains almost constant (only the provisioning times vary slightly, as shown in Figure 8.10), which is a significant improvement to former solutions.

[2]The actual price of a High-CPU medium instance amounts to \$0.145 per hour.

8.3.2 Data Aware Scheduling

Since the framework for data flow aware scheduling is based on a previous, load based scheduler (cf., Section 6.3), it is compared with this predecessor. The evaluation is performed within different scenarios and in each of them, static and/or Cloud resources are used. The static resources consist of two Core2Duo E6850 machines with 2 GB of memory running Fedora 11 Linux. Amazon's Elastic Compute Cloud was used as the Cloud infrastructure. The used resource type was "High-CPU Medium Instance" with 1.7 GB of memory and 5 EC2 Compute Units (split on two cores) hosting an Ubuntu Linux.

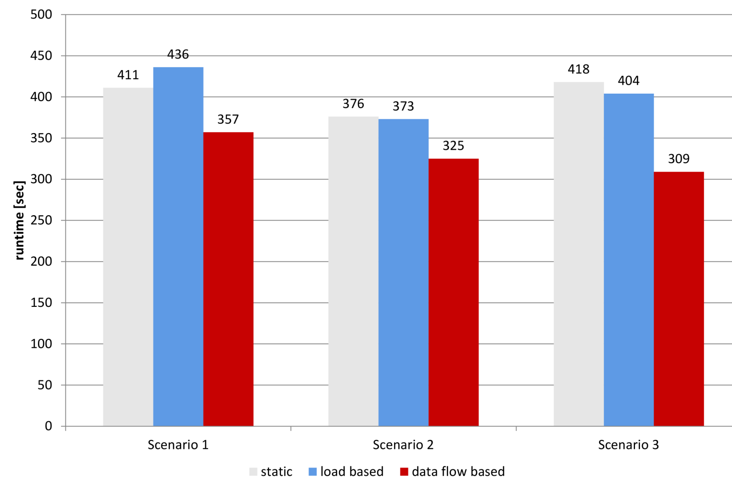
The framework is tested using the following three scenarios: scenario (1) simulates increasing load caused by the start of a new workflow every 30 sec. In scenario (2), at an interval of 90 sec, two workflows are started simultaneously to model an abrupt increase of resources demand. The scenarios are run up to the point that eight workflows have been started. Finally, to test the performance of the system in peak load situations, in scenario (3) four workflows are started concurrently.

For classifying the overall runtime of a scheduled workflow, the three scenarios are performed based on different setups. The first setup is static scheduling, i. e., resources are allocated manually within the BPEL process. The second setup uses the old, load based scheduler whereas the third setup applies the dataflow-aware approach. All measurements are obtained as the mean of 20 runs.

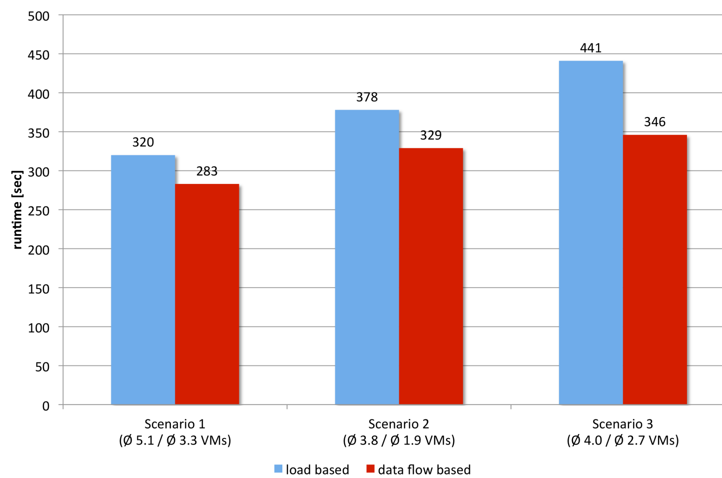
The diagrams in Figure 8.11(a) and 8.11(b) allow to compare the execution times for all scenarios in two different environments: while the first one does not allow to provide virtual machines, in the second setup new machines can be used. The measured values in Figure 8.11(a) indicate that the data flow based scheduling approach outperforms static resource allocation and the previous load based approach. A speedup of about 14% compared to static allocation in Scenario 1 & 2 and 26% in scenario 3 can be achieved. The load based approach is only insignificantly better (or even worse) than static allocation, since the benefit of execution time reduction on idle machines is equalized by the data transfer times induced by "hopping" between different resources.

In Figure 8.11(b), only load and data flow based scheduling are compared, since static allocation becomes pointless when Cloud resources are dynamically added to the pool of machines. The data flow based approach outperforms the load based solution in all three scenarios (the runtime decrease lies between 11% and 21%). Compared to the previous setup, the load based approach is by 26% faster in Scenario 1, less than 1% in Scenario 2 and by 10% *slower* in Scenario 3. Here, the system used 5.1, 3.8 and 4.0 virtual machines on the average. The data flow based approach is 21% faster in Scenario 1, 1% *slower* in Scenario 2 and 12% *slower* in Scenario 3. Here, the system used 3.3, 1.9 and 2.7 virtual machines in scenario 1, 2, and 3, respectively, on the average. Thereby, it is more cost-efficient than load based scheduling while at the same time execution times are reduced.

The effect that the runtimes *increase* in Scenarios 2 & 3 when Cloud resources are used is due to the startup delay of virtual machines (typically between 60 and 90 sec; cf., Section 8.3.1). The current implementation does not provide virtual machines in advance. This in particular means that whenever the system decides to set up a new virtual machine, the startup delay causes the execution of the initiating workflow to be prolonged. The overhead can be circumvented, if the *Reservation*



(a) Execution times using physical resources.



(b) Execution times using physical and Cloud resources.

Figure 8.11: Execution times of the medical use case workflow in different scenarios using a data flow aware scheduling component.

Agent replaces the aforementioned placeholder machines in such a way that the booting of the virtual machine has finished right before the corresponding reservation becomes active. However, the overhead pays off, since subsequently executed workflows can use the additional resources without any delay (as it happens in Scenario 1). Furthermore, for longer running and/or instance-intensive workflows, the provisioning overhead carries no weight.

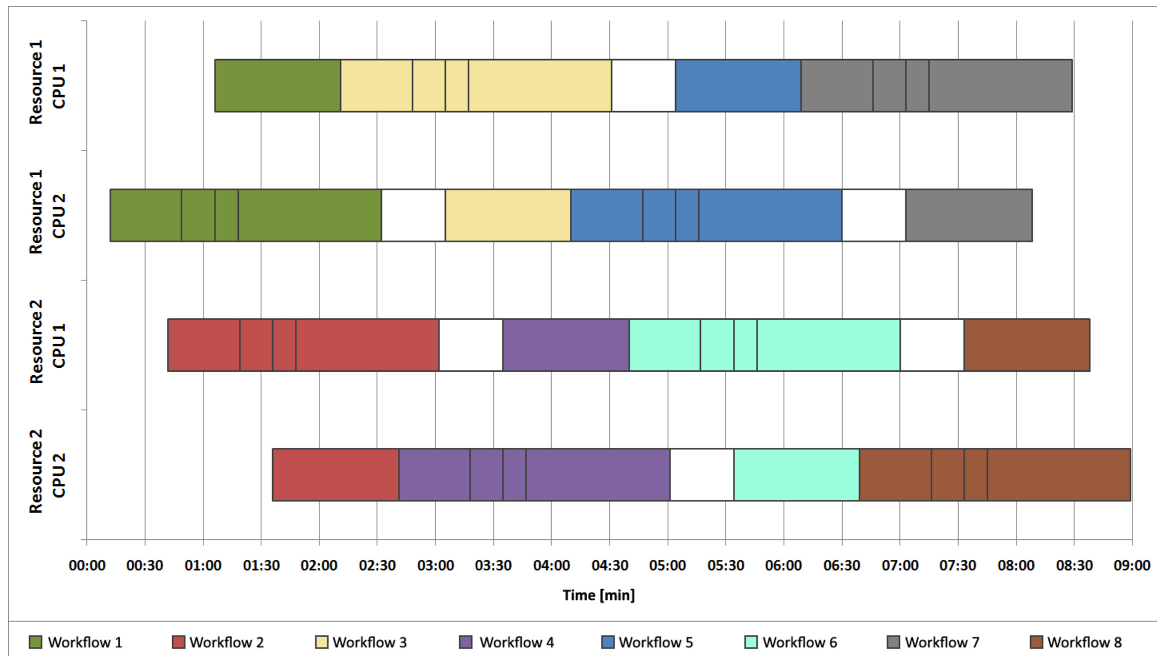


Figure 8.12: Gantt diagram of the resource usage for Scenario 1.

The Gantt diagram in Figure 8.12 illustrates the resource allocation for Scenario 1. It can be inferred that the system clusters invocations belonging to the same critical path, which in particular means that every workflow instance is executed on two different CPUs. In this case, those are even on the same machine since dual core machines were used. Furthermore, as the bars that represent the different workflow steps of almost equal length, it is apparent that the resources are utilized homogeneously.

The worst-case execution time of the genetic algorithm (cf., Listing 6.2) was simulated on an Intel Core2Duo E8600 with 3.33 GHz, 4 GB RAM, running a modern Linux, see Figure 8.13. The term worst-case refers to the fact that the evolutionary process generates all generations and is not stopped by one of the termination conditions (see Section 6.3.2). Practically speaking, the assignment result would repeatedly improve by at least 2% from iteration to iteration, which becomes more and more unrealistic after a certain number of iterations.

The overall runtime is dominated by the logarithm of the size of the population and the number of iterations. For reasons of simplicity, in this simulation only the number of resources was increased. For $r = 100$ resources and $n = 4$ nodes in the longest critical path, one obtains $\text{pc} = 100^4$ and

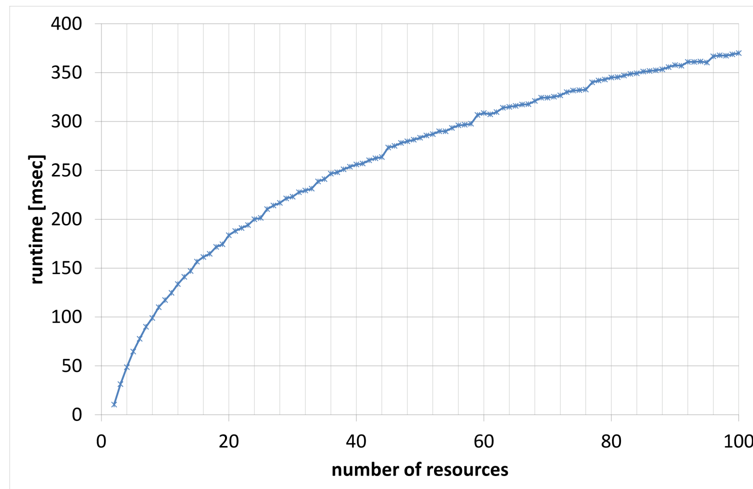


Figure 8.13: Scheduling overhead for different path complexities.

$\text{it}_{max} = \lfloor 10 \cdot \ln 100^4 + 50 \rfloor = 234$. Under these assumptions, the population size in every iteration amounts to $\lfloor 100 \cdot \ln 100^4 + 100 \rfloor = 1942$. The worst-case runtime for the algorithm is 370 msec. Note that the scheduling algorithm is single-threaded, i. e., it does not take advantage of the second CPU core of the machine.

To summarize, the simulation results for an increasing path complexity elucidate a good performance of the algorithm without significantly influencing the workflow execution time.

8.3.3 Multi-objective Scheduling

The workflow used to evaluate the solution for multi-objective scheduling in workflows has its origin in the medical data analysis use case. It basically performs an ECG (electrocardiogram) analysis and, based on the obtained results, conducts apnoea detection, cf., Section 2.2.2. The implementation uses the Physio Toolkit [155], which is a common set of open source tools in the biomedical sciences.

Since the data format (European Data Format, EDF) of the recorded vital signs (real, anonymized patient data) is different from the format required by the Physio Toolkit (WaveForm DataBase, WFDB), first the data has to be converted (*InvokeEDF*). Afterwards, Q-S peaks are detected within the ECG signal (*InvokeWQRS*) and the results are passed to the annotation reader service (*InvokeAnnotationReader*). This in turn decodes the input and transfers the results to the beat detection service *InvokeBeatDetection* that detects R waves within the signal. In parallel, the output of *InvokeWQRS* is passed to the apnoea detection service (*InvokeApnoea*) that analyzes the input signal and detects

respiration dropouts (to diagnose the sleep apnoea syndrome). Figure 8.14 illustrates the control as well as the data flow of this workflow.

