

AN UNDERLAY SYSTEM FOR DYNAMIC AND DISTRIBUTABLE WEB APPLICATIONS

vorgelegt von
Dipl.-Inform. Heiko Pfeffer

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Berichter: Prof. Dr. Prof. e.h. Dr. h.c. Radu Popescu-Zeletin
Prof. Dr. Hans-Ulrich Heiß
Prof. Dr. Andreas Polze

Vorsitzender: Prof. Dr. Odej Kao

Tag der wissenschaftlichen Aussprache: 18. März 2010

Berlin 2010
D 83

ABSTRACT

The Web has continuously developed towards a sophisticated platform for user-centric applications characterized by a rich graphical presentation and a high degree of user interactivity. This trend has been fueled by the increasing importance and heterogeneity of mobile user devices such as smartphones, PDAs and netbooks. In this case, the Web browser constitutes the unified platform that provides a high degree of independence from the underlying hardware and operating systems.

Mashups are composed Web applications that combine multiple services stemming from different 3rd party providers into a custom Web application. Web mashups can thus be considered as the Web's *service compositions*. However, such mashups are created by software developers by gluing together open APIs via scripting languages, so that Web mashups expose considerable shortcomings with regard to readability and reusability of code, dynamic integration of services, and methods for a user-centric creation beyond explicit programming techniques.

In this thesis, an *underlay system* for Web applications is introduced that features a resource-oriented view on services and data and thereby combines the advantages of classic service composition languages with the rich graphical presentation and user-centric nature of Web mashups. Here, the underlay system abstracts from concrete services and thereby can be dynamically deployed on different sets of heterogeneous devices.

The underlay system possesses two key properties that will enhance dynamicity within today's mashups. First, it can be dynamically distributed, so that each involved device is in control of its local services. This decoupling of devices increases the mashup's offline capabilities and provides a means for a private processing of user data. Second, the underlay system can be automatically created by common end-users, so that underlay systems can be employed for the rapid creation of Web mashups on top of end-user devices in a spontaneous manner.

The work on the underlay system for Web applications was a joint project of the TU Berlin and Fraunhofer FOKUS. The results have been employed within the scope of multiple national and international research projects funded by the BMBF (UST+), the European Commission (BIONETS), and industry partners (uMASH, LCAI). Parts of the work have also been patented.

ZUSAMMENFASSUNG

Das Web hat sich in den letzten Jahren zu einer Plattform für Applikationen entwickelt, die sich durch einen besonders hohen Grad an Benutzerinteraktivität und Medienreichtum auszeichnen. Die Bedeutung des Webs hat besonders durch die zunehmende Anzahl mobiler Endgeräte mit stetig steigender Leistungsfähigkeit wie z.B. Smartphones oder Netbooks zugenommen, für die der Web Browser einen einheitlichen Zugang zu Applikationen unabhängig von ihrer Hardwareausstattung und ihrem Betriebssystem darstellt.

Mashups sind Webapplikationen, die sich aus mehreren von Drittanbietern bereitgestellten Diensten zusammensetzen, so dass Mashups als *Dienstkompositionen* innerhalb des Webs verstanden werden können. Solche Mashups werden jedoch von Softwareentwicklern erzeugt, indem offene APIs anhand von Skriptsprachen miteinander verbunden werden. Diese statische Kombination von Diensten und die Verwendung von Skriptsprachen führt zu erheblichen Nachteilen von Mashups in Bezug auf die Lesbarkeit und Wiederverwendbarkeit des Programmcodes, der dynamischen Integration von Diensten und der Erzeugbarkeit solcher Applikationen durch Endbenutzer abseits von klassischen Programmiersprachen.

Im Rahmen dieser Arbeit wird ein *Underlay System* für Webapplikationen eingeführt, das sich durch seine ressourcenorientierte Sicht auf Dienste und Daten auszeichnet und dadurch die Vorteile von klassischen Dienstkompositionsansätzen mit denen von benutzer- und medienorientierten Web Mashups verbindet. Das Underlay System abstrahiert dazu von konkreten Diensten und ermöglicht so die Instantiierung von Mashups auf verschiedenen Gruppen unterschiedlicher Geräte.

Darüber hinaus besitzt das Underlay System zwei zentrale Eigenschaften. Zum einen kann es automatisch auf mehrere Endgeräte verteilt werden, so dass jedes Gerät genau diejenigen Dienste kontrolliert, die es lokal ausführt. Diese Entkopplung von Geräten erhöht die Fähigkeit von Mashups, auch ohne kontinuierliche Verbindung zum Web partiell ausgeführt zu werden und eröffnet die Möglichkeit, sensible Daten nur auf vordefinierten Benutzergeräten zu halten, ohne sie über das Netzwerk zu übermitteln. Zum zweiten kann das Underlay System über eine spezielle Anfrage von Endbenutzern automatisch erstellt werden, so dass Benutzer Web Mashups spontan erzeugen und verwenden können.

Die vorliegende Arbeit zur Thematik von Underlay Systems stellt das Ergebnis von Forschungsarbeiten sowohl im Rahmen der TU Berlin als auch des Fraunhofer Instituts FOKUS dar. Die Resultate wurden im Kontext von mehreren nationalen und internationalen Projekten, gefördert durch das BMBF (UST+), die Europäische Kommission (BIONETS) sowie Industriepartner (uMASH, LCAI), erarbeitet und verwendet. Teile der Arbeit wurden patentiert.

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Prof. Dr. Radu Popescu-Zeletin, who gave me the opportunity to work at both the TU Berlin and the Fraunhofer Institut FOKUS and thus let me experience an environment that gave me room for fundamental as well as applied research. I'm grateful for his continual openness, patience, and guidance during the past years.

I'm deeply thankful to all my colleagues at TU Berlin, Fraunhofer FOKUS, and those who worked with me on the BIONETS project over the last years. Their comments and criticisms during many fruitful discussions were of tremendous importance and help.

I would like to thank all the students who have graduated in the field of composability and dynamic Web applications. Special thanks go out to Louay Bassbous, Piotr Wrona, and Tobias Heger for their veritably outstanding work.

I would not have been able to complete my work without the continuous support, encouragement, and warmth of my family, Antje and Jule. I'm grateful for their ability to cheer me up and to refill my strength by simply being there.

CONTENTS

Abstract	iii
Zusammenfassung	v
Acknowledgments	vii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Contribution	3
1.4 Structure	3
2 Composability	5
2.1 The Principle of Composability	5
2.2 Mashups: Composed Applications within the Web	8
2.2.1 The Architectural Style of the Web	8
2.2.2 Towards Composed Web Applications	9
2.2.3 Mashup Roles	12
2.2.4 Mashup Technologies	12
2.2.5 Mashup Styles	17
2.2.6 Drawbacks of Mashups	21
2.3 Business Processes: Composed Web Services	23
2.3.1 Service Oriented Architectures: The Basis for Business Processes	23
2.3.2 Service Composition Languages: Programming in the Large	37
2.3.3 Orchestration Languages	38
2.3.4 Choreography Languages	42
2.4 Distinguishing Mashups and Business Processes	44
3 Towards Dynamic and Distributable Web Applications	47
3.1 Resource Integration and Orientation	47
3.1.1 Motivation	47
3.1.2 Discussion of Related Work	47

3.1.3	Novel Idea	50
3.2	Enabling Dynamic Creation of Service Compositions	51
3.2.1	Motivation	51
3.2.2	Discussion of Related Work	52
3.2.3	Novel Idea	61
3.3	Distributed Web Mashups	62
3.3.1	Motivation	62
3.3.2	Discussion of Related Work	62
3.3.3	Novel Idea	72
3.4	An Architecture for Dynamic Web Mashups	72
3.4.1	An Underlay System for Web Mashups: Requirements	72
3.4.2	Architectural Embedding of Underlay Systems	73
4	An Underlay System for Dynamic Web Mashups	77
4.1	Central Components of the Mashup Underlay System	77
4.2	Service Model and Basic Concepts	78
4.3	Service Composition Model	80
4.3.1	Workflow Graph	80
4.3.2	Dataflow Graph	84
4.3.3	Workflow and Dataflow Graph Interworking	84
4.3.4	From Sequential to Parallel Execution	86
4.3.5	Modelling Timeouts	90
4.3.6	Resource Orientation	92
4.4	Communication among multiple Workflows	94
4.4.1	Synchronization Channels	95
4.4.2	Broadcast Channels	96
4.5	Underlay Systems	100
5	Automatic Creation of Underlay Systems	103
5.1	Towards Creation Methods for Underlay Systems	103
5.2	Modeling Service Descriptions	105
5.3	An Algorithm for the Automatic Creation of Service Compositions	109
5.3.1	Request Definitions	110
5.3.2	Service Discovery	110
5.3.3	Composition Nodes as Partial Compositions	112
5.3.4	General Overview of the Algorithm	113
5.3.5	Inserting Operations	115
5.3.6	Updating Statements and Parameters	116
5.3.7	The Challenge of Cycles and Layers	117
5.3.8	Enabling Runtime Decision Making for Varying Effects	121

5.3.9	From Composition Nodes to Underlay Systems	122
5.4	Adapting the Composition's NodePool: Heuristics	124
5.5	Evaluation and Validation	126
5.5.1	Simulation Environment	127
5.5.2	Enhancing the Algorithm's Success Rate	128
5.5.3	Conclusion and Discussion	137
6	Distributing Underlay Systems	139
6.1	Motivation	139
6.2	State Management and Presentation Mapping	141
6.3	Partitioning Algorithm	143
6.3.1	General Terminology	143
6.3.2	Conceptual Process of the Partitioning Algorithm	145
6.3.3	Creating Workflow Partitions	149
6.3.4	Partitioning Parallel Execution Blocks	157
6.3.5	Dealing with Nondeterministic Choices	160
6.3.6	Dataflow Distribution Algorithm	162
6.3.7	Evaluation of Invariance	163
7	Specification	165
7.1	Architectural Embedding	165
7.2	Component Specification	167
7.2.1	Component Overview	167
7.2.2	Data Elements and Parameters	169
7.2.3	Request Processor	171
7.2.4	Markup Validator	173
7.2.5	Underlay Creation Engine	173
7.2.6	Late Binding Engine	173
7.2.7	Service Repository	175
7.2.8	QoS Evaluator and User Preferences Evaluator	177
7.2.9	Call Generator	177
7.2.10	Partitioning Engine	178
7.2.11	Underlay Execution Engine	178
7.2.12	View Controller	179
7.3	The Orchestration Synchronization Protocol (OSP)	180
7.3.1	General Overview	180
7.3.2	Negotiation Phase	182
7.3.3	Execution Phase	185
7.3.4	Communication and Synchronization	189
7.3.5	Recovery: Replacing Failed Services	195

8	Creating Dynamic and Distributable Web Mashups	197
8.1	Dynamic Integration of Resources	197
8.2	Executing Underlay Systems	201
8.2.1	The Workflow Model	201
8.2.2	The Underlay Execution Engine	208
8.3	Creating Underlay Systems	216
8.3.1	Semantic Description Specification and Discovery . . .	216
8.3.2	Composition Module	219
8.4	Application Scenarios	220
8.4.1	A Web Client for Effect Driven Mashup Creation . . .	221
8.4.2	sendAround: Incorporating Telecommunication Services	222
8.4.3	ScatterPoker: A Distributed Interactive Community Web Application	224
9	Conclusion and Future Prospect	227
9.1	Conclusion	227
9.2	Future Work	228
9.2.1	Establishment of the MashWeb JavaScript API	228
9.2.2	Client to Client Communication	228
9.2.3	Integration with sophisticated Policy Mechanisms . . .	229
9.2.4	Services Accessing Multiple Resources	229
9.2.5	Measuring the Web's Service Domain	229
9.2.6	Offline Capabilities through Caching	230
9.2.7	Instantiation of the Underlay System within the Do- main of SOA	230
A	A Resource-oriented Markup Language for Web Mashups	233
A.1	Motivation and Objectives of the Mashup Language	233
A.2	Language Specification	234
A.2.1	Integrating Hybrid Content	236
A.2.2	From Resource Integration to Service Composition . .	237
A.2.3	Dynamic Replacement of Services	238
A.3	Mashup Markup Validation	239
B	Resource Definitions	245
	Bibliography	273
	List of Figures	291
	List of Tables	295

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

During the last couple of years, the Web has emerged as a sophisticated platform for user-centric, highly interactive applications. This development has been fueled by both technological and social factors. While technologies like Ajax have revolutionized the responsiveness and interactivity of Web applications by enabling asynchronous client-server communications that virtually break the limitations of the Web's sequential request/response pattern, the increasing willingness of end-users to contribute data and information to social platforms within the Web has led to more personalized, user-centric, and presentation-enriched Web applications.

Mashups are composed Web applications that integrate data and services from multiple sources to create custom Web applications. These kind of Web applications have recently gathered considerable interest, since they enable the rapid creation of value-added applications without needing to develop the larger parts of the application logic itself.

Mashups also have a couple of serious limitations. First, they are created by combining dataflow between services via client and server side scripting languages. These languages are not designed to express combinations of and dataflow among services, but are rather tailored for specific concerns such as DOM scripting. Thereby, the code of mashups tends to be unclear and fails to provide a high degree of reusability. In addition, services cannot be integrated dynamically, since they are incorporated via concrete APIs. An abstraction layer, which supports the integration of a service on a conceptual basis, so they can be replaced by a matching service during deployment or runtime, is not available. Including multiple services that provide the same functionality in order to make an application responsive to user preferences or potential failures of single services thus requires an additional amount of effort, which corresponds in a linear fashion to the number of added services.

Above all, mashups are only capable of accessing server-sided Web services exposed by 3rd party providers via open APIs. Mashups cannot integrate services residing on common user devices and thereby haven't kept

up with the development of mobile computing devices in the user's everyday life. Today's mobile devices not only have advanced computing capabilities, but also feature hardware and related services not available on classic workstations. Integrated photo and video cameras, GPS devices for location retrieval, advanced acceleration sensors, and legacy telecommunication services for setting up a call or sending image-enriched messages are only a few examples of sophisticated services offered by modern mobile devices, which also make them important for service provisioning.

In addition, mashups cannot be executed in an offline mode, but rely on a continuous connection to the Web. Existing approaches to developing offline capabilities focus on caching online content to make it accessible during times of disconnection. However, there are no existing methods that overcome the necessity of calling on a remote Web service to perform a certain functionality since local services are not considered.

Although the exposure of open APIs is already the largest parts of application logic, there is still no method for common end-users to request the automatic creation of mashups.

1.2 OBJECTIVES

This thesis aims to define an underlay system that paves the way towards more dynamic and distributable Web applications. The proposed underlay system enables the paradigm of "programming in the large" for composed Web applications while retaining its capabilities with regard to graphical presentation and support of user-interactions.

There are multiple ways of creating an underlay system. Within the scope of this thesis, a method is highlighted that supports the automatic creation of underlay systems based on the set of effects that should be generated by the mashup. The algorithm leads to the creation of an underlay system that abstracts implementations from the actual service and then integrates services based on their functionality. Thereby, the same underlay system can be dynamically deployed on different sets of heterogeneous user devices, where the services are allocated on the various devices based on additional knowledge such as user preferences. Finally, the underlay system can be partitioned into multiple sub-underlay systems, so that every device is in control of its local services. The respective partitioning algorithm extends the single sub-underlay in such a way that their coherent distributed execution is ensured by a synchronization mechanism, which controls the execution order and data consistency within the single partitions of the underlay system. This distribution of underlay systems decouples the dependencies between

the single devices, so that a user's device can execute its local services in an offline mode as long as no communication is required with services running on remote devices.

1.3 CONTRIBUTION

The contribution of this thesis to the current state of research is threefold. First, the definition of an underlay system for Web mashups is novel with regard to its ability to combine a structured execution with presentation of services. The core of the underlay system is built on a workflow based services composition. Although the representation of service compositions by means of workflow graphs is not new in itself, the formal modeling of workflows by means of automata theory proposed in this thesis abstracts from language specific concerns and thus provides a basis for the instantiation of the found results within different application domains. Moreover, the proposed model provides a resource oriented view on services and data and thus allows a mapping from the current state of a service composition to a respective graphical presentation in form of a Web mashup.

Second, a user-centric algorithm for the automatic creation of underlay systems is presented. Here, the novelty lies within the definition of an adaptive algorithm that supports the consideration of a time boundary given by a user, which denotes the maximum amount of time the user is willing to wait for the creation of the Web application. During the creation process, the algorithm adapts to the remaining time and thereby measurably increases the chances of finding an appropriate underlay system within the given time span.

Third, the partitioning algorithm for underlay systems denotes a novel way to create sub-underlay systems that contain automatically generated synchronization messages. A respective communication protocol, referred to as the Orchestration Synchronization Protocol (OSP), ensures the coherent execution of a distributed underlay system –and thus mashup– independently of the allocation of services to devices.

1.4 STRUCTURE

The thesis is structured as follows. Chapter 2 introduces the concept of composability. Special emphasis is put on the discussion of the concept's realization within the domains of the Web and Service Oriented Architectures (SOA). Chapter 3 discusses the different, already existing approaches to the enhancement of dynamicity and distribution of Web mashups. It is concluded

with the proposition of an underlay system and its architectural embedding, which promises to overcome the previously identified shortcomings.

The relation between the main chapters is depicted in Figure 1.1. Chapter 4 introduces the underlay system for Web mashups, which builds upon a formal model in order to describe service compositions, and provides a resource-oriented view on services in order to map an underlay system's state onto a graphical presentation. In addition, the underlay system is defined in such a way so that it complies with two key requirements. First, the underlay system is automatically producible based on a given user request; an algorithm for this automated creation of underlay systems and thus Web mashups is subsequently discussed in Chapter 5. Second, the underlay system is distributable, so that services residing on different heterogeneous devices can be dynamically integrated and controlled by partial underlay systems executed on the respective devices. An algorithm for the distribution of underlay systems is given in Chapter 6.

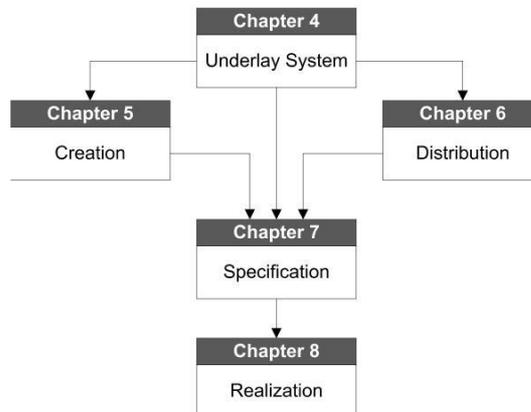


Figure 1.1: Relation between the thesis's main chapters.

Chapter 7 deals with the specification of the single components required for the realization of the underlay system, as well as its respective communication primitives. Special emphasis is placed on the specification of a protocol for the communication between parts of a distributed underlay system in order to ensure its coherent execution. Chapter 8 grounds the specification given in the previous chapter in the Web domain, in order to demonstrate the application of underlay systems to the control and representation of Web mashups.

CHAPTER 2

COMPOSABILITY

This chapter deals with state of the art of software and service composability. Therefore, the notion of composability is briefly introduced in section 2.1; definitions of the most important terms in this thesis are also provided. Section 2.2 then discusses the application and realization of composability within the Web, while section 2.3 focuses on the importance and role of service composability within business processes. Concludingly, section 2.4 discusses the differences between present realizations of composability with regard to Web applications and business processes.

2.1 THE PRINCIPLE OF COMPOSABILITY

Composability is a system design principle that deals with the integration of and relations among multiple software components [154]. Systems featuring a high degree of composability operate on a (commonly large) set of combinable software components, which can be aggregated, selected, and combined in order to meet a given system requirement. Combinable software components are characterized by being self-contained and stateless. Here, self-contained means that the deployment of a single component is independent from the other components, which makes the components easily replaceable. Statelessness means that each request is handled as an independent transaction. Thus, within the scope of a classic request/response pattern, where a transaction consists of a request and a corresponding response, there is no fixed relation between multiple requests, i.e. every request can be processed without any knowledge of previous requests or responses. Of course, statelessness is not a mandatory requirement for software components to be composable. However, the composition of stateful software components is much more difficult, since it requires the management of intermediate states for every involved software component.

Bergmans et al. Provide the very general definition for composability followed within the scope of this thesis [8]. The authors identify two major factors that determine the composability of components, namely *concerns* and a *composition scheme*. Here, the authors use the term concern to describe

a certain functionality, while abstracting from the realization by means of a software component or a service. A composition scheme defines the way concerns are composed. Therefore, it is possible that concerns can be composed on a logical level, while the applied composition scheme does not support their actual composability.

Two given concerns χ_1 and χ_2 may be composed to a concern χ_3 by means of the composition scheme \otimes , such that $\chi_3 = \chi_1 \otimes \chi_2$. The composition scheme can therefore be considered as a function \otimes operating on concerns, i.e. $\otimes : \chi \times \chi \rightarrow \chi$. Within the remainder of this section, concerns are identified with software components or services, while a general definition of service compositions instantiates the notion of composability.

Within the scope of this thesis, a *service* is considered as a function that is well-defined, self-contained, and does not depend on the context or state of other services. In addition, it is the endpoint of a connection and has some type of underlying computer system that supports the connection. Services may either be *concrete services*, i.e. implementations of a certain functionality, or *abstract services*, which describe a service's functionality on a meta-level. These abstract services serve as placeholders and can be substituted during multiple phases of a service's life-cycle with concrete services.

The term *service composition* is used rather loosely to denote a set of services combined by a certain set of control structures. Here, it is abstracted from the type of control structures and their capabilities. Analogously to single services, a service composition may either be abstract or not, i.e. may consist of a set of abstract services or a set of concrete services.

Figure 2.1 illustrates the relationship between abstract and concrete services and the services compositions they can build. Thus, a service composition may either consist of concrete services as shown in Figure 2.1 (a), or of abstract services as depicted in Figure 2.1 (b). In the latter case, the resulting abstract service composition is not executable, since it only encompasses meta-data on the general functionality of services, instead of real services itself. Therefore, an abstract service composition may be instantiated by one or more concrete service compositions by matching every abstract service against a concrete one. Since this second step does not necessarily have to be completed during the design time of service composition, but can be performed shortly before the deployment of the composition, this technique is referred to as *late binding*. A concrete service composition is also referred to as an *instance* of an abstract service composition.

A *service request* is considered as a description of a desired behavior that can potentially be met. In the case that a service request cannot be fulfilled by a single service, a composition of services may be capable of providing the requested functionality. Since a service request resembles the description of

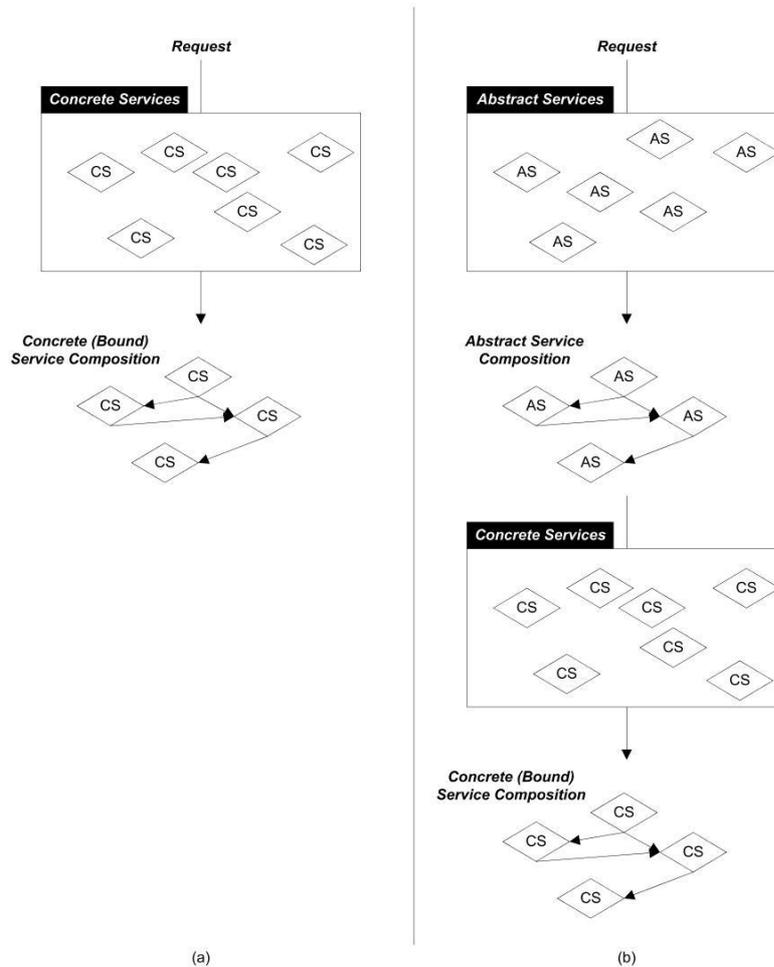


Figure 2.1: Late binding: The difference between abstract services (AS) and concrete services (CS).

a job that must be performed by some application logic, service requests are sometimes referred to as *tasks*.

Within the following sections, the role and influence of service composability is discussed for the Web and the Service Oriented Architecture (SOA) domain.

2.2 MASHUPS: COMPOSED APPLICATIONS WITHIN THE WEB

2.2.1 The Architectural Style of the Web

REpresentational State Transfer (REST) is an architecture style for distributed systems in general. The REST style was initially described by Fielding in 2000 [62], where he developed REST as an architectural style for hypermedia systems such as the World Wide Web (WWW). REST is a resource-oriented architectural style, where every object is considered a resource. REST is based upon five principles. *Addressability* allows every resource to be uniquely identified via an ID, while the property of *well-defined operations* implies the usage of standard operations to access and modify the available resources via a resource identifier. Resources possess multiple *representations* that represent the resource's current state, while representations can contain *links* that connect resources. *Statelessness* denotes communication that is stateless, i.e. every request is independent from the resources state and prior requests.

There are two roles housed within the Web [87]. A *provider* hosts information, i.e. resources, which are accessed by a *requestor*¹. Communication between two instances of these roles complies to the request/response pattern, i.e. every request is answered with a single response as illustrated in Figure 2.2.

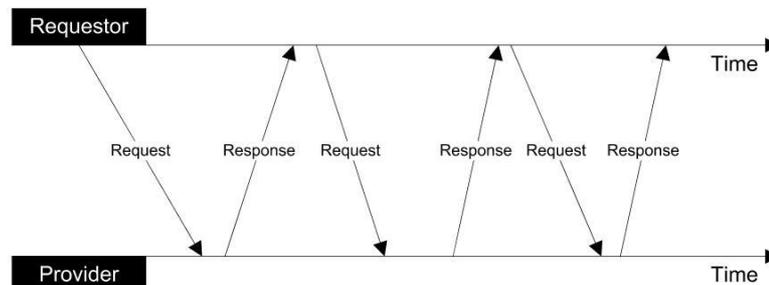


Figure 2.2: Request/response pattern as communication principle for provider and requestor.

The most prominent implementation of the REST architectural style is the Web itself, where the Hypertext Transport Protocol (HTTP)[61] is used for communication while Unified Resource Identifiers (URI) [21] are used for addressing resources. Resource manipulations are realized with

¹Requestors are sometimes also referred to as *consumer* in order to match the terminology with regard to the provider. Against a more physical background, these two roles are sometimes called *client* and *server*, respectively.

the HTTP methods, also called HTTP-verbs. HTTP-GET and HTTP-POST are widely known, but REST also makes use of HTTP-PUT and HTTP-DELETE for resource manipulation. HTTP-GET is used to request a resource, HTTP-POST to create one, HTTP-PUT for updates and HTTP-DELETE to destroy a resource [63]. An application therefore only needs the URI and the method to interact with a remote server that holds the information it uses [33]. Interfaces that follow the REST concept are referred to as *RESTful* interfaces. Several advantages over other communication protocols result from this simple approach: REST is independent from programming languages, it usually provides better response times compared to other protocols due to the support of caching, it avoids huge overheads due to its simplicity and HTTP authentication mechanisms, and allows the use of HTTPS connections. Moreover, server scaling is easy because common scaling technologies can be used. As representations of resources can hold links to other resources, additional discovery mechanisms are unnecessary [157].

Nevertheless, many of the so-called *RESTful* APIs in the Web follow the concept only partially. A more precise term would often be XML (or others) over HTTP. The HTTP-GET method to request data is used most often, while HTTP-PUT, HTTP-DELETE and HTTP-POST are not supported. Other APIs use HTTP for communication purposes, but do not deliberately use its methods [214].

2.2.2 Towards Composed Web Applications

A definition of mashups that can be considered as the least common denominator of today's definitions is that mashups are composed Web applications that integrate and combine data stemming from multiple service providers at runtime to create custom, value-added Web applications [90]. Ort, Brydon and Basler use this definition in their articles [135], [136]. Merrill [115] adds that mashups are usually "unusual or innovative compositions" that are "entirely new and innovative services" and "made for human (rather than computerized) consumption". Here, Merrill puts special emphasis on a mashup's ability to be spread across the Web and integrate data that lies outside of its organizational boundaries. The Gartner Research group refers to mashups as "a lightweight tactical integration of multi-sourced applications or content into a single offering" [143]. Xuanzhe et al. define Mashups as "ad hoc composition technology of Web applications that allows users to draw upon content retrieved from external data sources to create entirely new services" [211, p. 1].

The term mashup itself originates from the music domain. Here, mashups are

two or more entirely different tracks that are remixed into a new recording. The first well-known example is the Grey Album, a remix of The Beatles' White Album and Jay-Z's Black Album [211, p. 2]. In the web domain, mashups aggregate third party data in order to create new applications. This is achieved by the combination of several known technologies discussed later in Section 2.2.4. Eventually, web pages are organized in a more componentized manner, where some business logic can be implemented in the browser [211, p. 2]. The core features of mashups can be summarized as follows [211, p. 4]:

- Mashups are Web-based, i.e. JavaScript and Web services from other sites are leveraged and JSON or other data formats are used for data retrieval and remixing.
- Mashups are lightweight. They integrate data from publicly available sources (Application Programming Interfaces (API), Web services, Web sites, web applications, and others) and are built with as little code as possible.
- Mashups are user-centric, i.e. they are supposed to support easy programming for end-users.
- Mashups are more reusable compared to common SOA composition technologies because each building block is a combination of data, process and graphical representation.

A mashup therefore is a relatively new and lightweight kind of web application. Best-known are consumer mashups which are based on Web technologies, consumer-centric and strongly linked to the Web 2.0 paradigm defined by Tim O'Reilly. He identified seven core competencies that define Web 2.0 companies in 2007, among them cost-effective scalability, and "lightweight user interfaces, development models and business models" that allow for loosely coupled services [133, pp. 18 - 33]. Meanwhile, the RIA News Desk [85] already builds the bridge to service-oriented computing in 2006:

"Yet not only is Web 2.0 still very misunderstood, it's actually part of an even larger way of thinking about software in a fully service-oriented manner. This includes building composite applications, remixing data, building ad hoc supply chains, harnessing user involvement, aggregating knowledge, and more. Web 2.0 is becoming embodied in best practice sets such as service-oriented architecture (SOA)."

While consumer mashups get most of the attention, naturally businesses try to utilize Mashup advantages as well. A second category, the enterprise

mashups (also known as data mashups) cover a wider range of functionality. Consumer mashups focus on web-technologies (see 2.2.4), enterprise mashups, however, mash-up a broader range of data including databases, text-files, CSV data (Character/Comma Separated Values) and other data formats such as PDF, Excel and others [130].

In their attempt to extend SOA with mashups, Jin and Lee describe the relationship between SOA-style web services and mashups as “essentially a service composition style from SOA perspective” [90]. Xuanzhe et. al. point out that mashups provide an easy and flexible way for the composition of services compared to traditional web service composition technologies (e.g. BPEL and WSCI) used for corporate service composition. The increasing availability of APIs and web services that allow for remote invocation of services using standard Web protocols, mostly HTTP as communication protocol and XML for data delivery, led to an increase of web applications that utilize these possibilities [90].

Reasons for the quick spread of mashups are numerous. Merrill believes that “their popularity stems from the the emphasis on interactive user participation and the [...] manner in which they aggregate and stitch together third-party data” [115]. Easy and quick implementation and extensive data availability are reasons for developers to use mashups instead of proprietary solutions. The development of mashups can be directly linked to the importance of data for the future of the Web that O’Reilly identified [133, p. 27]. Developers can use a huge amount of data and sophisticated, already implemented and tested source code to experiment and create new web applications with much less effort as earlier. The sophisticated computations that may be necessary can be retrieved and executed externally, letting the developer concentrate on core features of an application. In business applications, service composition based on relatively complex WS-* standards can be observed. In the Web, however, “complex standards can get in the way, reduce interoperability, and stifle connectivity. [...] This has lead to simpler services such as REST and RSS instead of SOAP and WS-* standards” [85].

ProgrammableWeb² is the largest public directory of mashups and APIs. APIs enable developers to access data from providers in a standardized way, i.e. external access to data and systems is made possible in a specified way. Whether authentication is necessary and how authentication is handled as well as the extensiveness data is made available is at the provider’s discretion [33, pp. 17 sqq.]. Most APIs support different protocols and data formats.

²<http://www.programmableweb.com>

Today, REST is the favoured protocol and XML the most popular data format supported. 60% of all APIs registered on programmableweb.com support REST, only about 23% support SOAP and only 8% JavaScript. Other oft cited protocols are XML-RPC, which is supported by 3,8% of all listed APIs, ATOM (2,7%) and RSS (1,4%) and GData (2,1%). At first sight, this data can seem to say that the adjudicated importance of ATOM and RSS is overstated, but it must be kept in mind that GData is essentially RSS or ATOM and that ProgrammableWeb lists APIs, not data sources in general. While ATOM and RSS are obviously not important as data format for APIs, many mashups use RSS and ATOM feeds offered by other web sites as sources.

As mashups became popular, many tools for their creation were made available. The functional range differs vastly: some seem to be more like data filters with a new name (e.g. Yahoo! Pipes), others are sophisticated programming languages attached to (or built on top of) other languages (IBM's Sharable Code, formerly known as Swashup [114]). Other examples are Microsoft's Popfly³, IBM's QEDWiki⁴, Dapper⁵, and SUN's jMaki⁶.

2.2.3 Mashup Roles

Mashups architectures feature three elementary roles, as illustrated in Figure 2.3. The *content provider* is a source of data that can be accessed through open APIs over various Web protocols such as REST, RSS or SOAP. The *Mashup site* is the new Web application that requests content and services from various data sources and combines them in order to provide a value-added application to the user. The *client* is the user interface and presented within a Web browser. The user can interact with the mashup through client-side scripting languages like JavaScript.

In the following section, basic mashup styles are discussed, highlighting the varying responsibilities of clients and mashup sites in different approaches.

2.2.4 Mashup Technologies

Within the following section, the key technologies required for the development of mashups are outlined [211, 90, 135, 136, 115, 143, 133, 189, 77].

³<http://www.popfly.com/>

⁴<http://services.alphaworks.ibm.com/qedwiki/>

⁵<http://www.dapper.net/>

⁶<https://ajax.dev.java.net/>

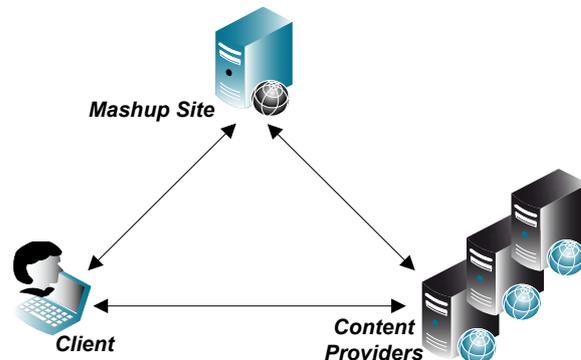


Figure 2.3: General roles of mashups.

Here, section 2.2.4.1 focuses on Ajax, a JavaScript-based technology for asynchronous communication between the client and the server, while section 2.2.4.3 introduces the most prominent data types returned as results from today's open APIs.

2.2.4.1 AJAX - Asynchronous JavaScript and XML

AJAX is not a specific technology, but rather a concept that comprises several technologies in order to achieve a seamless and interactive Web experience. These are XHTML and CSS for presentation and style, the Document Object Model (DOM) browsers for interaction and dynamic manipulation of data, the XMLHttpRequest API JavaScript for asynchronous data exchange [115], and JavaScript as client-sided scripting language. In general, Ajax can be used to implement asynchronous communication between the client and the server, which enables developers, for instance, to reload just the smaller parts of a website that have recently been changed instead of reloading the complete page. This asynchronous communication has sped up the capability for fast user interaction, and therefore the responsiveness of Web applications.

Figure 2.4 depicts the general model for the usage of Ajax within the Web. The left hand side of the figure shows the classic request/response pattern used within the Web, where a HTTP GET request is sent to a server, which, in turn, sends a representation of the requested resource in the form of an HTML document back to the client.

On the right hand side, the integration of Ajax is shown. Here, the client makes a JavaScript call to a client-side Ajax engine; today, this engine is realized by the *XMLHttpRequest* object, which is available in every browser less than four years old. This Ajax engine (i.e. the *XMLHttpRequest* object)

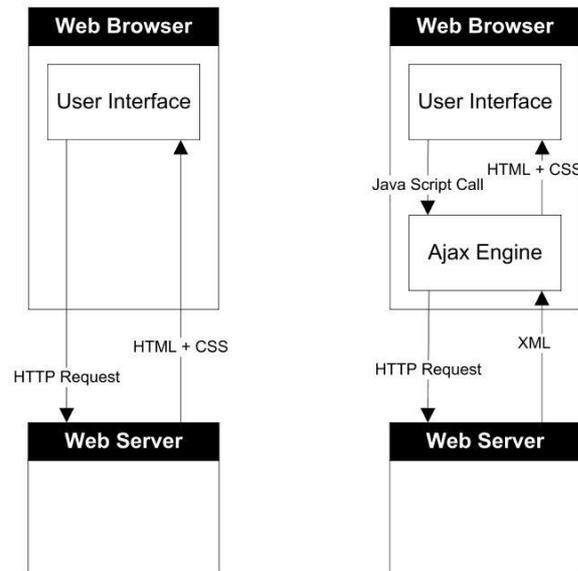


Figure 2.4: General Ajax model.

then transmits HTTP requests to the server and receives the respective responses. Through a callback functionality (realized through a property of the *XMLHttpRequest* object called *onreadystatechange*, which is assigned a callback function when transmitting a request to the server), asynchronous communication between client and server can be established.

This feature works well as long as the content that is requested asynchronously by the Ajax engine resides on the same server as the website that was originally requested. Letting a script making requests to servers other than the one from which it was originally obtained is forbidden by the so-called *same origin policy* of the Web browser, which thereby tries to prevent malicious behavior of scripts. This phenomenon, referred to as *cross-site scripting* (XSS), is illustrated in Figure 2.5.

This restriction is a considerable handicap when developing Web mashups, since data is commonly integrated from multiple 3rd party providers whose data does not reside at the same server as the mashup site. The same origin policy can be circumvented by installing a proxy on the mashups site that relies on the requests of top 3rd party providers. This approach is illustrated in Figure 2.6.

However, the installation of a proxy entails security threats, since the origin policy preventing the malicious behavior of JavaScript code is explicitly disabled. Therefore, the proxy should be configured in such a way that only hard-coded remote URLs can be accessed by the scripts via the proxy.

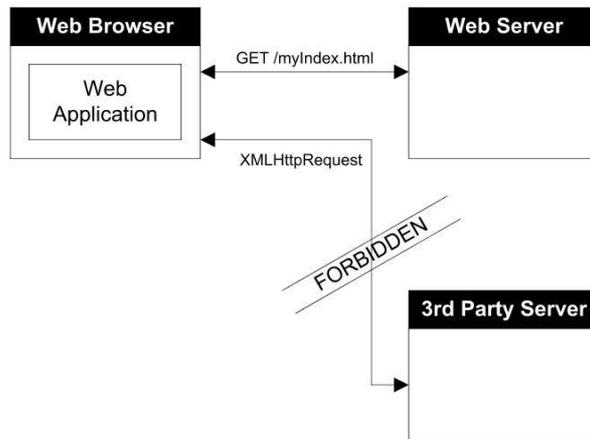


Figure 2.5: Cross site scripting: Violating the same origin policy.

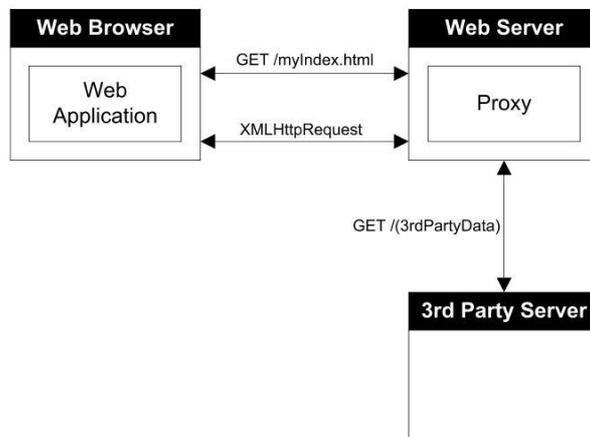


Figure 2.6: Cross site scripting: Usage of a proxy server.

There are some alternatives to the installation of a proxy, such as JSON with Padding (JSONP), an extension of JSON where the name of a callback function can be embedded as input argument into a function call. When data is returned, the passed callback function is invoked. The discussion of these techniques, however, would exceed the scope of this thesis.

2.2.4.2 Server Side Scripting Languages

Scripting languages have become increasingly important to the rapid development of Web applications. There is no approved definition of scripting

languages, and their advantages and shortcomings are never discussed without controversy. Different authors point out different, specific deficiencies of scripting languages, such as the fact that they usually lack an explicit compilation step and possess automatic memory management and powerful operations instead of relying on additional libraries [149].

In contrast to client-side scripting languages such as JavaScript that are executed within a user's Web browser, server-side scripts are deployed and executed on Web servers to generate Web pages dynamically and support user-interactivity. One of their main purposes is the server-side management of data, i.e. the integration of and access to databases. This consideration of user-centric data enables server-side scripts to generate highly-customized Web pages and applications based on the user's specific requirements and access rights [179].

Server-side scripting languages can be used to access services from 3rd party providers and combine their content on the server side, before transmitting it to the client. While classic scripting languages such as PHP, Perl, and Python [108] still play a major role, other languages such as Ruby have emerged and broadened the spectrum of server-side scripting languages.

2.2.4.3 Data Types of the Web

RSS and ATOM are both syndication formats based on XML. Clients subscribe to websites that provide RSS or ATOM feeds. The client can then check the feed for changes and present these appropriately. While RSS 1.0 was RDF based, version 2.0 is not anymore. ATOM is the newer standard created by the Internet Engineering Task Force (IETF) that provides more comprehensive documentation and allegedly maintains better metadata than the older RSS [115]. Both formats are widely used by blogs, developers that publish changelogs, news sites and Wikis, just to name a few.

JSON is neither a markup-language, a document format nor a general serialization format. It is a built-in feature of JavaScript that allows the literal notation of objects in programs. Parsers for many other languages are also available. It aims at similar needs as XML, namely the expression of structured data and its storage in a human-readable form, but the ratio between markup and content is better in JSON [36, p. 177].

YAML (YAML Ain't a Markup Language) is a superset of JSON. Therefore, each YAML decoder is also a JSON decoder. It is a "human-friendly, cross language, Unicode based data serialization language designed around the common native data structures of agile programming languages" [17].

GData is a simple protocol designed by Google for reading and writing on the web. The Google APIs using GData use either ATOM 1.0 and RSS 2.0 and a feed-publishing system based on the Atom publishing protocol with some extensions realized through Atom's standard extension model [69].

2.2.4.4 Local Service APIs

Access to client-sided services has become more important because of the increasing availability of mobile devices featuring considerable hardware and software resources. Under the umbrella of the Open Mobile Terminal Platform (OMTP), key telecommunication providers, device manufacturers, and research institutes have joined forces to define unified access interfaces to services and hardware resources of mobile devices. These activities have been developed within the scope of the BONDI project, which released its first approved version of the BONDI API specification on May 28th 2009 [132].

The specification encompasses descriptions of JavaScript interfaces in order to access the camera, file system, GPS module, or acceleration sensor of a mobile device. The reference implementation has already been developed for Windows Mobile™, but alternative implementations for other operating systems are planned.

2.2.5 Mashup Styles

Two basic styles of creating and executing mashups can be distinguished, both entailing different responsibilities for the client and the mashup site in order to execute the actual mashup. Both styles have significant advantages and disadvantages with regard to their performance, load distribution and security, and thus have to be considered carefully with regard to the requirements of the respective mashup application.

2.2.5.1 Server-side Mashups: Architecture

Within server-side mashups, services and content are integrated on a server, which plays the role of a proxy between the Web application on the client and other Web sites that are integrated into the mashup.

Every request or event originating from the client is forwarded to this proxy server, which then makes the calls on the respective Web sites. Because of this central server role acting as a proxy, server-side mashups are often referred to as *Proxy Mashups*. Figure 2.7 shows the general setting of a server-side mashup.

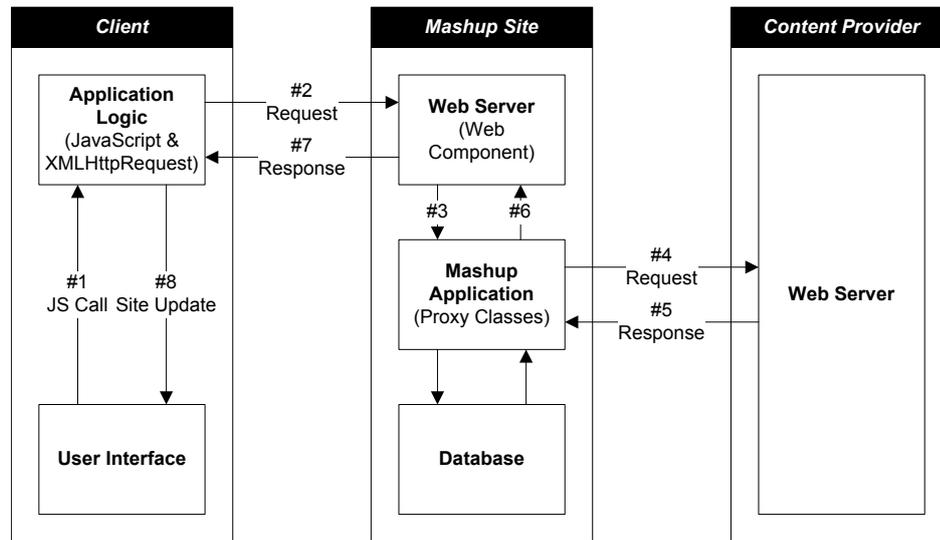


Figure 2.7: Server-side mashups: Architecture.

Whenever a user generates an event at the client Web browser, e.g. pushes a button or clicks on a map, the event triggers a JavaScript function (#1). This JavaScript function makes a HTTP request to the mashup site (#2). Commonly, this request is an Ajax request; the specific nature of this request will be discussed in greater detail later. The request is then received by a Web component like a Servlet or Java Server Page (JSP) on the mashup site. Based on the received request, the component calls on one or multiple methods, within a class or multiple classes containing the application logic, to make calls on 3rd party APIs (#3). Because they act as a proxy between the client's request and the request to the actual mashup servers, these classes are often referred to as Proxy Classes. Note that proxy classes are not restricted in the way they are realized, thus, they can be implemented as plain Java classes or large-scale J2EE components. Thus, the Proxy classes connect to the addressed mashup servers and request the respective services (#4). In turn, the mashup servers process the request and respond with the resulting data (#5). The proxy classes then receive the response and process the data before forwarding it to the Web component (#6). This option of processing data on the server side encapsulates most of the advantages of server-side mashups. For instance, data can be transformed into a format like JSON that is easier to process by the client, it can integrate different data sources and only send the integrated piece of data to the client, or data can be buffered or cached in order to improve the performance of later requests. Finally, the Web component sends the response to the client (#7).

Here, the view of the page at the client side is updated according to the response. In the event that the initial request has been an Ajax request in the form of an XMLHttpRequest object, this page update is executed through the callback function within the XMLHttpRequest, which manipulates the Document Object Model (DOM) accordingly.

2.2.5.2 Client-side Mashups: Architecture

Within client-side mashups, the integration of single services and data sources is performed at the client instead of using an additional proxy server. Thus, clients connect directly to 3rd party APIs in order to request services; a client-side mashup is abstractly depicted in Figure 2.8.

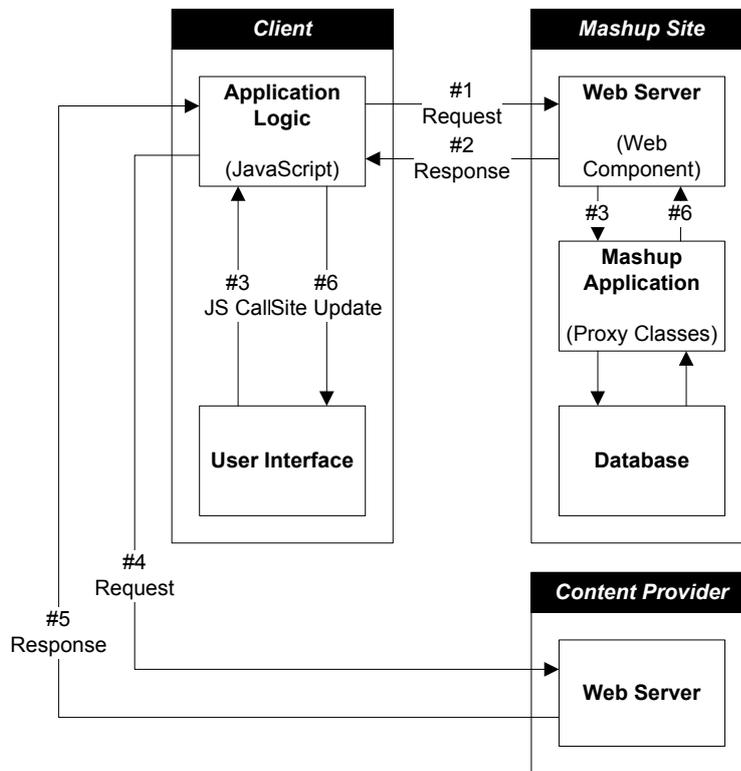


Figure 2.8: Client-side mashups: Architecture.

Initially, the client's browser makes a request to the Mashup site (#1), initiating the request for the server to load the requested Web page into the client (#2). This Web page provides access to JavaScript libraries that enable the client to directly call services at 3rd party APIs later, without

consulting the mashup site beforehand. There are three common ways to provide access to JavaScript libraries. First, the Web page may reference the JavaScript library from the respective 3rd party service provider, like Google Maps; here, referencing the library by a valid URL is sufficient. When the 3rd party provider does not expose an appropriate JavaScript library, the mashup developer can provide one of his own and make it available on the mashup site. Alternatively, other 3rd party libraries may be referenced in order to ease the mashup's creation. Specific actions on the Web page trigger the browser to call a function in the JavaScript library integrated within the Web page. This function dynamically creates a `<script>` tag in the Web site that points to the corresponding 3rd party server (#3). Afterwards, the client initiates an HTTP request based on the `<script>` tag, including the desired format of the response (#4). As discussed above, the format of the response provided by a service usually varies, ranging from XML to YAML or JSON. For client-side mashups, JSONP (JSON with Padding) is the most common response format since it can be easily evaluated (by the JavaScript function `eval()`) at the client. JSONP extends JSON by adding the capability of appending the name of a local callback function to the JSON object. Thus, in the case that the server provides the response in JSONP, a call is made to the callback function with the JSON object as parameter (#6). Finally, the DOM of the page is manipulated by the JavaScript function and the page is updated accordingly.

2.2.5.3 Comparison of Mashup Styles

Proxy-style mashups (PM) and client-side mashups (CM) are compared within Table 2.1 with regard to their advantages and respective shortcomings. Here, '+' indicates that a specific aspect is supported, '-' that it is not.

The implementation of client-side mashups is much easier, since developers only have to reference and include JavaScript libraries provided by 3rd parties. Server-side mashups on the contrary require the development of appropriate mashup classes at the mashup site. Moreover, client-side mashups commonly perform better since messages (request/response) are directly exchanged between the client and respective servers. Server-side mashups style can lead to significant delays, since every request and response is transmitted to the proxy first and then sent to the client or server, respectively. Thus, there are four steps in a classic request/response instead of two. In addition, client-side mashups cause less strain on the mashup site since messages are directly passed to the 3rd party providers.

Aspect	PM	CM
Easy Implementation	-	+
High Performance	-	+
Low Server Processing Load	-	+
Unlimited Asynchronous Calls	+	-
Buffering	+	+
Caching	+	-
Data Transformation	+	-
Data Transmission Optimization	+	+
Data Manipulation	+	+/-
Security Handling	+	-

Table 2.1: Comparison of proxy mashups (PM) and client-side mashups (CM).

However, these advantages of client-side mashups come at the expense of integration and aggregation capabilities provided by a mashup proxy. First, client-side mashups typically allow only two to three connections to 3rd party servers (i.e. XMLHttpRequests), while server-side mashups can establish a (nearly) unlimited number of asynchronous communication channels. Moreover, the proxy may serve as a buffer for incoming data from 3rd party services. Also the caching capabilities of a proxy are much more advanced than the limited cache size of clients. A proxy can also process data effectively before passing it on to the client. Thus, a proxy can transform data into a different format before it is transmitted to the client, may split up the data in smaller chunks to optimize the data size for transmission, and can manipulate data or combine it with other data sources before sending it to the client. Security handling is also easier on the server side. However, the proxy has to be secured against unauthorized access. Client-side mashups on the contrary, have to trust the 3rd party providers, since they execute the provisioned scripts directly at the client.

2.2.6 Drawbacks of Mashups

2.2.6.1 General Drawbacks

Mashups come with some technical drawbacks already identified by D. Merrill [115] in 2006. Data integration challenges are among the most important ones, i.e. drawbacks due to shared semantic meaning between heterogenous data sets, for which translation systems need to be developed. The architecture explained in section 2.2.5 is also fault prone. As Gartner Research states, mashups are relatively vulnerable to failure as they often depend on

external logic sources [211]. Therefore, Gartner sees mashups as useful for small and relatively simple application logic, while complex business process integration should not be realized with mashup technologies [143].

Other features yet to be addressed are single sign-on, several security concerns related to cross-domain applications, authorization and authentication features likely to be employed by content providers, and encryption to ensure data integrity apart from HTTPS connections.

Some of the characteristics of mashups, especially their quick and easy implementation, are rooted in the lack of complex standards. A drawback of this is that each service might use a different data format for its services. While this means that each service provider can use the format that fits its needs best, mashup developers cannot switch between different API and content providers easily. SOAP web services use complex standards that lead to a weak adoption, but switching between different services that use these standards is easier. The RIA New desk put it this way: “Web 2.0 seems to call out for discipline, while SOA seems mute and autistic” [85].

Component limitations can cause serious problems. Because client-side and server-side mashups are relatively different in their structure, this evaluation is split in two parts in order to go into detail on the involved components.

2.2.6.2 Drawbacks of Client-side Mashups

As AJAX uses the browser’s scripting possibilities together with its DOM or innerHTML properties to manipulate rendered pages, browser compatibility issues arise.

Another technical challenge related to the asynchronous update of content of already rendered pages are the URLs. Since the content is not directly linked to the called URL anymore, the ‘Bookmark’ and ‘Back’ Buttons of the browser do not work as expected by the user [115].

Although the client-side mashup style can increase performance because the Web application does not need to contact the host server for requests to remote sources, performance can also be limited [136]. First, the security sandbox of browsers limits the execution of XMLHttpRequests to two or three. Mashups often depend on more than two or three different sources, therefore this limitation can lead to performance constraints. Second, the browser needs to handle data in whatever format it is received. While JSON and JSONP can lead to increased performance, XML may decrease it. Servers may cache oft requested data or data from sources with long response times, but client-side caching is very limited. Moreover, the web application has to

handle any errors that occur during data retrieval from external sources. To use any AJAX-based functionality, JavaScript has to be enabled within the browser. Although only a few users have JavaScript disabled today, at least an emergency solution for them must be implemented. Moreover, security requirements are more difficult to manage client-sided [136].

2.2.6.3 Drawbacks of Server-side Mashups

All problems that arise with AJAX apply to server-side mashups as well. They are attributed to for client-side mashups because those mashups depend on AJAX and asynchronous updates. Server-side mashups may well use AJAX to achieve a more seamless user experience, but usually a Web application with similar features could be realized without excessive use of AJAX.

The biggest issue with server-side mashups is the increased response time. As explained above, the server undertakes all communication with external services and functions as proxy. That means that, unlike client-side mashups, each time the user invokes a task that requires an external service, it has to communicate with the host server, which in turn calls the external service. Eventually, the communication process has at least one more step unless some kind of server-side caching mechanism can fulfill the request by the client. Solid security measures have to be applied to the servers. Although security requirements should surely be applied to all servers and not only those that host mashups, additional precautions should be taken. Error-prone servers might lead to misuse of external services [136].

2.3 BUSINESS PROCESSES: COMPOSED WEB SERVICES

2.3.1 Service Oriented Architectures: The Basis for Business Processes

A service-oriented architecture (SOA) is an information technology architectural style that utilizes the advantages that result from distributed services available in a network. OASIS⁷ defines SOA as follows [129]:

[SOA is a] paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

⁷More about OASIS can be found at <http://www.oasis-open.org>

SOA emphasizes the implementation of software components as modular services. While service-orientation is mostly used for distributed systems, SOA can also refer to a single system where services might also be processes. More precisely, SOA refers to the design of a system, not its implementation. For reasons of simplicity, in this thesis the term SOA also refers to implementations in a service-oriented style.

Services, usually grouped by functionality around a business process, are shared in flexible way in a network and constitute the basic constructs of a SOA's implementation. Three different roles are introduced: service providers, service requesters-also called clients-and service registries or brokers. The main benefit of SOA is its loose coupling of services. Loose coupling means that clients are independent from the services' implementation. Clients communicate with services according to an interface description, i.e. the implementation of the services is unknown to the client and can be changed without the client's knowledge. Other key concepts that SOA fulfills are implementation neutrality, flexibility, persistence and granularity. These SOA specific characteristics are discussed in greater depth in section 2.3.1.3.

SOA is based on service-oriented computing – a “new computing paradigm that utilizes services as the basic constructs to support the development of rapid, low-cost and easily composed distributed applications even in heterogeneous environments” [139]. Applications following the service-oriented computing paradigm use services by composing them. Huhns and Singh explain that “[s]ervice-oriented computing provides a way to create a new architecture that reflects components' trends towards autonomy and heterogeneity” [81, p. 2]. The first implementation of SOA in a broader sense, although the term SOA did not exist then, was CORBA, the Common Object Request Broker. But due to its high-level of complexity and security issues, CORBA failed to gain prominence [130].

2.3.1.1 From Component-Based Development to Service Oriented Architectures

A special branch of software engineering focuses on the separation of concerns within software systems. Therefore, a software platform is considered to consist of a set of software components, where each component constitutes a module or software package encompassing a set of functions. According to Szyperski et al., those components are characterized by its interfaces and context dependencies, where an interface is a set of operations that can be invoked by clients, and context dependencies specify the components' requirements on the deployment environment to function [122]. Such software

engineering approaches that base on the notion of components are referred to as component-based development (CBD) or component-based software engineering (CBSE). As the paradigm of object oriented programming (OOP), CBD targets software economics that require a high degree of reusability in software development.

According to Crnkovic, those approaches' main goals are the support of systems that are built by assembling multiple components, where each component constitutes a reusable entity. The component orientation should facilitate the maintenance and upgrading of systems by providing means to replace single components [42]. CORBA [131], Sun's Enterprise Java Beans [137], and Microsoft's Component Object Model [71] are three of the most prominent technologies for component-oriented systems. Communication among components is realized via programmatic interfaces that are exposed to the system. These interfaces encapsulate a component's internal functionality, so that the implementation of the component remains hidden for the clients accessing it.

Service Oriented Architectures (SOA) add a layer of abstraction on top of IT environments and thereby overcome obstacles in terms of heterogeneity. Loosely coupled Web services as an implementation of such a SOA rest upon many concepts stemming from component-based approaches, such as the encapsulation of business relevant information and functionality into a component or service, respectively. However, CBD bases on tightly coupled APIs, where a modification of a component in general affects other pieces of the implementation that access this component, which makes CBD inflexible and hard to scale.

SOA focuses on a small aspect of component based systems by defining abstract interfaces that can be dynamically discovered and thus decouple service provider and consumer. Thereby, SOA extends CBD towards ubiquity and pervasive access by providing means to reach callable components via non-proprietary, interoperable protocols and service descriptions that loosen the interface from its implementation [26]. In the scope of SOA, a component can model a provider that exposes multiple functions to consumers, which dynamically discover and integrate these services when needed.

Thus, SOA focuses on an abstraction and dynamic integration of services on top of programmatic mythologies such as component models. It does not deal with many of the functionalities provided by components, such as the ability to move components between nodes. In that sense, the current model of mashups is closer to the abstraction level of SOA than to component oriented systems. Within the Web, services are made available via well-defined interfaces and concatenated by means of scripting languages. Moreover, the Web rests upon an architectural style that bases on the concept of simplicity,

where all operations are reduced to simple operations on resources. In its current state, the Web thus exposes functionality by means of well-defined interfaces rather than by means of components.

Within the remainder of this section, a closer introduction to Service Oriented Architectures (SOA) is given, bearing in mind their relation and inheritance from component based systems.

2.3.1.2 Major Layers of SOAs

Papazoglou and Georgakopoulos segment SOA services in three layers [138, pp. 26 sqq.] [139, pp. 5 - 6]. Three main characteristics, however, cut across the three layers: Quality of Service (QoS), semantics, and non-functional characteristics. Quality of Service encompasses functional as well as non-functional aspects: service metering and cost, performance metrics-e.g. response time, security attributes, integrity, reliability, scalability, and availability. For services to interact properly, they have to agree upon the service description and the semantics that govern the interaction between them. Non-functional characteristics include service modeling and service-oriented engineering, i.e. analysis, design and development methods and techniques crucial to the development of meaningful services and service compositions.

The basic services of a SOA are aggregated in the *Foundation Layer*. Service descriptions are used to delineate service capabilities, interface, behaviour and quality. This information is needed for service discovery and selection, binding, and composition of services. The service capability description states the conceptual purpose and expected results of the service. The service interface description outlines the nature of the interface, while the service behaviour description defines expected behaviour during execution [138, p. 26].

Composite services' functions are consolidated in the *Composition Layer*; these are coordination, monitoring and conformance.

Coordination, in this context, means control of the execution of composite services, management of dataflow and output of services. This is realized with specified workflow processes executed by workflow engines. Monitoring covers the subscription to information and events produced by the composite services and the publishing of higher-level events. This is realized by filtering, summarizing and correlating component events [138, p. 26]. The assurance of integrity of composite services (Conformance) is realized by constraints, data fusion, and matching of parameter types with those of the components.

The top layer, or *Management Monitoring Layer*, aims at two goals. First, managed services, the service operators, are provided to handle crit-

ical applications that require enterprises to manage the service platforms, service deployment and services themselves. They provide statistics about performance and effectiveness, aim to increase the transparency of individual business transactions, and usually feature sophisticated eventing mechanisms [138, p. 26]. Management of service level agreements (SLAs) are found on this layer as well.

Second, the management layer aims to support open service marketplaces. Open service marketplaces are open equivalents of vertical marketplaces. “Their purpose is to create opportunities for buyers and sellers to meet and conduct business electronically, or aggregate service supply/demand by offering added-value services and grouping buyer power.” [138, p. 27]. The management layer’s aim thereby is to enable enterprises to make their offerings visible and establish industry-specific protocols to carry out transactions.

2.3.1.3 Characteristics of Service Oriented Architectures

Connectivity is realized by HTTP(S); but the data format in SOAs is usually XML. While the understanding of different services is handled by XML and XML Schema, it cannot ensure the correct understanding of a message’s meaning. In legacy systems, XML was already used as standardized data format, but “it does not resolve how communicated data is to be understood and processed” [81, p. 4]. So, even if it adheres to standards, problems with messaging and semantics remain. Service-oriented computing provides the necessary abstraction and tools to model information and create cross-boundary processes [81, p. 3].

Service-oriented computing (SOC) can be viewed on several levels of abstraction: the intra-enterprise level, the inter-enterprise level, the infrastructure level and the software component abstraction level.

The most abstract level is the inter-enterprise level. It coordinates the cooperation between several independent business transactions across enterprise boundaries. SOC has gained in relevance with the rise of supply-chain management systems and flexible, on-demand manufacturing, which led to cross-enterprise processes. The intra-enterprise abstraction level is concerned with abstraction within the boundaries of an entity. Usually, within an entity different systems are used; some of them can be completely new, some can be legacy systems. A SOA requires policies to be made explicit and can therefore enforce compliance with necessary authentication and authorization enterprise policies that are usually deployed. The infrastructure abstraction level is concerned with building complex applications over distributed systems, just like Grid architectures. According to Huhns and Singh [81, pp.

4 - 5], the Grid research community sees Grid services and Web services as analogous. SOA abstracts the infrastructure level of an application and enables the efficient usage of distributed resources. The software component abstraction level focuses on autonomous software components that help improve software development. Abstraction through services leads to software modules with clear semantics specified in interfaces. “The result is that SOC provides a semantically rich and flexible computational model that can simplify software development” [81, p. 5].

Benefits that can be reaped by using a SOC system are the reduction of overhead, more efficient use of legacy systems, the prevention of redundant work and systems, dynamic selection of business partners, improvement of responses, simpler satisfaction of individual preferences and, eventually, the avoidance of unnecessary costs [81, pp. 2 - 3].

The main characteristics of Service Oriented Architectures are described briefly in the following.

Late binding – Late binding (also referred to as dynamic binding) denotes the linking of concrete services to abstract services at runtime, i.e. an allocation of actual services to service placeholders [147]. During development, abstract services are used to define the needed functionality. A service-oriented system is configured as late and flexibly as possible, i.e. the components should be bound as late as possible. Thus, dynamic changes according to predetermined requirements in the system are possible, which is considered a key benefit of service-oriented systems [176]. During runtime, the available services are scanned and the one matching the requirements best is bound. Common requirements are functionality and Quality of Service (QoS) [116]. In any case, the binding process should be transparent to the developer. In SOA, late binding is usually realized with the service registries holding service descriptions queried during runtime.

Loose coupling – Loose coupling in general refers to the coupling of services within a system. Loose coupling in the context of service-oriented systems in particular refers to the minimal knowledge services need about each other [159]. This means that, during development, the required functionality is specified, i.e. no fixed dependencies between clients and services are set. It is closely connected to late binding: actual services are discovered and bound when needed, i.e. if possible at runtime. It is based on minimal dependencies, which means implementation neutrality needs to be ensured. Service awareness is maintained through service discoverability, which is ensured through the use of registries in turn [139].

Clients communicate with services according to interface descriptions, i.e. actual service implementations become virtually irrelevant for service requestors. Xuanzhe et al. summarize loose coupling as follows: “Services

in SOA are loosely coupled: they are developed and hosted by different providers, described in specific standard interfaces (e.g. WSDL), published in an accessible registry (e.g. UDDI), and can be discovered and requested via standard protocols (e.g. SOAP) [211]. Most SOA implementations provide mechanisms that realize loose coupling between services with regard to location, time and protocols. Service-oriented computing provides the necessary abstraction and tools to model information, create cross-boundary processes, and assert properties in a loosely coupled way [81].

Reusability – The main driver for businesses to adopt the SOA approach is the reuse of business services. Legacy applications can be employed as services and reused for new business requirements with little effort. The main obstacles to reusing code were the unique characteristics of existing applications, accompanied by the necessity of fully understanding them in order to reuse the code. With its implementation neutrality, SOA provides the opportunity to integrate services in an easy way [176, p. 5]. The effort that is saved for developing functionality that already exists in the enterprise or at business partners, can lead to huge savings in development cost and time. The benefit of reuse grows with the number of services integrated into a SOA [134]. To maximize reuse, formerly monolithic business logic often needs to be divided into several independent services.

Distribution – The notions of late binding and loose coupling create the ability to distribute components and application logic. Since services are loosely coupled and the mechanisms that permit late binding are provided – standardization, discoverability, and replaceability in particular – Service-Oriented Architectures may be used to realize grids with relatively little additional effort. In contrast, distributed systems were mostly realized with proprietary technologies until not too long ago [176]. For the creation of applications that reach beyond organizational boundaries, the involved parties had to agree on technologies and protocols, thus leading to inflexible, and often proprietary solutions. This kind of application can be created more easily if the involved parties use service-oriented systems. Implementation details and location of the services as well as the environment do not concern the involved services as long as access is ensured. SOA provides the opportunity to move away from vertical, proprietary grid solutions, to standardized, cross-boundary service compositions [176].

Service Composition – Realizing business processes through the composition of multiple services, which are then offered as services themselves is one of the “ultimate ideas” behind SOA [111]. The resulting service compositions may be used by other services in further compositions, or they may function as complete applications. The composition of services is a logical and necessary step toward fully utilizing the potential of service-oriented

systems and especially toward realizing reusability. The ability to automatically compose services is an essential step in reducing development time and cost [139]. Nonetheless, automatic service composition faces some problems. These result from several aspects. First, usually complex applications lead to a high degree of interweaved processes. Second, the statelessness of Web Services poses a problem when creating complex services. Also, the quality of generated services is an important factor, e.g. generation and handling of errors, execution time of the scripts, Quality of Service, and many more [139].

2.3.1.4 Basic Architecture

Services are the basic components of each SOA, and communicate by exchanging messages. They can be accessed via a network, and provide functionality to a service requestor. Services are “self-describing, open components that support rapid, low-cost composition of distributed applications” [138]. The provided functionality can be a discrete function or a set of related functions. Services that aggregate a set of business functions are said to be ‘coarse-grained’. Moreover, multiple services can then be combined to form a composition. Service compositions, which are discussed in greater depth in section 2.3.2, can be used to satisfy more complex business requirements [134, p. 2]. This means that, although services are loosely coupled in a SOAs, they can depend upon other services or resources such as databases [176, p. 3]. Services advertise details about their capabilities, policies and supported protocols, but due to their loose coupling, implementation details are completely hidden.

Classic SOAs have three basic roles, as illustrated in Figure 2.9.

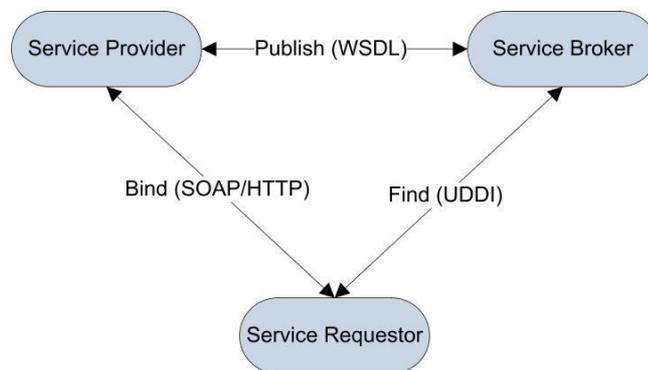


Figure 2.9: Classic roles of a SOA.

Service provider The service provider is the entity that offers a service. In a service-oriented environment, the implementation of the service is of no importance to the provided process itself. The interface of a service needs to be well-defined and achieved by using an appropriate service description language. The service then advertises its functionality to a well-known registry. Apart from that, no measures are undertaken to communicate the existence of a service. Once a service is contacted by a client, the functions of the service are executed accordingly and the result, if there is one, is sent back. Because of the nature of loose coupling, the actual implementation of a service can be changed. In fact, it can be replaced completely, as long as the interface remains the same.

Service registry or service broker The concept of SOA also includes an optional service registry, which basically contains information about the interface of the service. Clients can search for and find services that perform a specific task by examining the registry, i.e. the registry is responsible for making the interface of Web services available to any service requestors [178]. The registry can also be coupled with a repository component that contains additional information about the registered service, e.g. information about policy statements, preconditions, dependencies and business processes in general [134].

Service requester - client The client in a service-oriented system is the service consumer. It either knows the services' addresses or examines the registry. The communication between the client and the service provider is mostly realized by HTTP or HTTPS. Client and provider are bound via a communication protocol so that the client can make use of the remote service offered by the provider.

2.3.1.5 Web Services as SOA Implementation: Key Technologies

Web Services constitute today's most prominent way of realizing a Service Oriented Architecture, where the communication must be performed with open Internet protocols such as HTTP or FTP while the exchanged messages between services must be coded in XML [157].

In general, a Web Service is an application that is uniquely identifiable by an URI and its interfaces are formally defined and described. Here, Web Services can communicate with other applications through message exchange. The *W3C* defines Web Service as "a software system designed to support interoperable machine-to-machine interaction over a network" [199]. The

term Web Service is most often used synonymically for WS-* Web Services (also called *W3C* Web Services or SOAP Web Services) which use SOAP for communication and use WSDL as a description language for services interfaces, as recommended by the *W3C* [203]. UDDI is used to access the service repositories. These de-facto standards for Web Services are included in Figure 2.9 to specify their specific role in realizing a SOA.

Alternatively, services can be accessed with classic HTTP verbs (GET, PUT, POST, DELETE). Due to the architectural style underlying today's Web – the REpresentational State Transfer (REST) architectural style that finds its implementation in HTTP – these kind of Web Services are also referred to as REST Web Services.

In the following, an overview of the most important technologies for the implementation of SOA is given. In this thesis, SOA implementations are equated with the most often used implementations that meet WS-* standards.

2.3.1.5.1 SOAP

SOAP was originally an acronym for Simple Object Access Protocol, but since version 1.2 of SOAP this was officially dropped. It is an XML-based protocol that specifies envelope information, content and processing information for a message [209, 176], [44]. Technically, SOAP is the successor of XML-RPC, but transport and interaction neutrality as well as the envelope, the header and the body structure are taken from WDDX [156].

2.3.1.5.2 Web Service Description Language – WSDL

The Web Service Description Language is a platform, protocol and programming-language independent, XML-based description language for Web Services [38]. Specifically, it describes the interface of a Web Service, i.e. access URI, the available operations, protocol and the procedures to pass on messages and the structure of messages [156]. WSDL is usually stored in a repository and administered by registries [44].

Listing 2.1 shows the skeleton of a common WSDL description. Here, *portTypes* describe the operations performed by the Web service as well as the involved messages; thus, a *portType* can be compared to a class within classic object-oriented programming languages. A *message* is used by the Web Service and defines the data elements of an operation. Therefore, each message has 1 – *n parts*, which can be compared to a parameter of a function call within common programming languages. While *types* define the data

```
1 <definitions>
2
3 <types>
4   definition of types
5 </types>
6
7 <message>
8   definition of a message
9 </message>
10
11 <portType>
12   definition of a port
13 </portType>
14
15 <binding>
16   definition of a binding
17 </binding>
18
19 </definitions>
```

Listing 2.1: WSDL skeleton.

types used by the Web Services, the *binding* specifies the message format and protocol details for each port.

2.3.1.5.3 Universal Description, Discovery and Integration - UDDI

Universal Description, Discovery and Integration (UDDI) is a XML-based, open industry initiative registry sponsored by *OASIS*. UDDI-registries are set up to enable discoverability and manage information about service providers, service implementations and service metadata [188]. They can be queried by SOAP messages, and provide access to WSDL documents. Therefore, clients can easily gain the required knowledge about advertised Web Services by querying the registry [44].

2.3.1.6 Alternative Types of Service Description

Beside using WSDL to describe classic SOAP Web services, multiple other service descriptions with varying focuses exist. There are alternative interface descriptions such as the Web Application Description Language (WADL), which provides the means to describe the interfaces of REST-based Web services from a resource-oriented point of view. In addition, semantic service descriptions such as WSDL-S aim to extend the classic interface descrip-

tions such as WSDL or WADL towards a functional description of services. Within the remainder of this section, a couple of these service descriptions are outlined.

2.3.1.6.1 *Web Application Description Language – WADL*

WSDL is tailored for SOAP Web services and therefore provides an action-oriented perspective on services, while most services made available over open 3rd party APIs possess REST or JavaScript interfaces. There have been multiple proposals for possible description languages for REST services. Dr. Marc J. Hadley’s Web Application Description Language (WADL) is a XML-based format that describes Web applications in a platform independent manner, and constitutes a machine processable description of HTTP-based Web applications, typically REST services [73]. WADL for REST services can therefore be regarded as a counterpart to WSDL for classic SOAP Web services. A WADL description consists of a set of available resources, the relationship between those resources, methods that can be invoked upon each resource, and the data formats used [73]. Within his survey, Steiner evaluated these languages and identified WADL as the most suitable one, since it provides the most well-formed specification and expressiveness [180]. In addition, WADL also provides means to easily extend its scope from REST services, for instance towards JavaScript interfaces.

Listing 2.2 shows the skeleton of a WADL file, which consists of three main parts. The *resource* tag defines the Web resource that can be accessed via the included method. While the *method* tag defines the HTTP verb used by the method, the *representation* tag specifies the representation of the resource when returned.

2.3.1.6.2 *Resource Description Framework – RDF*

RDF is a collection of *W3C* specifications that provides syntactic structures for describing data [153]. While XML can be used arbitrarily, RDF provides a sufficient basis to encode concepts in a machine-readable way [115]. RDF enables the description of resources identified via a URI. Therefore, relations between resources can be expressed through *statements*. Such statements are triples, consisting of a subject, a predicate and an object. The subject is modeled as resource, which is then put in relation to the object by means of the predicate, which describes a property of the subject. For instance, the statement “Every photo has a caption.” can be expressed by

```

1 <resources base="http://example.company.org">
2   <resource path="employees">
3     <resource path="employee">
4       <method href="#getEmployee"/>
5     </resource>
6   </resource>
7 </resources>
8
9
10 <method id="getEmployee" name="GET">
11   <response>
12     <representation href="#employee"/>
13   </response>
14 </method>
15
16 <representation id="employee" mediaType="text/xml"
17   element="domain: employee"/>

```

Listing 2.2: WADL skeleton.

the RDF triple (`http://www.model.org\#photo`, `http://www.model.org\#has`, `http://www.model.org\#caption`).

RDF-Schema (RDF-s) is an extension of RDF that defines additional elements. For instance, resources can be classified and predicates can be assigned domain spaces.

2.3.1.6.3 OWL and OWL-S

The Web Ontology Language (OWL) is based upon RDF(-S) in order to model ontologies; it is recommended by the W3C as standard since 2004 [16, 202]. In general, an ontology constitutes a set of machine-readable objects and relations among these objects. In addition to the constructs defined within the scope of RDF and RDF-S, OWL introduces set operations and elements to help determine a set's cardinality.

OWL-S is based upon OWL and constitutes a newer version of Darpa Agent Markup Language (DAML-S) [11]. OWL-S is a language that describes the behavior of Web services [112]. A *Service* builds the core element of OWL-S; it possesses relations to a *ServiceProfile*, a *ServiceGrounding*, and a *ServiceModel*; these relations are depicted in Figure 2.10⁸.

The service profile allows for the definition of both the functional and non-functional properties of a service, which are stored as inputs, outputs,

⁸Image source: <http://www.w3.org/Submission/OWL-S>

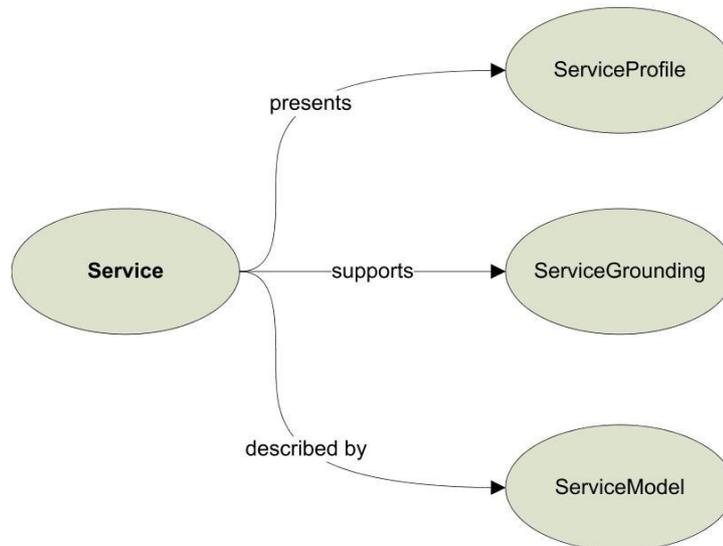


Figure 2.10: Top level of the service ontology according to OWL-S.

preconditions, and effects, and are generally referred to as IOPE descriptions [88]. While the service model specifies the services as a process, the service grounding encompasses access details like supported protocol and data formats.

2.3.1.6.4 WSDL-S

WSDL-S extends the WSDL standard with semantic expressivity. According to the *W3C*, this semantic expressivity is needed to adequately describe Web Services. WSDL-S incorporates semantic concepts already developed, for example in OWL-S [112]. The information WSDL-S holds, in addition to conventional WSDL descriptions, are preconditions, input, output, and effects of operations [4].

In contrast to OWL-S, WSDL-S is not based on a concrete domain model. Instead, WSDL-S supports the annotation of *input* and *output* parameters by concepts stemming from multiple different domains. In addition, WSDL-S operations can be extended to *precondition* and *effect* parameters.

2.3.1.7 Drawbacks of SOAs

In the explanations above, the major advantages of SOC were identified. Nevertheless, SOA, and especially the current implementations of SOA with

SOAP Web Services, have significant drawbacks when compared to other approaches for distributed systems.

Papazoglou et al. discuss future challenges for SOA in detail in [139]. Major challenges for the basic services aggregated in the foundation layer concern dynamically configurable runtime architectures, dynamic connectivity capabilities, end-to-end security solutions, infrastructure support for applications, data and process integration, and service discovery. Also, SOA implementations with Web Services do not support less standardized technologies common in the Web, e.g. REST-based services, and especially not the client-side orchestration that has recently gained momentum [211].

In the composition layer, autonomic composition, QoS-aware service compositions, business-driven automated composition and syntactic, behavioural and semantic conformance are identified as the challenges that need to be addressed in future research. Dynamic and automated creation of composite services has recently garnered extra attention [82]. Additionally, Huhns and Singh identify missing recovery mechanisms that would restore consistent states in service-oriented systems as major drawback [81].

For the top layer, the management & monitoring layer, Papazoglou et al. identify self-configuring, self-healing, self-optimizing and self-protecting services as significant challenges.

Apart from these mostly tangible obstacles, Malek and Milanovic [117] as well as Srivastava and Koehler [177] identify missing planning and correctness verification mechanisms, i.e. formal specifications, as a main problem. They admit that formal approaches are difficult to apply in real-world systems. Nevertheless, they hold that it would be advantageous to be able to analyze Web Service properties with elaborate mathematics. To realize this, it would have to be possible to translate WSDL and SOAP into mathematical solutions.

Next to these technical challenges, Xuanzhe et al. identify the need for highly skilled developers as an additional issue with SOA, especially for its implementation with SOAP Web Services [211, 107].

2.3.2 Service Composition Languages: Programming in the Large

The composition of multiple services to fulfill given business needs, which are offered as services themselves, is one of the ultimate goals for SOA [111]. The resulting service compositions may be used by other services in further compositions, or they may function as complete applications [139]. The composition of services is a logical and necessary step to fully utilize the potential of service-oriented systems, and especially to realize reusability. While the

operations grouped in the foundation layer of a SOA provide the basis for a service-oriented system, specifications in the higher composition layer provide functionality that leverages services and enables the integration of automated business processes [117].

For the creation of higher-level composite services, different approaches can be taken. Nevertheless, several technical requirements have to be met regardless of the approach: asynchronous service invocation, exception- and error-handling, as well as transactional integrity, the ability to compose already composite services and, in order to meet changing needs, the code has to be dynamic, flexible and adaptable [142].

Two ways of describing a service composition can be distinguished. While *service orchestrations* model service compositions from a central point of view, *service choreographies* focus on the interactions and message exchanges between multiple partners. The following two sections introduce these basic concepts.

2.3.3 *Orchestration Languages*

A service orchestration is a composition of multiple services, which are controlled by a single, central component, often referred to as *orchestrator*. The orchestration itself is commonly described by a high-level programming language that enables software developers to specify single services' execution order; therefore, these kind of descriptions are often referred to as *workflows* or *workflow descriptions*. They can be expressed by various languages featuring a structured expression of statements, such as, for instance, XML. Workflow descriptions of service compositions can be executed by an orchestrator, which is represented by a respective runtime environment.