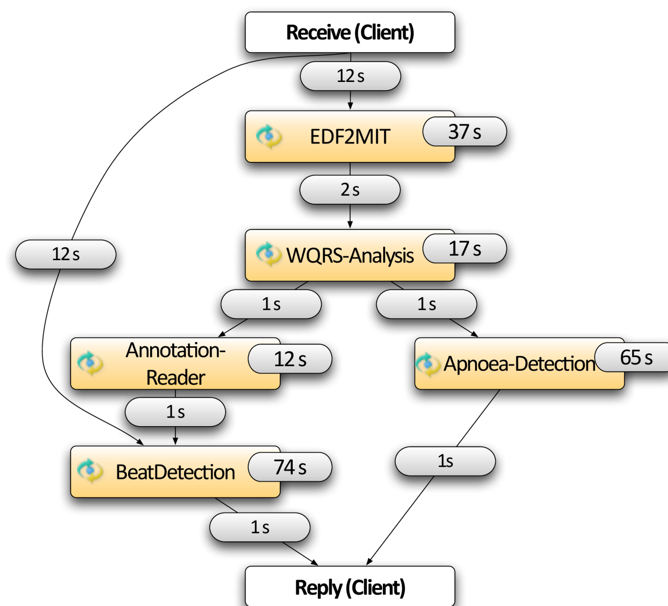


Figure 8.14: Simplified representation of the used sample workflow. BPEL elements that have no impact on execution time were left out for illustration purposes. Activities with orange background are to be executed, the gray ovals display execution times and data transfer times, respectively.

The total data amount to be transferred within the workflow is approximately 258 MB (plus additional 118 MB for data transfer between the client and the first service when the workflow is started). The net runtime of the services (without network overhead) on modern hardware (see next paragraph) amounts to 140 sec for the longer path and 65 sec for the shorter path. Note that both path can be executed in parallel, see Figure 8.14). Therefore, the minimum execution time including data transfers (using a 100 MBit/sec network and idle resources) adds up to around 180 sec. Though the runtime per instance is quite short, it has to be considered that typically many instances are performed in parallel by the physicians who use the workflow.

Experimental Results

As in the previous section, the evaluation is performed within different scenarios. Each of them uses four Cloud resources (EC2 small instances) as static/on-premises hosts, which are located in Amazon's *eu-west-1* and *us-east-1* regions. The scheduler is allowed to provision on-demand resources in the following regions: *us-east-1*, *eu-west-1*, and *ap-southeast-1*. As the initial connection speeds between the regions, the values depicted in Figure 6.11 are taken. In all regions, virtual machines of the types *m1.small*, *c1.medium*, and *c1.xlarge* are allowed to be used; all machines use Debian Linux

images (provided by the EC2 community) in which the provisioner deployed the web service stack for medical data analysis during the startup.

The framework is tested using the following three scenarios (comparable to those in Section 8.3.2): scenario 1 simulates increasing load due to the start of a new workflow every 15 sec. In scenario 2, at an interval of 15 sec, two workflows are started simultaneously to model an abrupt increase of resource demand. The scenarios are stopped when 50 workflows are running. Finally, to test the system in peak load situations, in scenario 3, 10 workflows are started concurrently (repeated five times with a 15 minutes pauses in between).

To classify the overall runtime of a scheduled workflow, the three scenarios were performed in different setups. As in Section 8.3.2, the first setup is static scheduling, i. e., two resources are allocated manually within the BPEL process. The second setup uses the existing load based scheduler as presented in Section 6.2 whereas the third setup is based on the approach presented in Section 6.4. All measurements represent the mean value of 20 runs.

The results for scenario 1 and 2 are similar: the load based approach (LB) was unable to complete the scenarios, since too many tasks were clustered per machine. Due to the initial low load value of a single machine, many workflows are scheduled to this very machine. Since the load value only increases slowly (also because of the initial data transfer that only causes small system load), more workflows are scheduled to the same machine. As a result, the machine becomes overloaded.

user weights (perf:cost)	**VMs used**	**cost in USD**	**runtime in sec**
10:1	0/0, 3/7, 0/0	1.615	124
1:1	0/0, 2/6, 0/0	1.33	187
1:10	0/0, 0/0, 0/0	0	1.224

Table 8.1: Results for scenario 1 (50 x 1 workflow). The number of machines in "VMs used" is encoded as "us, eu, ap; small/medium".

The figures in Table 8.1 clarify that the user defined weights have an impact on the workflow runtime and also on the cost: when performance is prioritized (setting 10:1), the average workflow runtime is 124 sec, and the cost is 1.615 USD for the whole scenario. If performance and cost are equally important (setting 1:1), the runtime increases to 187 sec (a plus of about 50%), while the cost is reduced by approx. 28 cents or 21%. When cost is prioritized (setting 1:10), no virtual machines are started at all – resulting in a runtime increase of about 1000%.

Scenario 2 delivers similar results as scenario 1 (see Table 8.2), but with higher absolute values. The graphical representation of the measurements (Figure 8.15) clearly demonstrates that (despite its increasing load) the use of virtual machines (and provisioning just-in-time) leads to almost constant runtimes.

Scenario 3 (Figure 8.16) simulates peak-load situations by starting 10 workflows in parallel (repeated 5 times with a 15 minutes pauses in between). This scenario was completed by all schedulers. However, for static resource allocation, a clear pattern occurs: the workflow runtime dramatically

user weights (perf:cost)	VMs used	cost in USD	runtime in sec
10:1	0/0, 3/14, 0/0	2.945	157
1:1	0/0, 1/12, 0/0	2.375	249
1:10	0/0, 0/0, 0/0	0	1.531

Table 8.2: Results for scenario 2 (25 x 2 workflows in parallel). The number of machines in “VMs used” is encoded as “us, eu, ap; small/medium”.

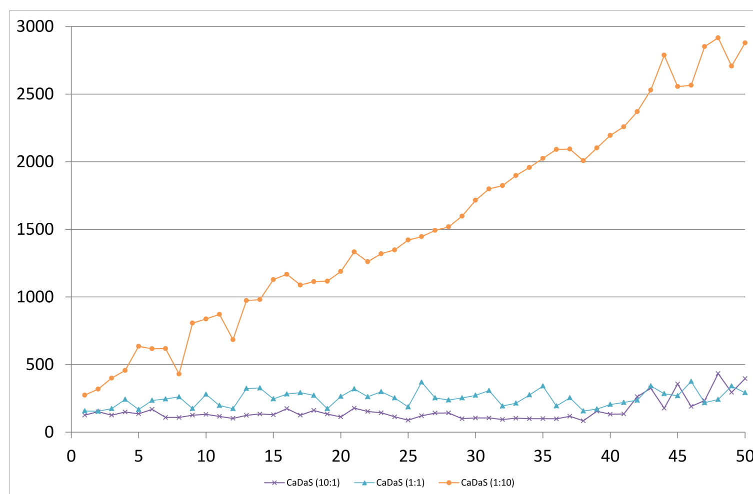


Figure 8.15: Workflow runtimes for scenario 2 with different user weights. The results clearly demonstrate that the use of virtual machines (and provisioning just-in-time) leads to almost constant runtimes in spite of increasing load.

increases (from about 600 sec to more than 1,000 sec) until the 10th (and last) workflow has finished. The same pattern can be recognized when CaDaS is used with a weighting of 1:10 (i. e., cost is dominant). In this situation, no virtual machines are provisioned, which causes the runtime to increase. The absolute values are shown below the ones of the static approach, since four instead of just two machines can be utilized.

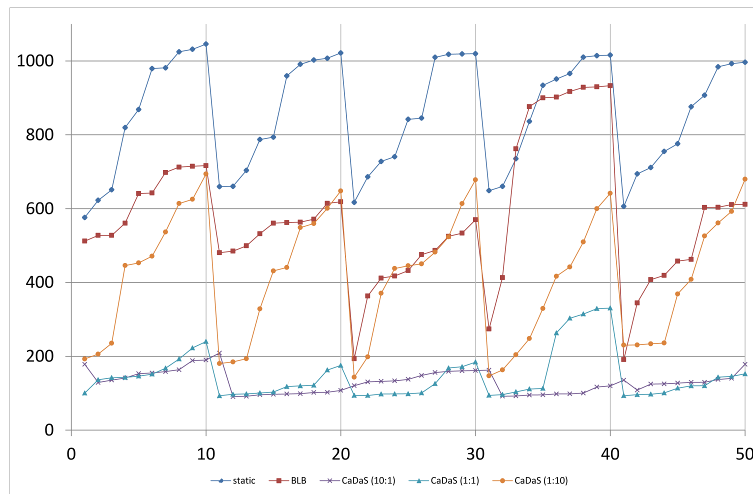


Figure 8.16: Workflow runtimes for scenario 3 (ten workflows in parallel, repeated five times with 15 minutes pause).

When virtual machines may be provisioned (LB, CaDaS 10:1, CaDaS 1:1), two effects emerge: First, though LB shows an increase in runtime within each iteration, the absolute runtime values decrease from iteration 1 to 3. In iteration 4 (workflow runs 31 to 40), new virtual machines had to be provisioned, since the previously used machines were de-provisioned (expiration of accounting period, no usage of machines during the pause). The second effect is that the runtimes using CaDaS only slightly increase within a single iteration. The reservation system first performs reservations on the four existing machines. Then, the algorithm calculates it to be more beneficial to provision additional machines than to wait for the existing machines to become available again. At this point, several virtual machines (0/0, 3/9, 0/0 for 10:1; 0/0, 0/9, 0/0 1:1) are started in parallel. To summarize, the proposed scheduling algorithm performs well, satisfies the previously defined requirements and is thus a good solution to the proposed problem.

8.3.4 Fault Tolerance

This section is concerned with the evaluation of the Fault Tolerance Module (cf., Section 6.5) applying the workflow of the medical use case (cf., Section 2.2.2). In addition to the non-functional

requirement of scalability, the evaluation of the fault tolerance module demonstrates that reliability as well can be handled from an infrastructural point of view and does not require any treatment on the business logic layer. The same workflow as in the previous evaluations, i. e., the beat and apnoea workflow is employed, Figure 8.17 shows this workflow during its development time in DAVO.

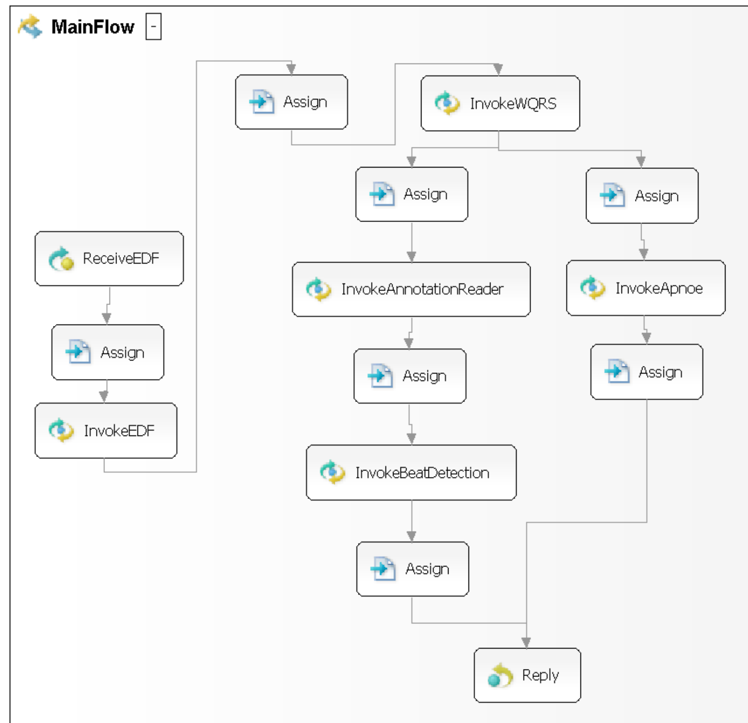


Figure 8.17: The beat and apnoea detection workflow.

Because of the relatively large data sets (approximately 120 MB per workflow instance), the need for efficient data transfer arises. Before transmitting binary data via SOAP messages, it has to be encoded as BASE64 – causing the message to be bigger than the data itself. In contrast, when Flex-SwA is used, only a reference is transmitted via the SOAP message, and the binary data is transferred from host to host by the Flex-SwA middleware. Thus in this scenario, the engine does not represent a bottleneck for data transfers anymore.

To provoke service faults during the conducted measurements, the web services were programmed in order to throw different SOAP faults with a probability of 30%. Thus, in particular, the workflow shown in Figure 8.17 succeeds in $0.7^5 = 0.16807 = 16.807\%$ the cases. In a first scenario, the workflows were executed 200 times on a pool of dedicated machines. In the second scenario, Cloud resources were added to provide spare hardware in case of *Substitution*.