The usage of workflow descriptions to express service compositions goes back to the classic workflow management systems as surveyed in 1995 by Georgakopoulos [67] and discussed extensively by van der Aalst and van Hee in [194]. Van der Aalst et al. evaluated existing workflow languages later on with regard to their usage as service composition languages and concluded, that it "is remarkable how much attention it receives while more fundamental issues such as semantics, expressiveness and adequacy do not get the attention they deserve" [193]. Additionally, modern workflow languages for the description of service compositions do not use established formalisms, but constitute technologies driven by concrete products and respective companies [192].

Within this section, the role of workflow-oriented orchestration languages used to define the execution order of single services within a service composi-

tion is discussed through the example of BPEL, which is already established as the de facto standard for service orchestration within today's SOAs.

BPEL stands for Business Process Execution Language and evolved from the combination of IBM's Web Service Flow Language (WSFL) [104] and Microsoft's XLANG [183], a block structured language that provides simple control structures for looping, conditions and sequential as well as parallel execution. In contrast to XLANG, WSFL is not block-structured, but is based on directed graph concepts. In 2002, Microsoft and IBM, together with BEA Systems, developed the Business Process Execution Language for Web Services (BPEL4WS) that combined XLANG and WSFL. The UN/CEFACT (United Nations Center for Trade Facilitation and Electronic Business) developed electronic business using XML (ebXML) [190]. ebXML could define the choreography and communication protocols between Web Services.

In 2003, BPEL was called BPEL4WS [96]; its version 1.1 was passed to OASIS for standardization [10]. In 2004 the standardization of BPEL4WS 1.2 started, which was then renamed to WS-BPEL 2.0 in order to align it with other Web Service related standards [92]. The version jump to 2.0 was chosen to reflect major changes in the language; moreover, version 2.0 is no longer compatible with version 1.1. The standardization of WS-BPEL version 2.0 was finished in April 2007. Within the following, the abbreviation BPEL is used for the current version 2.0 of WS-BPEL.

BPEL is a XML-based grammar on top of WSDL that coordinates the Web Services participating in a composition by describing the required control logic [142]. The WSDL description defines the interface of services, while BPEL specifies their execution order. The process description can be executed by a respective BPEL orchestration engine [82]. BPEL supports executable processes as well as abstract processes. Executable processes are those that can be executed by the orchestration engine. It basically models private workflows for the participating services, and these processes can therefore be called orchestration. Abstract processes define the public message exchange between services, essentially following the choreography approach. They are not executable and do not contain details about internal flows of the services. BPEL focuses on orchestration, while the Web Service Choreography Description Language (WS-CDL) focuses on choreography [142]; WS-CDL is later discussed within section 2.3.4.

BPEL composition results are called processes, participating services are partners, and communication via message exchange and transformation of intermediate results are called activity. Thus, processes consist of activities performed by partners. Interaction of a process with external partners is realized by WSDL interfaces. Processes are defined by two files and an optional third one. The BPEL source file, with the ending *.bpel*, describes

the activities and the process interfaces; later ones are specified in WSDL (*.wsdl*) as *portTypes* of the composed service. The third and optional file is a deployment descriptor in XML that contains information about the physical locations of partners [117].

BPEL language components can be grouped into basic activities, structured activities, and scopes. Basic activities defined in BPEL are listed within Table 2.2, while predefined structured activities are given in Table 2.3.

Basic Activity	Description
<code><invoke ></code>	Calls a Web service operation
<code><receive></code>	Waits for a message (blocked waiting)
<code><reply ></code>	Generates a reply for a I/O operation
<code><wait></code>	Waits for a predefined time span
<code><assign></code>	Copies data from one variable to another
<code><throw></code>	Throws an exception
<code><terminate></code>	Terminates the whole process instance
<code><empty></code>	Inserts NOOP (No Operations)

Table 2.2: BPEL: Basic activities.

Structured Activity	Description
<code><sequence></code>	Defines a sequence of actions
<code><switch></code>	Defines a branching based on given constraints
<code><while></code>	Defines looping
<code><pick></code>	Enables the selection of one of multiple alternative paths (through blocked waiting for a message/time-out)
<code><flow></code>	Definition of parallel processes

Table 2.3: BPEL: Structured activities.

Scopes bundle activities. Fault and event handlers can be assigned to scopes, as well as termination and compensation handlers. Listing 2.3 shows a BPEL skeleton. A process tag contains the whole definition of a BPEL process, wherein various header information and the actual BPEL process itself, i.e. a set of activities, are specified.

While *partnerLinks* define the roles of the single participants, the *variable* tag encloses the definition of variables required to represent the state of the BPEL process. *CorrelationSets* define properties of conversations; thereby, asynchronous messages can be correlated with each other so parts of their content can be associated. An example is given in Figure 2.11.

```

1 <process name="SellThings" targetNamespace=...
2   xmlns=... abstractProcess="no">
3
4   <partnerLinks> ... </partnerLinks>
5   <variables> ... </variables>
6   <correlationSets> ... </correlationSets>
7   <faultHandlers> ... </faultHandlers>
8   <compensationHandlers> ... </compensationHandlers>
9   <eventHandlers> ... </eventHandlers>
10  <!-- (Activities) -->
11
12 </process>

```

Listing 2.3: BPEL skeleton.

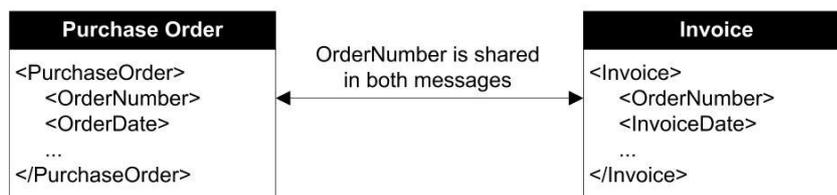


Figure 2.11: A correlation set in BPEL.

Here, the messages of both processes are correlated so that the variable *orderNumber* is common to both processes. While *faultHandlers* define actions performed in response to a predefined error, *compensationHandlers* specify compensation actions for irreversible errors. Event handlers specify activities triggered by specific events and have to be processed by another process executed concurrently. Finally, structured and basic activities describe the workflow itself.

Thus, BPEL is workflow based and its composition capabilities are characterized by the enabling of flexible integration of services, the ability of BPEL processes to be offered as services themselves, and the support of implicit life cycle management [96]. Flexible integration is realized through four characteristics. First, compositions are abstract, i.e. services are composed using their interfaces. This procedure allows the transparent exchange of service implementations without requiring adjustments of the service composition description itself and is referred to as late binding; it enables loose coupling of single services. Second, partners can be bound to the client at different times: at design time, deployment time or runtime. Third, through the availability of assignments in activities enables the manipulation of results of service for further use. Lastly, inlined logic or extensions, e.g. WS-BPELJ,

may further address services' incompatibilities. In this context, life cycle management means the management of composition instances and the ability to allocate inbound messages to the correct instances. At design time, the developer defines the events, i.e. incoming messages, that lead to the creation of new instances. Thus, life cycle management is implicit. The allocation is realized through common transaction mechanisms, e.g. IDs or tokens that are passed on with the messages. BPEL processes only communicate with Web Services, one of the main disadvantages of BPEL. BPEL processes' lack of ability to interact with humans led IBM and SAP to publish the BPEL extension BPEL4People, which covers interaction with humans. BPEL4People was first introduced in July 2005 by IBM and SAP in a joint Whitepaper. The problem that came up during the ongoing work on WS-BPEL 2.0 was its assumption that interactions are restricted to Web Services. People often participate in the execution of business processes and require the introduction of additional patterns. BPEL4People is defined on top of BPEL, so that its features can be used together with BPEL core features [3].

2.3.4 *Choreography Languages*

Choreographies provide a more abstract view on service compositions than orchestrations, tracking public, i.e. globally visible, message workflows among multiple participants, rather than the workflow of one specific business process. It specifies rules for interactions, agreements among multiple or all parties involved, and has a more collaborative nature than orchestration. The relation between choreographies and orchestrations is abstractly depicted in Figure 2.12. While orchestrations specify the internal processing of tasks by one or multiple services arranged within a workflow, choreographies define the message exchanged between single orchestrations, thereby enabling cross-enterprise collaboration.

The Web Service Choreography Description Language (WS-CDL) managed by the W3C is an XML-based language that describes Web Service peer-to-peer collaboration. Contrary to BPEL, it focuses on Web Service Collaboration [142]. It defines the common observable behavior from a global view, i.e. the ordered message exchange between all involved services [94]. WS-CDL does not distinguish between observable messages from applications and infrastructure based signaling. Here, WS-CDL describes the ordering rules for the messages, which dictate the order in which they should be observed. In contrast to successful orchestration languages such as BPEL, WS-CDL is not explicitly bound to WSDL. Thus, WS-CDL can be used to

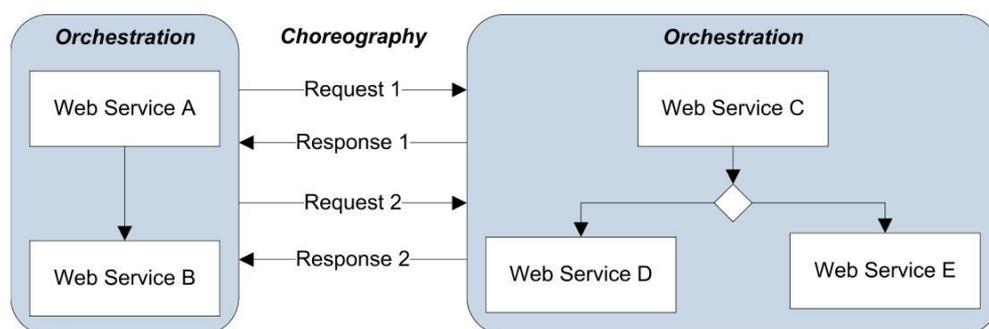


Figure 2.12: Relation between choreographies and orchestrations.

describe a global model for services with no WSDL descriptions as easily as describing services that do have or will have WSDL descriptions.

WS-CDL is a layered language that provides various means and levels of expressiveness to describe collaboration between multiple services. Here, WS-CDL focuses on the abstract description of message exchanges between different parties instead of providing the means to execute a set of collaboratively acting services. All choreographies described in WS-CDL include a number of types that need to be defined; a list of them is given in Table 2.4.

Features	Description
informationType	Describes general messages in interaction and variables.
roleType	Defines a service's behavior (e.g. by a interface descriptions such as WSDL).
participantType	Groups roleTypes into a physical representation of a service (e.g. aggregating multiple service descriptions into one location).
relationshipType	Describes how roles are connected and thereby defining the static linkage between roles.
channelType	Describe communication links and their constraints between roleTypes. Instances of such channelTypes can be used to interact between several roleTypes.
token	Alias for informationType.
tokenLocator	Describe the derivation of attributes from their carrying informationType.

Table 2.4: WS-CDL: Types.

Based on these type definitions, choreographies that build upon them can be specified. Beside the attribute's name and root (which defines a chore-

ography's entry point), WS-CDL also features a set of structured activities that can be used within an choreography description; these activities are summarized in Table 2.5.

Activities	Description
interaction	High-level realization of a collaboration between roles.
workunit	Set of interactions between two roles.
sequence	Defining a sequence of interactions.
parallelisation	Specifying interactions that can be executed in parallel.
choice	Specifies a choice between multiple actions.
assignment	Defines the assignment of a value to a variable.
silentAction	Used to hide certain values or conditions that give rise to one path being taken over another.
noAction	Required in case a message receiver should wait without performing any specific action.

Table 2.5: WS-CDL: Activities.

In addition to type definitions and structured activities to define choreographies based on the previously defined types, the specification of an *exceptionBlock* enables WS-CDL to deal with errors while a *finalizerBlock* can be used to finalize a choreography.

2.4 DISTINGUISHING MASHUPS AND BUSINESS PROCESSES

Business processes and Web mashups both constitute composed applications. However, they are created with different requirements. While business processes are tailor-made to describe large-scale business transactions among multiple enterprises, mashups are rapidly created Web applications for end-users aiming at a high degree of interactivity and graphical presentation, where content from 3rd party providers is also used to add value to the created Web application.

The fact that mashups are built on top of the Web protocol stack while business processes are designed for SOAs is also reflected in the underlying technologies for remote service invocation and service interworking shown in Table 2.6.

Moreover, Table 2.6 outlines the focus of both technologies. Business processes are defined by service composition languages such as BPEL, and provide late binding mechanisms for services that are described by languages like WSDL and stored in repositories where they can be accessed via UDDI. Enterprises can thereby modify the implementations of their single services

Feature	Business Processes	Web Mashups
Remote Invocation	Web services (WSA), XML-RPC	XMLHttpRequest (Ajax), JSON RPC, COMET
Service Interworking	.NET, J2EE	EcmaScript (Java Script)
Service Composition	Yes	No
Composition Languages	BPEL4WS, WSFL, WSCDL	
Composition Execution Engines	Bexee, Oracle BPEL Process Manager, IBM WBISF	
Late Binding	Yes	No
Service Description Languages	WSDL, WSDL-S, OWL-S	
Service Discovery	UDDI	
Interaction	No	Yes
Presentation		HTML, CSS, XML, Streamed audio and video
User Input		HTML Forms, XForms, JS Keyboard/Mouse Event Models

Table 2.6: Business processes vs. mashups.

(e.g. upgrade them to a new version) without any need to notify service consumers of their changes since the services are incorporated through their descriptions only and thereby decoupled from actual service implementation. Mashups on the contrary do not provide any kind of controlled service execution or late binding of services. Instead, Web developers directly integrate the concrete APIs of 3rd party providers into their mashups. In return, mashups provide a connection to the user by default: Web applications are built through a Web page that consists of HTML, CSS and Java Script and therefore possess a presentation and means to interact with the user. Application logic is only invoked when an event created by the user triggers a certain functionality.

The following can be concluded: While business processes enable software developers to create the application logic of a process without dealing with its graphical presentation and possible user interaction, mashups are created as interactive graphic presentations that react to user input and thereby invoke small parts of application logic to modify the presentation of the application.

CHAPTER 3

TOWARDS DYNAMIC AND DISTRIBUTABLE WEB APPLICATIONS

This chapter discusses related work within the scope of dynamic Web applications and proposes novel concepts to overcome identified shortcomings.

Sections 3.1-3.3 deal with the central shortcomings and limitations of today's Web mashups and discuss them based on related work; every section concludes with a proposition to overcome the outlined limitations. Section 3.4 then introduces an embedding of the previous propositions towards an architecture for dynamic and distributable Web mashups.

3.1 RESOURCE INTEGRATION AND ORIENTATION

3.1.1 Motivation

Section 2.4 highlighted the differences between service compositions within the domain of Service Oriented Architectures and the Web, respectively. Service composition languages provide the means to enable the paradigm of *programming in the large* and to abstract from concrete service implementations. Web applications, in contrast, do not possess such high-level programming languages, but feature a layer for graphical presentation and related user interaction. The first key shortcoming of today's Web architecture with regard to the support of dynamic Web mashups is thus the lack of a high-level description language for service compositions that not only supports the dynamic integration of services but also provides a mapping to a graphical presentation.

3.1.2 Discussion of Related Work

3.1.2.1 Service Composition and Late Binding Concepts for Mashups

In [140], C. Pautasso extends WS-BPEL 2.0 to support RESTful Web Services. He states that many assumptions made by existing languages for service compositions are not tenable anymore because of the following con-

ditions: first, most *RESTful* APIs do not use WSDL for service descriptions, making it impossible for most workflow languages to be applied directly. Second, languages that assume static bindings to communication endpoints do not cope well with the dynamic and changing set of *RESTful* services' URIs. In addition, the assumption that XML data will be dealt with is not applicable anymore, since REST services may well use JSON or other formats for data exchange. He argues that process-based languages can also be applied to *RESTful* services, resulting in service compositions that encompass both REST services and conventional Web services.

The extension of BPEL to include *RESTful* services may be sensible for extending existing SOAs, but it is not viable from a mashup point-of-view. Mashups have a different focus from SOA: whereas SOA service compositions languages such as BPEL focus on describing processes, i.e. workflows, mashups focus on resources, data integration, user-interaction, and data presentation. Therefore, BPEL and other service composition languages based on SOA are not suitable for mashups [181].

The approach taken by Pautasso is more relevant for enterprises that seek composed applications with characteristics close to current SOA implementations in terms of their leverage on the WS-* protocol stack. For instance, the approach ensures that security can be considered through the WS-Security specifications or message reliability through WS-Reliability. Nevertheless, an extended BPEL does not support composed Web applications' developers who seek to implement lightweight applications with a focus on presentation and user-interaction.

In [59] El-Gayyar, Alda and Cremers introduce TailorBPEL, a framework that enables users to adapt workflows during runtime in a personalized manner based on the ActiveBPEL engine. ActiveBPEL is an open source BPEL engine [184]. In principle, the framework consists of three layers: a client layer, an orchestration layer and a service provider layer. However, TailorBPEL focuses only on the first two layers, based on the assumption that services are described with WSDL and can therefore be orchestrated with BPEL. This assumption is justifiable in business environments, but it ignores Web technologies. While REST services can be described by WSDL 2.0 descriptions, services based on other formats common in the Web, e.g. Ajax, RSS and Atom, are completely ignored. Admittedly, the authors do not position TailorBPEL as a workflow engine for use with Web technologies, but since the user interface itself is based on Web technologies, in particular Ajax, a logical step would have been to include them. Their argumentation is based on the assumption that users want to be able to customize service workflows, but it ignores the probability that users could also want to include services that are not WS-* based. It therefore might be a reasonable

extension for BPEL engines in business environments, but it misses out on the convergence of SOA and the Web.

The DIANE (Dienste In Ad-hoc NETzwerken) project of the *University of Jena*, which is based on the service-oriented paradigm, “aims at developing and evaluating concepts that allow for an integrated, efficient, and effective use of resources in the form of services in ad hoc networks” [47]. In order to reach the goal of enabling ad-hoc service compositions, a language to semantically enhance the description of services named DSD (DIANE Service Description) was developed [98]. Based on that, service description language concepts for automatic service discovery, composition and invocation were developed [101]. Moreover, a service-oriented middleware was developed to cope with two different kinds of distributed service compositions: first, those with predetermined compositions, therefore statically bound services in their context. Second, compositions without a priori knowledge about the configuration, i.e. those only the desired effects were known for. Mechanisms for automatic service compositions as well as for late binding were necessary. Automatic service composition and late binding are important aspects of the project. Nevertheless, the focus of the DIANE project is on ad-hoc service networks in a SOA environment. Although the architecture presented incorporates concepts from SOA, e.g. service descriptions and late binding, it aims to develop Web applications. Therefore, a WS-* compliant SOA environment is not developed.

3.1.2.2 Abstraction Concepts for Web Applications

In [114] Maximilien et al. address the very problem of abstraction. They state that two paradigm shifts are currently occurring in the Web. The first is the increasing availability of APIs, which results in the increase of mashups. The second is a move towards dynamic programming languages and frameworks such as the *Dojo Toolkit* [79], *jQuery* [155], *Ruby on Rails* [160], *CakePHP* [31] and several others that lead to high-level abstractions. They argue that these two paradigm shifts are complementary in many ways, but frameworks that focus on abstraction, creation and deployment of mashups are missing. Specifically REST, SOAP, RSS and Atom expose its interface differently. In addition to that, common issues of distributed systems emerge. The proposed domain specific language (DSL) called Swashup for service mashups based on *Ruby on Rails* addresses some of the issues. Particularly, a shared interface representation for diverse service types was created, asynchronous and synchronous method invocations are supported, a uniform model for service data and service operations was developed and basic caching mech-

anisms were included. The authors state that differences between Swashup and mashup tools are found primarily in the focus: whereas *Yahoo! Pipes*, *QEDWiki* and several others try to make service composition in the Web iterative and collaborative, Swashup aims to provide common structures and a language for mashup creation and sharing. When comparing to service compositions in SOA via BPEL or other workflow languages, the authors deny similarities because BPEL is not geared towards mashups and Web applications in general.

Although Swashup introduces a way to abstract from different API interfaces, it still requires the developer to specify the service that should be used instead of introducing late binding mechanisms that select a service depending on the functionality a developer wants a service to have. Also, Swashup does not cope with services invoked via JavaScript APIs, therefore avoiding the client vs. server-side issues that emerge if those services are considered as well.

The Web Mashup Scripting Language (WMSL) [163] addresses two problems. First, there is no unified approach for code generation. Second, the mediation between different XML schemas is not supported by common mashup tools. WMSL aims to enable end-users to create mashups that integrate several services. To realize this, metadata and JavaScript code is added directly into the HTML source code. The mapping between different XML schemas is realized with mapping relations specified in the metadata.

Although WMSL has other objectives than the architecture presented here, the goal of abstracting from different XML schemas services return is inherently addressed by both approaches. Whereas WMSL overcomes differences by embedded metadata, the architecture presented here not only unifies the different return schemas, but also provides a mapping from abstract to concrete services that can be instantiated with multiple programming and description languages.

3.1.3 *Novel Idea*

None of the presented approaches has developed a model for the description of Web mashups that builds upon common service composition languages supporting a late binding mechanism, and provides means for the direct derivation of a respective graphical presentation.

This thesis proposes a definition of an underlay system for Web mashups that supports the structured execution of Web mashups based on a high-level programming language. Here, the basic concept of resources defined within the scope of the Web architecture [62] is overtaken in order to realize

a mapping from the mashups application logic to a graphical presentation. Every service is considered as an operation on a resource, where resources can possess multiple representations that serve as a basis for the graphical presentation of the application's current state, and provide means to interact with end-users.

Services within the underlay system are considered abstract, so that they can be replaced by concrete services dynamically. This proceeding enables the dynamic integration of resources residing on different user devices, since the application developer can then abstract from the location of a single service. For instance, the underlay system encompasses a service that derives the user's current position. Since this functionality has only been included conceptionally, the abstract service can either be replaced by a Web service that estimates the user's location based on his or her WLAN cell, or by a service hosted on a user device that has access to a local GPS device.

This abstraction thus enables the deployment of the same application into different environments, where different sets of devices and services are present. In addition, this concept supports the replacement of a service during runtime, in the case that a service becomes unavailable because its host device has left the other devices' connection range.

3.2 ENABLING DYNAMIC CREATION OF SERVICE COMPOSITIONS

3.2.1 *Motivation*

In general, the creation of applications is considered to be a software developer's task. However, due to the increasing presence of social platforms within the Web, user-generated content has gained importance. This development raises the question whether common end-users are as capable of generating services and applications for the Web as they are capable of generating content.

There are various approaches to support common end-users during the creation of Web applications. While a certain class of these approaches focus on the creation of a facilitated, intuitively accessible user-interface, a second class of approaches concentrates on the automatic creation of composed applications, based on a given user request. Latter approaches often deal with the determination of an optimal solution for a given request.

Since the automatic creation of applications is a complex problem, the contribution of this thesis within the field of automatic service composition concentrates on the automatic composition of services under the assumption of a fixed time constraint. Thus, a user requests the automatic creation of

an application fitting his or her needs, while defining an upper time limit for the system to come up with an appropriate solution.

3.2.2 Discussion of Related Work

3.2.2.1 Tools for Mashup Creation

Various tools that aim at the facilitated creation of mashups exist. Schroth and Christ categorize available tools according to three types of platforms [166]. First, those that facilitate syndication of gadgets with no connections between these gadgets, e.g. the iGoogle Website¹. Second, those that allow the connection and processing of resources that are compliant to certain standards, e.g. RSS or Atom, such as *Yahoo! Pipes* [148]. These tools usually collect data from syndication feeds and apply filters to them, therefore basically filtering the output. At best, geo-coded data can be positioned on a map as *Yahoo! Pipes* allows. Yet, even the sources that can be used are often restricted to those that belong to the providers, limiting these tools significantly. Finally, only few platforms exist that allow the mashing of data on an application logic level, such as *Kapow Technologies* [93].

Only the approaches falling into the third category can be considered as related to the approach taken in this thesis, since they provide a model for services and means to integrate functionality on service level. The tools that fall into the first and second category are of limited use for developers, because they either do not enable or concurrently limit the possibilities to mash data. The tools of the last category, however, offer the ability to integrate several sources and create application logic on a high level. Nevertheless, they require the developers to agree at least on the service providers used, and often restrict the possibilities for user-interaction and presentation possibilities.

Xuanzhe et al. propose a mashup architecture that extends present SOA models to incorporate mashup elements. They present a mashup component model that is supposed to help developers create compositions in a service component manner [211]. In their work, the conventional SOA roles service provider, registry and broker and client are extended to help handle mashup components. First, a Mashup Component Builder (MCB) that is responsible for creating Mashup Components (MC) for available services is introduced in order to map the different data formats of the various available services onto a proprietary JavaScript-based format used within their framework. Xuanzhe et al. distinguish three different subcomponents: UI components, a

¹<http://www.google.com/ig>

set of widgets for the browser, service components representing data manipulation interfaces, and action components that act as connectors between service and UI components. Second, a Mashup Server (MS) is composed of three elements: a service catalogue – basically a registry, a repository that stores the Mashup Components, and a monitoring tool. Finally, the mashup consumer is the client that uses the available components to create mashups.

Their argumentation is based on the assumption that mashups are compositions at the user-interface level only, whereas current SOA service compositions reside at the logic level. Whereas this assumption may be true for current mashups, they disregard the direction that service compositions on the Web may take in the future. Moreover, while assuming only superficial UI compositions, they do not provide an easy way to create mashups. Their model should have been aimed at developers with comparatively low programming skills. Nonetheless, it does require significant insight into their component model.

The presented model contains a service registry and repository for service discovery. However, the advantages of dynamic service integration are not leveraged. Developers can choose the components they want to use, but cannot abstract from actual services. In essence, the possibilities their component model offers are not thereby exploited. This can be explained with the example given in the paper. A mashup of the tracking service of *Fedex* together with a *Yahoo!* map is given. While the *Fedex* service can obviously not be replaced if a package delivered by *Fedex* needs to be tracked, the developer might be indifferent to the map provider. Still, he has to decide on one distinct map provider, likely opting for the best known one, and not the one best suited to his purposes. Both, the relatively complex usage and the missing abstraction could be addressed in future extensions.

3.2.2.2 *User Interfaces for Service Compositions*

In [175], Spillner, Braun and Schill analyze and extend WSGUI (Web Services Graphical User Interface), a set of concepts that provides the means to dynamically generate user interfaces in the context of Web Services by enriching the data model schema information. With Project Dynvocation an implementation of the concepts exists [151], which was first introduced by Spillner et al. in 2006 [174]. Spillner, Braun and Schill suggest that additional concepts for user interface generation and dynamic Web Service invocation may help the adoption of SOA for usage in the consumer domain. Nevertheless, until the end of 2008 the concepts relied solely on services describable with WSDL. Just recently, additional efforts were announced that

suggest that services described with WADL will be included [151]. With this additional support for WADL, common services in the Web may be included. Although Spillner et al. promise much more appealing user interfaces, the concepts still depend completely on conventional SOA concepts for service composition that have failed to be adopted in the Web not only because of the lack of feasible presentation possibilities, but also because of their complexity [167, 166].

In 2005, I. Gavran introduced the Simple Service Composition Language (SSCL) [66] – a statement-based programming language developed as part of the Programming Internet Environment (PIE), a framework for the design, development and execution of distributed Web applications [150]. Based on the assumption that the existing development infrastructure is insufficient to meet the increasing demands of users, the framework aims at consumer-level programming. Consumers themselves can then combine the large number of building blocks provided into more complex applications, so-called gadgets. The gadget composition is based on HUSKY, a spreadsheet-based methodology covered in [170]. This builds on a coordination layer in which the aforementioned SSCL is used. SSCL eventually translates to common service compositions based on CL (Coopetition Language) [171], a composition language based on BPEL and WSDL. Additionally, the constraints of service compositions based on orchestration, multi-agents and choreography are approached by a tree-tiered distributed architecture for service composition execution called USCA (Uncoupling Service Composition Architecture) [172].

Whereas the framework takes steps towards the easy creation of mashups and distributed Web applications, it does not address the heterogeneous types of services available in the Web. The iGoogle-based Gepetto user interface addresses the need for easy creation of distributed Web applications with the combination of building blocks in the form of gadgets. The underlying SSCL together with CL allows developers to create service compositions without knowledge of BPEL, WS-CDL or other workflow languages. Nevertheless, the architecture does not scrutinize RESTful services or JavaScript APIs. Consequently, late binding for Web applications is not addressed by PIE or the underlying technologies. Although similar issues are addressed, the architecture introduced later has a different focus.

3.2.2.3 Towards Automatic Service Composition

The dynamic creation of service compositions addresses the facilitated or (partially) automated creation of service compositions. Related approaches commonly strive for a reduction in composed application's development costs

of or for an increase in flexibility with regard to their responsiveness to failures, i.e. to enhance their ability to recover partially failed services.

The creation of service compositions can be achieved by directly assembling concrete services or by the composition of abstract services, where every abstract service is replaced by a concrete one during a second step. This proceeding has already been illustrated in Figure 2.1 of section 2.1. Within the remainder of this section, the intermediate step of abstract services is neglected, since it does not have an influence of the composition algorithms and strategies itself.

3.2.2.3.1 Classification of Service Composition Approaches

Figure 3.1 gives an overview of the classic service composition approaches based on [182]. Here, service composition approaches are divided into static and dynamic solutions. Static approaches assume a manually-created description of the service composition, which is bound when invoked by a request. After its execution has been started, monitoring and maintenance mechanisms enable adjustments of the composition on parameter level. Dynamic approaches enable either dynamic service discovery, binding during runtime or dynamic modification of the current service composition plan. This dynamicity is considered as a fundamental prerequisite to enable autonomous service composition, providing adaptability of compositions on a deeper level than the adjustment of predefined parameters.

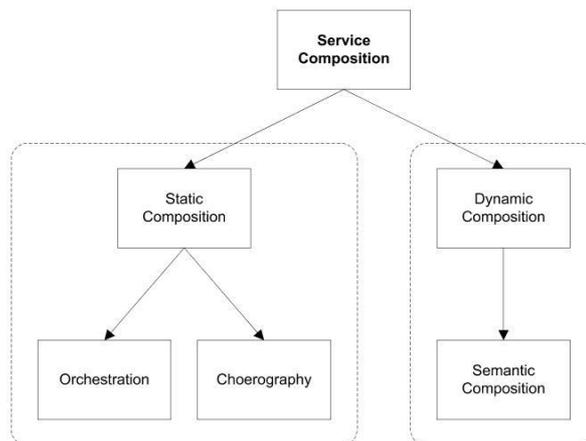


Figure 3.1: Classification of service composition approaches.

The ability to automatically compose services is an essential step in reducing development time and costs. The problems of automatic composition

of services are derived from several aspects. First, usually complex applications lead to a high level of interwoven processes. Second, the statelessness of web services poses a problem when creating complex services, very similar to the problems that had to be overcome in the Web to create complex applications. The quality (generation of errors, execution time of the scripts, etc.) of generated services is also an important aspect [111].

SOAs commonly provide means for static service composition, i.e. define service composition languages such as WS-BPEL as introduced in section 2.3.3 in order to develop composed applications manually. This concept is often referred to as “programming in the large”, demonstrating that the development of composed applications is still a programming task, though it is achieved on a higher level of abstraction. Within the following section, approaches towards the automatic creation of services composition are discussed.

3.2.2.3.2 *Automatic Service Composition*

The following distinguishes between semi-automatic and automatic composition of services.

Semi-automatic service composition commonly requires manual input to support decision making processes during composition time [105]. Sirin et al. proposed an algorithm of this type, which is capable of assisting end-users during the selection of appropriate services required within a composition [169]. Although this process is not fully automated, it makes user-profiles and preferences unnecessary. However, services still have to be described semantically, so the most complex problem of the service discovery, i.e. the matching of a request description to a certain service, remains. Today, semi-automatic service composition approaches are mostly found on specific domains, such as a composition algorithm operating on the *BioMoby Semantic Web Framework*, which supports the semantic interoperability among multiple life science services stemming from different scientific domains [48].

The area of automatic service composition approaches has a significant part of its roots in artificial intelligence (AI). Here, classic planning algorithms have been adapted so they can be used to automatically compose services. The planning algorithms rely on external knowledge of the services itself and their relations to each other [55]. For instance, SHOP2 constitutes a hierarchical task planners that operates on a set of tasks. There are primitive, composed, and goal tasks. Primitive tasks can be directly executed, while composed tasks constitute a sequence of primitive tasks. A goal task is a constraint that has to be evaluated as true by a combination of composed

and primitive tasks in order to verify the existence of a composition that fulfills the requested goal task [127].

Most service composition algorithms based upon classic AI planning algorithms employ the concept of forward or backward chaining. Within the domain of AI planning, such algorithms are started with a set of *constants*, *rules*, and *questions*. Here, constants denote information that is always true, while questions encompass the information that should be verified or falsified by the algorithm. The set of rules are applied to either the constants or questions to derive new information. When the rules are applied to the constants, the concept of forward chaining is referred to, otherwise the concept of backward chaining is employed [141]. The same approach can be followed within the domain of services described by their inputs and outputs. Here, services that only possess outputs are considered to be constants, while services that only encompass inputs are regarded as questions. The set of services available to the composition algorithm constitute the rules. Thus, the algorithm links outputs of some services with inputs of other services. Thereby, services are continuously added until the inputs of the question are generated via outputs of services, whose inputs are met by the outputs of the constants in turn. A forward chaining algorithm starts with the given inputs, i.e. the constants, and continuously adds services whose inputs are met by the already generated outputs, until the inputs of the question have been generated. A backward chaining algorithm, in contrast, starts with the question and adds services until all services leading to the creation of the question's inputs can be derived from the constants [34, 141].

Wu et al. have used this SHOP2 planner to show the automatic composability of Web services that are extended by semantic DAML-S [11] service descriptions [206]. Here, the authors derived a method to encode the service composition problem as a SHOP2 planning problem. Notably, these kinds of algorithms only create a sequence of service invocations as output capable of completing the requested task. Complex system behavior such as decision making and looping is not possible.

Many more algorithms have been proposed outside the domain of AI, which were all tailored for a specific type of service descriptions and thus restricted in their usage. Most approaches have already been discussed extensively in various surveys such as [54, 182, 152]. For instance, Sheshagiri et al. proposed a planning algorithm for services relying on DAML-S[11], a language for semantic Web services [168]. Zhang et al. opted for a similar approach, while focusing on the domain of manufacturing grids. Here, the external knowledge serving as the basis for the composition algorithm was represented by a novel ontology referred to as *MGOnto* [213].

Silva et al. also presented an algorithm for the automatic creation of service compositions, focusing on the combination of telecommunication services [45]. Their work relies on complex semantic service descriptions by means of the SPice Advanced service description language for TELecommunication services (SPATEL) [5]. The authors claim to support requests either formulated by end-users in natural language, or by software developers via a well-defined request language. However, the complexity of the request formulation increases proportional to the complexity of the composition problem itself, such that no confirmation is possible as to whether the proposed solution is able to reduce the actual development costs of composed applications.

In [19], Berardi et al. propose an approach for the automatic creation of service compositions by means of simulation. Here, they build upon a composition model introduced by Calvanese et al. [32], referred to as the Roman Model, that specifies services compositions as combinations of finite transition systems, which themselves define the functionality of the single available services. Berardi et al. consider the available set of services as *community* of services. They define a *community transition system* as the asynchronous product of the available services, such that this transition system expresses every possible behavior that can be provided by any combination of the available services. The authors define a service composition as a function that is capable of determining whether a service composition request, given as a sequence diagram itself, can be fulfilled by a set of available services. Therefore, the algorithm takes the requested service composition (as transition system) and the community transition system as input, and checks whether the requested service composition can be simulated by the service community transition system. This approach constitutes an example of the application of well-known model checking techniques to the domain of service compositions. In fact, the authors have defined a product-transition system and checked, whether it is simulation equivalent to the requested system [39]. The approach of Berardi et al. thus provides a way to facilitate the creation of service compositions by developing a high-level system, so that its realizability can be automatically checked based on a given set of services. The approach exposes two general weaknesses. First, it is still assumed that the creator of the application is capable of expressing his or her desired behavior with a finite transition system. Moreover, it has not yet been evaluated, which degree of facilitation can be reached through this kind of approach. Second, the solution is likely to remain theoretical, since one of the assumptions is that knowledge in form of finite transition systems is available for every service that is currently available.

3.2.2.3.3 *Heuristics for the Problem of Automatic Service Composition*

The most simple approach to finding a service composition for a given request is the brute-force search, i.e. the exploration of all possible combination of available services. However, the combinatorial explosion makes this kind of approach highly inefficient. Here, the explosion of the search state space is not driven mainly by the number of services itself, but by the theoretically unrestricted number of inputs and outputs they contain.

Kil et al. highlighted the complexity of the automatic service composition problem [97]. They base their work on the fact that, when the problem is solved based on a realistic model for service descriptions, the solution retrieval is double-exponential [13]. Therefore, the authors proposed an approximation-based algorithm that abstracts from the actual state space of the service composition problem by identifying internal variables of single sub-workflows not connected to parts of other sub-workflows. Here, the authors argue that a solution for the abstracted service composition problem also constitutes a solution for the origin problem. This, in fact, is true, since the paper proposes the development of a simulation-equivalent solution as known for the simplification of finite state machines or finite automata [39]. The simulation results are short and partially incomplete, but show that the composition procedure has sped up with the restricted number of variables. However, the authors focus on classic service composition problems, where the involved services are known beforehand and do not have to be explicitly discovered.

Such kinds of heuristics used by Kil et al. denote strategies to optimize search problems with respect to a certain metric. In most cases, heuristics aim at reducing the search space in order to speed up the composition process. Here, the search space is restricted in general during the composition process itself by guiding the search in a way that the conceivable following steps of the procedure are evaluated and the most promising one is selected.

Heuristics can be classified according to the knowledge sources they access and use for the composition process. Depending on the source and type of knowledge, different objectives with regard to the service composition algorithm's amelioration emerge. Figure 3.2 shows the classification of service composition algorithms with regard to information from the requesting user's side and knowledge on the service domain, i.e. the services itself.

Since heuristics require data to alter the decision making processes within the algorithm towards a more favorable direction, there are not many of them devoted to blind service composition, i.e. composition approaches that neither draw additional information from the user nor from the service domain. These heuristics are generally search strategies for the traversal of the search

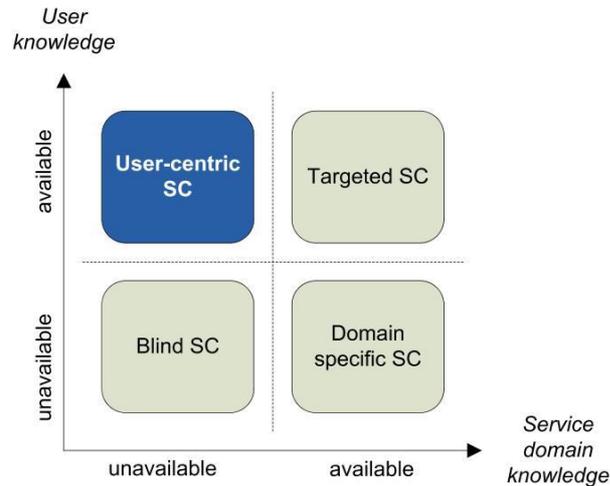


Figure 3.2: Different classes of service composition (CS) algorithms based on available knowledge.

space, which may, for instance, be given as a tree or list. Classic examples are depth-first search (DFS) and width-first search (WFS). The performance of both strategies is considerably affected by the size of the search space. The former strategy is afflicted with a high number of services, the latter one with a high number of services that can be added to the current service composition at every step [204].

Additional knowledge is used during the automatic composition of services for two main reasons. Either the algorithm's performance should be increased, or the resulting service composition has to possess a certain property, where it is ensured during the composition process that only those compositions that feature the respective property can result from the algorithm.

In the former case, i.e. domain specific service composition approaches, learning algorithms or aggregation mechanisms are employed to derive additional knowledge concerning services that are a good fit, so that those combinations are favored in following composition requests [60, 201].

In the latter case, information from the requestor side can be combined with knowledge on the service domain in order to create service compositions that possess certain properties. This approach is commonly used to ensure Quality of Service (QoS) properties of resulting service compositions.

User-centric service composition approaches, where information from the user side is available, but not on the service domain, have received nearly no attention thus far. In fact, it is a difficult problem to enhance a service

composition algorithm if no additional information on the services and their relation is available. Therefore, user-centric composition approaches often focus on the incorporation of preferences for the algorithm's search strategies [204]. Here, the user has to estimate the potentially optimal strategy for a given domain.

3.2.3 Novel Idea

Automatic service composition is a much covered research topic. However, novelty with regard to the composition procedure itself often lies in the usage of different semantic service descriptions and not the creation procedure itself. Many scientific publications concentrate on the semantic matchmaking process – i.e. on finding appropriate services for a given request or task description – but not on the creation of the control flow of the composition, which is often assumed to be already in place [32].

Algorithms dealing with the automatic creation of such control structures focus on two general problems. Some research groups concentrate on the reduction of the automatic creation process' complexity, since the problem has been proven to be double-exponential hard [13]. Most of the attention has been paid to finding optimal solutions for a given problem. These algorithms therefore have to be complete and computable, i.e. have to terminate in every case with either the solution or an *unsolvable* message as a reply.

Because of the composition problem's complexity, respective algorithms are consuming a lot of time in order to derive a service composition for a given request, especially when an optimal solution is requested. This makes the algorithms inappropriate for spontaneous usage by end-users. Within the scope of this thesis, the specification of a service composition algorithm enabling the rapid creation of small or medium scale Web applications is focused on. Here, a user-centric algorithm –as illustrated in Figure 3.2 and discussed above– is introduced that strives to enhance the composition algorithm based only on additional information from the user's side. The algorithm takes the upper time limit given for the creation of a service composition for a certain request into account, and adapts during the composition process in order to increase the odds of finding a solution within the given time span. Thus, the algorithm tries to solve the composition problem in a time-dependent, best effort manner instead of looking for an optimal solution within the complete search space.

Instead of relying on a specific service description language, the presented algorithm works on abstract structures, thereby defining the information that is required for the composition process. The algorithm therefore remains

independent from the concrete service descriptions and can be instantiated by using a service description of one's choice that is capable of expressing the abstract structures on which the algorithm operates.