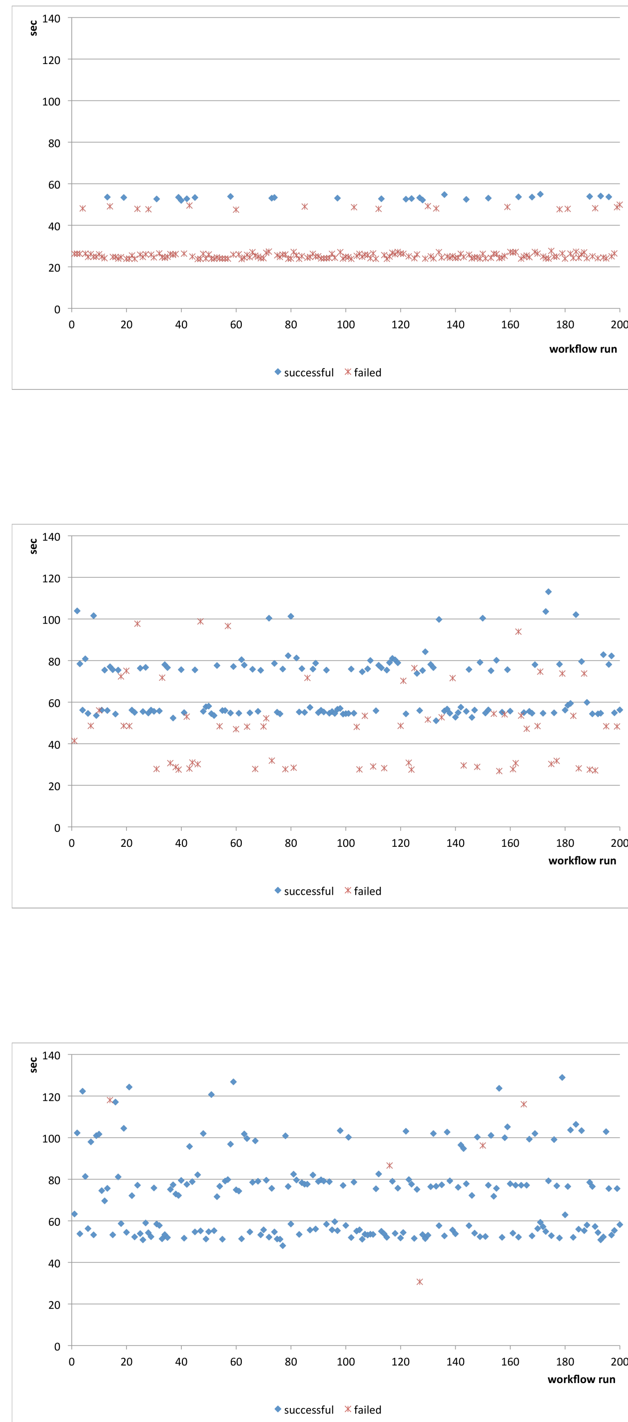


Figure 8.18: Runtimes of the apnoea analysis workflow using the Fault Tolerance Module for $r = 0$, $r = 1$ and $r = 3$ (from top to bottom).

Number of Retries	Theoretical Fail Ratio	Empirical Fail Ratio
0	0.83193	0.838
1	0.37597	0.3960
2	0.12790	0.15
3	0.039849	0.025
4	0.012091	0.018
5	0.0036397	0

Table 8.3: Fail ratios for the given apnoea analysis workflow, which consists of five service invocations (cf., Figure 4.2).

The dedicated machines were Core2Duo E6850 with 2 GB RAM, and as Cloud resources the so-called "small instances" (1.7 GB RAM, 160 GB hard disk, 32 Bit Linux, $0.10 per hour[3]) of Amazon's EC2 were used.

All policies for different fault categories were set up to perform the same number of retries. If q is the fault ratio of any service and r is the number of retries and substitutions specified in the policies, then the probability of an unsuccessful workflow run[4] is given as

$$\sum_{i=0}^{\#Services-1} q^{r+1} \cdot (1 - q^{r+1})^i$$

In Table 8.3, the theoretical and empirical values are shown. In Figure 8.18, the workflow runtimes for the non-Cloud scenario for different settings of the parameter r are shown: $r = 0$ means that no recovery is allowed, $r = 1$ means that only *one* retry or *one* substitution are feasible. For $r = 0$ it is obvious that the majority of workflows fails early (after approximately 25 sec). It turned out that the initial data transfer always succeeded; then, the subsequent invocation of the first service tended to fail frequently. In some cases, the workflows were abandoned at a later stage due to faults in subsequent service invocations.

The figures for $r = 1$ and $r = 3$ contain cluster points. These correlate with the different number of *Retry* and *Substitution* operations that had to be performed for a successful invocation. In some cases, this very number exceeded the configured r, meaning that the workflow terminated with failure. For $r = 3$, only five out of 200 workflow runs failed (2.5%), which is close to the theoretical value of 3.98%. It is fairly obvious that the majority of workflows run longer than just the raw execution time (approximately 56 sec, see successful runs for $r = 0$). This is, among others, due to the fact that input data first has to be transferred to the surrogate machine. As a rule, one can say as r increases so does the average runtime while at the same time the failure ratio decreases. For $r = 5$, in 200 runs none of the workflows failed.

[3] As of November 2013, the price of a small instance is $0.06 per hour.

[4] A workflow run is considered unsuccessful, i. e., the workflow engine terminates the execution of the workflow instance, if a service invocation returns a fault message. Using the FTM, a fault message is returned, if r invocations of a service have failed.

In Figure 8.19, the runtime of workflows with $r = 3$ *and* Cloud-based substitution is illustrated. The provisioning of Cloud resources (including the startup of middleware components) takes approximately 60 sec (cf., Section 8.3.3). Internal monitoring shows that four different Cloud resources (more precisely, in run 1, 13, 14 and 16) were started, which also explains the four different peak values during in the execution time. The average runtime (44.35 sec) lies below the average runtime of workflows on dedicated resources. While this might appear surprising at first (because of the lower computational power of the virtual machine), this is caused by the higher network bandwidth available at Amazon's EC2. The virtual machines within the EC2 are (guaranteed to be) connected via a 250 MBit/sec network, whereas the dedicated machines were connected by only a 100 MBit/sec network.

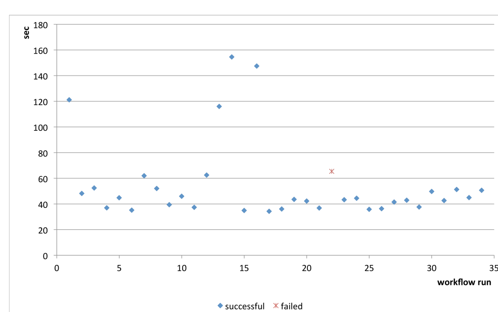


Figure 8.19: Runtimes of the workflow for $r = 3$ with Cloud-backed redundancy.

8.4 Request/Response Aspects

This section focuses on the evaluation of the framework for request/response aspects. First, the general applicability of the framework is demonstrated and afterwards its specific application to the multimedia data analysis use case is elaborated.

8.4.1 Applicability of Request/Response Aspects

In this section, the realization of three applications of request/response aspects is discussed. First, the realization of efficient data transmission as an aspect is considered. The final goal is to achieve efficient data transmission between web services with neither additional implementation efforts (except the one time development of each aspect), nor changes to the web services, nor the loss of modularity, nor a tighter coupling of web services. The second application realizes data compression and finally, motivated by the medical analysis use case, the third one provides a cryptographic data transfer via request/response aspects.

Data Transfer

For the realization of efficient data transfer as a request/response aspect, consider the general design shown in Figure 7.1: Before an aspect can handle the request of Service S_2, the response of Service S_1 must already have been handled by an aspect. In this way, the aspect woven into Service S_1 implements the `IResponseAdvice`. This implementation first takes the binary data returned by the service (Listing 8.4) and then creates a new Flex-SwA reference pointing to the data. Subsequently, this reference is encoded into the response so as to fit syntactically into the data structure, which the service returns. After this (response) message has been passed on to Service S_2, the aspect woven into this service eventually implements the `IRequestAdvice`. This implementation expects a Flex-SwA reference encoded into the data structure, which the service receives. The encoded reference is resolved, and finally the data is transferred.

For the evaluation of the behavior of this aspect, the corresponding services are kept as simple as possible with respect to their processing activity. This ensures that a potential speedup of the Flex-SwA advice is not mixed with a data dependent processing time of the services. For this reason, the services directly return the received data. For the test of the Flex-SwA advice on various data types, the following three different service implementations were investigated. While the first echo service implementation works on a byte array, the second service uses strings, and the third service is based on SOAP with Attachments.

```
public Object handleResponse(Object value) throws Exception {

  if (value == null) {
    return value;
  }

  Reference reference = new Reference();
  OutputStream os = reference.request(...);
  ObjectOutputStream oos = new ObjectOutputStream(os);
  oos.writeObject(value);
  oos.close();
  String referenceString = JSONSerializer.toJSON(reference);

  Object result = null;
  if (value.getClass().isArray()) {
    result = convert(value, referenceString);
  } else if (value instanceof String) {
    result = referenceString;
  } ...

 return result;
}
```

Listing 8.4: Response handling of the data transfer aspect.

These services represent a wide spectrum of data transfer mechanisms in SOAP environments [198]. The performance of request/response aspects is evaluated using these three web services and the

Flex-SwA advice mentioned before. This specific advice not only handles the request and the response message of the services, but also deals with their (different) message formats.

In order to perform the weaving of the aspect for efficient data transmission, a client that utilizes the presented framework only needs to call an aspect configurator service with an aspect (see Listing 8.5) as an argument.

It is expected that the weaving of the Flex-SwA aspect improves the workflow runtime significantly. The tests are performed on three machines running Fedora Linux, which are equipped with the same hardware, namely an Intel Core2Duo E8600, 4 GB of RAM and a 100 MBit/sec Ethernet network. One of these machines operates as the client, whereas the other two host the echo services. Different transmission types and file sizes are used, but not all tested transmission types support arbitrary data sizes. Due to the expensive XML encoding (with respect to runtime and especially to arrays) the available heap space for the Java virtual machine is the limiting factor for the test of the byte array service. More precisely, the heap space is set to 2 GB (`-Xmx2024m`) and the Java Garbage Collector is allowed to run concurrently (`-XX:+UseConcMarkSweepGC` and `-XX:-UseGCOverheadLimit`). Only with this setup, a test of the transmission of byte arrays containing up to 800, 000 single byte values is possible.

```
Aspect serviceaAspect = new Aspect(
   new QName("http://fb12.de/AosStringTestService",
     "AosStringTestService"), "echoStringA", "/data",
   Aspect.AOP_RESPONSE_MODE, "FlexSwAPlugIn")
```

Listing 8.5: Java bean constructor of an aspect.

In order to show the relative speedups of the the different echo service scenarios (each measurement was repeated 100 times), their workflow execution times were compared. When request/response aspects are used, the loss of effectiveness of the data transfer mechanism and the improvement in the relative runtime correlate. For the byte array, a large improvement of up to 50% could be achieved, i. e., the transmission of a byte array of size of 800 kbytes took about 95824 sec using plain SOAP communication and about 50190 sec using request/response aspects in combination with Flex-SwA. The explanation for the runtime improvement of up to 50% is as follows: As indicated in Figure 7.1, each SOAP transmission first requires a serialization, before the actual transmission over a network and finally a deserialization takes place (repeated in each step 1 – 4). Hence, the overall runtime time is determined as $T_{\mathrm{SOAP}}(n) = 4 * t(n) + \varepsilon$, where $t(n)$ is the network transmission time, including the serialization and deserialization time, n is the amount of transmitted data, and ε represents negligible processing times. In case of the Flex-SwA request/response aspect, in step 2 and step 3 only the woven reference instead of the actual payload is transported. Since the size of a reference is independent of the referenced amount of data, the corresponding runtime can be expressed as $T_{\mathrm{R/R}}(n) = 2 * t(n) + 2 * t' + t_D(n) + \varepsilon$, where t' is the corresponding time for transferring a single reference and $t_D(n)$ is the time needed to transfer the payload via Flex-SwA. For large n, one obtains $t' \ll t(n)$, $t_D(n) \leq t(n)$, which implies $T_{\mathrm{R/R}}^{n\to\infty}(n) = 2 * t(n) + t_D(n)$. The comparison of $T_{\mathrm{SOAP}}(n)$ and $T_{\mathrm{R/R}}^{n\to\infty}(n)$ indicates that the theoretical runtime improvement lies (slightly)

below 50%[5]. The measurements show that the proposed framework approaches this theoretical limit. Though the overhead introduced by the request/response aspect is noticeable for small data sizes when using strings or SOAP with Attachments (cf., Figures 8.20 and 8.21), for larger data sizes ($\geq$ 40 kbytes) the achieved runtime improvement outweighs this overhead by far.