3.3 DISTRIBUTED WEB MASHUPS

3.3.1 Motivation

Today's mashups are tightly bound to the client/server architecture of the Web. Here, one or more servers act as service and content providers for a client. This separation of roles hampers the integration of services residing on client side devices considerably, since a client has to act as a service provider in order to provide its service capabilities. Thus, a mechanism is required that enables clients and services to communicate bidirectly independent from their roles, so that different parts of the mashup can be executed on and controlled by multiple devices. The distribution of application logic entailed by this requirement can, at the same time, enhance the Web mashup's dynamicity with regard to offline capabilities and assurance of privacy concerns. If the application logic for a part of a mashup is executed on a single device, the respective part can be accessed without needing a connection to the Web. Moreover, private user data processed only by services residing on the user's device does not have to be transmitted over the network and is thereby better protected from misuse.

3.3.2 Discussion of Related Work

3.3.2.1 Communication and Resource Synchronization for Distributed Workflows

Andrews [9] discusses a wide range of resource synchronization mechanisms. Processes can communicate either directly by exchanging messages, or indirectly by reading and writing shared variables. They can also synchronize in two basic ways: directly by explicit signaling, or indirectly by testing and setting shared variables.

Synchronization and communication mechanisms can be divided into three classes: monitors-, buffered-messages- and synchronous(unbuffered)-messages-based. The first approach is not suitable for distributed systems, because it needs a memory area which can be shared among multiple communicating parties. Buffered messages are often employed for process interaction mechanisms, where the processes are able to work in a self-contained way for most of the time, so that synchronization is only rarely performed by pass-

ing messages. Here, messages interaction only takes place when either one party attempts to send a message when the buffer space is exhausted, or when a party attempts to receive a message and none is available. Systems based on synchronous message passing are based on direct unbuffered message exchange between communicating parties. Examples are communicating sequential processes (CSP) [78] and distributed processes (DP)[74]. For instance, within CSPs, messages are based on I/O commands. Here, a part of a process can be blocked by waiting for an output command of another process that matches the process' input command. Andrews states that each of parallel programming languages using synchronous message passing for communication controls synchronization by statements based on Dijkstra's guarded commands [50]. This means that using Boolean expressions like semaphores or conditional variables is preferable to explicit signals.

Nanda and Karnik [126] analyze coordination problems in decentralized orchestrations of Web Services. The analysis operates on three types of graphs:

- Workflow Dependence Graph (WDG): Describes a decentralized and concurrent orchestration.
- Control Flow Graph (CFG): Describes the workflow specification.
- Program Dependence Graph (PDG): Describes the control and data dependencies between the single components of a CFG.

Each node within a CFG describes a task that can be instantiated by a component, a Web service, or any other form of application logic. Given a start and an end node of the CFG, tasks can be executed according to the order defined by the CFG's edges until the end state is reached. A related PDG possesses nodes labeled with predicate expressions from the respective CFG and edges containing information on the single tasks' dependencies with regard to data and control flow.

The system's starting point is a centralized workflow specification given as CFG, from which a PDG is build, which is then partitioned to generate a WDG respectively as described in [125]. The term synchronization in the context of this paper means communication between nodes, i.e. between distributed workflow engines. The order and number of exchanged communication primitives between two nodes depends on the number of the receiver node's incoming edges. More than one incoming transition denotes the presence of concurrency. Depending on the WDG structure, two types of concurrency are identified for sequential code and five types for branched code. It is stated that concurrency in an iteration can be reduced to a combination of the above. Additionally, an algorithm that uses static analysis to

detect different forms of concurrency is introduced. To define during which points of the workflow's execution communication takes place between the single nodes, the following protocols are presented:

- Direct Deposit: sender deposits the data directly into the receiver's buffer.
- Request-Response: receiver pulls data after a control message that indicates the availability of data is sent.
- Combined Direct Deposit and Request-Response: a combination of the two above.
- Pipeline: data are pipelined to minimize the amount of necessary connections.

Experimental results show that there is no protocol that will perform best in all possible networks and with various concurrencies types.

Eder et al. [56] have discussed the problem of data synchronization in workflows. They claim that this issue has not yet garnered vast interest in the literature and it is often presumed that data used by a workflow instance cannot be changed by another instance or by an external application. Thus, workflow designers have to find some workarounds; possible methods are:

- developing a guarantee for all activities within a workflow, that data is changed in a way that may invalidate it,
- implementing condition checks in activities,
- defining pre- and post-conditions,
- rechecking data before critical actions,
- locking data for the execution time.

According to most authors none of above methods solves the general problem, but each of them may introduce new ones, e.g. locking is unacceptable if workflow executions take a very long time (e.g. months). They present another solution by introducing the concept of workflow data guards, a constraint defined over specified parts of a workflow. By means of these data guards, it can be ensured that certain changes in a data set are recognized by the system in order to guarantee the accuracy of a workflow's execution.

UPPAAL [191, 102] is a real-time model checker for timed automata based processes. Timed automata may be composed into a network of timed automata, where all members share a common set of clocks and variables. In order to achieve interoperability, UPPAAL extends timed automata with the concept of channel synchronization. There are two fundamental types of channels: binary and broadcast. In the first case synchronization messages are sent between exactly two members, in which both sender and receiver block until the synchronization is accomplished. In the second case

the sender may synchronize with an arbitrary number of receivers, in which only the receivers need to block. The synchronization can be considered as an exchange of control messages. Since each timed automaton has the same view on global variables, if one writes a variable, the update is immediately noticed by other automata and consequently there is no need to exchange any data implicitly.

3.3.2.1.1 *Offline Modes*

Nam et al. [124] introduce a distribution method and a Web application model to enable their execution in offline-mode. Four main requirements were identified. First, an application has to be easy to install and start in accordance with the "click & run" principle. Second, movable storage devices must be portable. Third, the application should be offline-capable through data synchronization. Last, it should be possible to develop a GUI easily by using XML-based languages.

In addition, a classification of Web applications has been made with respect to Internet connectivity and their characteristics. Considering the first aspect, *online* and *offline* with or without occasional synchronization classes are isolated. With respect to the second characteristic, applications are divided into *social* and *personal*. There are also some applications that can work both in online and offline mode, and manifest personal as well as social characteristics, e.g. Wikis or Blogs. Offline-based personal applications are considered to be the best fit for offline applications, since they are characterized by occasional connectivity and social characteristics. The publication presents a framework architecture that enables the creation, deployment, and execution of Web applications satisfying the requirements specified above. According to the authors, it demands only minimal modifications to existing software. Developers can build such software in the same way that the existing applications are made. After that, configuration files as well as initialization and launch scripts need to be created and packaged together with application files for installation.

The offline mode of a Web application's execution is realized via the Plain Old WebServer (POW) approach, which integrates a light-weight Web Server and embedded SQLite database into the Web Browser.

Gears [68] is a widely-used open source framework which extends a Web browser with the ability to store data locally. Web browsers' limitations with regard to restricted file system access are thereby overcome; it offers a lightweight alternative to the solutions introduced above, where databases are used at the client side to store local data.

The W3C also introduces a mechanism to execute Web applications offline [29]. The specification concentrates on the packaging and the configuration of Widgets. The term Widget is defined as “an interactive single purpose application for displaying and/or updating local data or data on the Web, packaged in a way to allow a single download and installation on a user’s machine or mobile device.” Thus, a Widget must always be installed in order to be executable on a certain device; a runtime environment is also needed to run downloaded Widgets. In general, Widgets do not constitute a novel category of Web applications, but rather a novel mechanism to package and deliver Web content.

3.3.2.1.2 Trust

Smetters [173] summarizes the data privacy problems that arise in Web 2.0 environment. He argues that they are magnified by spread of mashups, which combine data from several sources. In some cases mashups may reveal sensitive data by combining information from multiple providers, even though each of them does not seem to be meaningful on their own. Examples of this issue can be found in [120].

According to Smetters, the following policies have been used to restrict access to sensitive data:

- Avoidance: use only public or semi-public data,
- Reduction to a previously solved problem: the service operating on the data is either the service that holds it or one with whom user has existing trust relationship,
- Looking under the lamppost: identify *a priori* collections of data that may be accessed,
- Building a bridge: employ a dedicated financial data aggregator,
- Outsourcing: maintain a relationship with an identity provider or an authorization service.

Howell et al. [80] states that the rush mashup evolution has led to an “inadequate security model that forces Web applications to choose between security and interoperation”. According to the above, an application can either apply *no trust* and isolate itself, or *full trust* through incorporating 3rd party code as libraries. To fill this gap, the authors propose applying operating system principles to Web browsers. This should enable resource isolation and data-only message-based communication between service instances and display sharing.

Jackson and Wang [86] introduce mechanisms for the secure cross-domain communication. The interactions are sandboxed by a hidden IFRAME. This

approach has the advantage that it does not require any browser modifications.

3.3.2.2 Automata Decomposition

An automaton can be considered as a graph with labeled transitions [39]. In the case that a transition is identified with an operation, an automaton can control the behavior of a process by defining the order of the processes operations.

The objective of automata decomposition is to decompose a given automaton into several simple components so that their global behaviour realizes the behaviour of the original automaton [109]. The dependency between the single components is of utmost importance. Dependencies are realized by redirecting outputs from one automaton into input port of another one. Each automata acting independently from each other is referred to as a *parallel decomposition*. On the other side of the spectrum, if every automaton is sensitive to the behaviour of the others, it is referred to as *unbounded feed-back decomposition*. There are also more realistic intermediate interconnection models between those two extremes. Although it is desirable to have a decomposition in which the components are independent from each other, it is rarely possible. The intermediate model intends to minimize the scope of dependencies. One way to do this is to allow interconnections only between automata that are already in the same neighborhood. This presumes that the automata are located in a metric space. This is the case in cellular automata, neural nets and systolic computers.

Another way of limiting the scope of dependencies is partitioning the components into levels. Then components of higher levels may influence those of lower levels, but not vice versa. This hierarchical structure of dependencies is referred to as *cascaded decomposition*.

Work on automata decomposition started in the early 60s. In 1962, Krohn and Rhodes announced a method for the decomposition of automata. The proof for this method was presented in 1965 [99] using the semigroup theory. A semigroup is basically a set with an associative binary operation. A classic example is a sequence of symbols under the concatenation operation.

In contrast, Hartmanis and Stearns dealt with *parallel composition*. Their publication from 1966 [75] is sometimes referred to as the earliest work on systematic approach to decomposition [113]. They present a mechanism that tries to create non trivial orthogonal partitions of the originating automata. Here, two partitions are orthogonal if each pairwise intersection of the elements yields either the empty set or a singleton set. The partitions are

created by merging the original locations. A combination of them realizes the originating automaton. It is an additional requirement for found partitions that each of them has fewer locations than the originating automaton. Thus, the main aim of automata decomposition is to reduce a given system's complexity. This technique can be used to overcome the state explosion problem in program verification.

There has also been notable work on extending the standard decomposition theories of automata to timed automata [7]. For instance, Mason and Krishnan [113] present a decomposition method of timed automata by extending the standard theory of decomposition of finite automata to timed automata by Hartmanis and Stearns [75]. In contrast to previous theories, this paper not only addresses the decomposition of states, but also the decomposition of clocks. The authors found that an automaton has a parallel composition, if there is a different clock partition for each partition, such that the pairs of partition and according clock partitions are admissible.

3.3.2.3 Workflow Logic Distribution

[125] introduces a new code partitioning algorithm, which can be deployed to distribute a Web Service composition. The main objective of this algorithm is to maximize the throughput of multiple concurrent instances and minimize the communication costs. Since the centralized control of service orchestrations has proven to be a bottleneck, a workflow given in BPEL is partitioned into multiple processes, which can be executed in parallel manner. The algorithm operates on PDG representation and tries to find all possible partitions. However, it only merges the portable code, i.e. parts of the logic which do not belong to any of the Web Services. The best solution is found by the means of a cost function. Experimental results show that the distributed execution of a Web Service composition can be increased approximately from 30% under normal system load up to 200% under high load compared to execution at a single instance.

In this approach, distribution takes place at compile time. Moreover it is assumed, that parts of the distributed logic, can be deployed directly on the server that hosts the Web Service.

Liu et al. [106] note a lack of general methods for realizing cross-platform and distributed applications. In order to fill this gap, they present a workflow framework whose building blocks are Web services. It supports loose-coupling of services and is equipped with a distribution module. The framework consists of two components: a process definition tool providing workflow process definitions through a GUI, and workflow engine. The latter incorporates a

process definition parser, a Web service selector, and a Web service executor. The workflow description is given in BPEL4WS and extended with constructs to enable dynamic process definition.

Barret and Pahl [14] describe distribution patterns, i.e. reusable patterns, that describe how a composed system is assembled and subsequently deployed. They allow the modeling of the distribution of a system and thereby facilitate compliance with varying non-functional requirements. They therefore define where components of a distributed workflow should be deployed with respect to the location of Web services that are part of the workflow. This improves maintainability and comprehensibility of the system. The term distribution pattern originates from [205].

Three categories of patterns are identified:

- Core patterns
 - Centralized Dedicated-Hub
 - Centralized Shared-Hub
 - Decentralized Dedicated-Peer
 - Decentralized Shared-Peer
- Auxiliary patterns
 - Ring
- Complex patterns
 - Hierarchical
 - Ring + Centralized
 - Centralized + Decentralized
 - Ring + Decentralized

The centralized dedicated-hub pattern is simultaneously the most frequently applied and easiest to implement. However, the central point of control is a performance bottleneck. In contrast, the decentralized shared-peer pattern provides each cooperation partner with autonomy and reveals only data necessary to others while keeping sensitive data private. The price for lower communication costs is the increased development complexity and additional deployment overheads. The ring pattern provides load balancing and high availability; the hierarchical one facilitates organizations, whose management structure consists of multiple levels. The remaining complex patterns are built up as a combination of the further described, thereupon mixing up their properties.

Distribution patterns make up, alongside workflow control flow [161] and service interaction [15] patterns, the third group of defining building blocks that can be applied to building and analyzing service compositions. All of

them refer to the high level cooperation of components, even though the first one considers workflows as compositional orchestrations. In contrast, the two latter ones deal with compositional choreographies, where only the external messages flow is modeled.

3.3.2.4 Summary

The majority of approaches dealing with the distribution of application logic like [106, 37] are focused on extensions or modifications of the SOA stack; they do not consider integration of *Restful* Web Services or a fit with the REST architectural style. For instance, the approach to time error handling presented in [37] is defined in such a way that it can only be applied to the distribution of BPEL4WS fault handlers.

The area of late binding mechanisms also focuses on SOA technologies, as current approaches within the domain of mashups integrate 3rd party APIs into the service composition description in a fixed manner. The single services thereby cannot be exchanged during runtime, even though the current service might be not the one best suited to the user's requirements. Although both aspects of late binding and application distribution are named in publication of Liu et al. [106], there is no detailed explanation for how single modules are defined or implemented; other publications do not provide deeper information on this issue.

In contrast to the solution provided in this thesis, where services are dynamically integrated during runtime, the distribution approach described in [125] takes place at compile time and cannot be changed later on. Although the paper demonstrates a mechanism to partition a workflow given in BPEL into multiple workflows that can be executed in a distributed manner, the authors dramatically restrict the applicability of their algorithm. In fact, fixed and mobile nodes are differentiated between, where mobile nodes can be dynamically assigned one or another workflow partition while fixed ones cannot. The *invoke* construct required for the invocation and thus the execution of services is considered to be fixed, so that service execution cannot be distributed. The solution proposed in [125] thus does not deal with the distribution of workflows with regard to their functionality, but in terms of their data operations. These data operations are kept within mobile nodes and can be assigned to devices that host at least one fixed node, i.e. service, in order to reduce the number of necessary data updates that have to be communicated between the single involved devices. In that scope, the same authors have published experimental results on the evaluation of effective data synchronization methods between multiple devices [126].

Research in the area of distribution patterns [14] are of only limited use for the domain of Web applications, since it is assumed that parts of the distributed logic can be deployed directly at the Web service hosting server. However, mashups incorporate data and services from 3rd party providers, whose services cannot be modified by the mashup itself.

Although automata decomposition provides the means to partition a finite automata, it does not consider problems related to the distribution of the generated partitions. Moreover, the premises for the automata decomposition algorithms in general differ from the ones within this thesis, since automata decomposition tries to minimize the automaton's structure and not to achieve a functional distribution of automata and respective communication and synchronization methods.

In general, the main focus relies on the proof that each automaton can be built up from and decomposed to some basic structures. However, this approach of an arbitrary decomposition and thus distribution is not compatible with the one striven in this thesis since the locations of services, which take part in the composition, are predefined [75, 109].

No work was found on the distribution of a structured application execution between client and server, although it brings the advantage of truly parallel execution. Moreover, it enables binding of local services to meet abstract functionality descriptions given in service composition. In addition, it might increase privacy by allowing only the local services to operate on sensitive data.

Service recovery mechanisms in case of failures is either absent in the literature or only handled coarsely like in [37]. Even workflow data guards as introduced in [56] only guarantee that the guard conditions defined on data are not violated without being noticed. Consequently, when they are violated the developer has to take care of the situation by him- or herself.

Current approaches to extending Web applications with offline capability require the installation of additional software, e.g. a Web browser extension. POW, which was chosen to execute Web application in offline mode by authors of [124], integrates a light-weight Web server and embedded SQLite database into the Web browser. Also Gears [68], which requires a SQLite database, needs to be downloaded and installed. The same strategy was chosen for Widgets [30]. Modifying the Web browser or installing any extensions conflicts with objective of this thesis to guarantee compatibility with the broadest possible range of heterogeneous user devices by relying on standard Web browsers. Moreover, the software should be stored offline seamlessly without user interaction in order to provide the same feeling one has when using the same application online.

3.3.3 Novel Idea

In this thesis, an algorithm is presented that enables the distribution of a given underlay system among multiple devices, so that every device is in control of the locally available services. Here, the distribution algorithm automatically generates communication primitives for messages exchanged between the devices hosting a part of the underlay system, so that the distributed underlay system's execution provides the same functionality as the origin undistributed underlay system.

3.4 AN ARCHITECTURE FOR DYNAMIC WEB MASHUPS

This thesis aims to define an underlay system for Web applications that paves the way towards more dynamic and distributable Web applications. Within the remainder of this section, requirements for an underlay system are discussed that overcomes several shortcomings of today's Web mashups as laid out in sections 3.1-3.3. In addition, an architectural extension enabling the embedding of the underlay system within the mashup architecture is outlined.

3.4.1 An Underlay System for Web Mashups: Requirements

The underlay system itself has to meet five central requirements as abstractly depicted in Figure 3.3.

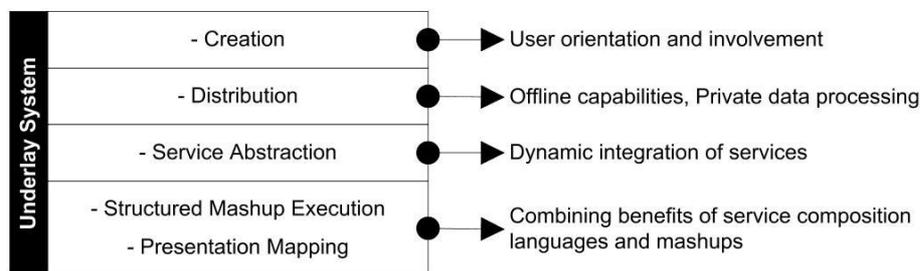


Figure 3.3: Requirements for the underlay system.

First, the underlay system itself has to support the structured execution of Web applications so that applications can be developed on a higher level of abstraction. Here, the underlay system has to bridge the gap between the composition languages available in other domains such as SOA and a user-

centric graphical presentation of the application stemming from the Web domain.

Mobile devices can spontaneously leave and enter the connection range of the user, leading to a changing availability of services. Therefore, the underlay system should encompass a mechanism that can respond to changing service availability and deal with the automatic replacement of services in a self-contained manner.

Instead of incorporating concrete services or APIs, the underlay system has to build upon abstract services so that an instantiation of the underlay system can be performed during the deployment of an application. Here, the underlay system should dynamically incorporate services from the available devices in order to create an application adapted to the user's needs and preferences.

The dynamic integration of services into a given underlay system forms the basis for the distribution of the underlay system, where the single parts of the underlay system should become responsible for the control of all services hosted on the single devices. This distribution of underlay systems promises to support certain offline capabilities of the Web application in the case that the only accessible services are those hosted and controlled on the local device. Moreover, private data does not need to be transmitted over the network when it is only processed by local services, so a certain degree of privacy can be ensured for selected data items.

To make the underlay system accessible for common end-users, there should be a mechanism to derive an underlay system and thus a respective Web mashup automatically from a given request. The goal here is to support common end-users in becoming creators and providers of Web applications in the same way that social networks and applications have enabled end-users to become data providers on the Web.

3.4.2 Architectural Embedding of Underlay Systems

The key aim of this thesis is to leverage access to services residing on multiple user devices and combine their capabilities within value-added Web applications. To ensure the seamless integration of services from highly heterogeneous devices into an underlay system, the Web is considered to be the unifying platform for service integration. Thus, the modifications at the client side should remain as small as possible to ensure a wide support of user devices. For mobile user devices, a common Web browser to access the Web is considered standard.

To achieve a seamless embedding of the underlay system, the present mashup architecture as introduced in section 2.2.5 is extended to enable the dynamic integration of services residing on different user devices and form a custom Web mashup spread among multiple devices. The respective architectural extension is illustrated in Figure 3.4.

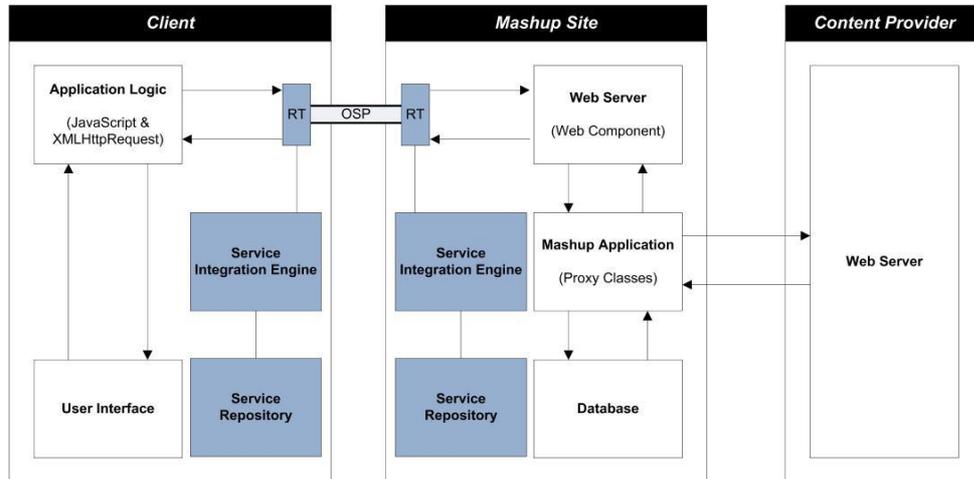


Figure 3.4: Extending the mashup architecture.

In order to abstract from service implementations, interface descriptions and behavioral descriptions are used that enable the addressing of services based on their functionality rather than on their endpoints themselves. These descriptions are kept in *Service Repositories* either at the client or the server side. Within Figure 3.4, the client's repository is drawn at the client side; however, this repository may as well be kept on a server so no additional data has to be present at the client side initially. Given a request for a specific service functionality, a *Service Integration Engine*—located both on server and client side—is capable of deriving a service from the local repository that meets this given request. Based on this abstract request, service classes featuring the same behavior are addressed instead of concrete service implementations so that the service integration becomes independent from the location of the service itself.

When all abstract services within an underlay system have been replaced by concrete services, the underlay system becomes executable; the *Runtime Environment* executes the underlay systems, in which parts of the system can be executed on one or more clients and a server.

Thus, the classic mashup architecture is extended by *Service Repositories* that hold service descriptions for the dynamic integration of services via the

Service Integration Engine. When deploying an underlay system to a set of devices, the abstract services are replaced by concrete services residing on the single devices. Afterwards, a sophisticated partitioning algorithm distributes the underlay system so every device is in control of its local services. To ensure that the distributed underlay system exposes the same behavior as the undistributed underlay system, the partitioning algorithm includes communication structures within the single parts of the underlay system. Its coherent execution and respective state management is ensured by a novel communication protocol referred to as *Orchestration Synchronization Protocol (OSP)*, which controls the communication and data passage between the single *Runtime Environments* executing parts of the underlay system.

The underlay system itself can be automatically generated based on a user request that specifies the effects the resulting mashup should provide. Here, the algorithm takes a time limit given by the user into account. The creation algorithm is tailored to increase the chances of finding an appropriate solution within the given time span.

All client-side components are realized in pure JavaScript, and can be pushed to the clients during the initialization phase of the distribution. Thereby, no modifications of the clients are required, so the distributed underlay systems are usable on every user device featuring a Web browser, such as smartphones, PDAs, or netbooks.

The remainder of this thesis is organized as follows. In chapter 4, an underlay system for integrated Web mashups is introduced that enables the structured execution of services within Web mashups. The underlay's perspective on services is resource oriented and thereby enables a direct mapping to a respective presentation as a Web mashup. Here, the underlay system is defined in such a way that it meets the lower three requirements illustrated in Figure 3.3. Chapter 5 deals with an algorithm for the automatic creation of the underlay system, while chapter 6 discusses its distribution. These chapters demonstrate the appropriateness of the underlay system for the remaining two requirements. Chapter 7 specifies the required components to support the previously introduced underlay systems and introduces a communication protocol for the distributed execution of Web mashups.

CHAPTER 4

AN UNDERLAY SYSTEM FOR DYNAMIC WEB MASHUPS

In this chapter, an underlay system for composed Web applications is introduced that decouples the development of applications from the heterogeneous devices hosting the application's respective services. In addition, it provides a direct mapping between the underlay system's state and a graphical presentation by introducing a resource-oriented perspective on services and data.

Section 4.1 derives the central components required for the realization of an underlay system. Section 4.2 then specifies a service model constituting the basis for service compositions, while section 4.3 defines the underlay system based on a bipartite graph concept. Here, special emphasis is put on the modelling of parallel service execution, the realization of realtime based timeouts that makes the underlay responsive to environmental changes, and a resource oriented view on services that enables the underlay's mapping to a graphical presentation. Section 4.4 then finally discusses the communication between multiple underlay systems.

4.1 CENTRAL COMPONENTS OF THE MASHUP UNDERLAY SYSTEM

The Web's architecture is distancing itself sharply from other architectural styles for distributed systems like the Service Oriented Architecture by relying on a simple design based on a communication protocol featuring a very limited set of interaction primitives and the concept of resources identified by URIs. Thereby, Web applications make a strong contrast to other composite applications such as business processes that rely on a heavy WS*-stack. It is thus aimed at keeping the definition of the underlay system in line with the Web's key concept of simplicity. In addition, the underlay is defined in an abstract and formal way, so it can be instantiated by various services composition languages and thus remains independent from a specific language itself.

In order to meet the the underlay system's requirements as discussed in section 3.4.1, the following components of the underlay system are defined within the scope of this section.

- Structured service composition execution – A model for the structured execution of services, supporting the paradigm of *programming-in-the-large*, is required to lessen dependency on specific scripting languages and to separate the concerns of service execution and data passage. The model must be defined so it can be automatically created and distributed by respective algorithms.
- Dataflow - One of the key aspects of mashups is their ability to aggregate content from various sources and to integrate it in order to create value-added Web applications. Therefore, a model that can capture the passage of data between services in an abstract way and does not rely on specific single services itself is required.
- Time Awareness – Within a domain featuring mobile devices that act as service providers, time awareness is considered an integral part of the underlay system so it can dynamically adapt to environmental changes and service availability without requiring external triggers. Thus, when a service is no longer responding, it must be dynamically replaceable during runtime without requiring external services or triggers.
- Presentation and Interaction Layer – Mashups are user-centric applications. Therefore, the underlay system must provide a means to derive a graphical presentation based on the current state of the single services that can support interaction with end-users.

Within the remainder of this section, the components listed above are shown to be sufficient to ensure an underlay system's structured execution, its presentation mapping, its responsiveness to failures, and its abstraction from service implementations. Within chapters 5 and 6, algorithms for the automatic creation and distribution of underlay systems are introduced, respectively, which prove that the underlay system defined within this section also meets the final two requirements derived in section 3.4.1.

4.2 SERVICE MODEL AND BASIC CONCEPTS

Services are considered as stateless, i.e. it is possible to create a response for every request leading to the invocation of a service without additional knowledge of the preceding or succeeding communication [62]. *RESTful* services can be considered as classic representatives of this kind of services. Note that –on a formal level– statelessness does not restrict the expressiveness of the

formal service composition model, since stateful services can be considered a composition of stateless services, where the states are encapsulated in the composition of the single service itself [18].

The service model comprises four general elements for service description, which are generally summarized in the following [88]. First, *inputs* and *outputs* of a service denote the data that is fed into and eventually retrieved from the services before and after its execution, respectively. Thus, services are considered as black boxes exposing a certain I/O behavior [53]. Inputs and outputs of a service are distinguishable by a unique identifier referred to as *port*, so an output port of a certain service can be connected to the input port of another. Here, connections between ports are restricted by *port types*, which denote the domain of the input or output data. For instance, a port type may be a primitive data type like Integer and String, or even a complex data type. Thus, only ports with the same port type can be connected; it is assumed that additional services will provide transformations between different (potentially compatible) data types; for instance, there may be a service to convert Doubles into Floats. Moreover, *preconditions* are conditional statements that have to be fulfilled before a service's execution, while *effects* constitute conditional statements fulfilled after a service's execution. These fourfold descriptions of services relying on inputs, outputs, preconditions, and effects, are commonly referred to as IOPE descriptions [88]. Therefore, services are described by their IOPE descriptions and are thus considered as *abstract services* defined within section 2.1.

This thesis aims to develop an underlay system that fits conceptually within the REST architectural style. Therefore, it is abstracted from concrete service descriptions such as WSMO [46] or OWL-S [112], which are capable of providing IOPE descriptions for services and already have been the focal point of extensive research [212, 187, 95]. Instead, it is focused on the conceptual definition of services by means of inputs and outputs, which are addressable by ports and can be assigned a certain type of data, and preconditions and effects, which are either true or false and can thus be regarded as a Boolean. Although the handling of preconditions and effects during the service composition's execution does not differ from Boolean inputs and outputs, they provide a special means to support the automatic composition of services by representing conceptual correlations between inputs and outputs of services. Section 5.2 addresses the capability in greater detail. Through this abstraction, the proposed model remains independent from current technologies and promises to have a longer scientific value. As a proof of concept, the proposed models are instantiated in chapter 8 dealing with the actual realization of the underlay system for Web mashups.

To represent service compositions properly as well as to control their execution, a bipartite graph concept, which is based on a workflow graph controlling the execution of the single components within a service composition and a dataflow graph defining the passage of data between services' output and input ports, is introduced. Each transition of the workflow graph corresponds to a service containing I/O parameters and a set of effects it generates during execution as described above.

When a service's precondition is met, the service can be executed. Thereby, an action representing the service functionality is performed, which consumes the service's input and generates a finite set of effects and a finite set of outputs. Transitions between locations are optionally annotated with guards, which restrict the transition's passage by requiring the satisfaction of specific preconditions or previously generated outputs. Notably, the output of a service is not necessarily required as input for a service reached in the next transition within the workflow graph; instead, an output may also become relevant after multiple other services have been executed. Therefore, a second graph is defined, specifying the dataflow within a service composition. This dataflow graph shares the set of locations with the workflow graph, but represents the flow of outputs from one service component to the input of another with its transitions. A guard for passing a transition within a workflow graph that would lead to the execution of the next service can thus depend on the presence of a set of effects (expressed through preconditions) and on the availability of all inputs that have to be created as outputs of other services before. Notably, the dataflow graph is not necessarily completely connected, thus may be a set of graphs.

In the following section, the service composition model is specified formally together with its respective execution semantics.

4.3 SERVICE COMPOSITION MODEL

4.3.1 *Workflow Graph*

There are several languages used to describe a composition of services; some of them have already been highlighted in 2.3.2. However, most of them are redundant in terms of expressiveness, and are driven by concrete products and respective companies [192]. Therefore, the workflow of the service composition, which is part of the underlay system, is specified according a well-defined and formal model for the representation of workflows.

Finite automata [22, 58] and Petri nets [121, 165] are the most prominent representatives for the formal modelling of service compositions. Both provide a mathematical basis to specify the execution order of single processes

or services. Both models have been extended through concepts to consider realtime during the execution of the overall system, leading to the definition of timed automata [6, 51] and time Petri nets [24]. In 2002, it was shown that timed automata and time petri nets are equivalent [72]. Although they are equivalent with timed petri nets in terms of timed language acceptance, timed automata are more expressive in terms of weak timed bisimilarity, to the extent that Petri nets have to be extended by the concept of priorities to obtain the same expressiveness as timed automata [23].

This thesis focuses on the usage of timed automata theory as underlying concept for the definition of the execution order of services within service composition, since automata do not require token concepts and are therefore more human-readable. In [144], a transformation from UML state machines [52] to timed automata has already proven that automata theory can be easily accessed by user-friendly modeling languages such as UML. In any case, all results stemming from this thesis that center on a timed automaton-based model can be transferred to models based on timed petri nets due to their functional equivalence; respective translations have already been proposed [35].

Timed automata are finite automata [22, 58] extended by a set of real-time valued clocks. The definition provided by Clarke et al. in [39] is referred to in the following.

Let X be a finite set of real-valued variables standing for clocks. Clock constraints are then defined as follows.

Definition 4.1 (Clock Constraints). A clock constraint is a conjunctive formula of atomic constraints of the form $x * n$ or $x - y * n$ for $x, y \in X$, $*$ $\in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. $\mathcal{C}(X)$ denotes the set of clock constraints. If φ_1 is in $\mathcal{C}(X)$, then $\varphi_1 \wedge \varphi_2$ is also in $\mathcal{C}(X)$.

A Timed Automaton is defined as follows.

Definition 4.2 (Timed Automaton). A timed automaton \mathcal{A} is a 6-tuple $\{\Sigma, L, l_0, X, I, T\}$, where

1. Σ is a finite alphabet (standing for actions),
2. L is a finite set of locations,
3. $l_0 \subseteq L$ are the initial locations (also called starting locations),
4. X is a set of clocks,
5. $I : L \rightarrow \mathcal{C}(X)$ assigns invariants to locations, i.e., provides a mapping from locations to clock constraints, and
6. $T \subseteq L \times \mathcal{C}(X) \times \Sigma \times 2^X \times L$ is the set of transitions.

As abbreviations, $l \xrightarrow{g, \alpha, \lambda} l'$ stands for $\langle l, g, \alpha, \lambda, l' \rangle$, i.e. the transition leading from location l to l' . The transition is restricted by the constraint g (often called guard); $\lambda \subseteq X$ denotes the set of clocks reset during the transition passage.

Notably, many model checkers such as UPPAAL [191, 102], operating on timed automata, restrict the location invariants to downwards closed ones, i.e. only allow invariants of the form $x \leq n$ or $x < n$ for $n \in \mathbb{N}$.

Infinite state transition systems (sometimes called infinite state transition graphs) are used as a model for a timed automaton \mathcal{A} . Deep introductions into the notion of transition systems can be found in [84, 110]. Here, the definition of Clarke et al. [39] is followed, specifying an infinite state transition system $\mathcal{T}(\mathcal{A})$ for a given timed automaton \mathcal{A} as a 4-tuple $\mathcal{T}(\mathcal{A}) = \{\Sigma, Q, Q_0, R\}$. A state $q \in Q$ of this transition system is defined as a pair (l, v) , where $l \in L$ is a location and $v : x \rightarrow \mathbb{R}^+$ is a clock assignment. The initial states are identified by all initial locations, where all clocks are set to zero, i.e. $Q_0 = \{(l, v) | l \in L_0 \wedge \forall x \in X [v(x) = 0]\}$.

The definition of the state transition relation is implied by the two following requirements. First, a notion is required to reset a clock to zero, i.e., for $\lambda \in X$, $v[\lambda := 0]$ is defined for mapping all clocks in λ to zero. For $d \in \mathbb{R}$, $v + d$ is defined as a clock assignment that maps the current value of v to $v(x) + d$ for all clocks $x \in X$. Based on this, two different types of transitions are defined, covering the passage of time and the triggering of actions. Time can pass while the system is in a specific location as long as the according state invariant is not violated. This elapsing of time is referred to as *delayed transition*, specified as $(l, v) \xrightarrow{d} (l, v + d)$, $d \in \mathbb{R}^+$, subject to the condition that the invariant $I(l)$ is not violated for every $v + e$, $0 \leq e \leq d$. The second type of transitions describes the actual execution of an action and is thus referred to as *action transition*. If there is a transition $\langle l, \alpha, g, \lambda, l' \rangle$ where v satisfies g and $v' = v[\lambda := 0]$, we note $(l, v) \xrightarrow{\alpha} (l', v')$ for a transition with $\alpha \in \Sigma$. Together, we receive the transition relation \mathcal{R} for $\mathcal{T}(\mathcal{A})$ as $(l, v) \mathcal{R} (l', v')$ (also written as $(l, v) \xrightarrow{\alpha} (l', v')$), if there are l'' and v'' so that $(l, v) \xrightarrow{d} (l'', v'') \xrightarrow{\alpha} (l', v')$ for some $d \in \mathbb{R}$. Thus, an action α can also be performed without elapsing time, i.e. in the case that $d = 0$. Moreover, α may also stand for the empty action $\tau \in \Sigma$, i.e. a transition can also be passed without entailing the execution of a specific action.

Since actions are supposed to model the behaviour of a service, the execution of an action α is mapped onto the consumption of inputs, the creation of outputs and the generations of effects. Inputs and outputs are regarded as typed variables bounded to specific ports, enabling the definition of I/O pas-

sage by means of a dataflow graph. The domains of variables thus constitute their primitive data type.

Workflow graphs Θ are modeled as timed automata as defined in Definition 4.2, extended by the ability of actions $\alpha \in \Sigma$ to operate on typed variables and effects. Typed variables are defined as 2-tuples $(x, \Gamma(x))$, where x is a variable and $\Gamma(x)$ its respective domain. Let \mathcal{V} be a finite set of typed variables. Regarding the treatment of effects, \mathcal{E} is defined as a final set of variables $\gamma^* \in \{0, 1\}$ indicating whether an effect γ has been created (γ^* set to 1) or not (γ^* set to 0). It is assumed that \mathcal{V} contains at least all variables of \mathcal{E} , thus, $\mathcal{E} \subseteq \mathcal{V}$.

Definition 4.3 (Guard Constraints). A real-value constraint is a propositional logic formula $x * n$, where $x \in \mathcal{V}$ is a typed variable, $n \in \mathbb{R}$ and $*$ $\in \{<, \leq, \geq, >, ==\}$. $\mathcal{C}(R \setminus \mathcal{V})$ denotes the set of real-value constraints containing only variables of \mathcal{V} .

If φ_1 and φ_2 are in $\mathcal{C}(R \setminus \mathcal{V})$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg \varphi_1$ are also in $\mathcal{C}(R \setminus \mathcal{V})$.

Let $\varphi_c \in \mathcal{C}(X)$ be a clock constraint as defined in Definition 4.1, φ_r a real-value constraint.

A guard constraint is a formula $\varphi = \varphi_c | \varphi_r | \varphi_c \wedge \varphi_r$ evaluating to a Boolean. $\mathcal{C}(X \circ R \setminus \mathcal{V})$ denotes the set of all guard constraints.

The workflow graph is defined as a timed automaton whose actions and guards can also operate on typed variables.

Definition 4.4 (Workflow Graph). A workflow graph Θ is a timed automaton as defined in Definition 4.2, where

1. Σ is a finite alphabet. The alphabet represents the actions that are identified by the single services within the service composition.
2. L is a finite set of locations defining the service compositions's current state,
3. $l_0 \in L$ are the initial locations defining the initial state of the service composition,
4. X is a set of clocks,
5. $I : L \rightarrow \mathcal{C}(X)$ assigns invariants to locations, restricting the system in the amount of time it is allowed to remain in the current state, and
6. $T \subseteq L \times \mathcal{C}(X \circ R \setminus \mathcal{V}) \times \Sigma \times 2^X \times L$ is the set of transitions, denoting the execution of a service (represented by an action α). The passage of a transition (and thus the execution of a service) thereby depends on whether the according guard is met.

In order to decide whether an output of a service component is required as an input for another one, a dataflow graph is introduced in the following section.

4.3.2 Dataflow Graph

Dataflow graphs represent the connections of output and input ports. They keep the same locations as the workflow graph introduced in Definition 4.4 and use labeled transitions to express inter-component data passing.

Definition 4.5 (Dataflow Graph Ω). A dataflow graph Ω is a labeled directed digraph $\Omega = \{N, P, E\}$, where,

1. N is a final set of labeled nodes, which is equivalent to the set of locations L held in the corresponding workflow graph Θ ,
2. P is a set of port mappings represented by 2-tuples $p = (p_1, p_2)$ indicating the passage of the output from port p_1 to the input port p_2 , and
3. $E \subseteq N \times P \times N$ is a final set of labeled directed transitions.

Each location represents the data generated by the execution of action α_i ; therefore a node is labeled with $d(\alpha_i)$ to identify the output data from action α_i . If a transition is passed within a workflow graph Θ entailing the execution of action α_i , *all* outgoing transitions from location $d(\alpha_i)$ within the corresponding dataflow graph Ω are passed. A transition passage $d(\alpha_i) \xrightarrow{(p_m, p_n)} d(\alpha_j)$ within Ω effectuates that the output at port p_m from action α_i is redirected to input port p_n of action α_j in the case that action α_i is executed within Θ .

The definition for a service composition can now be given as follows.

Definition 4.6 (Service Composition \mathcal{S}). A service composition \mathcal{S} is a 3-tuple $\mathcal{S} = \langle \Theta, \Omega, \Upsilon \rangle$, where

1. Θ is a workflow graph as defined in definition 4.4,
2. Ω is a dataflow graph as defined in definition 4.5, and
3. Υ is a finite set of abstract services.

The following section discusses a small example for service composition representation and control.

4.3.3 Workflow and Dataflow Graph Interworking

An exemplary service composition is abstractly depicted in Figure 4.1.

The involved abstract services are represented by rectangles labeled with an action α_i describing their functionality. The order of execution, i.e., the workflow, is depicted by bold arrows, while the passage of inputs and outputs

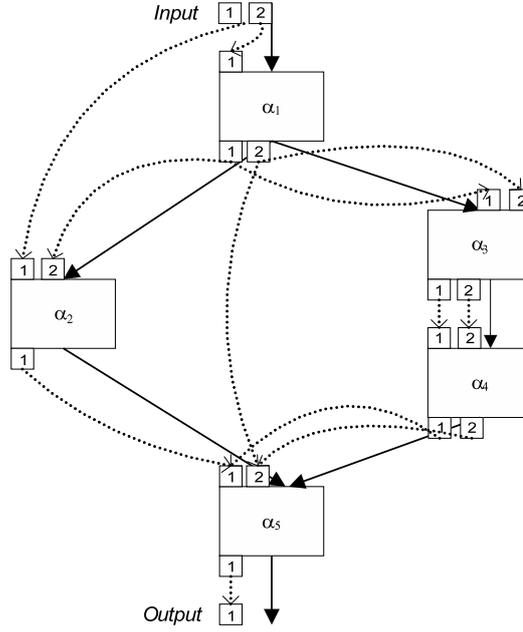


Figure 4.1: Abstract depiction of a possible service composition.

is indicated by dashed arrows. They connect the service's output ports, drawn as numbered squares, with the input ports of other services.

The example service composition contains five abstract services (labeled with $\alpha_1, \dots, \alpha_5$).

After α_1 has been started, either α_2 or α_3 and α_4 are executed. In the case that both execution paths are enabled, i.e. all premises in terms of I/O and effect availability are met, a path is chosen nondeterministically according to the notion of transition systems.

While service executions in common service composition representations like that in Figure 4.1 are represented as nodes within graphs, the service composition model defined above specifies service executions as actions performed during transitions between locations. The workflow graph deduced from the example service composition is illustrated in Figure 4.2 (a).

Guards restricting the transition from a location l to a location l' operate on the generated effects (through preconditions) and a final set of inputs and outputs. Outputs are not stored but redirected to input ports by means of the corresponding dataflow graph. The dataflow graph depicted in Figure 4.2 (b) corresponds to the workflow graph illustrated in Figure 4.2 (a).

For instance, the execution of α_1 implies the redirection of the output from the first output port of α_1 to the second input port of α_2 and the first

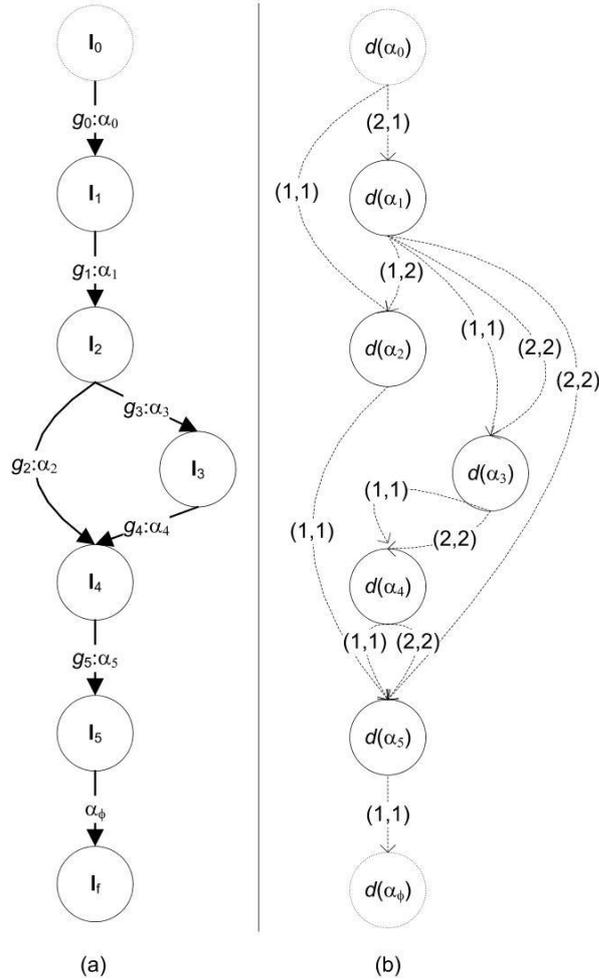


Figure 4.2: (a) An according workflow graph. (b) A dataflow graph representing the I/O passing.

input port of α_3 . Moreover, the output from the second output port of α_1 is forwarded to the second input port of α_3 and to the second input port of α_5 .

4.3.4 From Sequential to Parallel Execution

Because of their relation to automata and their given transition relation introduced in section 4.3.1, workflow graphs already support a wide set of control structures required to guide the execution order of services; the three most important ones are abstractly depicted in Figure 4.3. First, non-deterministic decisions between multiple possible services are modeled with a

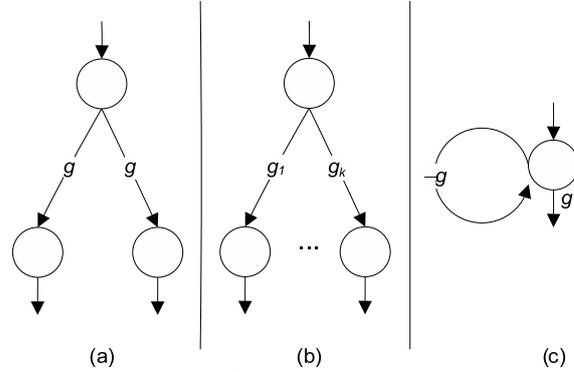


Figure 4.3: Basic control structures of automata semantics. (a) Nondeterministic selection. (b) Choice. (c) Looping.

finite set of outgoing transitions labeled with the same guard g . When the guard is met, all transitions are enabled and the transition relation discussed in section 4.3.1 selects a transition nondeterministically. Decisions such as **IF/ELSE** or **SWITCH/CASE** statements are represented by multiple outgoing transitions labeled with different guards g_1, \dots, g_k . Here, deadlocks can occur when all guards are evaluated as *false* under a given set of constraints. Moreover, multiple guards may be *true*, entailing a nondeterministic decision making between the enabled transitions. Last, looping can be modeled with transitions pointing back to the origin state as depicted in Figure 4.3 (c).

However, classic semantics for timed automata define actions as variable modifications performed during the passage of a transition. For instance, when a transition is labeled with the action $x = 1$, the variable x is set to one during the passage of the transition, so x equals one when the transition has led to the next location. This semantics do not allow the expression of parallel behavior, since only sequences of actions can be modeled. Thus, if actions are identified with service executions, parallel service execution cannot be modeled.

Therefore, the understanding of actions of timed automata is adapted in order to support both sequential and parallel service executions. This can be achieved by identifying actions with service *invocations* instead of service executions. Thus, when a transition is passed, the service is invoked so the finalization of the service's execution cannot be guaranteed when the following location is reached. Thereby, the automaton can directly pass the next transition, which implies the invocation of the next service. Note that the passage of transitions itself does not consume time, since it is considered as an action transition as introduced in section 4.3.1. Thus, services that are

modeled by a sequence of action transitions are invoked sequentially without time delay so that the services are executed in parallel.

This model for parallel execution of services is insufficient in the case that the biggest subset of a given set of services should be executed in parallel. Assume two services α_1 and α_2 are selected for parallel execution. When the services are initiated in the order $\alpha_1 \rightarrow \alpha_2$, it may be possible that α_1 cannot be started because an input is missing. The blocking of the transition would also entail that service α_2 does not start, although its guard may be met. Therefore, a control structure for parallel execution of services is required. The left-hand side of Figure 4.4 depicts an abstract workflow reflecting the previously described situation of two services α_1 and α_2 , which should be executed in parallel.

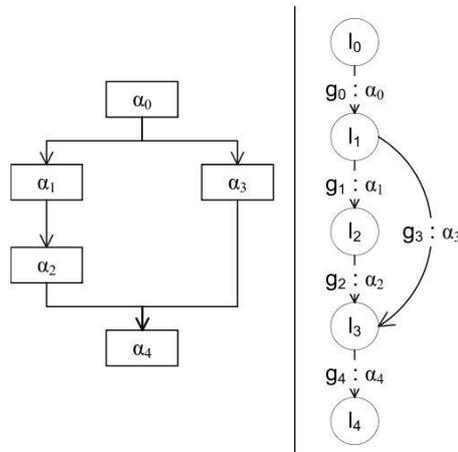


Figure 4.4: Timed automaton structure ensuring parallel execution.

The right-hand side of the same Figure depicts the timed automaton that corresponds to the abstract service composition. Here, the execution of α_2 would be forestalled in the case that guard g_1 is not met although g_2 may be fulfilled. In order to unblock this behavior, an additional *idle* state is automatically inserted during the generation of the workflow graph, which is entered after every execution of a service that may be executed in parallel to other services. The accordingly modified workflow graph is illustrated in Figure 4.5.

Because of this modification, the timed automaton moves into the idle state as soon as the guard of the next service is not met, so that the invocation of another service can be initiated whose guards have been fulfilled. The utilization of semaphores ensures that every service is only executed once.

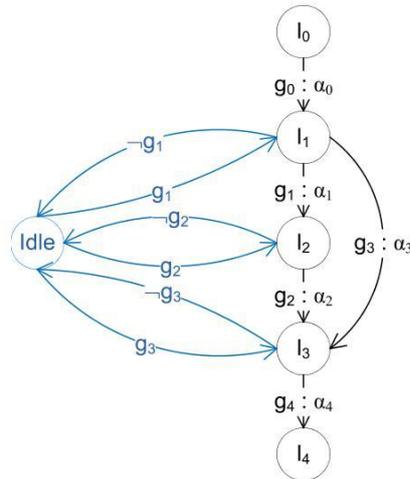


Figure 4.5: Extended timed automaton structure ensuring parallel execution.

The modelling of parallel service executions by means of sequential service invocations as introduced above entails the adaptation of the model for sequential service executions. Note that a sequence $\alpha_1 \rightarrow \alpha_2$ of two services is only executed sequentially, if the guard g_{α_2} forestalls the execution of α_2 until α_1 has finished its execution. When at least one output of α_1 is used as input for α_2 , this restriction is automatically enacted through the dataflow graph. However, when α_2 should not be executed before α_1 is finished, although both services do not have a data related dependency, this restriction has to be modeled explicitly. Therefore, every service invocation entails the creation of a Boolean containing a unique process identifier. As soon as α_1 is invoked, a variable pid_{α_1} is created and set to zero. When the execution of α_1 is terminated, pid_{α_1} is set to one. The guard protecting the invocation of α_2 is joined with $pid_{\alpha_1} == 1$, so that α_2 cannot be invoked before the finalization of α_1 's execution has been indicated. This model for the guarantee of finished service executions is also used to specify timeouts, which are introduced in section 4.3.5.

The technique of invoking services sequentially to support their parallel execution is based on the concept of asynchronous service calls and has been used more frequently within the Web since the appearance of Ajax and the XMLHttpRequest object as introduced in section 2.2.4.1. Asynchronous services are controlled by means of callback functions, i.e. whenever a service finishes its execution, it does not return a data set, but invokes a predefined function, the so-called callback function. This kind of eventing mechanism has been modeled with polled semaphores. Thus, each service α is assigned

a unique process identified pid_α represented by a Boolean, which is set to zero when α is invoked and set to one as soon as α has finished its execution. Thereby, the eventing mechanism realized by callbacks is expressed through a semaphore concept based on a shared memory, i.e. a set of shared variables. Section 8.2 later introduces a runtime environment for the underlay system defined in this section, highlighting the realization of asynchronous service invocations within the Web domain.

4.3.5 Modelling Timeouts

Workflow graphs are modeled through timed automata in order to make the underlay system capable of responding to environmental changes in a self-contained manner. A concrete service composition, i.e. an instance of an abstract service composition, may fail to work because one or more concrete services are no longer responding, either because they have crashed or are no longer available since the device they are hosted on has left the user's connection range. By allowing the definition of timeouts via clocks and guard constraints, such failed services can be dynamically recovered during runtime.

There are multiple ways to model timeouts within workflows, which are quickly discussed in the following. An abstract service equipped with a timeout mechanism so that a concrete service bound to this abstract service can be dynamically replaced when it does not respond within a given time span, is referred to as *protected service* in the following.

In general, a clock is created for every protected service invocation and reset to zero whenever a transition is passed that implies the invocation of such a protected service. This clock thus measures the time that has already elapsed since the services has been evoked. However, as introduced in section 4.3.4, the passage of a transition corresponds to a service invocation and not to a service execution, so that parallel execution of services can be expressed by a sequence of services, i.e. by a sequence of asynchronous service calls. Thereby, the current state of a workflow graph is unpredictable when a timeout elapses. The workflow graph can be virtually within every location that succeeds the transition corresponding to the invocation of a protected service. Thus, every location would require a method to respond to an elapsing timeout. This method can be realized by the addition of transitions leading from every location within the workflow graph to the location that precedes the transition, and labeled with the protected service. The transitions itself invoke a service that deals with the replacement of the protected service, then invoke the protected service, which is now instantiated

by means of another concrete service, and finally lead back to the current location of the workflow.

This approach exposes multiple weaknesses. First, $2 \times (n - 1) \times m$ additional transitions are required, where n denotes the number of locations within the workflow graph and m is the number of protected services. Moreover, m semaphores are required to guide a workflow to the replaced service invocation and back to the current location of the workflow graph. In addition, this proceeding can lead to the replacement of services although their substitution is not inherently necessary. Consider a protected service α with a given timeout of 500 milliseconds. Assume further that the results generated by α are not used by another service or the user within the next 700 milliseconds. Replacing the service after 500 milliseconds would thus neglect the chance that the service might respond in the following 200 milliseconds, which would not have affected the overall execution of the service composition.

Therefore, a time-bound and data-driven replacement is introduced, that 1) reduces the additional transitions required to realize a timeout and 2) only replaces a service in on-demand fashion, i.e. when the given time boundary has been exceeded and the results of the service are required within the next step. Moreover, this structure can be easily generated; appendix A.2.3 makes use of this property.

The timeout mechanism is realized by a so-called *timeout construct*; the extension is illustrated in Figure 4.6. Figure 4.6 (a) shows a transition with a guard g and an action α . In the case that α should represent a protected service, the transition's annotation is extended to set a special clock c_α back to zero; this clock measures the time that α has elapsed since its invocation. Figure 4.6 (b) illustrates this minor modification.

Figure 4.6 (c) finally shows the timeout construct. It is attached to every location that has an outgoing transition that is labeled with a service invocation which takes at least one output from α as input. Within the illustration, the invocation of β would thus require at least one output from α as input.

Asynchronous service calls are realized by callbacks as introduced in section 4.3.4. This callback is modeled by a Boolean pid_α that is set to one when the service has responded to the service invocation with the corresponding result; otherwise, the Boolean is zero. When the predefined amount of time granted for the execution of α has elapsed and the service has not yet responded ($pid_\alpha == 0$), the timeout construct is entered. Here, the concrete service is substituted with a special service $replace(\alpha)$. When no appropriate replacement can be found, the workflow moves into a failure state; the handling of this kind of exception follows the decisions of the software developer. If a replacement was found, the new service is incorporated and executed.

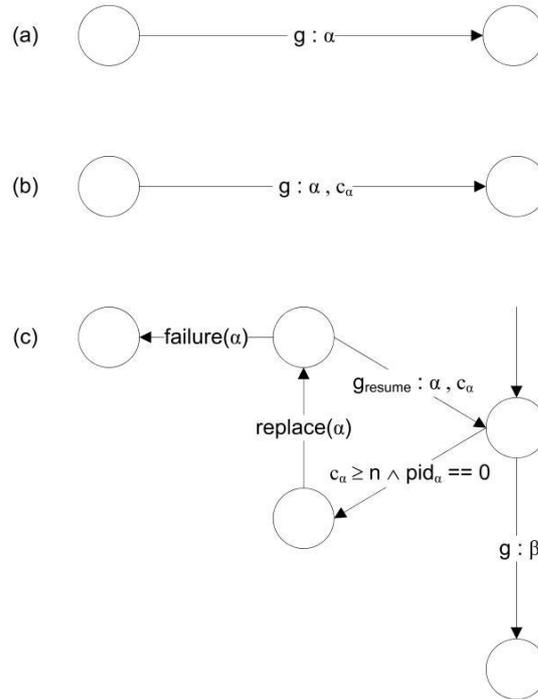


Figure 4.6: (a) Origin part of the workflow graph. (b) Modification of the workflow graph for the protected transition. (c) Timeout construct.

After the invocation of the new service, the workflow moves back into the current location and resets the respective timer.

Note that this timeout construct can exist multiple times. When β requires two inputs from two different services that are both protected, there is one timeout construct for the invocation of each service.

4.3.6 Resource Orientation

So far, a service composition has been defined as a 3-tuple consisting of a workflow graph, a dataflow graph, and a set of abstract services. Within this section, these service compositions are extended by the concept of resources and respective presentation mappings to form an underlay system for Web mashups that supports the generation of a graphic presentation of the underlying service composition to visualize the composition's current state and provide means for user interaction. Therefore, the notion of actions, which have been associated with service invocations so far, is extended by resources on which the actions are executed.

An *abstract resource* is defined by its *type*, a finite set of *attributes* that are expressed by key-value pairs, and a finite set of *presentation mappings*. For instance, a resource of the type *location* possesses two attributes *lat* and *long* that hold the coordinates of the location as Integers. The default presentation mapping may display the values as a String. A *concrete resource* possesses one well-defined type, one presentation mapping, and a unique identifier. The values of the single attributes can be set to specific values or remain uninitialized. For instance, the concrete resource *userLoc* can be of the type *location*, where *lat* = *x* and *long* = *y* denote the user's current position. The presentation mapping can be set to *none*, so that the resource does not expose a presentation but only serves as input for another service, like a map service that can visualize the user's location on a map.

All inputs and outputs of an underlay system are considered as attributes of resources. That way, a service takes a subset of one or more resources as an input and generates a set of outputs that constitute attributes of a resource. Although a service can take attributes from multiple different resources as input, its output attributes have to operate on a single resource only. This restriction simplifies the classification of services based on the resource they are modifying. The extension of resource modifications so that a single service can operate on attributes stemming from multiple resources is discussed within the scope of future work in section 9.2.

This resource-oriented perspective on data and services, which can be classified by the resource on which they are operating, allows the mapping of resources to presentations. These presentations are updated every time a resource is modified and thus provide a graphic presentation of the underlay system's state for every point in time. The current specification of resources and functions is listed in appendix B.

Within the remainder of this thesis, abstract resources and concrete resources are simply referred to as *resources* when the differentiation is either obvious or does not matter in the current context.

An action is defined as a 2-tuple consisting of an abstract service and a concrete resource. When the abstract resource is later replaced by a concrete service to make the underlay executable, the action is performed upon the given resource.

Both the abstract service and the resource are required for the replacement of an abstract service by a concrete one. A concrete service *matches* an abstract service if it can provide the abstract functionality, and operates on the same resources, i.e. has the same inputs and outputs.

Each underlay system operates on a final set of resources, which can be accessed and modified by the involved services; this set is referred to as *resource space* in the following. Resources possess two different types

of properties, which then become important when an underlay system is distributed among multiple devices. Section 6.2 addresses this issue.

Both the mapping from abstract services to concrete services and the mapping from resource to corresponding presentation are specified in more detail in chapter 7.

4.4 COMMUNICATION AMONG MULTIPLE WORKFLOWS

Workflows constitute a classic instance of service orchestrations where a single, central component, i.e. the workflow graph, manages the execution order of multiple services. One of the objectives of this thesis is to provide a means to automatically distribute an existing workflow dynamically onto multiple devices and to execute the single workflows so that they provide the same behavior and functionality as if the workflow was executed centrally on a single device.

As soon as those workflows are executed on different devices, this central control is lost. As a result of the distribution, multiple workflows reside on the single devices, thereby providing the means to centrally orchestrate the local services. Thus, the workflows have to be extended by a mechanism that enables the coordination of the workflows itself, resulting in a choreography of orchestrations expressed by multiple distributed workflows.

Communication between multiple parties can be characterized by three central aspects. First, the number of receivers of a sent message can vary; this aspect is covered by the notion of *unicast messages*, where a single sender transmits a message to a single receiver, and *multicast* or *broadcast messages*, where multiple receivers are addressed by a single message. Here, it is commonly differed between multicast messages that address a real subset of possible receivers, and broadcast messages that are transmitted to all possible receivers [9].

The second central aspect is the behavior of the sender after the transmission of the message. Here, it is differentiated between a blocking sender where the sender blocks the current process until the receiver consumes the message, and an unblocked sender, where the process can proceed with the current computations without waiting for the receiver. Due to the time dependencies between sender and receiver, these interaction styles are referred to as *synchronous communication* (blocked sender) and *asynchronous communication* (unblocked sender), respectively.

The third aspect is related to the receiver's behavior during the reception of a message. Here, the receiver may either block its calculation when a message is expected until the message is received. Alternatively, the incoming

messages may be stored and processed later on when the respective signal or data is required. Last, messages may be handled opportunistically, i.e. in the case that a message arrives when it can be managed, it is received and processed, while it is discarded if not [40].

Within the following sections, six communication patterns that can be used for inter-workflow communication are highlighted. A corresponding adaptation of the formal definition of workflows is then given in section 4.5.

4.4.1 Synchronization Channels

One of the central challenges within distributed systems is the remote execution of services. Thus, if a service α_1 requires some input that can only be provided by another service α_2 , α_1 may call α_2 and move into a waiting state, so that α_1 can be resumed as soon as α_2 provides the necessary input after its execution. This kind of synchronous communication is also referred to as *blocking read*, since the sender blocks its execution until it is resumed by the arrival of the expected piece of data from the called service. The probably most well-known application of this style is the Remote-Procedure-Call (RPC) [195].

Within the scope of loosely distributed workflows, synchronous communication is required for three key reasons.

- Access to shared resources (concurrency)
- Enforcement of proper execution order
- Synchronous data passage

The notion of two workflows interacting via synchronous communication is depicted in Figure 4.7.

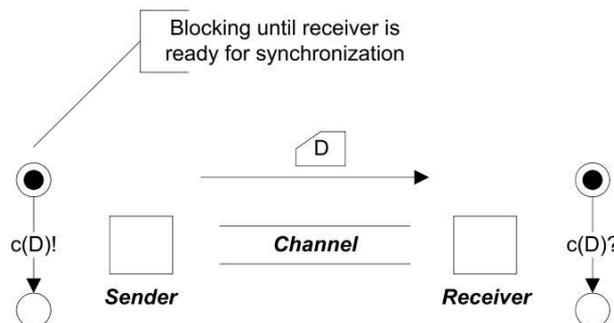


Figure 4.7: Synchronized unicast message.

Here, the synchronization takes place as soon as both the sender and receiver are in their synchronization locations s_1 and r_1 , respectively. In the event that one workflow enters the location before the other one, it blocks its processing until the other workflow also moves into the synchronization location. During synchronization, the sender and the receiver pass their transitions simultaneously. The action annotated at the sender's transition is executed first; afterwards, the receiver's action is initiated. This proceeding enables the sender to generate a piece of data D that can be directly processed by the receiver's action. In the case that the interaction is only required for signaling purposes, the data item D may be empty.

For example, a workflow may initiate the transmission of either an SMS or a MMS via a service residing on a smartphone, where the service is embedded within a respective workflow. The receiver is already in a receiving state because it is known a priori that either a message or image will be passed for transmission. The sender may build a data item $D = recipient, content$, that defines whether a text message or image should be sent, together with the content itself and a method $getContent(D)$ that aggregates the content from a user. The receiver may then receive the data item D and use it directly within a service $sendMessage()$ to transmit the message to the addressed *recipient*. The proceeding is illustrated in Figure 4.8.

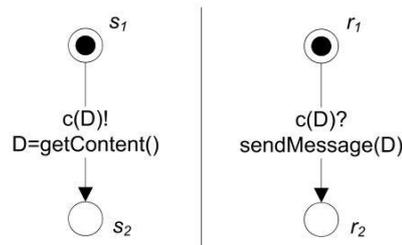


Figure 4.8: Passing data via synchronous channels.

Note that the interaction style of synchronous communication entails an implicit invocation acknowledgment for the sender, i.e. as soon as the sender can pass the transition that sends D to the receiver, the sender can be sure that the data will be directly processed at the receiver's side.

4.4.2 Broadcast Channels

Within this section, five broadcast communication patterns are derived. All patterns are defined as broadcast messages addressing a finite set of receivers. A specification of a single receiver denotes the usage of the broadcast message

as a unicast message. Note that synchronous messages as introduced before can entail serious problems with regard to deadlocks.

4.4.2.1 Queued Broadcast Channel

Blocking senders and receivers during synchronous communication can decrease the efficiency of the overall system, since processes do not continue their computation until a certain signal is received or can be transmitted, respectively. Alternatively, sender and receiver may communicate asynchronously, where the message is transmitted by the sender regardless of whether the receiver is able to process it directly or not. Instead, messages are stored at the receiver side and processed as soon as possible. There are multiple ways to realize such a data space for storing signals, e.g. a unstructured message heap, a stack or a queue. Here, the former approach requires an addressing structure for messages such that the required messages can be looked up by the receiver, while the latter two methods define a processing order of messages based on their arrival, either in Last In First Out (LIFO) or First In First Out (FIFO) manner, respectively.

Within this section, buffered asynchronous communication is realized by means of a signal queue as illustrated in Figure 4.9.

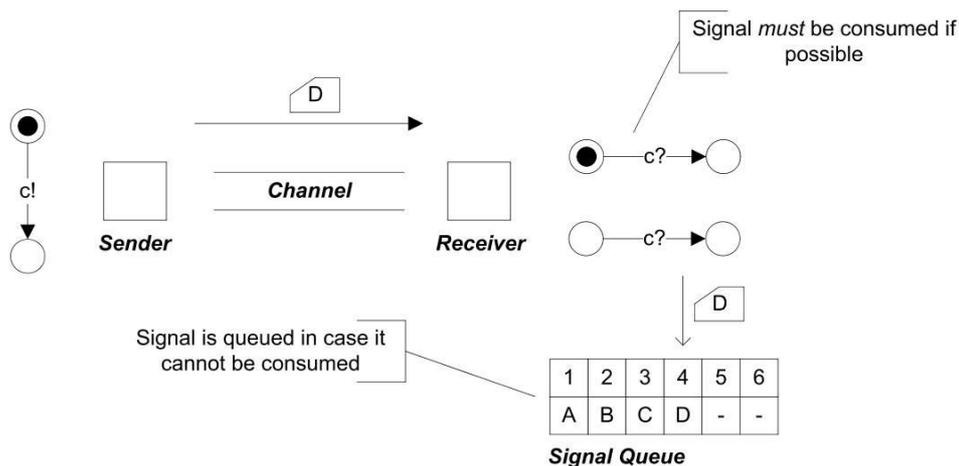


Figure 4.9: Multicast message with queued signals.

In this realization, all received messages are inserted into a queue at the receiver side and processed in the same order. Notably, messages never lose their importance, i.e. there is no general mechanism to skip a message at the head of the queue. This is a general problem when a sender transmits

a message in regular steps in order to communicate the current state of a certain resource. Here, a message that has not yet been processed by the receiver may become obsolete as soon as a newer message containing a more up-to-date state of the resource is received. The following section addresses the issue of message updates.

4.4.2.2 Updating Broadcast Channel

Updating broadcast channels are queued broadcast channels where messages within the receiver's queue are replaced by newer incoming messages of the same type in the event that they haven't been processed yet. Thereby, the temporal relevancy of each queued message item can be ensured. The general proceeding is illustrated in Figure 4.10.

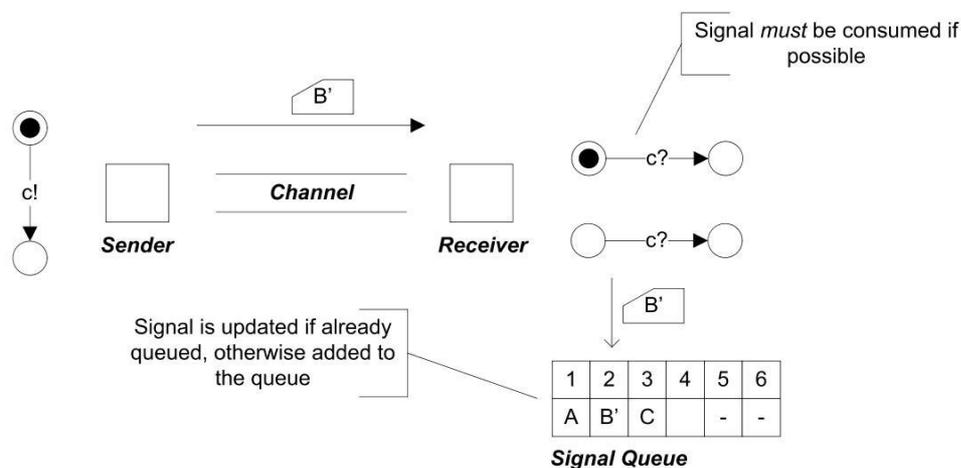


Figure 4.10: Multicast message with updated signals.

However, these kinds of channels require additional identification or addressing schemes, since the message queue has to be searched whenever a new signal arrives at the receiver side in order to identify older versions of the same message. This identification mechanism for equal messages can be avoided by defining a *virtual channel* for every message type, so that the queue always holds no or exactly one message, which is then replaced by a newly incoming one.

4.4.2.3 *Temporally Blocking Broadcast Channel*

Synchronous communication channels imply the blocking of either the sender or the receiver in order to wait for the other workflow's synchronization message, which may reduce the performance of service compositions on single nodes where the workflow is blocked while waiting for a specific synchronization channel although it may process other requests in the meantime.

Moreover, many scenarios do not require a hard synchronization of messages, but rather expect a tight temporal coupling of communication. Here, a sender may request a piece of data from multiple services or users and is only interested in information that is returned within a certain time span. For instance, a workflow may send out messages to multiple nodes to request whether the receiver is interested in joining a pervasive game. The sender may opt to wait for a time span of 20 seconds for positive replies and then initiates the game with the set of users that have confirmed to join. Replies that arrive after the given period are discarded since they can no longer be processed. Alternatively, a method may be defined that processes messages shortly after the given time span has elapsed to notify the users that their reply came late.

This communication style is realized by a temporally blocking sender as depicted in Figure 4.11.

The sender transmits a message that is stored in the signal queue of the single receivers. Up to this point, the mechanism is equivalent to the pattern of queued signals. In addition, another process is defined that deletes messages from the queue after a certain time span has elapsed. This time span either may have been previously negotiated between sender and receivers, or was part of the signal message.

4.4.2.4 *Discarding Broadcast Channel*

When messages are considered optional notifications that may be processed by the receiver if it currently resides within an idle state, they may be sent via an discarding channel as depicted in Figure 4.12. Here, messages that cannot be directly processed by the receiver are dropped.

This type of messages represents a method for best-effort addressing and can, for instance, be used to search for nodes that are currently available for a certain processing task.

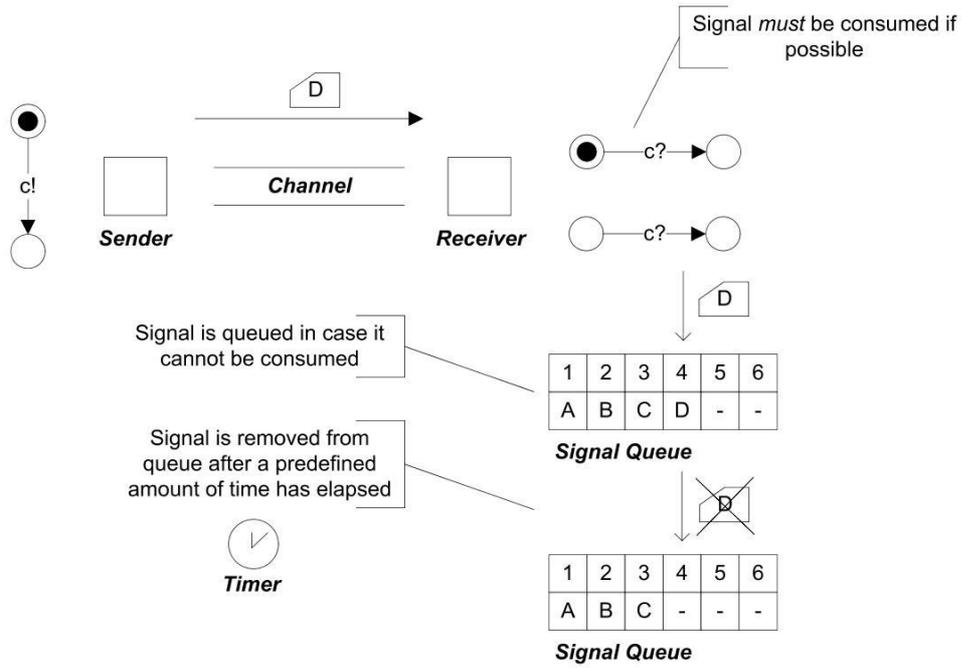


Figure 4.11: Multicast message with temporally blocked sender.

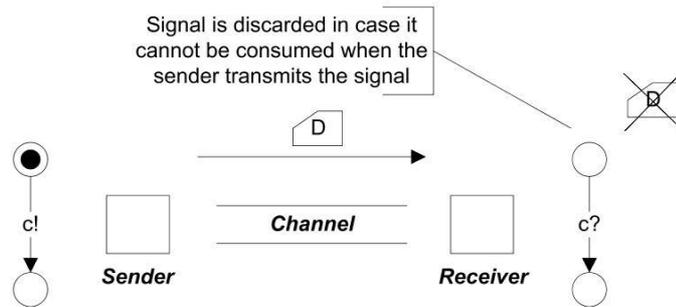


Figure 4.12: Multicast message with discarded signals.

4.5 UNDERLAY SYSTEMS

Within this section, the definition of an underlay system for distributable Web application is summarized.

Definition 4.7 (Underlay System). An underlay system Ψ is a service composition $\mathcal{S} = \langle \Theta_*, \Omega, \Upsilon \rangle$ as defined in definition 4.6, where the modified workflow graph Θ_* contains resource-oriented actions and supports communication via virtual channels.

Θ_* is a timed automaton as defined in Definition 4.2, where

1. Σ is a finite alphabet. The alphabet represents the actions that are given as 5-tuples as outlined in section 4.3.6,
2. L is a finite set of locations defining the service compositions's current state,
3. $l_0 \in L$ are the initial locations defining the initial state of the service composition,
4. X is a set of clocks,
5. $I : L \rightarrow \mathcal{C}(X)$ assigns invariants to locations, restricting the system in the amount of time it is allowed to remain in the current state, and
6. $T \subseteq L \times \mathcal{C}(X \circ R \setminus \mathcal{V}) \times \Sigma \times S \times 2^X \times L$ is the set of transitions, denoting the execution of a service (represented by an action α). The passage of a transition (and thus the execution of a service) thereby depends on whether the according guard is met.
7. R is set of resources,
8. S is a set of synchronizations, and
9. C is a set of channels types.

As abbreviations, $l \xrightarrow{g, \alpha, s, \lambda} l'$ stands for $\langle l, g, \alpha, s, \lambda, l' \rangle$, i.e. the transition leading from location l to l' . The transition is restricted by the constraint g (often called guard); $\lambda \subseteq X$ denotes the set of clocks that is reset during the transition passage. s denotes a signal that is transmitted via the affiliated channel. While $s!$ denotes the sending of a signal, $s?$ stands for the receiving of a signal. When the channel cannot be derived from the signal's name itself, s/c denotes the transmission of a signal s via a channel $c \in C$.

AUTOMATIC CREATION OF UNDERLAY SYSTEMS

This chapter deals with the automatic creation of underlay systems. While section 5.1 discusses general approaches for the creation of underlay systems, the remainder of this chapter deals with an algorithm for the automatic creation of underlay systems based on a given user request. Special emphasis is put on enhancing the chances of finding an underlay system to match the user's request when given an upper creation time limit. Here, heuristics are introduced and evaluated that decouple the dependencies of the algorithm's performance from the underlying service domain and thereby increase the algorithm's average success rate.

5.1 TOWARDS CREATION METHODS FOR UNDERLAY SYSTEMS

There are three general approaches to the creation of underlay systems, which feature different advantages and disadvantages; these approaches are summarized in Table 5.1.

First, the underlay system can be directly created by software developers through a graphical user interface. Here, tools such as the realtime model checker UPPAAL [102] can be used as graphical front-end to create the workflow of the underlay system; a transformation has been developed that supports the translation from the notion of UPPAAL to the notion of underlay systems as introduced in section 4.3. This engineering approach allows the rapid creation of Web applications with the same expressiveness as any common programming language. However, fundamental programming skills are required to understand the concept of service compositions and service invocations, so that a usage of this approach is limited to software developers only. But because of its expressiveness and flexibility, the engineering approach is recommended for large-scale complex applications.

Since the challenge of this approach's realization mostly lies within the area of software engineering rather than in the area of research, the approach is not discussed in more detail. However, an example for an engineering approach towards the creation of underlay systems is presented later within

Method	Applicability	Expressiveness	Required Skills
<i>Underlay Engineering</i>	Large-scale applications	Classic programming language	For software engineers only
<i>Markup Language</i>	Medium-scale applications	Sequential and parallel execution via markup language; other expressions can be easily incorporated with JavaScript.	For skilled end-users with basic knowledge on computers and simple programming languages.
<i>Effect-driven Underlay Creation</i>	Small applications	Sequential and parallel execution, as well as decision making (if/else or switch/case) possible, but no looping.	For common end-users or software developers of larger applications to automatically generate an application skeleton.

Table 5.1: Methods for the creation of underlay systems.

section 8.4, where some application scenarios based on the concepts developed within this thesis are presented.

Second, a markup language can be used that abstracts from actual service implementations and thereby supports the dynamic integration of services of a given underlay system although the services might be residing on multiple heterogeneous devices. Here, the semantics of the markup language are restricted to sequential and parallel execution of services. The markup language requires a very basic knowledge of markup languages such as HTML; additional knowledge of simple scripting languages such JavaScript support a greater expressiveness of the language on top of parallel and sequential execution. Since rudimentary programming skills are required to develop Web applications via the markup language, it is limited to software developers or skilled end-users (so-called power users). The syntax and the semantics of the markup language can be found in appendix A.

The main focus of this chapter relies on the third approach, which extends the concept of dynamic integration of services. In fact, by using the markup language, the software developer or power user already defines the structure of the workflow implicitly through the order of the single elements of the markup language. The third approach is based on an effect-driven creation of underlay systems, where the complete underlay system (including the workflow and the dataflow graph) is automatically created based on a

given user request that defines the behavior of the desired Web application by naming the *effects* that it should create. This approach does not require explicit programming skills and is thus open to common end-users. However, since the complexity of the request formulation is linearly dependent on the size of the requested application and the creation method does not support the concept of loops within applications, this approach is best suited to the rapid creation of small Web applications.

As discussed in section 3.2, the complexity of creating applications automatically, i.e. the problem of automatic service composition, has been proven to be double-exponentially difficult [13]. Thus, a major shortcoming of existing algorithms for the automatic creation of service compositions is their complexity and entailed time consumption, which hinders their execution on mobile user devices since users are not willing to wait for an indefinable time span to possibly create the desired application. Therefore, the contribution of this work puts special emphasis on the time restricted automatic creation of underlay systems, where it is focused on the creation of a functionally correct system within a short time span instead of concentrating on the creation of an optimal solution. Here, an algorithm is presented that features heuristics to increase the probability of finding an underlay system for a given request and a given time boundary. For instance, in the event that a user is willing to wait three seconds for the automatic creation of an application at most, the algorithm adapts during composition time based on the currently developed partial solutions and the remaining time in order to increase the chances of successfully coming up with a solution before the given time elapses.

The following sections discuss the automatic creation of underlay systems in greater detail and evaluate it with regard to its ability to comply with a given time limit. Here, section 5.2 introduces service descriptions on an algorithmic basis that are required for the automatic composition algorithm, while section 5.3 discusses the creation algorithm itself in detail. Section 5.4 introduces a heuristic for the dynamic adaptation of the composition algorithm to the remaining time, while simulation results and a discussion of the algorithm's performance are outlined in section 5.5.

5.2 MODELING SERVICE DESCRIPTIONS

Service descriptions can occur on multiple levels, ranging from descriptions for programmatic interfaces to descriptions defining a service's behavior. A couple of these descriptions have already been introduced in section 2.3.1.6. In order to enable an automatic creation of underlay systems, three key pieces

of information on services have to be available. The related descriptions are abstractly depicted in Figure 5.1.

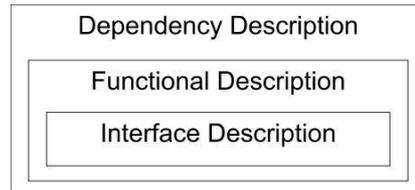


Figure 5.1: Three layers of service descriptions.

First, an interface description is required that supports the formal description of endpoints of a service. These descriptions are essential for the automatic generation of a respective service call. Second, a functional description allows the dynamic integration of services by making services addressable by the function they provide instead of relying on the concrete interfaces they expose. Third, dependency descriptions are required to formalize the relation between multiple services, i.e. to relate their input and output data.

Within this section, an interface description as well as a description capturing a service's functionality and relation to other services are introduced on an algorithmic basis. Both descriptions may later be instantiated and realized by different existing service descriptions, so all introduced algorithms remain independent from concrete service descriptions. Instead, the structures defined in this section can be considered requirements for appropriate service descriptions enabling the automatic creation of underlay systems.

Endpoint descriptions operate at the syntactic level of Web services, and therefore lack the semantic expressiveness to represent the requirements and capabilities of Web services necessary to enable the discovery and automatic composition of Web services. Input, Output, Precondition, and Effect (IOPE) descriptions significantly facilitate the automatic composition of Web services, where preconditions and effects are described as triples of subject, predicate and object [88, 28]. The inputs and outputs are the semantic annotations of the service parameters. However, such a language to describe IOPEs in this way does not yet exist. WSDL-S provides a similar concept to associate semantic annotations with Web services described using WSDL [4]. However, WSDL-S only extends the IO descriptions of existing WSDL documents with semantic annotations; the definition of preconditions and effects as triples over inputs and outputs is not possible. The Resource Description Framework (RDF), which is a framework for representing information in the Web, adapts the concept of triples, each consisting of a subject, a predicate and an object [1]. A set of such triples builds a RDF graph. Therefore, both

approaches are combined to a language that bases on the triple concept of RDF and semantic IOPE annotations for services.

The basic elements of these service descriptions' model are the two data types *Parameter* and *Statement* listed in Algorithms 5.1 and 5.2, respectively. The keyword *structure* is used to define a new data type. Basic data types like *STRING* or *BOOL* are written in capital letters and are used according to their general meaning. A *Parameter* has the property *name* used as a label. The property *modelRef* annotates the parameter with an *URI* identifying the data type of the parameter similar to the data typing concept in RDF. The property *type* identifies whether the parameter is an input (*I*) or an output (*O*), while the property *internal* indicates whether the parameter can only be used within an *operation*.