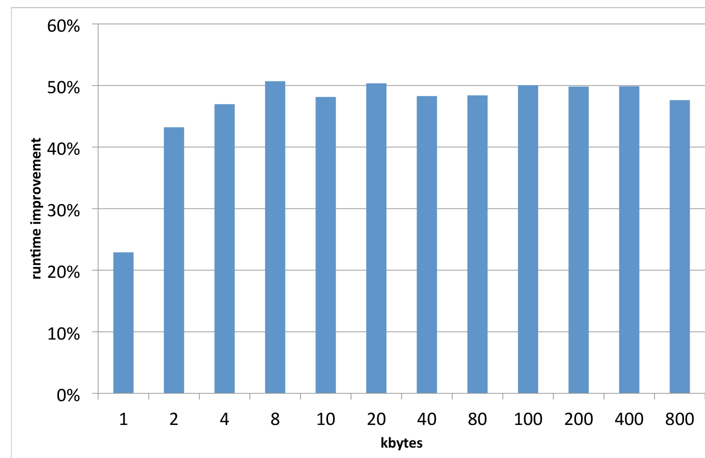


Figure 8.20: Byte Array – relative runtime improvement using Flex-SwA aspects.

It is worth mentioning that not only is the execution time of a workflow of the described kind accelerated by a significant factor, but the development time is also shortened considerably. This reduction in development time is caused by the facts that simple data types can be used for the development of the services and that the more complex reference component for data transfer can be introduced using the proposed aspect framework.

Data Compression

Text detection in videos is another workflow of the motivating multimedia analysis scenario: here, (depending on a given input video) large text documents might be generated. These documents can be transported via Flex-SwA again, but at the same time it is desirable to compress the text data. In the detailed approach, a special advice was implemented to compress data and afterwards encode it with BASE64 in order to embed it again in an XML document. This aspect can reduce the message size to 60% of the original size. Another possibility is to use the message-based compression offered by the web service container. However, aspect-based compression allows the usage of field-specific compression algorithms for different parts of the message, which is possible by the field operator of the pointcut description. This is reasonable since different data types (text, images, videos, etc.)

[5]Assuming that $2 * t_D(n) \geq \varepsilon$, which holds since the passing data inside the service implementation (ε) requires less time since the actual transmission of this payload.

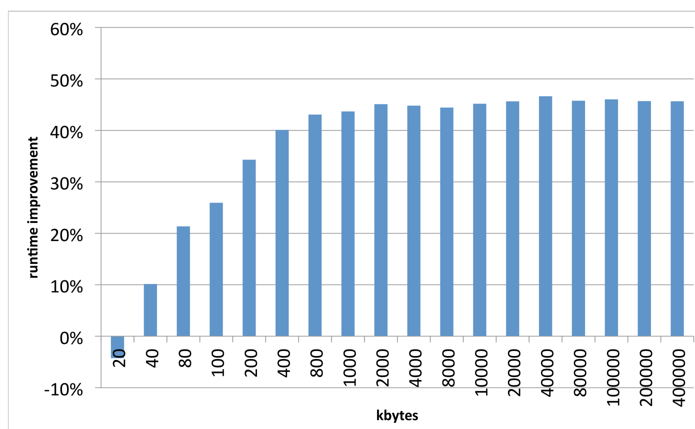


Figure 8.21: SOAP with Attachments (SwA) – relative runtime improvement using Flex-SwA aspects.

may require different compression techniques. If a better compression algorithm becomes available, it is much easier to replace the applied compression using the proposed aspect-oriented solution as compared to using a message based compression. In this case, a new compression approach would have to be realized in all implementations of related web services.

The above example indicates that the incorporation of new non-functional requirements via an advice is a straightforward procedure, since it effectively represents a filter operation applied to basic data types. For employing such advice by the proposed framework, they only need to be applied to the primitive data types. If, due to a revision, maintenance, or other updates, the service changes over time, only (changed porttypes, field names, etc.) the pointcut (cf., Listing 8.5) needs to be updated.

Data Encryption

Secure messaging is another crosscutting concern that can also be tackled by the means of request/response aspects. This problem is exemplified by a workflow that has been developed during a cooperation with medical researchers (cf., Section 2.2.2). When performing patient studies, it is important that personal patient data is kept private by either making it anonymous or encrypting it. For this reason, an advice that performs an encryption at the data source has been developed. The encryption is realized such that none of subsequent services is able to read sensitive data. Eventually, the decryption aspect is located at the service that merges the results into a patient's record. The use of such an aspect allows for encrypting services without neither changing the service implementation nor configuring the middleware.

Besides anonymously recorded vital signs, sensitive patient data might be sent to (and received from) the services. Due to legal regulations, such data must not be transferred unencryptedly.

```
protected Object handleString(String value) throws Exception {
  Cipher cipher = Cipher.getInstance("RSA");
  cipher.init(Cipher.ENCRYPT_MODE, getPublicKey());
  byte[] encrypted = cipher.doFinal(value.getBytes());
  return new BASE64Encoder().encode(encrypted);
}
```

Listing 8.6: Response handling (of strings) of the encryption advice.

The data exchanged between the services contains not only the actual ECG measurements but also several identification attributes. To prevent the misuse of these attributes, in this approach a privacy advice has been developed relying on public key cryptography. The support of the wildcard operator allows for the encryption (and decryption) of all of the patient related data. This data is encrypted when it is initially retrieved from a database. During its processing by the mentioned services, the personal information is encrypted (while the ECG data remains unencrypted). Finally, when the analysis is finished and the result has been stored in the database, the corresponding advice decrypts the personal data.

Further Applications of Request/Response Aspects

There are other areas of the web service standards that can benefit from the application request/response aspects. For example, the reliable transmission of a message is another crosscutting concern that can be handled by request/response aspects. The Web Services Reliable Messaging Protocol (WS-ReliableMessaging) [145] describes the reliable transfer of messages between nodes that use this protocol in case of software component, system, or network failures. Reliable Messaging (RM) is based on so-called RM Sources and RM Destinations to transfer messages reliably. Instead of implementing the establishment of RM Source and RM Destination in the client and the service, it is realizable by means of request/response aspects. Reliable messaging operates with control messages in the SOAP header, e. g., to start a reliable message sequence, to acknowledge received messages etc. The reliable message aspect can inject these header messages and it can also react to incoming control messages.

Another area for request/response aspects can be found in the profiling of service calls that can provide a basis for accounting. Such an aspect uses the wildcard operator to profile all service calls within a specific porttype, or even within several porttypes within a middleware container. When such an aspect is woven into one or more web services, it enables the profiling and the tracing of web service calls performed by particular users.

8.4.2 Request/Response Aspects for Multimedia Data Analysis

In this section, request/response aspects are evaluated for the multimedia data analysis use case (cf., Section 2.2.1). Instead of evaluating synthetic web service compositions as in the preceding section,

the use of real-life service compositions allows for a more realistic estimation of the leverage of request/response aspects on the communication of web services.

To assess the proposed approach, an implementation for face detection that consists of three main components is analyzed. First, for a given input video, an MPEG decoder sequentially produces a series of frames. After the decoding, each frame is processed by the face detection algorithm (mainly consisting of feature extraction and classification). The results of this algorithm are one or more bounding boxes that are stored in a simple list. Finally, in a last step, when the whole input video has already been processed in this manner, the final list is given to a specific component that uses this information to build an MPEG-7 result file.

A straightforward decomposition of this implementation is realized by wrapping each of the components as an individual web service, resulting in three services: namely an MPEG decoder service, a face detection service, and an MPEG-7 converter service. The purpose of the last service is to collect the coordinates of detected faces and to integrate them into an existing MPEG-7 file containing the shot segmentation of the video. The decoded frames, which are produced by the MPEG decoder service are represented as integer arrays along with some meta-information such as frame number, width and height. Operating on the level of single frames supports yields to a high level of interoperability with other services and with clients from different scenarios. All three services are deployed into an Apache Axis 1.4 SOAP processing engine that runs within an Apache Tomcat 6. Using DAVO (see Figure 8.22), a BPEL workflow, which is composed of these services, has been modeled. This workflow processes a complete video file by a sequential call of the services for each frame.

To utilize the proposed solution, a user has to graphically annotate the given multimedia workflow only once. This means that the data flow to be optimized by the aspect for efficient data transfer is simply marked by the user. This information is saved within the BPEL process. If, later on, this process is executed by the BPEL engine, the aspect invoke handler (AIH) interprets this very information and performs the corresponding calls to the Aspect Configurator. These calls take the aspect description of Listing 7.6 as its input. This procedure is repeated at all services that are supposed to use the data transfer optimization. After the configuration phase just described, the aspects are woven and the workflow can be executed without further modifications. Since for other scenarios it is also possible to explicitly use a Java library that provides an API for weaving request/response aspects, it is not necessary to use annotations to weave aspects. In particular, the procedure of weaving aspects is a another web service call of the Aspect Configurator with the aspect bean (cf., Listing 7.1) as input argument.

Referring back to Figure 2.3, the decoder service is responsible for the data pre-processing step. In the presented setup, an actual data transformation is not necessary, because the decoded frames can be directly processed by the face detection service. Thus, this step can be omitted. Both, the feature extraction and classification, are performed by the face detection service relying on face detector of OpenCV. Finally, as the post-processing step, all meta-information generated by the face detection service is aggregated by the MPEG-7 service.

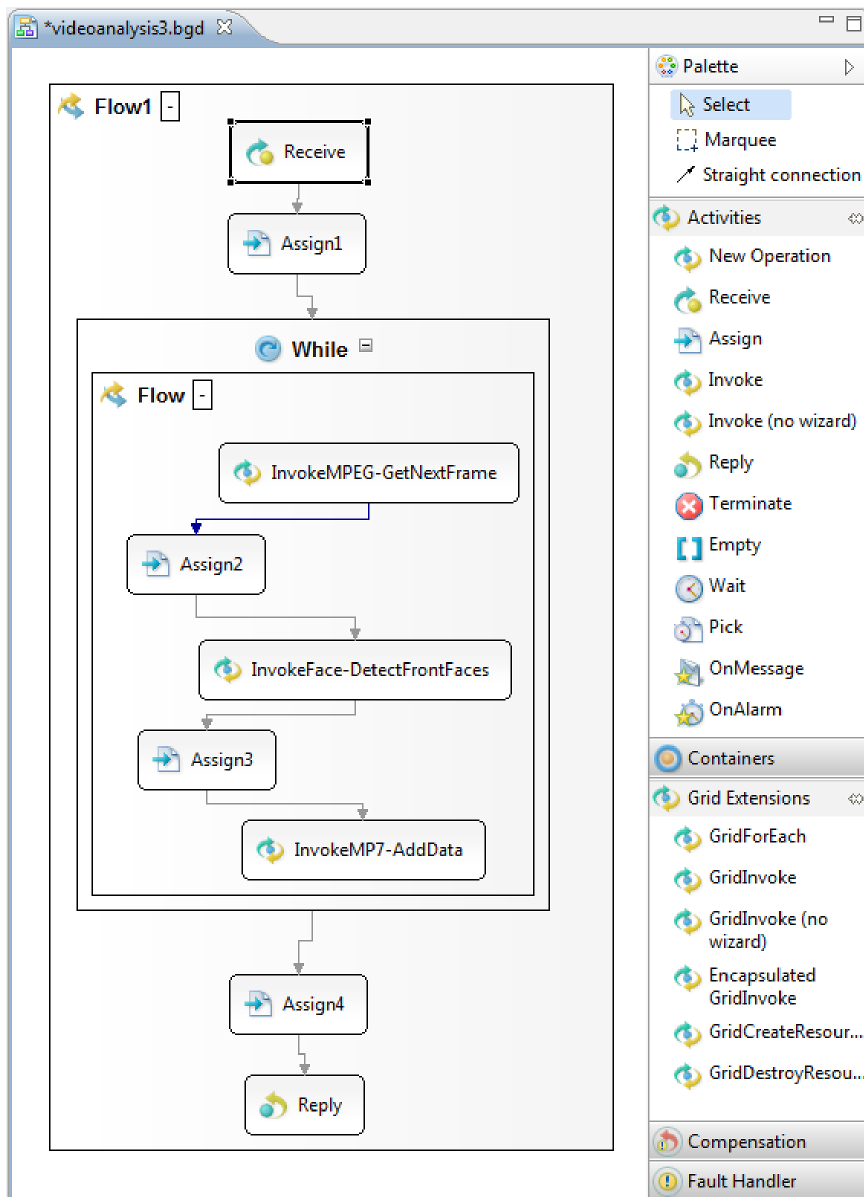


Figure 8.22: Modeling the face detection workflow with DAVO. A video is passed to the workflow and taken as input. The MPEG decoder segments the video into single frames, which afterwards are analyzed by the face detector. This is repeated until the last video frame, the while loop terminates and the workflow finishes.