Algorithm 5.1 *Parameter* structure

```

1: structure Parameter
2:   name : STRING
3:   modelRef : URI
4:   internal : BOOL
5:   type : {I, O}
6: end structure

```

The structure *Statement* links two parameters together using a *URI*, building a triple of *subject*, *predicate* and *object*. *subject* and *object* are parameters within the same operation as the statement, while the *predicate* is a *URI* reference as defined in RDF. The property *type* identifies if the statement is a precondition (*P*) or an effect (*E*). The structure *Operation* listed

Algorithm 5.2 *Statement* structure

```

1: structure Statement
2:   subject : Parameter
3:   predicate : URI
4:   object : Parameter
5:   type : {P, E}
6: end structure

```

in Algorithm 5.3 defines the IOPE behavior of a service similar to the IOPE concept in OWL-S [185]. The properties *inputs* and *outputs* are both sets of *Parameter* and the *preconditions* and *effects* are sets of *Statement*. A *Statement* within an *Operation* is a triple over parameters of the same operation.

Algorithm 5.3 *Operation* structure

```

1: structure Operation
2:   inputs :  $\mathcal{P}(\text{Parameter})$ 
3:   outputs :  $\mathcal{P}(\text{Parameter})$ 
4:   preconditions :  $\mathcal{P}(\text{Statement})$ 
5:   effects :  $\mathcal{P}(\text{Statement})$ 
6: end structure

```

A *Service* is built by a set of operations as shown in Algorithm 5.4. If the value of the property *user* is not empty, the defined service works as a *context service* for this user. Context services are special services that only contain operations encompassing output parameters and effects. Since these services do not contain operations featuring preconditions or input parameters, its inputs are available and its effects are true without any restrictions. Within the scope of section 3.2.2.3.2, these services were defined as constants, i.e. denote information that does not require any prerequisites. Thus, context services can be used to model information that is always available for a given user, such as his or her personal preferences. A *ServiceRepository* is

Algorithm 5.4 *Service* structure

```

1: structure Service
2:   operations :  $\mathcal{P}(\text{Operation})$ 
3:   user : USER
4: end structure

```

defined as a container of services as listed in Algorithm 5.5. A simple ex-

Algorithm 5.5 *ServiceRepository* structure

```

1: structure ServiceRepository
2:   services :  $\mathcal{P}(\text{Service})$ 
3: end structure

```

ample demonstrating the usage of this model is depicted in Figure 5.2. The service operation 'upload_photo' has three input parameters where i_1 represents a photo ($modelRef=http://example.org/photo$), i_2 the file that should be uploaded ($modelRef=http://example.org/file$), and i_3 the title of the photo ($modelRef=http://example.org/title$). Furthermore, the operation has one output parameter o_1 representing the URL of the photo after the upload ($modelRef=http://example.org/url$). The two preconditions $p_1 = (i_1, http://example.org/has, i_2)$ and $p_2 = (i_1, http://example.org/has, i_3)$ are used to ensure that the file and the title belong to the same photo. The same

applies to the effect $e_1 = (i_1, \text{http} : // \text{example.org/has}, o_1)$ which ensures that the output parameter URL denotes the same photo as represented by parameter i_1 .

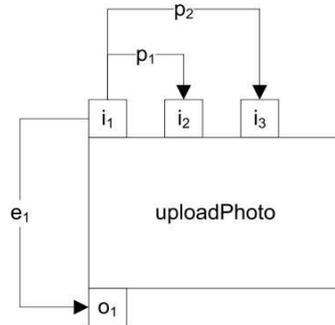


Figure 5.2: Example for semantic service descriptions.

5.3 AN ALGORITHM FOR THE AUTOMATIC CREATION OF SERVICE COMPOSITIONS

Based on the lightweight model for the semantic description of services outlined in the previous section, an algorithm for the automatic composition of services is introduced in the remainder of this section. The algorithm conceptually combines forward and backward chaining principles and thereby reduces the algorithm's dependencies on a certain search strategy. The algorithm can develop multiple services compositions for a given request in parallel and decide within every step which partial service composition is chosen for the next step. Section 5.5 evaluates possible configurations of the algorithm and identifies strategies to enhance the algorithm's success rate for a given time boundary.

Data types necessary to describe composition requests are defined in section 5.3.1, which represent the entry point of the algorithm. In section 5.3.2, a service discovery mechanism is discussed that enables the lookup of services that can be added to the partial service composition within a next step. Section 5.3.3 deals with the concept of composition nodes that describe partial service compositions during the composition process. In addition, appropriate service discovery requests are derived based on the current state of the service composition nodes. A detailed description of the composition algorithm can be found in sections 5.3.4-5.3.9. Within section 5.4, a pool of composition nodes is introduced that can be dynamically adapted in its

size by means of heuristics and supports the development of multiple service compositions in parallel. Section 5.5 finally evaluates different heuristics operating on the pool of compositions nodes with regard to their capability of increasing the odds to find an appropriate service composition within a predefined time span.

5.3.1 Request Definitions

The composition request builds the entry point of the composition algorithm, which creates its initial context (using *CreateInitialNode()*) based on the information represented by this composition request. Such requests define available parameters and preconditions available as inputs in the produced final composition, and desired parameters and effects that represent the respective outputs of the composition. The structure *CompRequest* listed in Algorithm 5.6 encapsulates the properties necessary to define a new composition request. The property *avInputs* denotes a set of parameters representing the available inputs, and *deOutputs* is a set of parameters representing the desired outputs. Similarly, *avPreconditions* constitutes a set of statements representing the available (or fulfilled) preconditions and *deEffects* holds a set of statements describing the desired effects of the composition. The property *requestor* contains an identifier from the user making the request and is necessary to incorporate context information of this user during the composition process.

Algorithm 5.6 *CompRequest* structure

```

1: structure CompRequest
2:   avInputs :  $\mathcal{P}(\text{Parameter})$ 
3:   deOutputs :  $\mathcal{P}(\text{Parameter})$ 
4:   avPreconditions :  $\mathcal{P}(\text{Statement})$ 
5:   deEffects :  $\mathcal{P}(\text{Statement})$ 
6:   requestor : user
7: end structure

```

5.3.2 Service Discovery

The service discovery function is used by the Automatic Service Composition (ASC) algorithm to look up new services during the composition process. In each composition step, the algorithm generates a new discovery request and sends it to the discovery module using the interface *discover()*. The discovery module operates on the semantic repository where the semantic

descriptions of all available services are stored. Based on the received request, the discovery algorithm initiates the search for suitable service operations that are compatible with the information in the request defined in the data type *DiscoveryRequest* listed in Algorithm 5.7. The property *avParameters* of a discovery request contains a set of parameters available during the current composition step, and *opParameters* holds a set of parameters that are not available before this composition step. Coevally, the properties *avStatements* and *opStatements* define two sets of available and open statements before the current composition step. The property *requestor* has the same meaning as in Algorithm 5.6 and identifies the user who has originated the request. The *requestor* is necessary to restrict the selection of services, which acts as source of semantically described context information related to the user defined in the property *user* of Algorithm 5.2. Such special kind of services are referred to as *context services* and expose the following restrictions in contrast to non-context services:

$$cs.user \neq \mathbf{nil} \quad (5.1)$$

$$\forall op \in cs.operation : op.inputs = \phi \wedge op.preconditions = \phi \quad (5.2)$$

where *cs* is a context service. For the composition algorithm, there are no differences between context and non-context services because it acts on service operation and not on services itself. In contrast, the discovery algorithm has to consider these restrictions by selecting context services. The following

Algorithm 5.7 *DiscoveryRequest* structure

- 1: **structure** *DiscoveryRequest*
 - 2: *avParameters* : $\mathcal{P}(\text{Parameter})$
 - 3: *opParameters* : $\mathcal{P}(\text{Parameter})$
 - 4: *avStatements* : $\mathcal{P}(\text{Statement})$
 - 5: *opStatements* : $\mathcal{P}(\text{Statement})$
 - 6: *requestor* : *user*
 - 7: **end structure**
-

statements must be fulfilled for each service operation *op* in the result set *rsp* of the discovery algorithm for a discovery request *req*:

$$\forall in \in op.inputs : (\exists op_1 \in Operation : (op \neq op_1 \wedge in \in op_1.outputs) \vee in \in req.avParameters) \quad (5.3)$$

$$\forall pre \in op.preconditions : (\exists op_1 \in Operation : (op \neq op_1 \wedge pre \in op_1.effects) \vee req.avStatements) \quad (5.4)$$

$$\begin{aligned}
& (op.outputs \cap req.opParameters \neq \phi) \wedge \\
& (op.inputs \cap req.avParameters \neq \phi) \wedge \\
& (op.effects \cap req.opStatements \neq \phi) \wedge \\
& (op.preconditions \cap req.avStatements \neq \phi)
\end{aligned} \tag{5.5}$$

$$\begin{aligned}
& \forall op_1 \in rsp, s \in Service : s = op_1.service \wedge \\
& s.user \neq \mathbf{nil} \Rightarrow s.user = req.requestor
\end{aligned} \tag{5.6}$$

Equation 5.3 denotes that each input *in* of the operation *op* can be connected with at least one output parameter of any other operation in the repository or with an available parameter from the discovery request. The same applies to preconditions and effects described in Equation 5.4. Equation 5.5 makes sure that the IOPE descriptions of the operation *op* can be connected with at least one parameter or statement of the discovery request. Equation 5.6 constitutes that only context service operations of the same user as in the discovery request can be selected.

5.3.3 Composition Nodes as Partial Compositions

Partial compositions describe uncompleted compositions during the composition process. Here, a partial composition contains at least one open input parameter or one open precondition and is represented by the data type *CompNode* listed in Algorithm 5.8. The property *operations* defines a set of service operations used in the current composition. Furthermore, the property *dataflow* defines the flow of data between different operation parameters and statements of the partial composition and is defined as a set of *Parameter-Parameter* or *Statement-Statement* mappings. An input parameter *in* or a precondition *pre* in a composition node *n* is defined as 'open' if the following statements are fulfilled for *in* and *pre*:

$$\forall op \in n.operations : \forall out \in op.outputs : (out, in) \notin n.dataflow \tag{5.7}$$

$$\forall op \in n.operations : \forall eff \in op.effects : (eff, pre) \notin n.dataflow \tag{5.8}$$

Equation 5.7 means that the input parameter *in* is not connected with any output parameter of any operation in the composition node. The same applies to the precondition *pre* for effects of any operation in the composition node as stated in Equation 5.8.

The *layer* assigns a natural number to each service operation in the partial composition representing the layer of the operation in the composition node. It is necessary to detect cycles in the composition. The following

Algorithm 5.8 *CompNode* structure

```

1: structure CompNode
2:   operations :  $\mathcal{P}(\text{Operation})$ 
3:   dataflow :  $\text{Parameter} \rightarrow \text{Parameter} \cup \text{Statement} \rightarrow \text{Statement}$ 
4:   layer :  $\text{Operation} \rightarrow \mathbb{N}$ 
5: end structure

```

statement must be fulfilled for all operations of a composition node n :

$$\begin{aligned} & \forall op_1, op_2 \in n.operations : \\ & (op_1.outputs \times op_2.inputs \cup op_1.effects \times op_2.preconditions) \quad (5.9) \\ & \cap n.dataflow \neq \phi \Rightarrow n.layer(op_1) < n.layer(op_2) \end{aligned}$$

Equation 5.9 states that if at least one outputs parameter or effect of an operation op_1 is connected to an input parameter or precondition of another operation op_2 of the same composition node, the layer number of op_1 must be smaller than the layer op_2 . Thus, the operation op_2 can only be executed when the execution of op_1 is finished. If the node contains at least one cycle, then the statement defined in Equation 5.9 is never fulfilled for any layer numbering of operations in the composition node. This means that when cycles are created, the cyclic dependencies between input and output parameters forestall the service composition's execution at runtime. The challenge of layers is discussed in section 5.3.7 in more details.

5.3.4 General Overview of the Algorithm

Based on the elementary data types, procedures, and semantic service descriptions defined in the previous sections, this section deals with the automatic service composition algorithm in detail. The main structure of the algorithm is listed in Algorithm 5.9. The single input of the algorithm is a composition request $cReq$ as defined in Algorithm 5.6. The algorithm returns a set of composition nodes $cRsp$, representing the response of the algorithm, i.e. a set of complete service compositions. If the result set is empty, the algorithm has not found a composition for the given request. When the result set contains multiple nodes, MAX_NODES represents the upper bound of the number of nodes in the result set.

During the composition process, the algorithm stores open nodes in a pool named *NodePool* which provides the methods *insertNode()* and *removeNode()* to insert or remove a node to or from the pool. The algorithm terminates directly when the pool is empty. The variable *node* represents the active node in each composition step. The initial node is created from the composition

Algorithm 5.9 Composition algorithm

```

1: procedure COMPOSE(cReq : CompRequest)
2:   cRsp :  $\mathcal{P}(\text{CompNode}) := \phi$ 
3:   dReq : DiscoveryRequest
4:   dRsp :  $\mathcal{P}(\text{Operation}) := \phi$ 
5:   pool : NodePool new
6:   busy : BOOL := true
7:   node := CREATEINITIALNODE(cReq)
8:   copy : CompNode
9:   INSERTNODE(pool, node)
10:  while pool.openNodes  $\neq \phi \wedge$  busy do
11:    node := REMOVENODE(pool)
12:    dReq := NEXTDISCOVERYREQUEST(node)
13:    dRsp := DISCOVER(dReq)
14:    for all op  $\in$  dRsp do
15:      copy := node
16:      if INSERTOPERATION(copy, op) then
17:        if ISOPEN(copy) then
18:          INSERTNODE(pool, copy)
19:        else
20:          cRsp := cRsp  $\cup$  {copy}
21:          if |cRsp|  $\geq$  MAX_NODES then
22:            busy := false
23:            break
24:          end if
25:        end if
26:      end if
27:    end for
28:  end while
29:  return cRsp
30: end procedure

```

request using *createInitialNode()* and constructs the entry point for the next composition steps.

In a composition step, the algorithm removes a node from the pool using *removeNode()*. This node represents the active node in this step. Afterwards, a discovery request is generated from the active node using *nextDiscoveryRequest()* and sent to the discovery module using the interface *discover()*. The procedure *nextSearchRequest()* derives the discovery request as defined in Algorithm 5.7 from a partial composition represented by the composition node *n*.

The first step for the generation of a new discovery request is the selection of all open inputs and preconditions from the partial composition represented by the composition node n . The data is stored in the sets $opParameters$ and $opStatements$ of the discovery request, respectively. The same applies to available outputs and effects. The value of the property $requestor$ in the discovery request is the same as the requestor in the composition request. The discovery algorithm as represented in section 5.3.2 searches for new service operations in the semantic repository and holds the result set in the discovery response $dRsp$.

For each service operation op of the discovery response, the algorithm clones the active node and tries to insert op in the copy of the active node using $insertOperation()$. In the event that the operation can be successfully inserted and the node is still open, the algorithm adds the new node to the pool. The procedure $isOpen()$ checks if a composition node is open or not. If the new node is no longer open, the algorithm inserts it into the final result set.

5.3.5 Inserting Operations

The operation $insertOperation()$ listed in Algorithm 5.10 describes the insertion of a new operation into a partial composition represented by a composition node. It constitutes the elementary operation to update an existing node with a new operation. Initially, the new operation is inserted in the

Algorithm 5.10 Insert new operation in the partial composition

```

1: procedure INSERTOPERATION( $node : CompNode, op : Operation$ )
2:    $node.operations := node.operations \cup \{op\}$ 
3:   UPDATESTMTS( $node, op$ )
4:   UPDATEPARAMS( $node, op$ )
5:   return UPDATALAYERS( $node, op$ )
6: end procedure

```

operations set of the given node. Afterwards, the dataflow of the node is updated using the operations $updateStmts()$ and $updateParams()$ listed in Algorithm 5.11 and Algorithm 5.12, respectively. The insertion of an operation is successfully accomplished if it does not entail the creation of a cycle within the resulting dataflow. To check the creation of cycles caused by added operations, a layer concept is introduced in the following section that assigns a layer to every operation and thereby detects cyclic dependencies between the dataflow of multiple services; the exact proceeding is detailed in the operation $updateLayers()$ listed in Algorithm 5.13.

5.3.6 Updating Statements and Parameters

The operation *updateStmts()* listed in Algorithm 5.11 specifies the update mechanism for the statement mapping in the dataflow of a given node after inserting a new operation. First, all open and available statements are selected from the given node using the operations *selectOpStmts()* and *selectAvStmts()*. Afterwards, the update operation checks for each effect of the operation and for each open precondition in the node, whether both triples can be connected using *canConnect()*, i.e., whether a new connection from the effect to the precondition can be added in the dataflow graph. If a connection between the two statements can be added, the mappings between subjects of the statements and mappings between objects of the statements are also added. The same applies to the mapping between available state-

Algorithm 5.11 Update statements after inserting new operation

```

1: procedure UPDATE_STMTS(node : CompNode, op : Operation)
2:   opStmts := SELECT_OP_STMTS(node)
3:   avStmts := SELECT_AV_STMTS(node)
4:   for all eff ∈ op.effects do
5:     for all pre ∈ opStmts do
6:       if CAN_CONNECT(eff, pre) then
7:         node.dataflow := node.dataflow ∪ {(eff.subject, pre.subject)}
8:         node.dataflow := node.dataflow ∪ {(eff.object, pre.object)}
9:         node.dataflow := node.dataflow ∪ {(eff, pre)}
10:      end if
11:    end for
12:  end for
13:  for all eff ∈ avStmts do
14:    for all pre ∈ op.preconditions do
15:      if CAN_CONNECT(eff, pre) then
16:        node.dataflow := node.dataflow ∪ {(eff.subject, pre.subject)}
17:        node.dataflow := node.dataflow ∪ {(eff.object, pre.object)}
18:        node.dataflow := node.dataflow ∪ {(out, pre)}
19:      end if
20:    end for
21:  end for
22: end procedure

```

ments in the given node and preconditions of the operation to insert. Similar to the operation *updateStmts()*, the operation *updateParams()* listed in Algorithm 5.12 updates the parameters after the insertion of a new operation. Here, the operation selects open and available parameters from the

given node using *selectOpParams()* and *selectAvParams()* and checks if output parameters of the new operation can be connected with open parameters or not. When two parameters can be connected, a new mapping between these parameters is added to the dataflow of the given node. The same applies to input parameters of the inserted operation and the available parameters in the given node. The helper method *canConnect()* checks if two

Algorithm 5.12 Update parameters after inserting new operation

```

1: procedure UPDATEPARAMS(node : CompNode, op : Operation)
2:   opParams := SELECTOPPARAMS(node)
3:   avParams := SELECTAVPARAMS(node)
4:   for all out ∈ op.outputs do
5:     for all in ∈ opParams do
6:       if CANCONNECT(out, in) then
7:         node.dataflow := node.dataflow ∪ {(out, in)}
8:       end if
9:     end for
10:  end for
11:  for all out ∈ avParams do
12:    for all in ∈ op.inputs do
13:      if CANCONNECT(out, in) then
14:        node.dataflow := node.dataflow ∪ {(out, in)}
15:      end if
16:    end for
17:  end for
18: end procedure

```

parameters or statements in a composition node can be connected or not. When called with two parameters, *canConnect()* returns *true* if the parameters have the same *modelRef*. In the case of statements, the method returns *true* if the parameters representing the subjects and the objects of the statements can be connected and both statements contain the same predicate.

5.3.7 The Challenge of Cycles and Layers

The following example shows why cycles in the dataflow graph are a problem for the composition algorithm and why it is necessary to detect and eliminate cycles from a composition node. To simplify the example, a semantic repository with the following service operations and the IO parameters *DOC*, *PS* and *PDF* representing a Word-, PS- and PDF-document, respectively, is assumed:

- $doc2ps : DOC \rightarrow PS$
- $ps2pdf : PS \rightarrow PDF$
- $pdf2ps : PDF \rightarrow PS$

The composition request only contains DOC as available input and PDF as desired output. Figure 5.3 (a) shows the initial composition node generated by the composition algorithm. The composition node in Figure 5.3 (b) represents a partial composition after inserting the service operation $ps2pdf$. In the next composition step, the algorithm tries to insert the operation $pdf2ps$ without success because the method $updateLayers()$ detects a cycle in the dataflow graph. If the composition algorithm ignores this cycle, the composition node depicted in Figure 5.3 (c) is returned as result in the response set of the algorithm, since there are no open inputs or preconditions in this node. Here, the algorithm would have produced an incorrect composition, since the plan described in Figure 5.3 (c) is neither executable nor does it generate the desired output PDF .

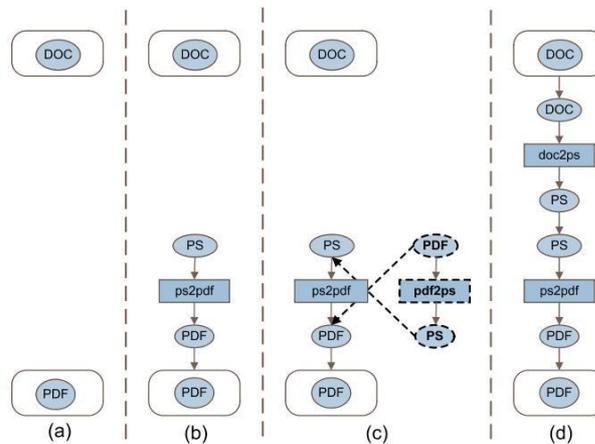


Figure 5.3: Cyclic dependencies between I/O parameters.

Figure 5.3 (d) shows the composition after inserting the service operation $doc2ps$. After updating parameters and statements of the given composition node, the layer numbering for operations of the current node is no longer consistent with the statement defined in Equation 5.9, since the layer number of the new operation is undefined. The $updateLayers()$ method listed in Algorithm 5.13 tries to change the layer numbering so that the statement defined in Equation 5.9 is correct.

The starting point of this method is to set a default value as initial layer number for the new operation. Afterwards, a new initially empty set is

declared to store all moved operations during the update layer process. An operation is moved if its corresponding layer number is changed. After setting an initial value for the layer of the new operation the main algorithm calls *updateLayersRec()*, which then tries to update the layers recursively. First,

Algorithm 5.13 Update Layers after inserting new operation

```
1: procedure UPDATELAYERS(node : CompNode)
2:   node.layer := node.layer  $\cup$  {(op, DEF_LAYER)}
3:   mvOps :  $\mathcal{P}(\textit{Operation})$  :=  $\phi$ 
4:   return UPDATELAYERSREC(node,mvOps)
5: end procedure
```

the algorithm checks whether the layer of the source operation (operation with output *out*) is greater than or equal to the layer of the target operation (operation with input *in*) for each mapping (*out*,*in*) in the dataflow of the given node. In this case, the algorithm tries to move the source operation in front of the target operation and mark the source operation as moved, or to move the target operation behind the source operation and mark the target operation as moved. In both cases the algorithm completes the updating of the layer recursively. Finally, the algorithm returns *true* if the statement defined in Equation 5.9 is correct, otherwise *false*.

```

6: procedure UPDATELAYERSREC( $n : \text{CompNode}, mvOps : \mathcal{P}(\text{Operation})$ )
7:   for all ( $out, in$ )  $\in n.dataflow$  do
8:      $src := operation(out)$ 
9:      $trg := operation(in)$ 
10:    if  $n.layer(src) \geq n.layer(trg)$  then
11:      if  $src \notin mvOps$  then
12:         $oldL := n.layer(src)$ 
13:         $newL := n.layer(trg) - 1$ 
14:        if  $newL > MIN\_LAYER$  then
15:           $n.layer := n.layer \cup \{(src, newL)\}$ 
16:           $mvOps := mvOps \cup \{src\}$ 
17:          if UPDATELAYERSREC( $n, mvOps$ ) then
18:            return true
19:          else
20:             $mvOps := mvOps / \{src\}$ 
21:             $n.layer := n.layer / \{(src, newL)\} \cup \{(src, oldL)\}$ 
22:          end if
23:        end if
24:      end if
25:      if  $trg \notin mvOps$  then
26:         $oldL := layer(trg)$ 
27:         $newL := layer(src) + 1$ 
28:        if  $newL < MAX\_LAYER$  then
29:           $n.layer := n.layer \cup \{(trg, newL)\}$ 
30:           $mvOps := mvOps \cup \{trg\}$ 
31:          if UPDATELAYERSREC( $n, mvOps$ ) then
32:            return true
33:          else
34:             $mvOps := mvOps / \{trg\}$ 
35:             $n.layer := n.layer / \{(trg, newL)\} \cup \{(trg, oldL)\}$ 
36:          end if
37:        end if
38:      end if
39:    end if
40:    return false
41:  end for
42:  return true
43: end procedure

```

5.3.8 Enabling Runtime Decision Making for Varying Effects

In general, it is assumed that effects are always generated during a service's execution, i.e. that the value of the annotated effect is *true*. In this case, the notion of "effect" is sufficient. However, in some cases the generation of an effect cannot be predicted during the design time of a composition. Instead, the decision of whether a service generates an effect or not may only be made during runtime (since it may, for instance, depend on the service's inputs). Based on the current realization of the composition algorithm and the semantic model for service description, it is only definable if an operation produces some effects during design time, although multiple cases exist where the generation of effects depends on input parameters of the service operation. Therefore, the semantic model is extended to support *runtime effects* and *runtime preconditions*. Those runtime statements can either be *true* or *false* during design time, so that both cases have to be covered by the connected preconditions or effects, respectively. Thus, either a runtime effect is connected with a runtime precondition so that the decision making based on the value of the returned effect is considered in scope of the following service, or the runtime effect is connected with two preconditions of different services via an IF/ELSE branch so that both possible values of the runtime effect can be considered. For this reason, the two datatypes *Operation* and *Statement* of the semantic model are extended as listed in Algorithm 5.14 and Algorithm 5.15, respectively. Within the updated definition, the datatype

Algorithm 5.14 Modified *Statement* structure

```

1: structure Statement
2:   subject : Parameter
3:   predicate : URI
4:   object : Parameter
5:   type : {P, E}
6:   correct : BOOL
7: end structure

```

Statement contains the new property *correct* to indicate whether the statement must be correct to be considered as fulfilled or not. If the value of *correct* is undefined, the related statement is either a runtime precondition or a runtime effect. The value of *correct* is then available at runtime. The adapted structure *Operation* contains the new properties *rtPreconditions* and *rtEffects* that define two sets of statements representing the runtime effects and runtime precondition of the operation. There are four possibilities for mappings between (runtime) preconditions and (runtime) effects:

Algorithm 5.15 Modified *Operation* structure

```

1: structure Operation
2:   inputs :  $\mathcal{P}(\text{Parameter})$ 
3:   outputs :  $\mathcal{P}(\text{Parameter})$ 
4:   preconditions :  $\mathcal{P}(\text{Statement})$ 
5:   effects :  $\mathcal{P}(\text{Statement})$ 
6:   rtPreconditions :  $\mathcal{P}(\text{Statement})$ 
7:   rtEffects :  $\mathcal{P}(\text{Statement})$ 
8: end structure

```

1. Mapping from a runtime effect to a runtime precondition ($rtEff \rightarrow rtPre$): This mapping is similar to the mapping between output and input parameter. The value of the runtime effect is forwarded to the runtime precondition at runtime.
2. Mapping from a runtime effect to a precondition ($rtEff \rightarrow pre$): This mapping defines a decision (IF or ELSE) at the composition time, since the value of the runtime effect is not available at the design time but only at the runtime so that the composition algorithm must make sure that the precondition is fulfilled at the composition time.
3. Mapping from an effect to a runtime precondition ($eff \rightarrow rtPre$): In this case the value of the runtime precondition can directly be set at the composition time.
4. Mapping from an effect to a precondition ($eff \rightarrow pre$): The value of *correct* must be equal for both effect and precondition.

5.3.9 From Composition Nodes to Underlay Systems

The result of the composition algorithm is a set of composition nodes. Within a final step, every node is transformed into an abstract service composition. An abstract composition is represented by the *dataflow* and *workflow* graphs defined in the datatype *AbstractPlan* of Algorithm 5.16. The dataflow is the same as that in *CompNode* and connects parameters. The connection $(o, i) \in \text{dataflow}$ denotes that the output parameter o is forwarded to the input parameter i when it becomes available. In contrast, the workflow connects operations in order to define their execution order.

The procedure *createAbstractComposition()* describes the creation of an abstract composition from a composition node and the transformation from a resulting dataflow into a workflow. The first step in the algorithm is the addition of a connection between two operations o_1 and o_2 if there is at least one output parameter of o_1 connected with an input parameter of o_2 in the

dataflow. The goal of the following step is to eliminate redundant connections in the workflow using the operation *isReachable()*. A connection between two operations o_1 and o_2 is redundant if there is a path to reach o_2 from o_1 .

Algorithm 5.16 *CompNode to AbstractPlan*

```

1: structure AbstractPlan
2:   dataflow : Parameter  $\rightarrow$  Parameter
3:   workflow : Operation  $\rightarrow$  Operation
4: end structure

5: procedure CREATEABSTRACTPLAN(node : CompNode)
6:   p : AbstractPlan new
7:   p.dataflow := node.dataflow
8:   p.workflow :=  $\phi$ 
9:   src : Operation
10:  trg : Operation
11:  for all (out, in)  $\in$  p.dataflow do
12:    src := operation(out)
13:    trg := operation(in)
14:    if (src, trg)  $\notin$  p.workflow then
15:      p.workflow := p.workflow  $\cup$  {(src, trg)}
16:    end if
17:  end for
18:  for all (src, trg)  $\in$  p.workflow do
19:    if ISREACHABLE(p, src, trg) then
20:      p.workflow := p.workflow / {(src, trg)}
21:    end if
22:  end for
23:  return p
24: end procedure

25: procedure ISREACHABLE(p : AbstractPlan, src : Operation, trg :
   Operation)
26:   if  $\exists op \in$  Operation :  $op \neq src \wedge op \neq trg$ 
27:    $\wedge (src, op) \in p.workflow \wedge$  ISREACHABLE(p, op, trg) then
28:     return true
29:   else
30:     return false
31:   end if
32: end procedure

```

5.4 ADAPTING THE COMPOSITION'S NODEPOOL: HEURISTICS

The definition of the composition algorithm so far is independent from the realization of the node pool, which is used as a container for composition nodes with two operations *insertNode()* and *removeNode()* to insert into or remove a composition node from the pool. Here, a queue constitutes one possible realization of the node pool. The disadvantage of a queue is that it works according to the FIFO principle, where *removeNode()* always removes the first node from the pool, but not necessary the best node. Furthermore, the size of the queue can grow quickly during the composition. To reduce the time and space complexity, a heuristic-based concept is introduced to adapt the composition node pool. The idea behind heuristics is to use a rating function to evaluate the quality of each node in the pool. Nodes with higher rating values have better chances to be selected in the next composition steps. The data type listed in Algorithm 5.17 declares properties necessary to define a new *NodePool*. The property *node* defines a mapping between order values of nodes in the pool and composition nodes itself. The *rating* defines a mapping between composition nodes in the pool and rating values. Furthermore, the pool stores a limited number of nodes described in the property *capacity*. The property *access* represents the access count to the pool. The

Algorithm 5.17 *NodePool* structure

```

1: structure NodePool
2:   node :  $\mathbb{N} \rightarrow \text{CompNode}$ 
3:   rating :  $\text{CompNode} \rightarrow \mathbb{R}$ 
4:   capacity :  $\mathbb{N} := \text{MAX\_CAPACITY}$ 
5:   access :  $\mathbb{N} := 0$ 
6: end structure

```

node pool provides the two operations *insertNode()* and *removeNode()* listed in the Algorithms 5.18 and 5.19, which can be used to manipulate the pool during the composition. The function *insertNode()* calculates the rating of the new composition node using the operation *calcRating()*, inserts the new node in the pool, and resorts the pool by rating values. If the size of the pool is greater than the maximum capacity after inserting the new node, the node with the smallest rating value is removed from the pool. The procedure *removeNode()* removes a node from the pool and returns it as output. After every call of this operation, the access counter is increased. If the access limit has been reached (*access* > *ACCESS_LIMIT*), the pool capacity is decreased every *ACCESS_STEP* steps to make sure that the composition algorithm terminates after a certain time. The index of the node to remove depends on the strategy used, which is defined in the method *getIndex()*. If

Algorithm 5.18 Insert node in the *NodePool*

```

1: procedure INSERTNODE(pool : NodePool, node : CompNode)
2:   r := CALCRATING(node)
3:   pool.rating := pool.rating  $\cup$  {(node, r)}
4:   idx := max(domain(pool.node)) + 1
5:   pool.node := pool.node  $\cup$  {(idx, node)}
6:   SORT(pool)
7:   if |pool.node|  $\geq$  pool.capacity then
8:     idx := max(domain(pool.node))
9:     tmpNode := pool.node(idx)
10:    pool.node := pool.node / {(idx, tmpNode)}
11:   end if
12: end procedure

```

the “First Fit” strategy is used, the method *getIndex()* returns always the index of the node with the best quality value. However, it is possible that nodes other than the first one will derive the end result. For that reason, heuristics are applied to get the index of the node that is selected for removal. The probability of removing a node from the head of the list is greater than the probability of removing it from the tail. The probability value of selecting a node is calculated using the rating value of the node. These kind of algorithms feature a probability distribution for the selection of promising individuals, and are often referred to as evolutionary algorithms [12], since they follow the “survival of the fittest” principle introduced by Darwin in 1860 [64]. The rating value of a node depends on multiple factors such as the

Algorithm 5.19 Remove node in the *NodePool*

```

1: procedure REMOVENODE(pool : NodePool)
2:   idx := GETINDEX(pool)
3:   node := pool.node(idx)
4:   pool.node := pool.node / {(idx, node)}
5:   pool.access := pool.access + 1
6:    $\Delta$  := pool.access - ACCESS_LIMIT
7:   if  $\Delta > 0 \wedge \Delta \bmod$  ACCESS_STEP = 0 then
8:     pool.capacity := pool.capacity - 1
9:   end if
10:  return node
11: end procedure

```

number of operations in the partial composition, the number of open parameters and statements, the number of available used and unused parameters

and statements, and so forth. The operation *calcRating()* listed in Algorithm 5.20 shows a part of the rating function. *R_PARAM*, *R_OP_PARAM* and *R_AV_PARAM* are rating constants for available used parameters, open parameters and available not used parameters, respectively. A used parameter denotes that the parameter is at least involved in one mapping within the dataflow. In this case, it is possible to give available and used parameters a better rating value than available but unused parameters. The same applies for the rating of preconditions and effects.

Algorithm 5.20 Rating function

```

1: procedure CALCRATING(node : CompNode)
2:   rating :  $\mathbb{R}$  := 0.0
3:   for all p  $\in$  Parameter do
4:     if  $\exists p1 \in$  Parameter : (p, p1)  $\in$  node.dataflow
5:      $\forall (p1, p) \in$  node.dataflow then
6:       rating := rating + R_PARAM
7:     else if ISOPEN(p) then
8:       rating := rating + R_OP_PARAM
9:     else
10:      rating := rating + R_AV_PARAM
11:    end if
12:  end for
13:  for all s  $\in$  Statement do
14:    ...
15:  end for
16:  return rating
17: end procedure

```

5.5 EVALUATION AND VALIDATION

In this section, how the algorithm introduced above can be tailored to increase the chances of finding an appropriate solution in a given time span is demonstrated. Therefore, the algorithm's pool size, i.e. the number of service compositions that are developed in parallel, is dynamically adapted. This adaptation is performed during the algorithm's runtime based on the actual time left for the finalization of a service composition matching the user's request. Thereby, the algorithm also becomes tailored to the computing capabilities of the single devices on which it is executed, since it adapts based on the absolute time left and therefore neglects the time required by the different devices to perform a single step of the algorithm.

for a given service repository, so that which percentage of the given requests can be fulfilled can be evaluated. Requests can have a different difficulty, depending on the number of parameters they are requesting. The set of requests also encompasses a time limit t_{limit} , which denotes the maximal time span allowed to complete a service composition for a given request.

The third element is built by the composition algorithm itself. The algorithm takes a service request and a service repository as input, and tries to create service compositions matching the given requests in time.

5.5.2 *Enhancing the Algorithm's Success Rate*

Operating on a single partial service composition leads to a low success rate, since the chances are high that the integration of service into the current partial composition makes the resulting solution harder or even impossible to find. Classic forward and backward chaining constitute greedy algorithms that do not support means to reevaluate a prior decision during runtime.

Successful algorithms for the automatic creation of service compositions operate on a set of partial service compositions [12].

The algorithm presented in the previous section holds a set of partial service compositions within a node pool. Here, the single nodes are evaluated after every iteration and sorted according to their fitness. A higher fitness of a node results in a higher probability for the selection of the node for the next iteration. Since the pool size is limited, only the most promising partial service compositions are kept. Such algorithms featuring a probability distribution for the selection of promising individuals are often referred to as evolutionary algorithms [57, 12].

The development of multiple service compositions in parallel outperforms blind forward and backward chaining algorithms in every simulation setting. Heuristic-based approaches that dynamically select between multiple partial service compositions within every step are evaluated to be extremely dependent on the request and service domain. In most cases, either an heuristic based backward chaining algorithm considerably outperforms the heuristic based forward chaining algorithm or vice versa. The algorithm presented in this chapter combines forward and backward chaining mechanisms and thereby eases the dependency on the respective search strategy.

Figure 5.5 shows an exemplary simulation result for all five algorithms. Here, the backward chaining approaches perform better than their pendants based on forward chaining; nevertheless, other simulation results demonstrate the exact opposite behavior. All simulations have in common that the combination of both strategies as proposed by the algorithm introduced above

performs slightly worse than the optimal search strategy, but much better than the other search strategy. Thus, without knowledge on the service domain, the proposed algorithm performs better on average than the straight forward or backward chaining strategies.

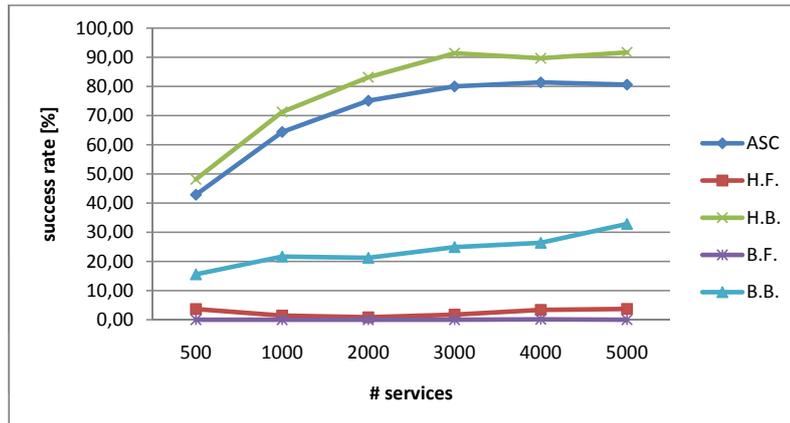


Figure 5.5: Comparison of the success rates for blind forward chaining (BF), blind backward chaining (BB), heuristic forward chaining (HF), heuristic backward chaining (HB), and the novel algorithm for the automatic service composition creation (ASC) introduced in this chapter.

The main objective of the presented algorithm is the reduction of the required time for a given service composition request. Here, the node pool's size has a considerable impact on the speed of the algorithm. While a smaller pool size promises to find a solution more quickly, it increases the risk of running into local optima. Within the remainder of this section, different strategies for the modification of the automatic service composition algorithm's pool size during runtime are investigated. Thus, multiple functions are defined in relation to the user's given time span in order to evaluate the influence of the node pool's size on the composition speed and success rate. The hypothesis followed here is that the pool size should be initially set to a higher value to decrease the chances of running into a local optimum, while it is continuously decreased based on the remaining time, in order to enhance the chances of finding a composition within the given time span.

5.5.2.1 Simulation Settings

A general overview of the simulation setting is depicted in Figure 5.6.

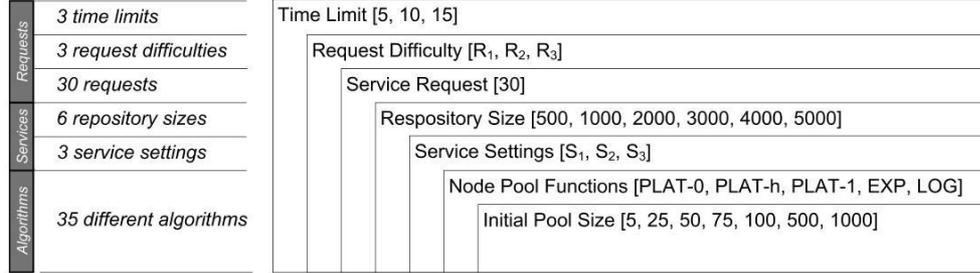


Figure 5.6: Abstract overview of the simulation setting.

For each combination of 3 different time limits and 3 different request difficulties, 30 requests are generated; thus, there are 810 requests in total. The time limits are chosen as 5, 10 and 15 seconds.

The three types of request difficulties are defined by the varying number of inputs, outputs, effects and preconditions they possess. A higher number of parameters leads to a more complex composition process, since the search space has been expanded. Table 5.2 lists the three requests difficulties.

Parameter	R_1	R_2	R_3
Input range	[1 – 5]	[2 – 6]	[3 – 7]
Output range	[1 – 3]	[2 – 4]	[3 – 5]
Precondition range	[1 – 4]	[1 – 4]	[1 – 4]
Effect range	[1 – 4]	[1 – 4]	[1 – 4]
Average number of inputs	3	4	5
Average number of outputs	2	3	4
Average number of preconditions	2	2	2
Average number of effects	2	2	2

Table 5.2: Simulation settings for request difficulties.

The requests are applied to six different service repositories containing 500, 1000, 2000, 3000, 4000, and 5000 services, respectively. The services within the repositories are generated based on multiple different settings; the values assumed for the generation of the services itself are listed in Table 5.3 for two exemplary settings S_1 and S_2 .

Parameter	S_1	S_2
Input range	[1 – 2]	[2 – 2]
Output range	[1 – 2]	[5 – 5]
Precondition range	[1 – 2]	[2 – 2]
Effect range	[1 – 2]	[5 – 5]
Average number of inputs	1	2
Average number of outputs	2	5
Average number of preconditions	1	2
Average number of effects	2	5
Input types	10	10
Output types	10	10
Precondition types	10	10
Effect types	10	10
Verb types	10	10

Table 5.3: Simulation settings for the generation of services.