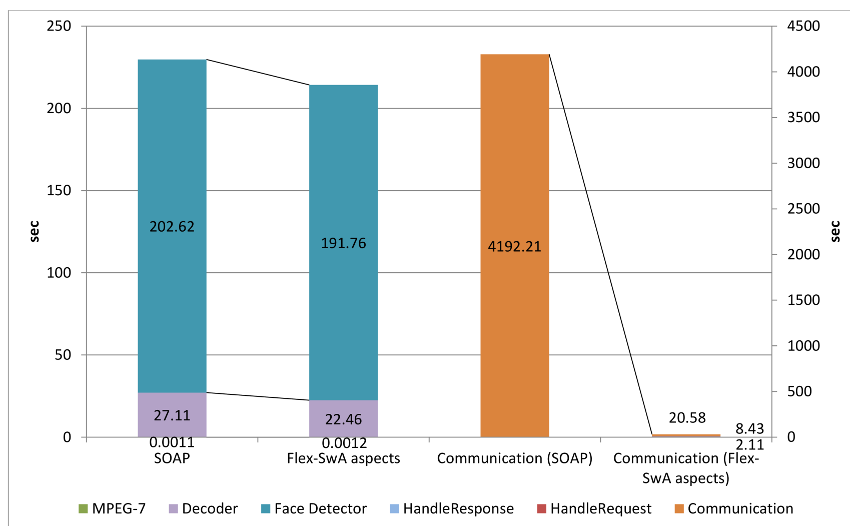


Figure 8.23: Runtime divided into computational and data communication efforts; left y-axis: service runtime without data communication, right y-axis: time needed for data transfer/ communication (orange bars).

One of the basic problems to be solved in the considered scenario is the sequential invocation of two subsequent services in the workflow, one of which produces (large amounts of) data and consumed by the other one. The available network bandwidth of the machine hosting the corresponding BPEL engine can very quickly become a bottleneck. The transfer of huge data via the BPEL engine, can be avoided by using the proposed aspect framework.

Experimental Setup and Results

All tests were performed in the Amazon Elastic Compute Cloud (EC2) which allowed for easily setting up a homogeneous environment to perform the analyses. Although other authors have had rather negative experiences with the reliability of EC2 computing resources [162], the obtained results show only a small standard deviation. The tests are performed on two and three EC2 machines of the type High-CPU Medium Instances, respectively, that have a guaranteed interconnection rate of 250 Mbit/sec. Each of these instance types come along with 1.7 GB memory, 5 EC2 compute units, 350 GB storage and a 32-bit platform. At the time the tests are performed, an EC2 compute unit is proclaimed to provide the same CPU capacity as a 1.0 – 1.2 GHz 2007 Opteron or a 2007 Xeon processor. To determine the overall runtime, the client records the overall process runtime. The aspect framework records the actual execution time of the service's logic. By substracting these two values (minus possible runtimes for aspect handling), one obtains the communication time.

In the first scenario, no aspects were used. Thus, the data exchange between the BPEL engine and the service consists of plain SOAP messages. Only two machines are used and needed, respectively, one hosting the BPEL engine, and one hosting the three services.

In the second scenario, the developed aspect framework is tested. Each run uses three EC2 instances, one for the BPEL engine and two as worker nodes. One worker hosts the decoder service, while the other one hosts the face detector and the MPEG-7 converter service. On both worker nodes, the aspect framework and also the Flex-SwA aspect are available. Since the decoder and the face detection service carry the biggest fraction of data exchange, the request/response aspect based on Flex-SwA is woven into this communication. It substitutes the array containing the frame data, which is returned by the decoder through a reference. This setup leads to the highest number of data transfers (compared to a monolithic implementation), because each service invocation triggers a new data transmission between different hosts.

Each scenario carries out the analysis of one news video (approx. 100 MB, 20051209_125800_CNN_LIVEFROM_ENG.mpg) from the TRECVID 2006 test set [169]. The results of the conducted measurements are shown in Figure 8.23. Each value represents the average of 35 runs that are performed for both scenarios. With respect to the three considered web services, the face detector service has the longest runtime, with 202.62 sec in the first and 191.76 sec in the second scenario, followed by the MPEG decoder service with 27.11 sec and 22.46 sec, respectively. Regarding the accuracy of measurements, no difference can be detected for the runtimes of the third service, the MPEG-7 collector service, i. e., 1.1 msec to 1.2 msec. Nevertheless, the client runtimes differ significantly. When using plain SOAP communication, the runtime is 4421.94 sec, whereas in the second

scenario the runtime only amounts to 245.34 sec (see Table 8.4). To summarize, one observes that service runtimes are almost equal while the times needed for communication differ significantly. Compared to the average SOAP runtime, this yields to an average speed-up factor of 18.02 and of 15.98 and 18.85 in the worst and the best case, respectively. When using SOAP, the communication takes 4192.21 sec. By means of the request/response aspect, this communication overhead could be reduced to to 20.58 sec. The use of request/response aspects results in an overhead of 10.5 sec: 2.1 sec for the request message and 8.4 sec for the corresponding response message. The standard deviation in the SOAP setup was approximately 3.3% and 5.2% in the other case for the runtime measurements. In particular, the EC2 machines have proven reliable with respect to the computational resources for the given scenarios.

	Plain SOAP	**Flex-SwA aspects**
Video Decoder	27.1	22.4
Face Detector	202.6	191.7
MPEG-7 Converter	0.001	0.001
Communication	4192.2	20.5
Client	4421.9	245.3

Table 8.4: Runtimes in seconds. The client runtimes is comprised of the different service invocations and the communication time. In case of the plain SOAP client, the communication time obviously dominates the overall runtime. Using request/response aspects the overall runtime can be significantly reduced without additional development efforts and service downtimes, respectively.

The large difference between the time needed for communication in the plain SOAP scenario compared to the aspect scenario can mainly be explained by the following two reasons. The first one is that the transfer of the data array is doubled (from the decoder to the workflow engine and from there to the face detector) when plain SOAP communication is used. The second reason can be found in the serialization and deserialization of the data array. Since video frames encoded as an array usually contain many items/elements (several 100.000 entries that represent the encoded pixels of each frame), these are further time-consuming tasks. However, the overhead using request/response aspects (2.1 sec on the request side and 8.4 sec on the response side, respectively) is negligible due to the massive runtime improvements achieved by the request/response aspects (245.34 sec compared to 4421.94 sec).

Due to the following two reasons, the proposed system allows to reduce the communication overhead. On the one hand, the absolute number of large data transmissions is reduced, and on the other hand the type of transmitted data has changed. To be more precise: assuming a system in which the response of the decoder service is processed by multiple post-processors (e. g., face detection and text detection/text recognition), this particular response must be sent to several services. The proposed approach not only reduces the number of transmitted messages from $(n + 1)$ (from the decoder via the BPEL engine to the n successor services) to n, but in addition yields a significant

reduction of the overhead for serialization and deserialization. In the original case, the XML representation must be serialized and deserialized by the decoder and the BPEL engine, respectively. As a consequence, the data is transmitted over the network in this rather memory and bandwidth consuming representation (the same is also true for the data transmission from the BPEL engine to the successor services). In the proposed aspect-oriented solution, however, only a (relatively) small reference has to be transferred and – in a streaming-like approach – the bulk data is then directly passed from the decoder service to its successors.

8.5 Summary

In this chapter, a comprehensive evaluation of the components presented and developed in this thesis is conducted. The evaluation shows that the postulated requirements are accomplished and it furthermore shows the versatility of the proposed solutions.

9
Conclusions

Contents

9.1 Summary **181**
9.1.1 Service Development 182
9.1.2 Adaptable Service Workflows 182
9.1.3 Request/Response Aspects 183
9.1.4 Evaluation 184
9.2 Future Work **184**
9.2.1 Service Development 184
9.2.2 Adaptable Service Workflows 185
9.2.3 Request/Response Aspects 185

This chapter concludes this thesis on *Runtime Adaptation of Scientific Service Workflows* and indicates some directions for future work. During its course, methods as well as solutions in the form of implementations have been presented that allow to adapt service compositions at runtime in service-oriented architectures in a general way. For this aim, different techniques and mechanisms were developed and used, respectively, all fulfilling the goal of being easy-to-use without the introduction of a new layer of complexity.

9.1 Summary

In this thesis, first three different use cases stemming from the different areas of multimedia data analysis, medical research and systems biology have been presented. A subsequently conducted requirements analysis identified the necessity of a general approach for runtime adaptation of service workflows in service-oriented architectures. A review of related work revealed that no such solution existed yet. An architectural blueprint then pointed out how the handling of non-functional requirements is realizable and thereby eradicating this flaw. The details were elaborated in the succeeding chapters, as summarized in the following.

9.1.1 Service Development

The LCDL framework, an extensible approach for wrapping legacy code, was presented. It supports different types of legacy code, being based on a plug-in mechanism, also support several input and output types as well as binding mechanisms. A use case from the area of speech processing has been presented to illustrate our approach. (Section 5.2.)

It was argued that modeling scientific workflows by means of a workflow language like BPEL is a promising approach, since scientists can rely on industry-proven tool support and execution engines. However, BPEL is a powerful, but still complicated, general-purpose language. To foster adoption of BPEL in the scientific community, a simplified modeling solution was developed that builds on domain-specific process fragments that are created by scientists with workflow skills. The design and implementation of the SimpleBPEL Composer was discussed. (Section 5.4.)

9.1.2 Adaptable Service Workflows

The design and implementation of a BPEL-based workflow system that supports on-demand resource provisioning was presented. The approach automatically schedules workflow steps to underutilized hosts and provides new hosts using Cloud computing infrastructures in peak load situations. An implementation based on the ActiveBPEL engine and Amazon's Elastic Compute Cloud was provided. (Section 6.2.)

It was argued that when workflow languages like BPEL are used for service orchestration in data-intensive applications, there is a lack of data flow aware scheduler. Existing DAG-based scheduling approaches cannot be applied, since BPEL is a non-DAG (Turing-complete) language. The presented approach distinguishes between of development time and execution time components: at development time, the workflow developer may annotate the workflow with information about the data flow and the execution time. At execution time, these annotations are processed to create a data flow graph that in turn serves as input for the developed genetic algorithm that performs the assignment of BPEL workflow steps to resources. A genetic algorithm operates on critical paths of the workflow instead of on the complete workflow, yielding a reduction of runtimes. The algorithm is able to simulate whether the makespan of a workflow would decrease if further machines (i. e., newly provisioned Cloud resources) were added to the pool of available machines. The implementation does not require any changes to the BPEL standard and has been shown to significantly reduce the execution times. (Section 6.3.)

A novel workflow scheduling algorithm as well as its implementation are tailored towards the needs of Cloud-based workflow applications: in particular, if the workflow tasks are widely geographically distributed, data transmission can form the main bottleneck. To avoid this, the developed algorithm takes data dependencies between different workflow steps into account and depending on the preferences of the user (i. e., depending on which of the two conflicting objectives of low cost and low execution time the user favors) these workflow steps are scheduled to Cloud resources. Several implementation details (e. g., provisioning, reservation system, branch crossover)

have been described. The implementation is based on, but not limited to, the ActiveBPEL engine and Amazon's Elastic Compute Cloud. (Section 6.4.)

Additionally, a novel approach for the transparent handling of infrastructural faults when long-running and computationally-intensive workflows in distributed computing environments are model with BPEL has been presented. Classes of faults that can be automatically handled have been identified. The approach is based on policies to configure the automatic behavior without the need for adding explicit fault handling mechanisms to a BPEL process. It provides automatic redundancy of services, since the use of a Cloud computing infrastructure allows for the substitution of defective services. An implementation based on Amazon's Elastic Compute Cloud was presented. (Section 6.5.)

An approach that includes the orchestration engine in the fault handling was also presented. Using sophisticated Cloud technologies, the issues of the single point of failure at the workflow engine, which would usually remain in a service-oriented architecture, can thereby be avoided. (Section 6.6.)

9.1.3 Request/Response Aspects

Request/response aspects for web services were introduced that allow developers of service-oriented applications to easily enrich web services with additionally required functionalities, such as efficient data transmission, data compression, secure messaging, or other crosscutting concerns. The need of request/response aspects is motivated by the described applications in the use case section and the requirements analysis section. Additional functionalities can be dynamically woven into remote web services without changes to their implementations or interfaces. The presented framework that offers support for request/response aspects includes a pointcut description for join points in SOAP-based web service environments. Additional functionalities can be dynamically woven into remote web services without changing their implementations or their interfaces. The presented framework supporting request/response aspects includes a pointcut description for join points in SOAP-based web service environments. Request/response aspects are advantageous in several ways: They allow to add functionalities at the web service invocation and communication levels, thereby offering the possibility to execute aspect code on the remote host where a called web service is running. Furthermore, they are independent of the implementation language of a specific web service and do not require the web service implementation to provide related functionalities. By using the aspect framework, the development of web services is simplified considerably. This is demonstrated, for example, by creating the functionality of efficient transmission of large amounts of data in a web service workflow by circumventing the bottleneck at the client or workflow engine. (Section 7.3.)

In particular, an aspect-oriented solution to this issue for multimedia web services was provided. The developed request/response aspects are woven into the message handler chain of web services and leave the web service implementation unchanged. References are used for the orchestration of data between multimedia services. The solution is based on the Flex-SwA framework in conjunction with the BPEL workflow language. The proposed aspect-oriented solution significantly reduces the

required software development efforts for implementing efficient data transmission between web services. Hence, software developers of multimedia applications benefit from the advantages that are offered by service-oriented environments. (Section 7.6.)

9.1.4 Evaluation

Finally, an evaluation of the presented solutions showed and emphasized their benefits and advantages. In addition, the evaluation verified that all six requirements stated in the requirements analysis (see Section 2.3) are covered by the presented solutions.

By using a service-oriented architecture based on web-services in combination with the industrial-proven workflow environment BPEL, in combination with the presented LCDL and the SimpleBPEL approach fulfill Requirement 2 (see Chapter 5). Requirement 3 is achieved by the non-functional extensions for workflow engines and especially by the multi-objective scheduling approach (see Section 6.4f). The reliability Requirement 4 is accomplished by the fault tolerance module (see Section 6.5). Security (Requirement 5) is covered by the choice of web services and the WS-* universe. The handling of changing and cross-cutting concerns (Requirement 6) is satisfied by request/response aspects (see Chapter 7).

Taken all the presented methodologies and solutions of this thesis into account, Requirement 1 is achieved.

9.2 Future Work

This section indicates areas of future work for the solutions presented in this thesis. Future work that focuses on the specific realizations is discussed chapter wise.

9.2.1 Service Development

Considering the LCDL framework, the following areas of future work arise: (1) currently, the invocation of methods that are provided by a library can only use primitive data types. Since libraries are typically used within programming code, the occurrence of composed and thus more complex data types is not uncommon. However, a programming language and target platform independent way of describing such complex types is necessary. For this aim, the use of XML schemata can provide means to describe such complex data types. The Environment element can be extended (2) so as to be able to handle JSDL information, e. g., hardware requirements, so as to (3) foster an automatic deployment of compute nodes by using Cloud Computing or desktop pools [90]. (4) The incorporation of additional interfaces – bindings for Mathematica, Matlab interfaces or an integration into (Java) Message Bus Systems is conceivable.

There are also several areas for future work concerning SimpleBPEL. From an implementation perspective, the exchange of profiles could be realized by means of a centralized database repository instead of a file-based manual exchange. Conceptually, we are currently investigating how semantic validation of SimpleBPEL fragment compositions can be achieved. The goal is to compute the

semantics of a fragment just from the semantic description of the enclosed services, and to validate whether the fragments semantically fit or not whenever the user wants to connect two fragments. “Semantic Validation of BPEL Fragment Compositions” developed by Harbach et al. [88] presents a solution to this question.

9.2.2 Adaptable Service Workflows

Future work for the basic scheduling approach includes the integration of further backend implementations, such as Globus Virtual Workspaces/Nimbus. For WSRF services, it is at the moment not possible to make use of the Grid Security Infrastructure (GSI), since GSI needs host-based certificates, but host names in Cloud environments are subject to continuous change. Instead of providing machines when a call is to be executed, the system could perform advanced provisioning depending on the overall system load. Having a spare machine would be especially useful for workflows with many tasks that only have short execution times. The execution time of a service call would remain unaffected by the provisioning time.

Another area of future work for both the data flow aware scheduling and the multi-objective scheduling architecture is concerned with the integration of data caching mechanisms into the described service-oriented environment. To ease the development and to increase the accuracy of workflow annotations, an investigation on how automatic assignments and determination of data flow graph annotations can be achieved. Finally, interesting properties of DAG-based schedulers should be studied and if possible adapted to the BPEL schedulers.

A goal of future work for the fault tolerance module is to eliminate the single point of failure that is located at the registry of the Load Balancer. A possible solution for this issue consists in the distribution of the registry. The adaptation (during runtime) to new failures and fault groups forms another field of future work, in order to overcome the restriction to fixed set of faults. Furthermore, the policy and monitoring framework should be extended so as to allow the definition, monitoring and enforcement of quality-of-service (QoS) parameters, which is of particular interest for time-constrained web service architectures [129].

9.2.3 Request/Response Aspects

The framework for request/response aspects is also subject to future work. For example, instead of transferring the aspect-ID (plugin-ID), it is also easily conceivable to copy the whole aspect either as (Java) binary code or as an interpretable description to the remote service. Concepts like sequence pointcuts and shared states between aspects executed on different hosts [139] are other interesting enhancements of the presented approach. Finally, to prevent a congestion of services with aspects over time, a sophisticated life cycle management (e. g., according to wall-clock time or communication patterns) is another area of further research.

Regarding the application of request/response aspects, it is interesting to investigate, model and evaluate further aspects and their applicability to the domain of distributed service-oriented multimedia content analysis. Thus, one target is the automatic integration of cross-cutting concerns

such as reliable messaging and profiling. Further possible investigations concern the usage of the framework in the field of time-constrained web services and for the modeling of soft real-time requirements. Also some kind of context-awareness will be considered, where the localization of the interacting services should be recognized to optimize the data flow by eliminating unnecessary data transport. Another direction of future work is the investigation of aspects that aggregate two or more services into a new one to reduce communication costs. Such an approach will introduce the principle of locality in composed service-oriented environments.

•

> The White Rabbit put on his spectacles. 'Where shall I begin, please your Majesty?' he asked. 'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'
>
> Alice's Adventures in Wonderland

— • —

Bibliography

[1] ActiveEndpoints. ActiveBPEL Business Process Execution Engine. `http://www.activebpel.org`, 2011.

[2] ActiveEndpoints. ActiveVOS Business Process Management. `http://www.activevos.com`, 2013.

[3] A. Akram, D. Meredith, and R. Allan. Evaluation of BPEL to scientific workflows. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, 2006.

[4] A. Akram, D. Meredith, and R. Allan. Evaluation of BPEL to Scientific Workflows. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 269–274. IEEE Press, 2006.

[5] Amazon Web Services LLC. Amazon CloudFront . `http://aws.amazon.com/cloudfront/`.

[6] Amazon Web Services LLC. Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[7] Amazon Web Services LLC. Amazon Simple Storage Service (S3). `http://aws.amazon.com/s3/`.

[8] Amazon Web Services LLC. Auto Scaling. `http://aws.amazon.com/de/autoscaling/`.

[9] Amazon Web Services LLC. Elastic Load Balancing. `http://aws.amazon.com/de/elasticloadbalancing/`.

[10] K. Amin, G. V. Laszewski, M. Hategan, N. J. Z. S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *Proceedings of 37th Hawai'i International Conference on System Science*, pages 5–8, 2004.

[11] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services Version 1.1*. Microsoft, IBM, Siebel, BEA und SAP, 1.1 edition, 2003.

[12] Apache Foundation. Apache Axis. `http://ws.apache.org/axis/`.

[13] Apache Software Foundation. Apache ODE. `http://ode.apache.org/`.

[14] AspectJ. `http://eclipse.org/aspectj/`.

[15] U. Aßmann. *Invasive Software Composition*. Springer-Verlag New York Inc, 2003.

[16] F. Baligand and V. Monfort. A Concrete Solution for Web Services Adaptability Using Policies and Aspects. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 134–142. ACM, 2004.

[17] A. Barker and J. V. Hemert. Scientific Workflow: A Survey and Research Directions. *Lecture Notes In Computer Science, Springer*, pages 746–753, 2008.

[18] W. Binder, I. Constantinescu, and B. Faltings. Service Invocation Triggers: A Lightweight Routing Infrastructure for Decentralized Workflow Orchestration. *International Conference on Advanced Information Networking and Applications*, 2:917–921, 2006.

[19] A. Black, J. Huang, and J. Walpole. Reifying Communication at the Application Level. In *Proceedings of the 2001 International Workshop on Multimedia Middleware*, pages 32–35. ACM, 2001.

[20] M. Blow, Y. Goland, M. Kloppmann, F. Leymann, G. Pfau, D. Roller, and M. Rowley. BPELJ: BPEL for Java. *Whitepaper, BEA and IBM*, 2004.

[21] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *J. of Parallel and Distributed Computing*, 61(6):810 – 837, 2001.

[22] M. L. Brodie and M. Stonebraker. *Migrating legacy systems: gateways, interfaces & the incremental approach*. Morgan Kaufmann Publishers Inc., 1995.

[23] C. Bussler. The Fractal Nature of Web Services. *Computer*, 40(3):93–95, 2007.

[24] G. Canfora, A. Fasolino, G. Frattolillo, and P. Tramontana. A wrapping approach for migrating legacy system interactive functionalities to Service Oriented Architectures. *Journal of Systems and Software*, 81(4):463–480, 2008.

[25] V. Cardellini and S. Iannucci. Designing a Broker for QoS-driven Runtime Adaptation of SOA Applications. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 504–511. IEEE Press, 2010.

[26] W. Cazzola. Remote method invocation as a first-class citizen. *Distributed Computing*, 16(4):287–306, 2003.

[27] K. S. M. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A Fault Taxonomy for Web Service Composition. In *Proceedings of the 3rd International Workshop on Engineering Service Oriented Applications (WESOA07), Springer LNCS*, 2007.

[28] A. Charfi and M. Mezini. Aspect-oriented Web Service Composition with AO4BPEL. In *Proceedings of the European Conference on Web Services*, pages 168–182. Springer-Verlag, 2004.

[29] M. R. Chernick. *Bootstrap Methods: A Guide for Practitioners and Researchers (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2 edition, 2007.

[30] L. Chung and J. do Prado Leite. On Non-Functional Requirements in Software Engineering. *Conceptual Modeling: Foundations and Applications*, 5(4):363–379, 2009.

[31] M. A. Cibrán and B. Verheecke. Dynamic Business Rules for Web Service Composition. In *2nd Dynamic Aspects Workshop (DAW05)*, pages 13–18, 2005.

[32] C. Courbis and A. Finkelstein. Towards an Aspect Weaving BPEL Engine. In *The Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Lancaster, UK*, pages 1–5, 2004.

[33] D. Cooper and S. Farrell and S. Boeyen and R. Housley and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. `http://tools.ietf.org/html/rfc5280`.

[34] T. Dalman, T. Dörnemann, E. Juhnke, M. Weitzel, M. Smith, W. Wiechert, K. Nöh, and B. Freisleben. Metabolic Flux Analysis in the Cloud. In *Proceedings of IEEE eScience 2010*, pages 57–64. IEEE Press, 2010.

[35] T. Dalman, T. Dörnemann, E. Juhnke, M. Weitzel, W. Wiechert, K. Nöh, and B. Freisleben. Cloud MapReduce for Monte Carlo Bootstrap Applied to Metabolic Flux Analysis. *Journal of Future Generation Computer Systems (FGCS)*, 29(2):582–590, 2013.

[36] T. Dalman, E. Juhnke, T. Dörnemann, M. Weitzel, K. Nöh, W. Wiechert, and B. Freisleben. Service Workflows and Distributed Computing Methods for ^{13}C Metabolic Flux Analysis. In *Proceedings of 7th EUROSIM Congress on Modelling and Simulation*, page (online), 2010.

[37] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.

[38] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In M. Dikaiakos, editor, *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer-Verlag, 2004.

[39] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk. GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing*, 3(1):75–90, 2005.

[40] M. Di Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. D. Nitto. WS Binder: a Framework to Enable Dynamic Binding of Composite Web Services. In *Proceedings of the 2006 International Workshop on Service-oriented Software Engineering*, pages 74–80. ACM, 2006.

[41] G. Dobson. Using WS-BPEL to Implement Software Fault Tolerance for Web Services. In *Software Engineering and Advanced Applications, 2006. SEAA'06. 32nd EUROMICRO Conference on*, pages 126–133. IEEE Press, 2006.

[42] M. Döhring, B. Zimmermann, and L. Karg. Flexible Workflows at Design- and Runtime Using BPMN2 Adaptation Patterns. In *Business Information Systems*, pages 25–36. Springer-Verlag, 2011.

[43] T. Dörnemann, T. Friese, S. Herdt, E. Juhnke, and B. Freisleben. Grid Workflow Modelling Using Grid-Specific BPEL Extensions. In *Proceedings of German e-Science Conference (GES)*, pages 1–8, 2007.

[44] T. Dörnemann, E. Juhnke, and B. Freisleben. On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '09)*, pages 140–147. IEEE Press, 2009.

[45] T. Dörnemann, E. Juhnke, T. Noll, D. Seiler, and B. Freisleben. Data Flow Driven Scheduling of BPEL Workflows Using Cloud Resources. In *Proceedings of 3rd IEEE International Conference on Cloud Computing (IEEE CLOUD)*, pages 196–203. IEEE Press, 2010.

[46] T. Dörnemann, M. Mathes, R. Schwarzkopf, E. Juhnke, and B. Freisleben. DAVO: A Domain-Adaptable, Visual BPEL4WS Orchestrator. In *Proceedings of the 23rd IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 121–128. IEEE Press, 2009.

[47] T. Dörnemann, M. Smith, and B. Freisleben. Composition and Execution of Secure Workflows in WSRF-Grids. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '08)*, pages 122–129. IEEE Press, 2008.