In total, 35 different types of node pool functions, and therefore algorithms are investigated. The five functions listed in Figure 5.4 are invoked with 7 different initial pool sizes: 5, 25, 50, 75, 100, 500, and 1000.

Thus, 810 different requests are applied to 18 different service repositories; two values are measured. First, the *success rate* denotes the percentage of requests for which a service composition was found within the given time limit. The *duration* is the average time that was consumed by an algorithm for the processing of all requests. For instance, in the event that none of the given 30 requests can be met within a given time limit of 5 seconds, the success rate is zero and the average duration of the algorithm equals 5 seconds.

The simulations were performed on a cluster server with 4GB of RAM and 4 kernels running at 2,33 GHz each.

5.5.2.2 Node Pool Functions

Five different functions are evaluated that are defined in accordance to the time limit t_{limit} given by the user, where p denotes the initial pool size; the functions are listed in Table 5.4 and illustrated in Figure 5.7.

Name	Function
PLAT-0	$\text{PLAT-0}(t) = p - p \frac{t}{t_{limit}}$
PLAT-h	$\text{PLAT-h}(t) = \begin{cases} p, & 0 \leq t < t_p \\ p - p \frac{2 \cdot t - t_{limit}}{t_{limit}}, & t \geq t_p \end{cases}$
PLAT-1	$\text{PLAT-1}(t) = p$
EXP	$\text{EXP}(t) = 1 + p \cdot e^{a \cdot t}, a = \frac{\log(1+p)}{t_{limit}}$
LOG	$\text{LOG}(t) = p \cdot e^{a \cdot t}, a = \frac{\log(\frac{0.99}{p})}{t_{limit}}$

Table 5.4: Functions operation on the algorithm's pool size.

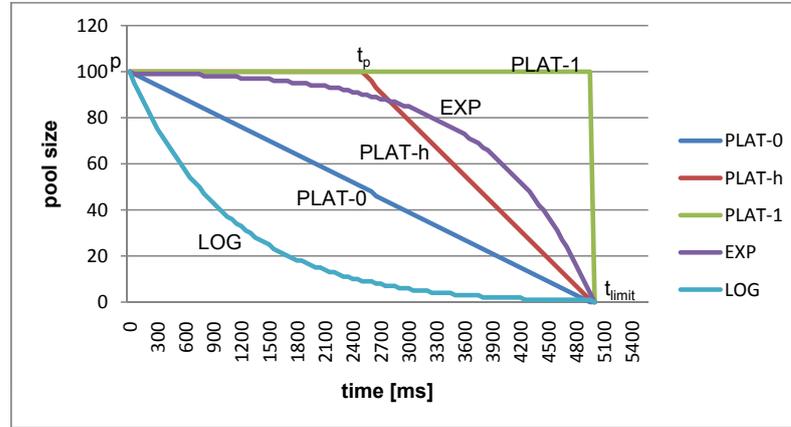


Figure 5.7: Functions operating on the composition algorithm's pool size.

There are three different types of plateau functions, which start at t_0 with an initial pool size p and remain on the same level until t_p is reached. Henceforward, the function falls linearly, so that p equals 1 at t_{limit} . The different types of plateau function are derived from the definition of t_p . The plateau function either falls linearly from the very beginning ($t_p = t_0$), falls linearly after half the given time limit has elapsed ($t_p = \frac{t_{limit}}{2}$), or does not fall at all, i.e. remains constant ($t_p = t_{limit}$). The exponential function EXP remains close to the initial pool size and falls shortly before t_{limit} is reached, while LOG rapidly decreases the initial pool size and runs asymptotical to the ordinate for most of the given time.

In addition to the function for the modification of the pool size itself, the initial size of the node pool has an impact on the respective algorithm's performance. Note that an initial pool size of 1 makes the algorithm greedy, since only one partial service composition is developed over the whole time span.

For the remainder of this section, the node pool functions are considered to imply a novel algorithm. Thus, *algorithm EXP*(p) denotes the automatic service composition algorithm introduced above that uses the EXP function to regulate the algorithm's pool size, where the initial pool size is set to p .

In the following section, the quality of the single algorithms is evaluated.

5.5.2.3 Evaluation

In section 3.2, the difference between optimization heuristics was discussed. Here, user-centric algorithms are characterized by the use of additional information stemming from the user, but not from the service domain. In fact, assuming the possession of knowledge on services within the Web domain is highly unrealistic. There are no numbers or evaluation reports on the average number of service parameters. In addition, such results would become rapidly inaccurate due to the continuously changing availability of services and their appearance.

To illustrate the difference between user-centric and targeted algorithms, assume that the number of services within a repository is known by the algorithm. In that case, the optimal initial pool size can be chosen for every function. When this optimal pool size can be selected for every algorithm based on the knowledge of the repositories' size, all functions lead to a similar success rate; the comparison is illustrated in Figure 5.8.

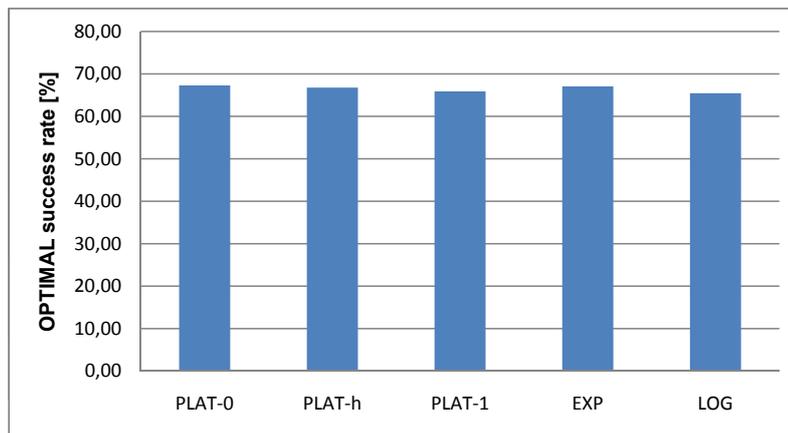


Figure 5.8: Comparison of the algorithms' success rate when the service repository's size is known.

It is notable that, while the success rate for all functions is similar when their initial pool size is optimally chosen, the algorithms differ in their dura-

tion. Figure 5.9 shows the average time consumption of the different functions within a given time limit of 5 seconds.

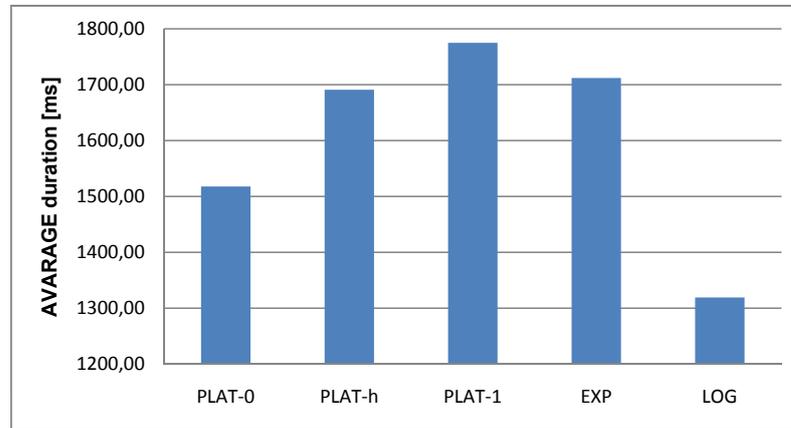


Figure 5.9: Duration of the single functions in average, where the initial pool size is optimally chosen in dependence of the service repository’s size.

The logarithmic function considerably outperforms the remaining ones. In average, the algorithm equipped with the LOG function is 26.9% faster than the any other algorithm. Notably, the LOG algorithm outperforms the PLAT-1 algorithm, which does not perform any modifications on the node pool’s size, by 34.5%. The fact that the second best performance is reached by the linearly decreasing function supports the theory that a greater initial pool size and its fast reduction over time enhances the composition algorithm.

Since the functions should be optimized for a user-centric algorithm, no additional knowledge of the service domain is assumed to be present within the remainder of this section. Therefore, the average over all repository sizes is built for the following evaluations. When neglecting the service repositories’ size, it can be observed that all algorithms perform the best with a similar initial pool size. However, the optimal initial pool size for the single functions depends considerably on the complexity of the services within the service repositories. Figure 5.10 shows the different optimal initial pool sizes for the single algorithms with settings S_1 and S_2 , respectively.

The evaluations above have shown that the optimal initial pool size is dependent on the service repository’s size and the included services. Thus, without domain knowledge, the optimal initial pool size cannot be determined. However, the initial pool size has a great impact on the performance of the composition algorithm. Figure 5.11 shows the performance of the single algorithms with regard to their initial pool size. Here, it can be observed that an initial pool size between 25 and 100 leads to good success rates, while

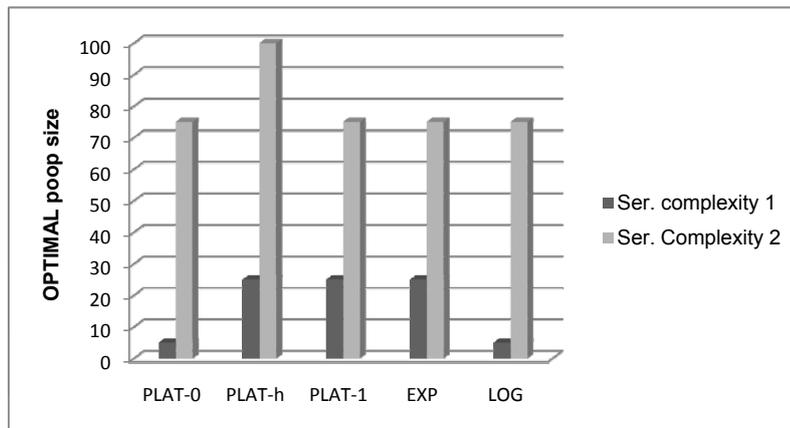


Figure 5.10: Different optimal pool sizes depending on the service settings.

a smaller pool size or a pool size greater than 100 leads to a considerably lower success rate.

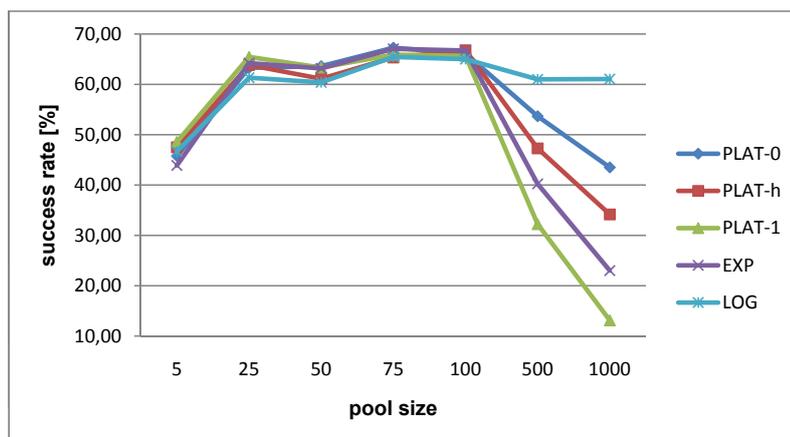


Figure 5.11: Success rates for all algorithms based on their initial pool size.

Since there is no additional information on the service domain, the single functions are evaluated with regard to their independence to the given domain.

In the next evaluation step, the deviation between the single algorithms' performance with different initial pool sizes and the algorithms' performance with an optimal initial pool size is computed. It can be observed that the logarithmic function is less dependent on the initial pool size as the other functions; Figure 5.12 illustrates this independence. Especially higher pool

sizes can be better absorbed by means of a logarithmic function. Figure 5.13 shows the average deviation of the single algorithms by a varying initial pool size. The figure confirms the conclusion that functions with faster decreasing pool sizes expose less deviation in their performance. While the LOG algorithm performs on average only 5.32% less than the optimal algorithm, the PLAT-1 algorithm dramatically depends on a proper initial pools size and performs on average 15.29% less than with an optimal initial pool size.

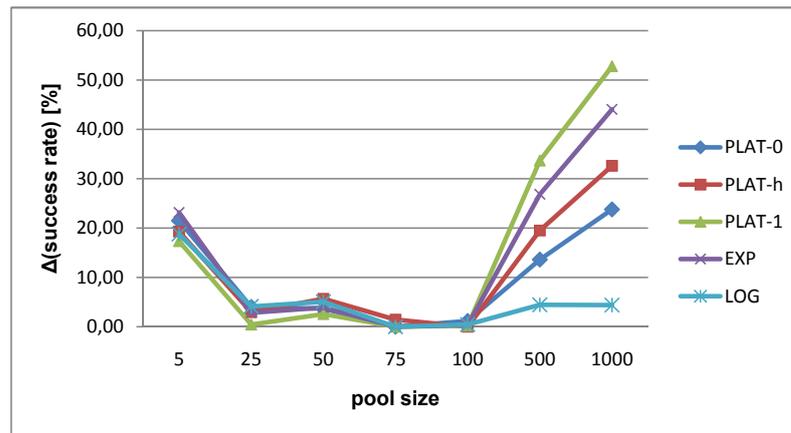


Figure 5.12: Deviation of the algorithms' success rates depending on their initial pool size.

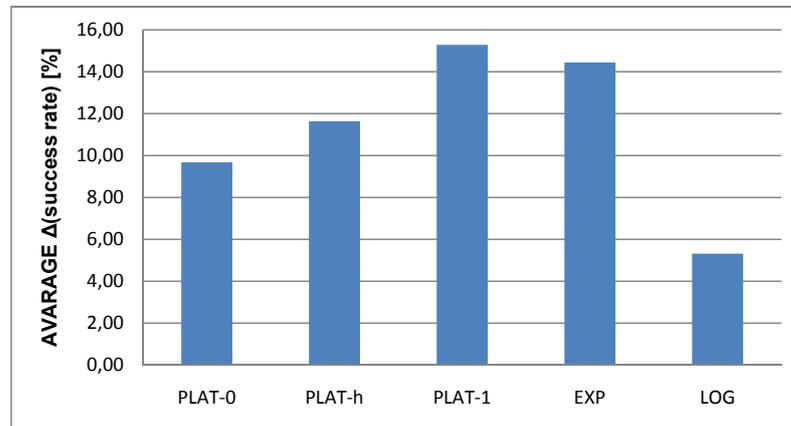


Figure 5.13: Average deviation of the algorithms' success rate depending on their initial pool size.

The same result holds for the duration of the single algorithms. Figures 5.14 and 5.15 illustrate the algorithm's total and average deviation with regard to their duration, respectively.

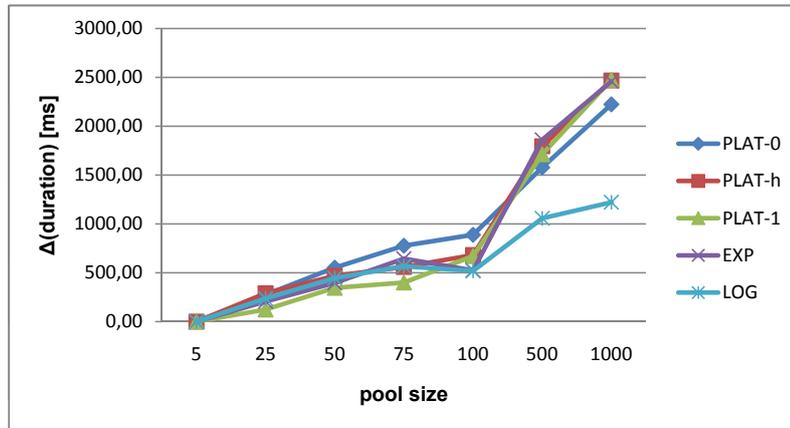


Figure 5.14: Deviation of the algorithms' duration depending on the initial pool size.

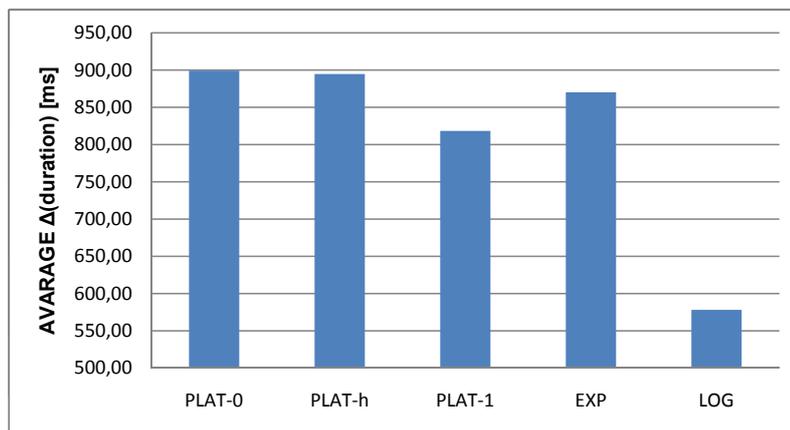


Figure 5.15: Average deviation of the algorithms' duration depending on the initial pool size.

5.5.3 Conclusion and Discussion

When an upper time limit is given by the user, functions can be defined that adapt the algorithm's pool size based on the remaining amount of time. Thus, after every iteration of the algorithm, the pool size is decreased in order to enhance the chances of finding a solution for the given user request within the predefined time span. This optimization was evaluated for user-centric algorithms, i.e. algorithms that do not build upon additional knowledge on the service domain, but only receive information from the user's side.

Notably, the evaluated algorithms expose no change in their success rate when domain knowledge is available. However, the configuration of the algo-

rithm, especially the initial pool size, denoting the number of partial service compositions developed in parallel, is highly dependent on the service domain itself.

It can be observed that the logarithmic function leading to a rapid diminution of the node pool's size is less dependent on the initial node pool size and consequently on the service domain. Thus, algorithms incorporating a logarithmic function for the adaptation of the algorithm's node pool size over time considerably increase the odds of finding an appropriate service composition for a given request and speed up the creation of the respective composition at the same time.

CHAPTER 6

DISTRIBUTING UNDERLAY SYSTEMS

This chapter deals with the automatic distribution of underlay systems to support the distributed execution of Web applications. Three key phases are distinguished. First, a protocol handles the allocation of concrete services to abstract services. Thus, multiple devices, each hosting a finite set of concrete services, negotiate the allocation of concrete services to the abstract services of the underlay system. Second, the underlay system is partitioned so that each device receives a sub-underlay system that is responsible for the execution of its local services. Third, the sub-underlay systems are deployed on the single devices and executed in parallel, where the partitioning algorithm has split up the given underlay system in a way that preserves its original undistributed behavior.

This chapter concentrates on the partitioning algorithm itself. The specification of the communication protocol ensuring the negotiation of services bindings, as well as the coherent execution of the distributed underlay system is discussed in chapter 7.

6.1 MOTIVATION

The benefits of distributing underlay systems are mainly threefold. First, services are incorporated dynamically during runtime, which enables a QoS-based and personalized selection of services and supports the dynamic recovery of a Web application during runtime when a single service fails. By incorporating local services preferentially, parts of the application logic can be executed in an offline mode, i.e. without communication to Web servers. Second, the execution of Web applications can be optimized by the distribution of application logic, where services requiring a high degree of user interaction can be outsourced to the client side. Third, private user data can be handled locally or at a single, trusted site, and do not necessarily have to be sent to untrustworthy or unknown 3rd party service providers.

In the event that multiple concrete services are available to replace an abstract one in a given underlay system, multiple strategies may be applied to guide the decision making process. A user may want to create a mashup

that possesses offline capabilities. In that case, services are always integrated at the client side so that they do not require Web access and thus are part of the application that remains executable in times of disconnection.

Alternatively, a user may want to reduce the communication during the applications's execution in order to increase its responsiveness and limit data transfer and thus costs. In that case, the allocation of services can be performed in a way that the greatest possible set of services possessing data related dependencies are executed on single devices so no inter-device communication is required.

There are many other methods to determine the allocation of concrete services to abstract ones. Most of them (except the ones outlined above) require additional information on the services itself. The most simple additional information on services may be a *server* or *client* tag attached to a service, which indicates whether a service is better suited for a client or server-side execution. Moreover, additional information on the quality of a single service may be present and could be employed for the decision making process. For instance, it is possible to measure the resource consumption of a service during its execution and to store it in a lightweight model so that the information of a service's quality can be considered during succeeding decisions. A fuzzy logic based model for the representation of such non-functional service properties, and a proof-of-concept application by employing the model for the measurement of a service's memory consumption and response time have been published in [146] and [100], respectively. However, since the scope of possible additional information on services is broad and requires the conceptual extension of a service's description, this thesis focuses on the definition of interfaces for the integration of additional knowledge of services and does not deal with the respective mechanisms themselves.

The partitioning algorithm presented in the next section thus abstracts from the decision making process that has led to the allocation of concrete services to abstract services. It operates on a given underlay system, whose abstract services have already been replaced by concrete ones, so every transition can be uniquely imputed to the device that hosts the annotated service.

During the specification of the components required for the realization of the underlay systems conceptually introduced in chapter 7, the allocation of services is exemplarily performed by the means of additional QoS properties and user profiles.

6.2 STATE MANAGEMENT AND PRESENTATION MAPPING

Within the Web, state management and presentation mapping is designed according to the Model View Controller (MVC) pattern [103]. The MVC pattern defines a *model* that captures the application logic, and a *controller* that communicates changes in the application's state to the *view*, which can then be adapted accordingly.

Within the scope of the discussed approach, the underlay system represents the application logic and is thus identified with the model. As introduced in section 4.3.6, the underlay operates on a set of resources, which is then referred to as resource space. Every resource is represented by a set of key-value pairs that specify the current state of the resource. The controller of the MVC pattern is conceptually realized by a resource listener operating on the resource space, i.e. whenever a resource is modified, the modification is transmitted to the view, i.e. the client-side representation of the application. In the event that the modified resource possesses a presentation, the application's presentation, i.e. its view, is adapted accordingly.

For instance, the resource space may contain a resource `map`, which contains its position given by coordinates of the center of the map and a zoom level. When the position of the map is modified, e.g. because a user has searched for a city and a service has centered the map accordingly, the coordinates of the resource are modified by the service's execution. The modification is translated to the view, which then performs an update so that the map is centered on the city the user has requested.

The following section introduces a mechanism to split up an underlay system into multiple sub-underlay systems, which can then be executed in a distributed way while retaining the same behavior as the original underlay system. Here, every device hosting one of the sub-underlay systems holds a copy of the application's resource space. In the event a resource is modified within the scope of a service execution of a sub-underlay system, the changes have to be communicated to the other involved devices. In order to reduce the communication overhead, resource modifications are only updated if they are required directly as input for a service invocation within a remote sub-underlay system or when the resource modification possesses a presentation that requires an update of the view at the clients' side. The following section addresses this problem in greater detail.

In order to support the protection of private data within composed applications as well as to deal with multi-user scenarios, two additional properties are assigned to resources. First, a resource may be declared *private*. A private resource can only be read or modified by services of the device that hosts the resource. Moreover, no service that receives input from a remote

service can be applied to a private resource, since this external input may cause a behavior in the local service that modifies the resource in a malicious manner.

Multi-user scenarios are supported by defining a *scope* for every resource, which specifies the set of devices aware of the resource's state. By default, this property is set to *all*, so that every device holds a copy of the resource's state. In the event that a service accesses a resource, it first aggregates the most recent state of the resource, locks access to it, executes its operation on the resource and finally releases it again. Thereby, every service ensures that it is operating on the most up-to-date version of a resource. When a resource is only of interest to a subset of devices, the scope can be reduced so only the selected subset of services has access to the respective resource. Section 7.3 discusses access to resources in greater detail.

A possible scenario for the usage of resources with restricted scopes is a competitive geo-caching game. In this example, each user gets a map that is centered on his or her current position, together with a marker on the map that indicates a target location. The users compete to reach the given location, where the user entering a certain vicinity of the given target location first is granted a point and a new target location is generated. Given n clients and a server, $n + 1$ *location* and n *map* resources are defined. The scopes of the resources are restricted in a way that every client has access to a personal map that is centered on his or her current location. All clients are shown the same location by a marker on their maps. The first client that enters a predefined vicinity of the target location gets a point and the target location is then changed by the server. The scopes of the single resources are depicted abstractly in Figure 6.1.

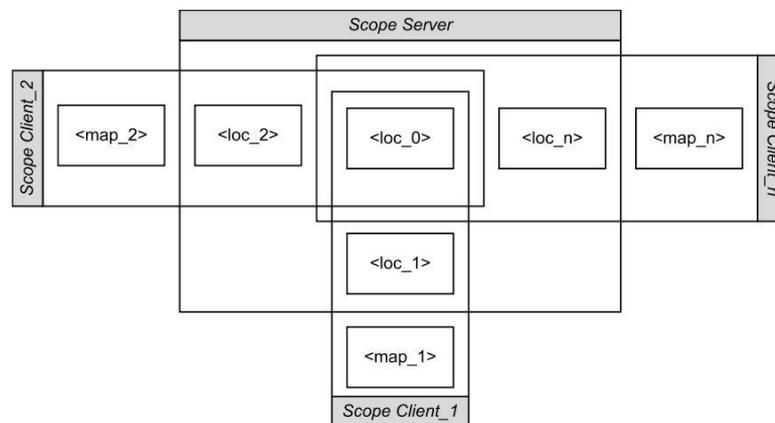


Figure 6.1: Visualization of a possible resource scope for a multi-client geo-caching game.

Here, each client i holds its current location within the resource $\langle loc_i \rangle$ and its personal map $\langle map_i \rangle$ that is centered on $\langle loc_i \rangle$. The other clients do not have access to the other clients' maps or locations and are thus unaware of their position. The target location $\langle loc_0 \rangle$ is defined within the scope of all clients and the server, so that all clients share the same target location. The server is also capable of taking the single clients' locations and matching them against the target location and thereby determines which client has reached the target location first.

6.3 PARTITIONING ALGORITHM

In this section, an algorithm for the automatic distribution of underlay systems is given.

The algorithm's main objective is the generation of multiple sub-underlay systems from a single underlay system, so that the single parts of the underlay can be distributed among multiple devices and executed concurrently. The algorithm ensures that the distributed execution of the underlay system is equivalent to the execution of the holistic underlay, independent of the allocation of the involved services.

The distribution of a given underlay system must thus be invariant with regard to

1. the execution order of services as defined by the workflow graph,
2. the data passage between the single services as defined by the dataflow graph,
3. the communication between workflows, and
4. the invariants and guards defined within the workflow including clocks.

6.3.1 General Terminology

The algorithm takes an underlay system, as defined in chapter 4, as input. Within chapter 7, a method for the allocation of concrete services stemming from different devices to abstract services is introduced. For the distribution algorithm presented in this section, it is assumed that this allocation has already been performed. Here, the result of this allocation is considered abstractly as a *coloring* of the underlay system's workflow graph. Each transition¹ of the workflow graph is assigned a color, which reflects the allocation

¹A transition is commonly related to a service invocation that is initiated when the transition is passed. Since this chapter deals with the distribution of workflow structures, it is focused on the respective terminology of timed automata. Nevertheless, a colored transition is always considered a service invocation.

of the respective service to a device. All transitions with the same color invoke services that are hosted on the same device. A coloring can thus be considered as a function $\varphi : T \rightarrow C \cup \text{null}$ operating on a finite set of transitions T and a finite set of colors C that assigns a color to each transition. In case a transition is assigned `null`, the transition is considered as *uncolored*.

Figure 6.2 shows an exemplary workflow graph, whose transitions are marked with two different colors.

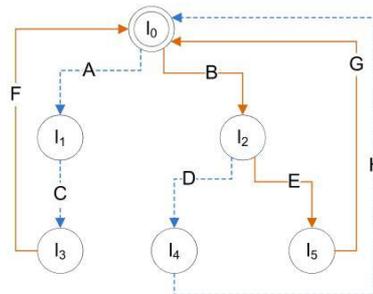


Figure 6.2: Colored transitions. A , C , D and H build the first partition (blue dashed arrows), B , E , F and G the second one (orange bold arrows).

The distribution algorithm operates mainly on an underlay system’s workflow graph and derives the dataflow graph from the distributed version of the workflow. The result of the distribution algorithm is a set of *partitions*. A partition is a workflow graph that results from the distribution algorithm and is capable of the execution of all services of one specific device. The number of partitions that are created is therefore dependent on the number of different colors present in the original underlay system’s workflow graph.

A *transition group* is defined as a set of transitions which possess the same coloring. Every transition group is part of exactly one partition. Given a partition p and a colored transition t , t is considered as *local transition* if it belongs to the transition group that is identified with p ; otherwise, it is considered as *remote transition*. Uncolored transitions are neither local nor remote.

Given the exemplary workflow graph depicted in Figure 6.2, the distribution algorithm creates two portions, since there are two different colors of transitions. The transitions A , C , D and H build a transition group and are thus part of the same partition p . They are local to p , while the transitions B , E , F and G are remote to p .

6.3.2 Conceptual Process of the Partitioning Algorithm

Within this section, the general process of the algorithm is introduced and discussed with the background of a concrete example.

6.3.2.1 General Proceeding

The algorithm creates a partition that contains the initial location for every color found within the origin workflow graph. Starting with the initial location, the partitioning algorithm traverses the origin workflow graph in Breath-First-Search (BFS) manner, while it adds every passed transition to one of the partitions so that all partitions only encompass transitions from one transition group. This distribution of transitions breaks the inter-service communication determined by the original structure of the workflow graph. Therefore, additional synchronization messages are generated and added to the single partitions, which ensure that the execution and communication flow between the single partitions remains invariant with regard to the origin workflow.

For instance, given a sequence of service executions represented by transitions t_1 and t_2 , the origin workflow ensures that t_2 is not invoked before t_1 has finished its execution. When t_1 and t_2 are labeled with different colors, an additional communication primitive is required to notify the partition containing t_2 that the execution of the services invoked via t_1 has been finished.

Although the distribution of underlay systems provides advantages, as discussed in section 3.4, it might also lead to increased communication costs. Their size depends on the allocation of services to devices and the dependencies between services with regard to both data passage and execution order. Thus, the level of communication depends on the coloring of the underlay system's workflow graph.

Despite the fact that the coloring of transitions takes place before the partitioning mechanism begins, the partitioning algorithm aims at a reduction of the communication overhead caused by the partitioning itself, thus, aims at partitions that cause the least possible amount of additional message exchange.

In particular, a synchronization signal that notifies other partitions should not be issued after each service execution. Instead, the synchronization message is held back as long as the partition can still execute local services. When the partition can no longer invoke local services, a synchronization is sent that represents the set of service invocations performed by the partition. This mechanism is referred to as *delayed synchronization* in the following.

The process of copying transitions to the partitions works as follows. For each location of a partition p , all outgoing transitions are examined in the origin workflow graph. The transitions with the matching color are copied. Other transitions may be copied as well in order to provide exchange of synchronization primitives or checking of conditions. Nevertheless, they are deprived of the annotations given in the origin workflow graph.

The new target location is determined by a *getNext()* procedure. The procedure takes a partition p and a transition t as input. If t is local to p , the target location of t within the origin workflow graph is returned. Thereby, the transition and its corresponding service invocation are taken into the partition. When t is remote to p , the algorithm follows the path indicated by t in Depth-First-Search (DFS) manner until a location is reached, which either has an outgoing transition local to p or possesses multiple outgoing transitions. By combining BFS and DFS, all transitions that do not belong to p and form a path within the origin workflow graph can be substituted by a single transition within p .

In the case that the DFS ends because a location with a local outgoing transition was found, the location becomes the target location within p . In the case that a location initiating a split is discovered, i.e. a location possessing multiple outgoing transitions, a BFS is applied to discover the next transition local to p .

Within the following section, the general proceeding of the algorithm is discussed by means of a working example.

6.3.2.2 Working Example

This section discusses the application of the partitioning algorithm to the workflow graph depicted in Figure 6.2. The core functionality of the algorithm is given in Algorithm 6.1.

The algorithm operates on a number of sets. Here, *partitions* keeps the workflow's partitions, which are continuously built up during the algorithm's execution. The set *currentLocations* encompasses all locations that are added to partitions within the next *step* of the algorithm. Here, steps of the partitioning algorithm are defined as iterations over all locations found within the last BFS step, which have not yet been discovered during a prior step. In other words, a step encompasses an iteration over all members of *currentLocations*. The procedure runs until the set is empty. At the beginning, it contains the initial location of the originating workflow. After each step, the set is updated with new locations found during the BFS.

In addition, each workflow partition holds a corresponding *nextLocations* set. It contains locations, whose outgoing transitions are investigated next and are, if required, copied into the partition. These locations are targets of transitions which were added in the last algorithm's step. They are obtained by calling the *getNext()* procedure and are required to check the relevance of the current locations for each partition. Initially, it contains the origin workflow's initial location.

Within the remainder of this section, the partitioning of the workflow graph given in Figure 6.2 is discussed. Since the originating graph's transitions are two-colored, two new partitions p_1 and p_2 are created by the algorithm. Table 6.1 shows the state of the sets *currentLocations* and the two partitions' *nextLocations* for every step of the algorithm. Table 6.2 lists the parameters for the invocation of *getNext()* after each step and its respective result. The partitions resulting from each step are illustrated in Figure 6.3.

During every step of the algorithm

1. each partition's *nextLocations* are extended with results returned by *getNext()* for the specified partition, and
2. *currentLocations* are subtracted from *nextLocations*.

Finally, *currentLocations* is overwritten with new locations found by the algorithm's BFS.

Initially, all sets contain the workflow's initial location l_0 .

Within the first step, *getNext()* is called for each outgoing transition from each partition's *nextLocations*' member. As there are two outgoing transitions from l_0 , which is the only member of both sets, the *getNext()* function is called four times. The function's outputs are added to *nextLocations*. At the end of the first step, p_1 's *nextLocations* contains l_1 and l_2 , since they were returned by *getNext()* and called with the parameter p_1 . l_0 was removed from the set, because it was contained in *currentLocations*. The other *nextLocations* were modified accordingly.

Transition A is copied to p_1 without any modifications. Transition B is also added to p_1 ; however, its action is removed, since it is executed within the scope of partition p_2 . Instead, it is annotated with a receive synchronization $b?$, which keeps the partition within the current location until a corresponding signal is received. As soon as B has been executed within p_2 , a synchronization $b!$ is sent to indicate the finalization of B 's execution. In the same way, p_2 is assigned the transition A , but the action is removed because it is invoked within p_1 . Instead, a receive synchronization $c?$ is inserted, which suspends the invocation until the action C is executed. *currentLocations* is updated to l_1 and l_2 , since they were returned by the BFS algorithm.

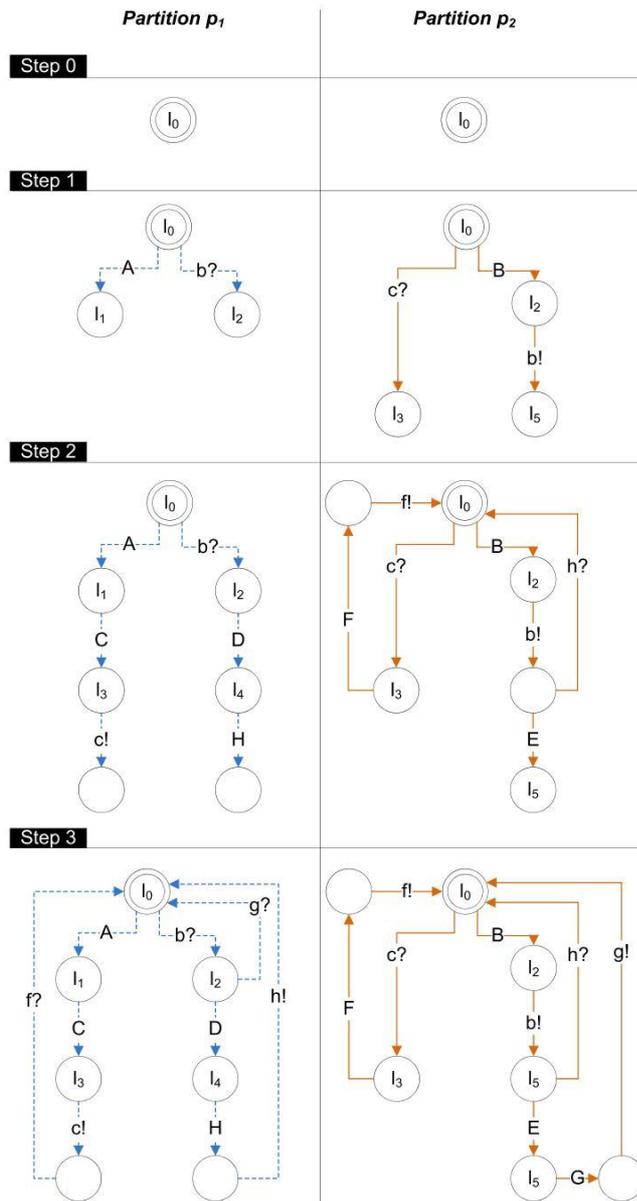


Figure 6.3: The two partitions of the example workflow shown in 6.2 after every step of the partitioning algorithm.

Steps 2 and 3 of the algorithm are performed similarly, as outlined in Table 6.1 and 6.2. At the end of step 3, $getNext()$ only returns l_0 , which has already been visited. Therefore, $currentLocations$ is empty in step 4, so that $getNext()$ is not invoked. The algorithm is thus terminated.

Step	Current locations	p_1 's next locations	p_2 's next locations
1	l_0	l_0	l_0
2	l_1, l_2	l_1, l_2	l_2, l_3
3	l_3, l_4, l_5	l_3, l_4	l_3, l_5
4	\emptyset	l_0	l_0

Table 6.1: Partitioning Algorithm: Content of the key structures.

Step	Parameters	Result	Comment
1	A, p_1	l_1	A is local to p_1
	B, p_1	l_2	l_2 has local outgoing transition
	A, p_2	l_3	l_3 has local outgoing transition, but l_1 has not
	B, p_2	l_2	B is local to p_2
2	C, p_1	l_3	C is local p_1
	D, p_1	l_4	D is local p_1
	E, p_1	l_0	l_0 has local outgoing transition, but l_5 has not
	D, p_2	l_0	l_0 has local outgoing transition, but l_4 has not
	E, p_2	l_5	E is local to p_2
3	F, p_1	l_0	l_0 has local outgoing transition
	H, p_1	l_0	H is local to p_1
	F, p_2	l_0	F is local to p_2
	G, p_2	l_0	G is local to p_2

Table 6.2: Partitioning Algorithm: Calling the getNext() method.

6.3.3 Creating Workflow Partitions

In the following, the workflow distribution algorithm is described in detail, together with its key subroutines *getNext()* and *copyTransition()*. Some steps are discussed on the basis of the workflow graph depicted in Figure 6.4.

6.3.3.1 The Core Algorithm

Algorithm 6.1 lists the partitioning algorithm in pseudo-code notation. The *main()* procedure builds the core of the partitioning algorithm and takes a workflow *workflow* and a set *transitionGroups* as input. The latter parameter contains groups of the same-colored transitions within the origin workflow graph. The procedure traverses *workflow* and distributes it into multiple partitions.

Four sets of data are defined. Here, *partitions* keeps the single partitions and *currentLocations* holds the locations that are entered within the next step of the algorithm; they have already been introduced in section 6.3.2.2.

Algorithm 6.1 Core Algorithm for the Distribution of Underlay Systems

```

1: procedure (workflow : Workflow, transitionGroups :  $\{x|x : \{y|y : Transition\}\}$ )
2:   partitions :=  $\emptyset$ 
3:   currentLocations :=  $\{workflow.initialLocation\}$ 
4:   visitedLocations :=  $\emptyset$ 
5:   newCurrentLocations :=  $\emptyset$ 
6:   for all transitionGroup  $\in$  transitionGroups do
7:     partition := newWorkflow()
8:     partition.initialLocation := workflow.initialLocation
9:     partition.nextLocations :=  $\{workflow.initialLocation\}$ 
10:    partition.transitionGroup := transitionGroup
11:    partitions := partitions  $\cup$   $\{partition\}$ 
12:  end for
13:  repeat
14:    for all currentLocation  $\in$  currentLocations do
15:      for all transition  $\in$  currentLocation.outgoings do
16:        newCurrentLocations := newCurrentLocations  $\cup$ 
           $\{transition.target\}$ 
17:        for all partition  $\in$  partitions do
18:          if currentLocation  $\in$  partition.nextLocations then
19:            nextLocation := GETNEXT(transition, partition)
20:            if nextLocation  $\neq$  null then
21:              COPYTRANSITION(partition, transition, nextLocation)
22:              partition.nextLocations :=
                partition.nextLocations  $\cup$  nextLocation
23:            end if
24:          end if
25:        end for
26:      end for
27:      for all partition  $\in$  partitions do
28:         $\triangleright$  remove current location from next locations
29:        partition.nextLocations := partition.nextLocations  $\setminus$ 
           $\{currentLocation\}$ 
30:      end for
31:    end for
32:    visitedLocations := visitedLocations  $\cup$  currentLocations
33:    newCurrentLocations := newCurrentLocations  $\setminus$  visitedLocations
34:    currentLocations := newCurrentLocations
35:  until currentLocations  $\neq$   $\emptyset$ 
36:  return partitions
37: end procedure

```

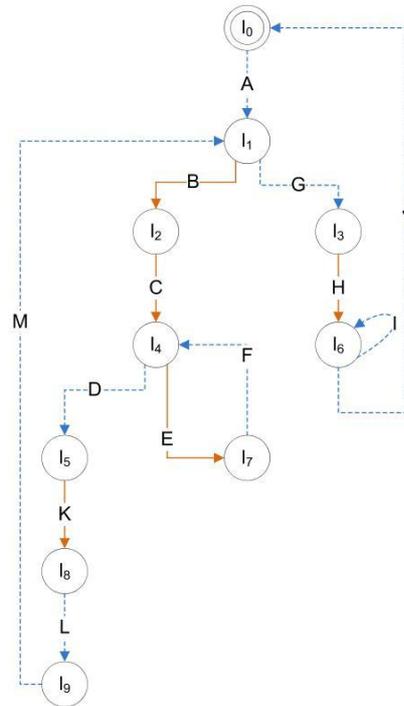


Figure 6.4: Example of a colored workflow graph.

A step of the algorithm is defined as an iteration over all members of this set. Its content is overwritten with locations found within the next BFS's step as soon as all of the current locations are visited. Initially, the set contains the initial location of the origin workflow. For instance, when the algorithm is applied to the workflow depicted in Figure 6.4, *currentLocations* would initially contain l_0 , l_1 in the first step, and l_2 and l_3 in the second.