[48] T. Dörnemann, M. Smith, E. Juhnke, and B. Freisleben. Secure Grid Micro-Workflows Using Virtual Workspaces. In *Proceedings of 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 119–126. IEEE Press, 2008.

[49] ^{13}CFlux. https://www.13cflux.net.

[50] D. Ecklund, V. Goebel, T. Plagemann, and E. Ecklund Jr. Dynamic end-to-end QoS management middleware for distributed multimedia systems. *Multimedia Systems*, 8(5):431–442, 2002.

[51] Eclipse BPEL Team. Eclipse BPEL Project. http://www.eclipse.org/bpel.

[52] Eclipse Foundation. Eclipse Modeling Framework Project. http://www.eclipse.org/modeling/emf/.

[53] Eclipse.org. ATL Use Case – Model Driven Performance Engineering: From UML/SPT to AnyLogic. http://www.eclipse.org/m2m/atl/usecases/UML2AnyLogic/.

[54] Eclipse.org, Tobias Widmer, IBM Rational Research Lab Zurich. Unleashing the Power of Refactoring. http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html.

[55] B. Efron and R. Tibshirani. *An Introduction to the Bootstrap (Monographs on Statistics and Applied Probability)*. Chapman & Hall/CRC, 1 edition, 1994.

[56] V. S. W. Eide, F. Eliassen, O.-C. Granmo, and O. Lysne. Scalable Independent Multi-level Distribution in Multimedia Content Analysis. In *Proceedings of the Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems (IDMS/PROMS)*, pages 37–48. Springer-Verlag, 2002.

[57] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task scheduling in parallel and distributed systems*. Prentice Hall, 1994.

[58] T. Elrad, R. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[59] J. Elson and J. Howell. Handling Flash Crowds from your Garage. In *ATC'08: USENIX 2008 Annual Technical Conference*, pages 171–184. USENIX Association, 2008.

[60] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. Price. Grid Service Orchestration using the Business Process Execution Language. In *Journal of Grid Computing*, volume 3, pages 283–304. Springer-Verlag, 2005.

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

[62] T. Erl. *SOA: Principles of Service Design*. Prentice Hall Press Upper Saddle River, NJ, USA, 2007.

[63] R. Ewerth and B. Freisleben. Semi-Supervised Learning for Semantic Video Retrieval. In *Proceedings of the 6th ACM International Conference on Image and Video Retrieval*, pages 154–161. ACM, 2007.

[64] R. Ewerth, M. Mühling, and B. Freisleben. Self-Supervised Learning of Face Appearances in TV Casts and Movies. In *Proceedings of the Eighth IEEE International Symposium on Multimedia*, pages 78–85. IEEE Press, 2006.

[65] R. Ewerth, M. Mühling, T. Stadelmann, J. Gllavata, M. Grauer, and B. Freisleben. Videana: A Software Tool for Scientific Film Studies. *Digital Tools in Media Studies – Analysis and Research*, pages 145–160, 2007.

[66] O. Ezenwoye and S. M. Sadjadi. TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services. In *In Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST 2007)*, 2007.

[67] I. D. Falco, D. Maisto, U. Scafuri, E. Tarantino, and A. D. Cioppa. An Innovative Perspective on Mapping in Grids. In *BADS '09: Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, pages 27–36. ACM, 2009.

[68] N. Fallenbeck. *Virtual Machine Image Management for Elastic Resource Usage in Grid Computing*. PhD thesis, Philipps-Universität Marburg, 2011.

[69] M. Ferber, S. Hunold, and T. Rauber. Load Balancing Concurrent BPEL Processes by Dynamic Selection of Web Service Endpoints. *2009 International Conference on Parallel Processing Workshops*, pages 290–297, 2009.

[70] FFmpeg. `http://ffmpeg.org/`.

[71] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[72] J. L. R. Filho, P. C. Treleaven, and C. Alippi. Genetic-Algorithm Programming Environments. *Computer*, 27:28–43, June 1994.

[73] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag, 2006.

[74] I. Foster, T. Freeman, K. Keahy, D. Scheftner, B. Sotomayer, and X. Zhang. Virtual Clusters for Grid Communities. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 513–520. IEEE Press, 2006.

[75] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 2003.

[76] J. Frey. Condor DAGMan: Handling Inter-Job Dependencies. *University of Wisconsin, Dept. of Computer Science, Tech. Rep*, 2002.

[77] J. García-Fanjul, J. Tuya, and C. De La Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services using SPIN. In *International Workshop on Web Services–Modeling and Testing (WS-MaTe 2006)*, page 83, 2006.

[78] D. Gibbon and Z. Liu. Large Scale Content Analysis Engine. In *Proceedings of the First ACM Workshop on Large-scale Multimedia Retrieval and Mining*, pages 97–104. ACM, 2009.

[79] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the Challenges of Scientific Workflows. *Computer*, 40(12):24–32, 2007.

[80] T. Glatard, D. Emsellem, and J. Montagnat. Generic web service wrapper for efficient embedding of legacy codes in service-based workflows. In *Grid-Enabling Legacy Applications and Supporting End Users Workshop (GELA'06), Paris, France*, 2006.

[81] M. Glinz. On Non-Functional Requirements. *15th IEEE International Requirements Engineering Conference (RE 2007)*, pages 21–26, 2007.

[82] J. Gllavata, E. Qeli, and B. Freisleben. Detecting Text in Videos Using Fuzzy Clustering Ensembles. In *Proceedings of the 2006 IEEE International Symposium on Multimedia (ISM 06), San Diego, USA*, pages 283–290. IEEE Press, 2006.

[83] Globus Toolkit, Project Homepage. `http://www.globus.org/toolkit/`.

[84] I. Gorton, A. Wynne, Y. Liu, and J. Yin. Components in the Pipeline. *Software, IEEE*, 28(3):34–40, 2011.

[85] Grimoires, Project Homepage. `http://www.grimoires.org`.

[86] C. Griwodz and M. Zink. Dynamic Data Path Reconfiguration. In *Proceedings of the 2001 international workshop on Multimedia middleware*, pages 72–75. ACM, 2001.

[87] I. Habib, A. Anjum, R. Mcclatchey, and O. Rana. Adapting Scientific Workflow Structures Using Multi-Objective Optimization Strategies. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(1):4, 2013.

[88] M. Harbach, T. Dörnemann, E. Juhnke, and B. Freisleben. Semantic Validation of BPEL Fragment Compositions. In *Proceedings of the fourth IEEE International Conference on Semantic Computing (ICSC2010)*, pages 176–183. IEEE Press, 2010.

[89] Y. Hassoun, R. Johnson, and S. Counsell. Reusability, Open Implementation and Java's Dynamic Proxies. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, pages 3–6. Computer Science Press, Inc., 2003.

[90] M. Heidt, T. Dörnemann, K. Dörnemann, and B. Freisleben. Omnivore: Integration of Grid Meta-Scheduling and Peer-to-Peer Technologies. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '08)*, pages 316–323. IEEE Press, 2008.

[91] S. Heinzl, M. Mathes, T. Friese, M. Smith, and B. Freisleben. Flex-SwA: Flexible Exchange of Binary Data Based on SOAP Messages with Attachments. In *Proceedings of the IEEE International Conference on Web Services, Chicago, USA*, pages 3–10. IEEE Press, 2006.

[92] S. Heinzl, D. Seiler, E. Juhnke, and B. Freisleben. Exposing Validity Periods of Prices for Resource Consumption to Web Service Users via Temporal Policies. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pages 233–240. ACM and OCG, 2009.

[93] S. Heinzl, D. Seiler, E. Juhnke, T. Stadelmann, R. Ewerth, M. Grauer, and B. Freisleben. A Scalable Service-Oriented Architecture for Multimedia Analysis, Synthesis, and Consumption. *International Journal of Web and Grid Services*, 5(3):219–260, 2009.

[94] Hey!Watch – Video encoding web service. `http://heywatch.com/page/home`.

[95] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good. On the Use of Cloud Computing for Scientific Workflows. *2008 IEEE Fourth International Conference on eScience*, pages 640–645, 2008.

[96] Y. Huang, I. Taylor, D. Walker, and R. Davies. Wrapping Legacy Codes for Grid-Based Applications. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 7 pp.–, 2003.

[97] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, 2006.

[98] IBM. Business Process Manager. `http://www-01.ibm.com/software/integration/business-process-manager`.

[99] IBM. Web Services Flow Language, 2001.

[100] IBM developerworks, Brian Goetz. Java theory and practice: Decorating with dynamic proxies. `http://www.ibm.com/developerworks/java/library/j-jtp08305/index.html`.

[101] B. Ihle, S. Kirch, E. Juhnke, T. Dörnemann, D. Seiler, and B. Freisleben. A Workflow Management Platform for Media Analysis in BPEL-based Grid Environments. In *Central Europe (CEUR) Workshop Proceedings*, page (online), 2011.

[102] The Official Microsoft IIS Site. `http://www.iis.net/`.

[103] Intalio. Process Designer. `http://www.intalio.com/process-designer`.

[104] JBoss Community. jBPM. `http://www.jboss.org/jbpm`.

[105] jBoss Drools. `http://www.jboss.org/drools/`.

[106] Java Message Service. `https://jcp.org/en/jsr/detail?id=343`.

[107] Java Native Access (JNA). `https://jna.dev.java.net/`.

[108] Java Native Interface Specification. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html`, 2003.

[109] N. Joncheere, D. Deridder, R. Straeten, and V. Jonckers. A Framework for Advanced Modularization and Data Flow in Workflow Systems. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 592–598. Springer-Verlag, 2008.

[110] E. Juhnke, T. Dörnemann, and B. Freisleben. Fault-Tolerant BPEL Workflow Execution via Cloud-Aware Recovery Policies. In *Proceedings of 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 31–38. IEEE Press, 2009.

[111] E. Juhnke, T. Dörnemann, S. Kirch, D. Seiler, and B. Freisleben. SimpleBPEL: Simplified Modeling of BPEL Workflows for Scientific End Users. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 137–140. IEEE Press, 2010.

[112] E. Juhnke, T. Dörnemann, R. Schwarzkopf, and B. Freisleben. Security, Fault Tolerance and Modeling of Grid Workflows in BPEL4Grid. In *Proceedings of Software Engineering 2010, Grid Workflow Workshop (GWW-10)*, LNI, pages 193–200. GI, 2010.

[113] E. Juhnke, T. Dörnemann, D. Böck, and B. Freisleben. Multi-Objective Scheduling of BPEL Workflows in Geographically Distributed Clouds. In *Proceedings of the 4th IEEE International Conference on Cloud Computing (IEEE CLOUD)*, pages 412–419. IEEE Press, 2011.

[114] E. Juhnke, D. Seiler, R. Ewerth, M. Smith, and B. Freisleben. Request/Response Aspects for Web Services. In H. Mouratidis and C. Rolland, editors, *Advanced Information Systems Engineering*, volume 6741 of *Lecture Notes in Computer Science*, pages 627–641. Springer-Verlag, 2011.

[115] E. Juhnke, D. Seiler, T. Stadelmann, T. Dörnemann, and B. Freisleben. LCDL: An Extensible Framework for Wrapping Legacy Code. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, pages 648–652. ACM, 2009.

[116] D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. Buchmann. Extending BPEL for Run Time Adaptability. In *EDOC '05: Proceedings of the Ninth IEEE International EDOC Enterprise Computing Conference*, pages 15–26. IEEE Press, 2005.

[117] B. Kemp and J. Olivan. European data format 'plus' (EDF+), an EDF alike standard format for the exchange of physiological data. *Clinical neurophysiology*, 114(9):1755–1761, 2003.

[118] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[119] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, and I. Trickovic. WS-BPEL Extensions for People – BPEL4People. *Joint white paper, IBM and SAP*, 2005.

[120] J. D. Knowles and D. W. Corne. The Pareto Archive Evolution Strategy: A New Baseline Algorithm for Multi-Objective Optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 98–105. IEEE Press, 1999.

[121] Y. K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[122] G. Lee, N. Tolia, P. Ranganathan, and R. Katz. Topology-aware Resource Allocation for Data-Intensive Workloads. In *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems*, volume 41, pages 1–6. ACM, 2010.

[123] Y. Liu, I. Gorton, and A. Wynne. Architecture-Based Adaptivity Support for Service Oriented Scientific Workflows. In *SOSE*, pages 309–314, 2013.

[124] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[125] R.-Y. Ma, Y.-W. Wu, X.-X. Meng, S.-J. Liu, and L. Pan. Grid-Enabled Workflow Management System Based On BPEL. *International Journal of High Performance Computing Applications*, 22(3):238–249, 2008.

[126] A. Maier, B. Mitschang, F. Leymann, and D. Wolfson. On combining business process integration and etl technologies. In *BTW*, volume 5, pages 533–546, 2005.

[127] S. Majithia, M. Shields, I. Taylor, and I. Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *IEEE International Conference on Web Services, 2004. Proceedings*, pages 514–521, 2004.

[128] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 2–28. Springer-Verlag, 2003.

[129] M. Mathes, S. Heinzl, and B. Freisleben. Towards a Time-Constrained Web Service Infrastructure for Industrial Automation. In *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 846–853. IEEE Press, 2008.

[130] A. Mdhaffar, R. B. Halima, E. Juhnke, M. Jmaiel, and B. Freisleben. AOP4CSM: An Aspect-Oriented Programming Approach for Cloud Service Monitoring. In *Proceedings of the 11th IEEE International Conference on Computer and Information Technology*, pages 363–370. IEEE Press, 2011.

[131] MediaGrid. `http://www.mediagrid-community.de/`.

[132] Microsoft. BizTalk Server. `http://www.microsoft.com/biztalk/en/us/default.aspx`.

[133] Microsoft. Windows Workflow Foundation. `http://msdn.microsoft.com/en-us/netframework/aa663328`.

[134] Microsoft. XLANG – Web Services for Business Process Design, 2001.

[135] R. Mietzner and F. Leymann. Towards Provisioning the Cloud: On the Usage of Multi-Granularity Flows and Services to Realize a Unified Provisioning Infrastructure for SaaS Applications. In *Proceedings of IEEE Congress on Services - Part I*, pages 3–10, Los Alamitos, CA, USA, 2008. IEEE Press.

[136] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceedings of the 17th international conference on World Wide Web*, pages 815–824. ACM, 2008.

[137] MPEG-7 Overview. `http://mpeg.chiariglione.org/standards/mpeg-7/mpeg-7.htm`.

[138] H. Naguib and G. Coulouris. Towards automatically configurable multimedia applications. In *Proceedings of the 2001 international workshop on Multimedia middleware*, pages 28–31. ACM, 2001.

[139] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly Distributed AOP using AWED. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 51–62. ACM, 2006.

[140] J. Nesvadba, P. Fonseca, A. Sinitsyn, F. de Lange, M. Thijssen, P. van Kaam, H. Liu, R. van Leeuwen, J. Lukkien, A. Korostelev, J. Ypma, B. Kroon, H. Celik, A. Hanjalic, U. Naci, J. Benois-Pineau, P. de With, and J. Han. Real-Time and Distributed AV Content Analysis System for Consumer Electronics Networks. In *Proceedings of International Conference on Multimedia and Expo*, pages 1549–1552. IEEE Press, 2005.

[141] E. Newcomer and G. Lomow. *Understanding SOA with Web services*. Addison-Wesley, 2005.

[142] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 7–15. ACM, 2004.

[143] Noelios Technologies. Restlet – RESTful web framework for Java. `http://www.restlet.org`.

[144] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '09)*, pages 124–131. IEEE Press, 2009.

[145] OASIS. Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.2. `http://docs.oasis-open.org/ws-rx/wsrm/v1.2/wsrm.pdf`.

[146] OASIS. Web Services Resource Framework Specification. `http://www.oasis-open.org/specs/index.php#wsrfv1.2`, 2006.

[147] Open Compuer Vision Library, Project Homepage. http://opencv.willowgarage.com/wiki/.

[148] OpenNebula. http://opennebula.org/.

[149] Oracle. BPEL Process Manager. http://www.oracle.com/technetwork/middleware/bpel/overview/index.html.

[150] Oracle. Code Conventions for the Java Programming Language. http://www.oracle.com/technetwork/java/codeconv-138413.html.

[151] Oracle. Dynamic proxy classes. http://download.oracle.com/javase/1.3/docs/guide/reflection/proxy.html.

[152] Oracle. Java Remote Method Invocation – Distributed Computing for Java. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html.

[153] C. Ouyang, M. Dumas, A. ter Hofstede, and W. van der Aalst. From BPMN Process Models to BPEL Web Services. *Proceedings of the 4th International Conference on Web Services (ICWS), IEEE Computer Society, Chicago IL, USA*, pages 285–292, 2006.

[154] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

[155] PhysioNet. PhysioToolkit. http://www.physionet.org/physiotools/.

[156] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[157] R. Prodan and T. Fahringer. Dynamic Scheduling of Scientific Workflow Applications on the Grid: A Case Study. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 687–694. ACM, 2005.

[158] R. Prodan and T. Fahringer. *Grid Computing: Experiment Management, Tool Integration, and Scientific Workflows*. Springer-Verlag, 2007.

[159] R. Rose, E. Hofstetter, and D. Reynolds. Integrated Models of Signal and Background with Application to Speaker Identification in Noise. *IEEE Transactions on Speech and Audio Processing*, 2:245–258, 1994.

[160] Y. Rubner, C. Tomasi, and L. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision*, 40:99–121, 2000.

[161] SAP. Netweaver Process Integration. http://www.sap.com/platform/netweaver/components/pi/index.epx.

[162] J. Schad, J. Dittrich, and J. Quiane-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[163] H. Schneiderman and T. Kanade. Object Detection Using the Statistics of Parts. *International Journal of Computer Vision*, 56(3):151–177, 2004.

[164] C. Schridde, T. Dörnemann, E. Juhnke, M. Smith, and B. Freisleben. An Identity-Based Security Infrastructure for Cloud Environments. In *Proceedings of IEEE International Conference on Wireless Communications, Networking and Information Security (WCNIS2010)*, pages 644–649. IEEE Press, 2010.

[165] D. Seiler, S. Heinzl, E. Juhnke, R. Ewerth, M. Grauer, and B. Freisleben. Efficient Data Transmission in Service Workflows for Distributed Video Content Analysis. In *Proceedings of the 6th International Conference on Advances in Mobile Computing & Multimedia (MoMM2008)*, pages 7–14. ACM and OCG Book Series, 2008.

[166] D. Seiler, E. Juhnke, R. Ewerth, M. Grauer, and B. Freisleben. Efficient Data Transmission Between Multimedia Web Services via Aspect-Oriented Programming. In *ACM International Conference on Multimedia Systems*, pages 93–104. ACM, 2011.

[167] N. Seo. OpenCV haartraining (Rapid Object Detection With A Cascade of Boosted Classifiers Based on Haar-like Features). `http://note.sonots.com/SciSoftware/haartraining.html`.

[168] S. Shankar and D. J. DeWitt. Data Driven Workflow Planning in Cluster Management Systems. In *HPDC '07: Proceedings of the 16th International Symposium on High Performance Distributed Computing*, pages 127–136. ACM, 2007.

[169] A. F. Smeaton, P. Over, and W. Kraaij. Evaluation Campaigns and TRECVid. In *MIR '06: Proceedings of the 8th ACM International Workshop on Multimedia Information Retrieval*, pages 321–330. ACM, 2006.

[170] R. Smith. An Overview of the Tesseract OCR Engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2*, volume 2, pages 629–633. IEEE Press, Sept. 2007.

[171] H. Sneed. Integrating legacy Software into a Service oriented Architecture. *Conference On Software Maintenance*, 2006.

[172] M. Sonntag, K. Görlach, D. Karastoyanova, and N. Currle-Linde. Towards Simulation Workflows With BPEL: Deriving Missing Features From GriCoL. In *Proceedings of the 21st IASTED International Conference*, volume 696-704, page 341, 2010.

[173] M. Sonntag, D. Karastoyanova, and F. Leymann. The Missing Features of Workflow Systems for Scientific Computations. In *Software Engineering (Workshops)*, pages 209–216, 2010.

[174] T. Stadelmann, S. Heinzl, M. Unterberger, and B. Freisleben. Webvoice: A toolkit for perceptual insights into speech processing. In *Proceedings of 2nd International Conference on Image and Signal Processing CISP'09*, pages 1–5. IEEE Press, 2009.

[175] E. Stehle, B. Piles, J. Max-Sohmer, and K. Lynch. Migration of Legacy Software to Service Oriented Architecture. *Conference On Software Maintenance*, pages 1–5, 2008.

[176] A. Streule. Abstract Views on BPEL Processes. Master's thesis, Institute of Architecture of Application Systems, University of Stuttgart, Germany, 2009.

[177] S. Subramanian, P. Thiran, N. Narendra, G. Mostefaoui, and Z. Maamar. On the Enhancement of BPEL Engines for Self-Healing Composite Web Services. In *Proceedings of the International Symposium on Applications and the Internet, 2008. SAINT 2008.*, pages 33–39, 2008.

[178] D. Suvee, W. Vanderperren, and V. Jonckers. JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 21–29. ACM, 2003.

[179] É. Tanter. Expressive Scoping of Dynamically-Deployed Aspects. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, pages 168–179. ACM, 2008.

[180] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana Workflow Environment: Architecture and Applications. In *Workflows for e-Science*, pages 320–339. Springer-Verlag, 2007.

[181] THEME ICT-2007.8.2. ALLOW – Adaptable Pervasive Flows. `http://www.allow-project.eu/`.

[182] Transport Layer Security. `http://datatracker.ietf.org/wg/tls/charter/`.

[183] Typica, Project Homepage. `http://code.google.com/p/typica/`.

[184] J. Ullman. NP-Complete Scheduling Problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[185] Universität Stuttgart. SimTech. `http://www.simtech.uni-stuttgart.de/`.

[186] R. Van Engelen and K. Gallivan. The gSOAP Toolkit for SOAP Web Services and XML-Based Applications. In *ccgrid*, page 128. IEEE Press, 2002.

[187] C. Vecchiola, S. Pandey, and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications. *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 4–16, 2009.

[188] B. Verheecke, M. Cibran, W. Vanderperren, D. Suvee, and V. Jonckers. AOP for Dynamic Configuration and Management of Web Services. *International Journal of Web Services Research*, 1(3):25–41, 2004.

[189] B. Verheecke, W. Vanderperren, and V. Jonckers. Unraveling Crosscutting Concerns in Web Services Middleware. *Software, IEEE*, 23(1):42–50, 2006.

[190] P. Viola and M. J. Jones. Robust Real-Time Face Detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[191] W3C. XML Schema. `http://www.w3.org/XML/Schema`.

[192] L. Wang, H. J. Siegel, V. R. Roychowdhury, and A. A. Maciejewski. Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22, 1997.

[193] X. Wang, R. Buyya, and J. Su. Reliability-Oriented Genetic Algorithm for Workflow Applications Using Max-Min Strategy. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 108–115. IEEE Press, 2009.

[194] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel. *Workflows for e-Science*, chapter Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling, pages 428–449. Springer-Verlag, 2007.

[195] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall, 2005.

[196] W. Wiechert. ^{13}C Metabolic Flux Analysis. *Metababolic Engineering*, 3(3):195–206, 2001.

[197] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of Scientific Workflows in the ASKALON Grid Environment. *ACM SIGMOD Record*, 34(3):56–62, 2005.

[198] World Wide Web Consortium (W3C). SOAP Messages with Attachments. `http://www.w3.org/TR/SOAP-attachments`.

[199] World Wide Web Consortium (W3C). W3C SOAP Specification. `http://www.w3.org/TR/soap/`.

[200] World Wide Web Consortium (W3C). Web Services Definition Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`.

[201] World Wide Web Consortium (W3C). Web Services Policy 1.5 – Framework. `http://www.w3.org/TR/ws-policy/`.

[202] World Wide Web Consortium (W3C). XML Path Language (XPath), Version 1.0. `http://www.w3.org/TR/xpath`.

[203] Web Service Specifications. `http://en.wikipedia.org/wiki/List_of_web_service_specifications`.

[204] ws-universe.com. http://www.ws-universe.com/index.html.

[205] F. Xhafa and A. Abraham. Meta-heuristics for Grid Scheduling Problems. *Metaheuristics for Scheduling: Distributed Computing Environments, Studies in Computational Intelligence*, pages 1–37, 2008.

[206] J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34(3):44–49, 2005.

[207] J. Yu and R. Buyya. Scheduling Scientific Workflow Applications with Deadline and Budget Constraints using Genetic Algorithms. *Scientific Programming, IOS Press*, 14(3):217–230, 2006.

[208] J. Yu and R. Buyya. Gridbus Workflow Enactment Engine. *Grid Computing: Infrastructure, Service, and Applications*, page 119, 2009.

[209] J. Yu, M. Kirley, and R. Buyya. Multi-objective Planning for Workflow Execution on Grids. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 10–17. IEEE Press, 2007.

[210] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *Proceedings of the IEEE Congress on Services, 2007*, pages 199–206. IEEE Press, 2007.

[211] Y. Zhao, Y. Li, W. Tian, and R. Xue. Scientific-Workflow-Management-as-a-Service in the Cloud. In *Proceedings of the Second International Conference on Cloud and Green Computing (CGC), 2012*, pages 97–104. IEEE Press, 2012.

[212] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, ETH Zurich, May 2001.

[213] E. Zitzler and L. Thiele. An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach. Technical Report 43, ETH Zurich, May 1998.

[214] Y. Zou and K. Kontogiannis. Web-Based Specification and Integration of Legacy Services. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2000.