The set *newCurrentLocations* has an auxiliary function, as it builds the basis for the content of *currentLocations* during the next step. It stores locations that can be reached from a current location within one hop. For each of these locations, an incoming transition exists, which has a *currentLocations*'s member as source. For instance, when *currentLocations* encompasses l_4 , *newCurrentLocations* consist of l_4 , l_5 , and l_7 at the end of the iteration.

At the beginning of the algorithm, the *partitions* set is initialized; a new partition *partition* is created for each given transitions group. It is initialized as a new, empty workflow and is put into the *partitions* set. Each partition is assigned its corresponding transition group and the respective *nextLocations* set is created. It contains locations, whose outgoing transitions are copied into *partition* within the next step.

The loop (line 13), which builds the core of the algorithm, repeats until the set *currentLocations* is empty. Within the loop, the outgoing transitions for all locations of *currentLocations* are explored; their target locations are added to *newCurrentLocations*.

Afterwards, the algorithm iterates over all partitions. When the *currentLocation* is an element of the partition's *nextLocations* set, all outgoing transitions of the current location are investigated. Their target location is derived by calling the *getNext()* procedure and is stored within *nextLocation*. The input of the *getNext()* procedure consists of the outgoing transition of *currentLocation* and the partition itself. After deriving the target location of the transition with regard to the given partition, the *copyTransition()* procedure is called to include the transition into the partition. It takes the partition, the transition itself, and *nextLocation* as input and creates a transition or a set of transitions that is capable of representing the origin transition's functionality within the given partition. Finally, *nextLocation* is added to the partition's *nextLocations* set. After all outgoing transitions of *currentLocation* have been copied, it is removed from the *nextLocations* sets of all partitions.

After the algorithm has iterated over all members of the *currentLocation* set, its locations are moved to the *visitedLocations* set. The *visitedLocations* are then removed from the *newCurrentLocations* set, so that it only contains locations that have not yet been visited. Finally, the content of *currentLocations* is overwritten with *newCurrentLocations*. If the set is empty, the algorithm finishes and the distributed partitions are returned. Otherwise the iteration continues.

6.3.3.2 The *getNext()* Procedure

The *getNext()* procedure discovers the new target location of a given transition within a selected partition. The target location depends on the coloring of the origin workflow graph and is retrieved by a combination of a BFS and DFS. The procedure is shown in Algorithm 6.2.

Initially, *visitedLocations* is initiated, so locations that have once been visited by the procedure are not visited again. Thereby, the termination of recursive procedure calls is ensured, given a final set of locations within a workflow graph.

The procedure returns the location *next*. It may be *null* if *transition* does not have to be copied. This is the case if the transition is remote and its source equals its target, i.e. if the transition is a remote loop (line 4). *next* is set to the current *transition*'s target if the target has either no or just a

Algorithm 6.2 Combined BFS and DFS for the retrieval of relevant locations for each step.

```

1: procedure GETNEXT(transition : Transition, partition : Workflow)
2:   next := null
3:   visitedLocations := partition.locations  $\cup$  {transition.source}
4:   if transition.source = transition.target and transition  $\notin$ 
   partition.transitionGroup then
5:     next := null
6:   else if transition.target  $\in$  visitedLocations then
7:     else
8:       targetOutgoings := transition.target.outgoings
9:       if targetOutgoings =  $\emptyset$  or transition  $\in$  partition.transitionGroup or
       #targetOutgoings = 1 and {targetOutgoings  $\cap$  partition.transitionGroup}  $\neq$ 
        $\emptyset$  then
10:        next = transition.target
11:      else if #targetOutgoings > 1 then
12:        if {targetOutgoings  $\cap$  partition.transitionGroup}  $\neq$   $\emptyset$  or
        isLastInParallelBlock(transition.target) then
13:          next := transition.target
14:        else
15:          locations :=  $\emptyset$ 
16:          for all outgoing  $\in$  outgoings do
17:            if !ISIDLE(outgoing.target) then
18:              locations := locations  $\cup$  {GETNEXT(outgoing, partition)}
19:            end if
20:          end for
21:          if #locations > 1 then
22:            next := transition.target
23:          else if #locations = 1 then
24:            next := location | location  $\in$  locations
25:          end if
26:        end if
27:      else
28:        next := GETNEXT(outgoing  $\in$  targetOutgoings, partition)
29:      end if
30:    end if
31:    if next  $\neq$  null and ISIDLE(transition.source) and next  $\neq$ 
    transition.target then
32:      next := null
33:    end if
34:    return next
35: end procedure

```

single local outgoing transition, or *transition* is local. Otherwise, if the target has a single outgoing transition, *getNext()* is called recursively on the target location.

When the target location has multiple outgoing transitions, wherefrom at least one is local, or the target is the last location in a parallel execution block, *next* is set to *transition*'s target. The handling of structures expressing the parallel execution of services is discussed later in section 6.3.4. Otherwise, *getNext()* is called concurrently on each outgoing transition in order to find *transition*'s new target. Finally *next* is returned.

6.3.3.3 The *copyTransition()* Procedure

The *copyTransition* procedure requires a partition *partition*, a transition *transition* and a location *target* as input parameters. It copies *transition* into the specified partition, while setting its target location to *target*. The procedure's functionality is listed as pseudo-code in Algorithm 6.3.

First, a new transition *copy* is created. Its source location and guard are set to the source location and guard of *transition*, respectively. The target is set to *target*. The source and target location of *copy* are then added to *partition*. Afterwards, whether the current transition has an action or not is checked, since the presence of an action may render the addition of synchronization messages necessary. In the event that the *transition* is labeled with a local action α , the action is copied to *copy*. If there is a remote outgoing transition from *transition*'s target location, a send synchronization is added to the partition in order to signal the finalization of α 's execution to other workflow partitions responsible for a service invocation that succeeds α .

Therefore, a new transition *newTransition* and a new location *newLocation* are added to *partition*. *newLocation*, which is the source location of *newTransition*, becomes the target of *copy*. Consequently, *newTransition*'s target location is set to *target*. The new send synchronization is added to the new transition. That way, a new transition containing a send synchronization is inserted between *copy* and its target location that indicates the finalization of α 's execution. Finally, *transition*'s origin synchronization is copied to *copy*.

If *transition* is not local, a receive synchronization is added to *copy*. This receive synchronization can be considered a counterpart to a send synchronization as described above, which is part of the partition for which *transition* is considered local. The receive synchronization ensures that the partial workflow graph blocks its execution until a send synchronization has indicated the

Algorithm 6.3 Copying transitions into workflow partitions.

```

1: procedure COPYTRANSITION(partition : Workflow, transition :
   Transition, target : Location)
2:   copy := newTransition()
3:   copy.source := transition.source
4:   copy.target := target
5:   copy.guard := transition.guard
6:   partition.locations := partition.locations  $\cup$  {transition.source, target}
7:   targetOutgoings := transition.target.outgoings
8:   if transition.action  $\neq$  null then
9:     if transition  $\in$  partition.transitionGroup then
10:      copy.action := transition.action
11:      if ISONETRANSITIONREMOTE(transitions, transitionGroup) then
12:        newLocation := newLocation()
13:        newTransition := newTransition()
14:        newTransition.source := newLocation
15:        newTransition.target := copy.target
16:        newTransition.synchronization :=
   newSendSynchronization(copy.action)
17:        copy.target := newLocation
18:        copy.synchronization := transition.synchronization
19:      else if !(#targetOutgoings = 1 and
   HASRECEIVESYNCFROMTRANSITIONGROUP(target, transitionGroup))
   then
20:        copy.synchronization := GETLASTREMOTEACTION(transition)
21:      end if
22:    end if
23:  end if
24: end procedure

```

finalization of the preceding service's execution. During execution, *workflow* blocks until a send synchronization primitive is issued for the *transition*'s action. Note that a receive synchronization is only added if the succeeding transition in *partition* does not already possess a receive synchronization. In that case, the addition of a synchronization to *copy* is dispensable, since the succeeding signal can be used to acknowledge the execution of both services and would only lead to a communication overhead.

Thus, *copyTransition()* copies a transition from the origin workflow graph to a given partition, where the action on the transition entails the addition of a send or receive synchronization. The former one is used to indicate the finalization of a local service's execution, while the latter one ensures that the workflow is blocked until the preceding service's execution has finished.

Transitions, which neither have an action nor a synchronization, do not have to be joined to any partition.

As the algorithm finishes, each partition of the workflow contains the following transitions.

1. Its local transitions,
2. its remote transitions, if
 - (a) they are followed by a local transition, or
 - (b) their source locations has multiple outgoing transitions.

They are deprived of their action as well as the original synchronizations, but can be supplied with a new send or receive synchronization, thus they are used to synchronize the control flow of the workflow's execution.

3. Transitions connected with the *IDLE* location, and
4. transitions, which were not present in the original workflow graph. They are added for synchronization reasons, but may also be employed to optimize some execution characteristics, e.g. to minimize the synchronization overhead.

6.3.3.4 Examples for Workflow Partitions

To illustrate the varying appearance of partitions resulting from the presented algorithm, two examples for distributed workflows are given. Figure 6.5 shows the algorithm's output for the workflow shown in Figure 6.4.

Here, both partitions look almost like copies of the originating workflow graph, which is mainly caused by the coloring of the workflow graph, where neighboring transitions have different colors in most cases. This leads to intensified synchronization overhead, since a synchronization message is exchanged whenever the control flow is passed between two partitions. The transitions and locations added for synchronization reasons enlarge the single partitions more.

Figure 6.6 shows the same workflow as in Figure 6.4, but with a different coloring.

The result of the distribution algorithm is illustrated in Figure 6.5.

The structure of the partitions is less complex, since the dependencies between the single services of different partitions are small. Thus, the synchronization overhead is minimal, since there are long sequences of transitions with the same color. Consequently, synchronization messages are only rarely required to synchronize the state of the single partitions.

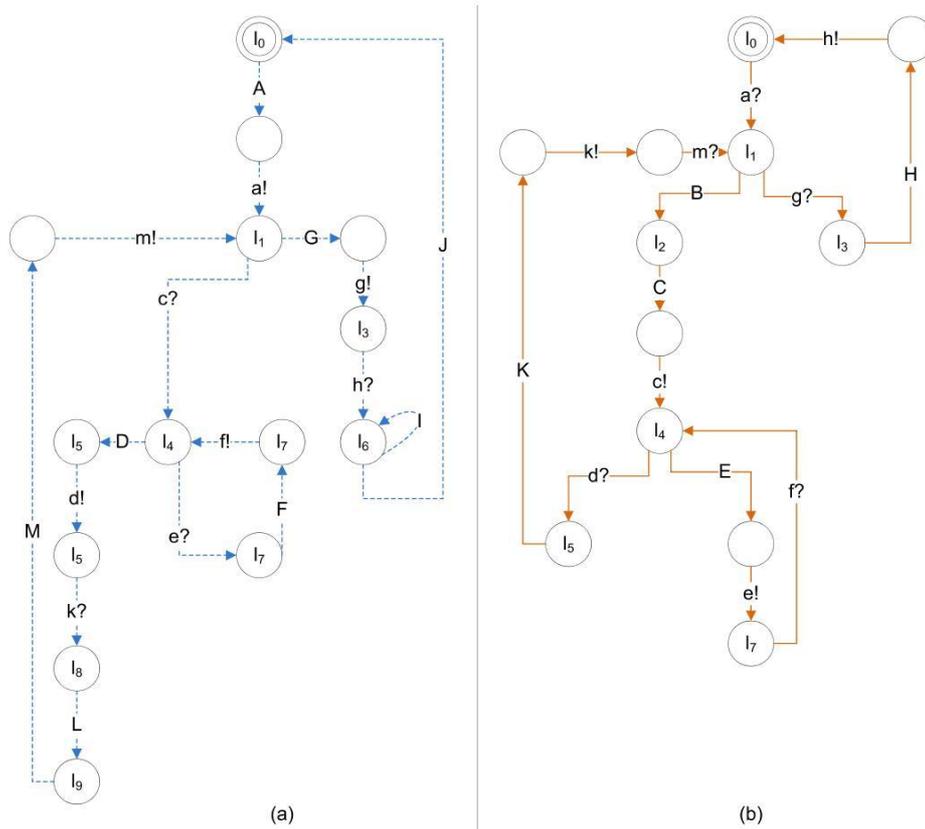


Figure 6.5: Two partitions derived from the workflow depicted in Figure 6.4.

Note that the coloring of transitions corresponds to the allocation of services to devices, i.e. the decision making with regard to the replacement of abstract services with concrete ones possibly residing on different devices. As demonstrated by the examples above, the allocation has a considerable influence on the workflow's partitions and overall synchronization. For instance, when a user prefers an application that can execute larger parts of its application logic in an offline mode, it is favorable to allocate sequences of services to the same device, as done for the workflow depicted in Figure 6.6, which leads to a clearer separation of the workflow structures and thus to fewer synchronizations.

6.3.4 Partitioning Parallel Execution Blocks

In section 4.3.4, a structure supporting the parallel invocation of services has been introduced. This special structure, based on the concept of an IDLE

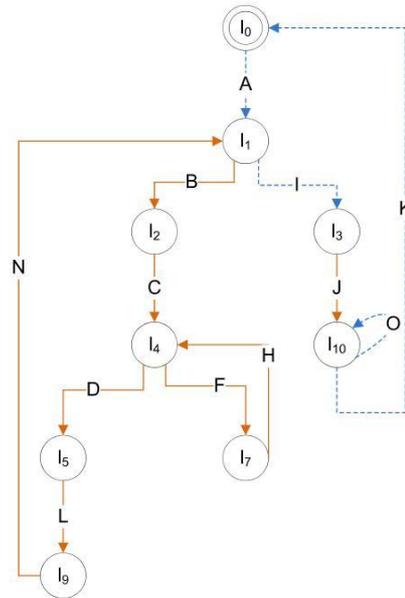


Figure 6.6: Exemplary workflow as depicted in Figure 6.4, but with different coloring.

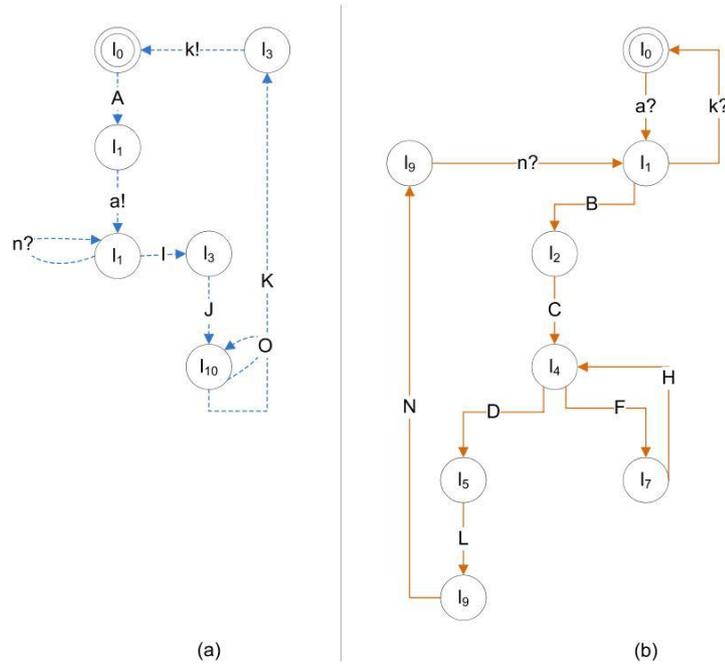


Figure 6.7: Two partitions derived from the workflow graph depicted in Figure 6.6.

The guard, which checks if B has terminated, is not explicitly given, but is attached to the new transition.

In order to remove this drawback, each new location added to the parallel execution block has to be connected with the *IDLE* location, as depicted in Figure 6.9. Here, actions in parallel branches can be executed in the time between B 's invocation and termination.

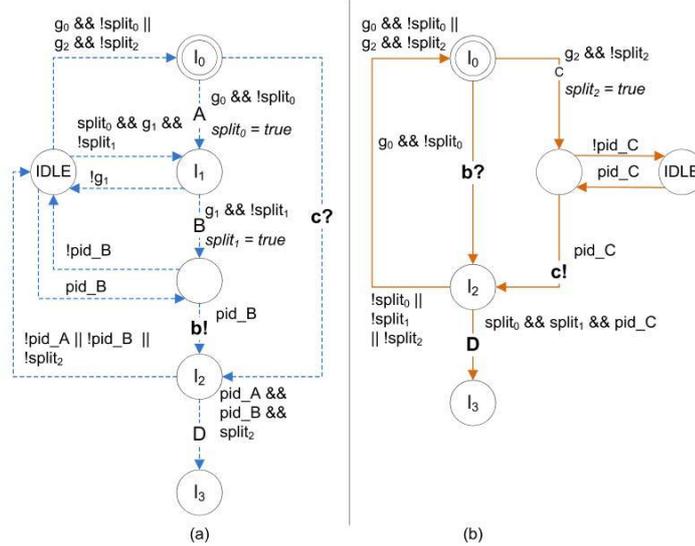


Figure 6.9: Modified partitions of the parallel execution block shown in Figure 6.8 (a). (a) Shows the modification of the partition depicted in Figure 6.8 (b). (b) Shows the modification of the partition depicted in Figure 6.8 (c).

6.3.5 Dealing with Nondeterministic Choices

The partitioning algorithm presented so far assumes that choices between transitions are deterministic, i.e. that the guards on outgoing transitions from a single location split the decision domain disjointly. In the event that the nondeterministic choice between services should be supported, the algorithm has to be adapted accordingly.

Assume that all outgoing transitions A , B , and C of l_0 depicted in Figure 6.10(a) are enabled. The partitions resulting from a distribution according to the algorithm 6.1 are presented in Figure 6.10(b)-(d), respectively. Here, each partition is assigned its local action and synchronization primitives in order to synchronize with the other partitions. There is a receive synchronization for every remote transition; complementary, a notification is sent after a local action's execution.

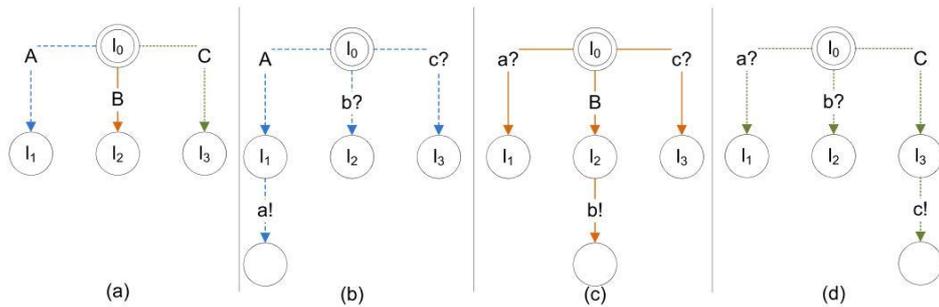


Figure 6.10: Resolving nondeterministic choices. (a) A nondeterministic choice between three transitions. (b)-(d) Three possible partitions.

The nondeterminism of the origin workflow is manifested within every resulting partition, where the choice of the single partitions is decoupled from the decision making within the remaining partitions. Therefore it is possible that each partition resolves the nondeterministic choice differently, which potentially leads to incoherent states of the partitions and no longer guarantees a behavior equivalent to the behavior of the origin workflow graph.

In general, the problem of nondeterministic choices can only be solved by inspecting and evaluating the guards of all outgoing transitions from the same location. Without additional knowledge of the guards and the domain space they are covering, it cannot be decided whether multiple outgoing transitions from a single location represent a deterministic or non-deterministic choice. When at least one of the outgoing transitions does not possess a guard, the choice can be considered as nondeterministic. If all transitions are labeled with guards, the guards' evaluation cannot be circumvented.

The evaluation of guard domains is considered to be out of scope of this thesis, since nondeterminism is usually undesirable in the application logic. Especially workflows, which are generated from the mashup language as introduced in appendix A or via the algorithm for the atomic creation of underlay systems as discussed in section 5.3, never expose such behaviour. In the following, the process of localizing nondeterminism is not considered; instead, it is assumed that the affected locations are known beforehand. For instance, they may be marked by the software developer when the underlay system is directly engineered. Note that this assumption only simplifies the detection of locations entailing nondeterministic choices; the treatment of such locations and their respective outgoing transitions remains the same as described below.

There are multiple ways to avoid an incoherent behavior of partitions during execution. For instance, partitions could compete for the right to

resolve the nondeterministic decision. Here, the competition would result in the assignment of the decision directly to one partition, so that the remaining partitions have to follow its decision. The competition process would require an additional workflow graph that serves as a central entity to coordinate the competition procedure.

The definition of a master partition constitutes a similar possibility. Here, one partition –the master partition– is responsible for the decision making and the communication of the decision to the remaining partitions. This alternative generates less communication and leads to smaller partition graphs, since no additional workflow graph is required and the competition phase is substituted by a plain assignment of the master role to one partition. The application of this approach to the nondeterministic workflow depicted in Figure 6.10(a) is illustrated in Figures 6.11(a)-(c), where the partition shown in Figure 6.11(a) is selected as the master. It can either execute the action *A* or let a remote action be executed. The decision is then communicated to other partitions. As a complement, the partitions shown in Figure 6.11(b) and Figure 6.11(c) forestall the execution of any action until a synchronization signal is received.

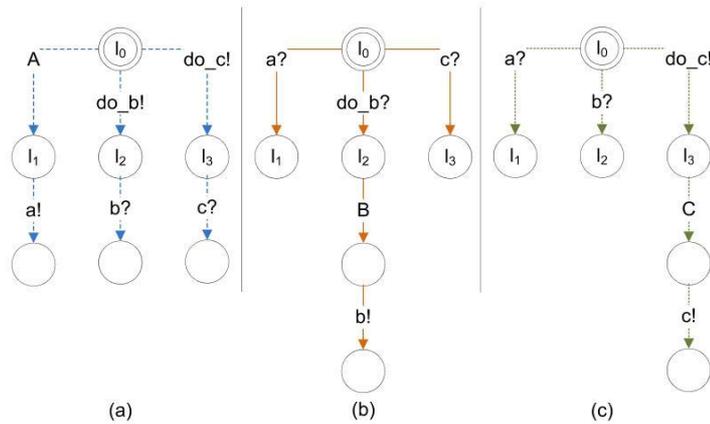


Figure 6.11: Resolving nondeterministic choices with a master automaton. (a) The master partition derived from the nondeterministic automaton shown in Figure 6.10(a). (b)-(c) The two slave partitions derived from the nondeterministic automaton shown in Figure 6.10(a).

6.3.6 Dataflow Distribution Algorithm

The distribution of the dataflow graph is directly derived from the partitioning of the workflow graph. For every service within a partition, the mapping

between the service's outputs and the inputs of other (possibly remote) services has to be known. The respective algorithm for the extraction of the necessary information from the origin dataflow graph is listed in pseudo-code notation in Algorithm 6.4.

The algorithm takes the dataflow *dataflow* of the origin not-distributed workflow and one of its partitions *workflow*, which was returned by the workflow distribution algorithm listed in Algorithm 6.1. As result of the algorithm's execution the part of *dataflow* relevant for services present in the given *workflow* is returned.

Algorithm 6.4 Distribute Dataflow

```

1: procedure DISTRIBUTEDF(dataflow : Dataflow, workflow : Workflow)
2:   newDataflow :=  $\emptyset$ 
3:   for all location  $\in$  dataflow.locations do
4:     if location.action  $\in$  workflow.actions then
5:       newDataflow.locations := newDataflow.locations  $\cup$  {location}
6:     end if
7:   end for
8:   return newDataflow
9: end procedure

```

The structure *newDataflow* holds the created sub-dataflow graph. The algorithm iterates over *dataflow*'s locations and copies those to *newDataflow* which refer to actions present within the given workflow, i.e. one of the partitions resulting from the workflow partitioning algorithm presented above. Finally, the new dataflow is returned.

The algorithm has to be invoked once for each partition of the original workflow graph in order to distribute the dataflow completely.

6.3.7 Evaluation of Invariance

Distributed underlay systems have to be functionally equivalent to the origin underlay system. Thus, the following aspects have to remain invariant to the distribution:

1. Invariants and guards defined within the workflow, including clocks,
2. the execution order of services as defined by the workflow flow,
3. communication between workflows defined before the distribution process, and
4. the dataflow.

Although the origin workflow's structure changes due to the distribution, the effects that are enforced by invariants and guards persist. An action

is never allocated to another transition so that the guard, which is associated with the action and consequently with the transition, remains at the right spot and must be evaluated before the action is executed. Despite the fact that a transition's target location may change, the source location holds potential invariants that must be met before entering the location and therefore the outgoing transition never changes. Since a timed automaton only defines upper bounds for a clock as invariants, they do not get violated, as the relative order between transitions and therefore actions is persistent.

However, since the underlay system's workflow graphs have a timed automaton structure, the synchronization of all distributed clocks must be ensured. In the distributed architecture that underlies the approach presented in this thesis, a time synchronization protocol such as the Network Time Protocol (NTP) [118, 91] is assumed for the management of the clocks' synchronization.

The execution order of services is constant, since the control flow in the distributed execution is controlled by synchronization primitives. When the origin workflow executes an action α_1 after an action α_0 and the services are allocated to different partitions, synchronization primitives are generated for both partitions that ensure the services' coherent execution order. A send synchronization transmitted via channel c is added to the transition that entails the invocation of α_0 or the succeeding transition if the previous transition already holds a synchronization. In the same way, the other workflow is extended by a receive synchronization that listens on the same channel c . It is appended to the transition preceding the transition labeled with α_1 . As a consequence α_1 cannot be executed until α_0 's execution has finished.

The algorithm for the distribution of the underlay system's dataflow graph discussed above simply extracts the part of the I/O mapping that is relevant for the services of every device. Thus, each partition is granted access to a sub-dataflow graph, which defines the passage of the local services' output data to the inputs of other services. Thereby, dependencies between services' inputs and outputs are resolved as if the dataflow and the services were executed in a centralized manner.

CHAPTER 7

SPECIFICATION

This chapter deals with the specification of components that enable the automatic creation and distribution of underlay systems among multiple devices. Section 7.1 deals with the embedding of single components into the extended mashup architecture discussed in section 3.4. While section 7.2 focuses on the single components and their respective communication interfaces, section 7.3 deals with the specification of a communication protocol that enables the negotiation of service bindings required to apply the partitioning algorithm introduced in chapter 6 and the succeeding coherent execution of the resulting sub-underlay systems.

7.1 ARCHITECTURAL EMBEDDING

For the specification of single components as well as the communication protocol, the roles introduced in the architectural extension of server-side mashups depicted in Figure 3.4 are overtaken. A *client* is considered as a user device that generates the request to start an application. A *proxy* acts as intermediate component that deals with the aggregation and transformation of data retrieved from one or more *servers*, i.e. 3rd party content providers.

In that scope, only the distribution between a client and a proxy is focused on, where the proxy encapsulates the additionally required components and thereby frees the client from necessary extensions. The same mechanisms can be applied to the distribution between multiple parties without any changes in the presented algorithms itself.

Figure 7.1 gives a general overview of the single activities that dynamically create, distribute, and execute Web applications.

Initially, the mashup is requested by a client via a URI (#1). The URI may contain information on the IOPE description of the requested service or point to a Web page that contains markup tags. Based on the given information, i.e. either the requested service's IOPE descriptions or the markup, an underlay system is created at the proxy-side. When the mashup's underlay system has been directly engineered by a software developer, this step is skipped (#2). The proxy checks the availability of services from 3rd party

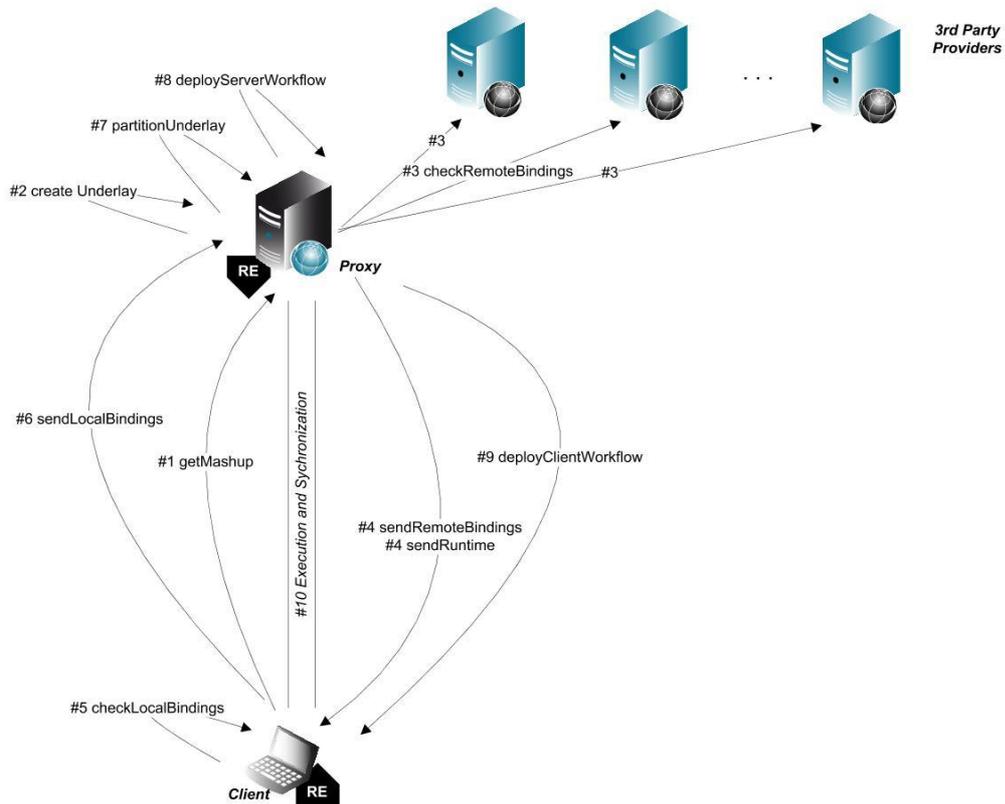


Figure 7.1: Distributing Web applications: General overview.

providers that may serve as implementations for abstract services of the underlay system (#3). Information on the discovered services is sent back to the client, together with a special runtime environment that allows the client to execute a part of the underlay system locally later (#4). The client checks out which services should be integrated from the client side (#5) and communicates its decision to the proxy (#6). Based on the available services on the proxy and the client side, the proxy allocates concrete services to the abstract services within the underlay system. Based on the allocation of the single services, the underlay system is split up into a client-side partition for the controlling and execution of client side services, and a partition that is executed on the proxy side controlling the integration of services stemming from remote 3rd party providers (#7). Finally, the sub-underlay systems are deployed at the proxy and client side and their execution is initiated (#8, #9). The distribution of the services between client and proxy as well as the coherent execution of the dynamically distributed underlay system is performed by the means of a novel communication protocol (#10).

The specification contains two central parts. Section 7.2 deals with the specification of the involved components and their respective communication interfaces. Here, most components are required exactly once, so that they can be deployed on the proxy of the extended mashup architecture introduced in section 3.4. Some components must be available multiple times, i.e. an instance of the respective components is required for each involved device. Section 7.3 concentrates on the communication among the single instances of those components by specifying a communication protocol that supports the allocation of concrete services stemming from different devices to an abstract underlay system. Moreover, it ensures the coherent execution of a partitioned underlay system distributed among the single devices.

7.2 COMPONENT SPECIFICATION

7.2.1 *Component Overview*

In total, eleven functional components are defined. These are mainly hosted on a proxy server; only a smaller subset of these components has to also be present on the client side. An overview of these components and their respective interaction interfaces is given in Figure 7.2.

A user requests a Web application via a URI, which either contains information on a requested service by means of its IOPE descriptions or points to a Web page possibly containing markup tags. The Web page is forwarded to the *request processor*, which then determines whether an underlay system has to be created from a IOPE based request or markup tags (#1). In the latter case, the syntax of the markup is validated by a respective *validator* (#2). The IOPE descriptions of the service request or the validated set of markup tags are then passed to the *underlay creation engine*, where a corresponding underlay system is generated based on the algorithms introduced in chapter 5 (#3). The descriptions of the single abstract services of the underlay system are passed to the *late binding engine* (#4); there is a late binding engine on both the client and the proxy side. Both engines communicate via the orchestration synchronization protocol (OSP) to negotiate which services will be allocated to which device (#5). Given the respective abstract services, the late binding engines look up concrete services from the *service repositories* (#6). In the event that there are multiple services matching an abstract service, the user preferences can contain information on the preferred service integration style as introduced in section 6.1 (#7.1). For example, when the user prefers the create an application with the highest possible offline capability, services from the client side are favored. Alternatively, additional information on the services' quality may be employed and

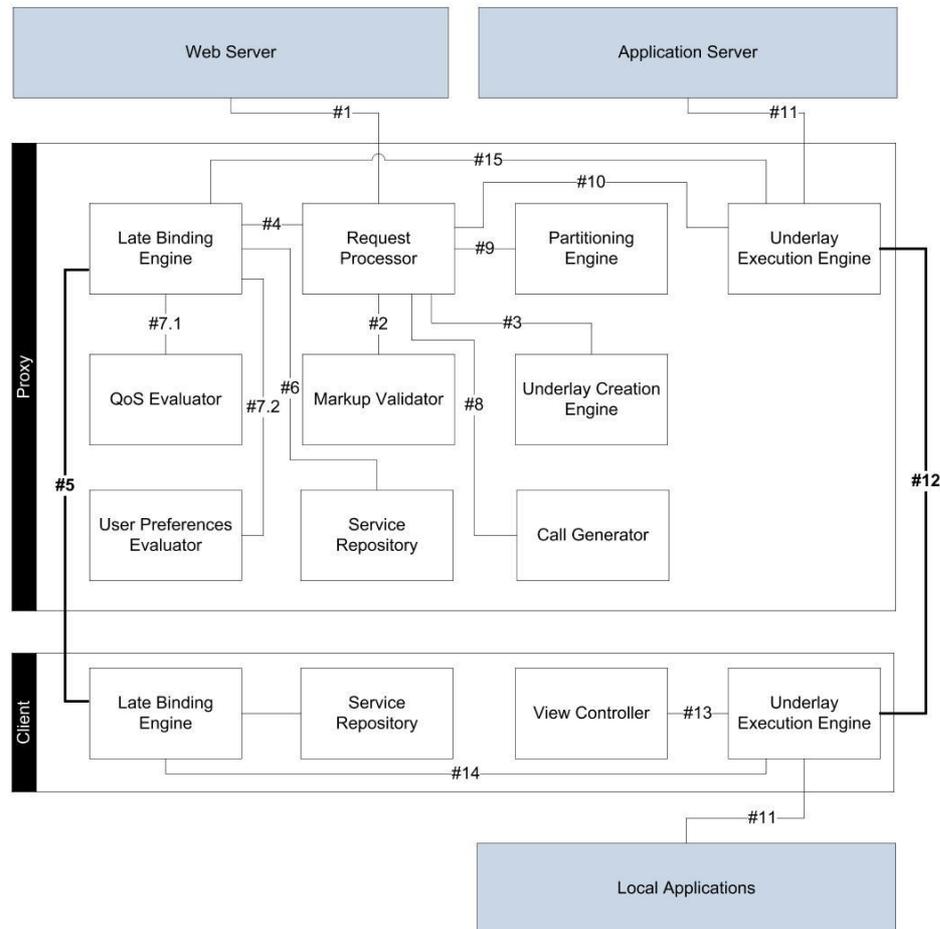


Figure 7.2: Component overview.

considered against the user's preferences, so that the most suitable service can be selected (#7.2). After every abstract service has been substituted by a concrete service either from the client or the proxy side, the *call generator* is accessed to automatically create service calls from the services' interface descriptions (#8). The resulting concrete underlay system is then passed to the *partitioning engine*, which can split up a given underlay system based on the allocation of services to devices as introduced in chapter 6 (#9). The single sub-underlay systems are finally deployed in the *underlay execution engines* hosted on the devices that provide at least one concrete service to the given underlay system (#10); the single services involved in the underlay system are invoked by the *underlay execution engines* when required (#11). The coherent execution and respective synchronization of the dynamically created sub-underlay systems is ensured through the *orchestration synchro-*

nization protocol (OSP) (#12). Every time the execution of a service leads to a modification of a resource that possesses a representation, the user's interface is adapted accordingly via a *view controller*, which holds a listener on the resource space of the client and can therefore respond to resource changes (#13). When a service fails during runtime, the service can be replaced dynamically via the local late binding engine (#14) or by communicating the failure via the OSP (#12) to call the proxy side late binding engine (#15), which then deals with the recovery of the failed service through the OSP's negotiation phase (#5).

In general, a solution that requires no modification of the client is the goal. Here, an echo server holds the descriptions of the locally available services present at the client side, i.e. the client's repository can be stored at a respective server. All remaining components can either be transferred as an executable Java Script library to the client during the initialization process—such as the execution engine and the late binding engine— or the respective functionalities are provided by the proxy.

In the following, the single components and their respective communication primitives are described. All primitives are understood as local service calls with regard to the setup depicted in Figure 7.2, i.e. as communication between components hosted on the same device. In order to render the specification independent from the concrete setup of the components, the style of the defined communication primitives is not differentiated between.

There is a special set of primitives coercively exchanged between multiple devices. With regard to the abstract depiction of the components presented in Figure 7.2, the allocation of services to the single devices is negotiated between late binding engines located on different devices (#5). Moreover, the coherent execution of a distributed underlay system within multiple execution engines also requires communication primitives sent over the network to communicate data and synchronize application states (#12). These primitives are specified as part of the Orchestration Synchronization Protocol (OSP) in section 7.3.

7.2.2 Data Elements and Parameters

The components introduced in the previous section interact by the means of well-defined communication primitives. Their possible data types and parameters are listed and explained in Table 7.1. Table 7.2 encompasses an overview of the error codes that may originate in a primitive's execution.

In the following sections, the single components required to support the distribution and respective execution of underlay systems are specified, fo-

Parameter name	Description
File	Data file that can contain markup tags (MashupXML) or abstract service requests (ServiceRequest)
MashupXML	XML document containing markup
Workflow	Serialized workflow graph
Dataflow	Serialized data flow graph
Underlay	Workflow graph, dataflow graph, and set of abstract services
ServiceDescription	Concrete service specified by its interface description, e.g. a WADL file
ServiceCall	Service call, e.g. a REST call
UserPreferences	Description of the user's preferences.
ServiceQuality	Information on the service's quality.
Mapping	One-to-one mapping of an abstract service to a concrete service
AbstractService	Abstract service description (IOPE)
ServiceRequest	Abstract service requested by the user (IOPE)
TimeLimit	Time limit given by the user for the automatic creation of an underlay system
MashupID	ID of a mashup as String
InputID	ID of an input parameter as String
ChannelID	ID of a channel as String
ChannelType	Defines the type of a virtual channel; the type may correspond to one of the inter-workflow communication patterns introduced in section 7.16.
ResID	ID of a resource as String
ResState	Object containing a resource's state
Object	Data object that is passed from one service's output to another service's input port.
Location	Location in workflow
Transition	Transition in workflow
Action	Action associated with an abstract service or markup tag; the action corresponds to a service invocation.

Table 7.1: Parameters and data types.

cusing on their interaction primitives that operate on the previously listed parameters and data types.

Error Code	Primitives	Description
100	all	ACK (No error). Returned in case the operation was successful.
201	checkXML()	XML parsing error.
202	checkMarkup()	Markup parsing error.
301	createUnderlay()	Underlay cannot be created.
401	mergeBindings()	At least one abstract service cannot be replaced.
501	createService()	Service cannot be created.
502	deleteService()	Service cannot be deleted.
503	updateService()	Service cannot be updated.
504	createMapping()	Mapping cannot be created.
505	deleteMapping()	Mapping cannot be deleted.
506	updateMapping()	Mapping cannot be updated.
507	getService()	No service available.
601	evalServiceQos()	QoS cannot be evaluated.
602	evalServicePref()	Preferences cannot be evaluated.
701	generateCall()	Call cannot be generated.
801	partitionUnderlay()	Partitioning error.
901	deploy()	Underlay system cannot be deployed. No correct underlay system.
902	deploy()	Underlay system cannot be deployed. Other underlay system running.
903	remove()	Underlay system cannot be removed.
904	replaceService()	Service cannot be replaced.
905	executeAction()	Action cannot be executed.
1001	replace()	Service cannot be recovered.
1101	update()	View error.

Table 7.2: Error codes.

7.2.3 Request Processor

The request processor is the central component for the processing of a user request by means of distributed applications. Initially, the component transforms the given user request into a corresponding underlay system and is henceforth responsible for its execution. Here, a request may either be a service request given as an abstract service, i.e. containing IOPE descriptions, or a Web page that contains markup tags. In the former case, the user has requested a service via a user interface that supports the specification of desired outputs and effects that should be created by the application, as well as the definition of available inputs and preconditions. These statements

are encapsulated within the URI and extracted at the proxy side. In turn, these statements are included in a special environment within the header of the Web page, where they can be extracted from the request processor when the page is initially loaded. In the latter case, the request processor can detect the presence of mashup markup by parsing the file and searching for `<mashup>` tags as introduced in appendix A.

In general, the request processor executes the following activities (in the given order):

1. Retrieval of an abstract service request from the Web server (#1).
2. Validation of the markup with regard to the XML and markup schema definitions in the event that markup was detected (#2).
3. Creation of an underlay system based on the identified markup tags or the abstract service request via the algorithm for automatic underlay creation as introduced in section 5.3 (#3).
4. Replacement of abstract services with concrete services via the late binding engine. (#4).
5. Requesting service calls for the selected concrete services via the call generator (#8).
6. Requesting the creation of multiple sub-underlays from the partitioning engine (#9).
7. Deployment of the sub-underlay systems within the workflow execution engines (#10).

Table 7.3 holds the primitive for initiating the parsing and interpretation process of a Web application; it is called directly when the Web page is loaded. In the event the document contains markup, the tags are extracted together with a dataflow description between the single services; the tags are stored as abstract services. If the document encompasses a service request, i.e. an abstract service description, it is returned within `ServiceRequest`. The single return types are `null` when the respective information cannot be found.

Primitive	Description	Caller
<code>parseMashup(File):</code> <code>list<AbstractService></code> , <code>Dataflow</code> , <code>ServiceRequest</code>	Parses the Website when loaded and extracts additional information describing a service request or markup tags.	Application Server

Table 7.3: OSP component: Request processor.

Note that the dataflow between the single markup tags can be directly derived from the nesting of the single tags as introduced in appendix A.

7.2.4 Markup Validator

The markup validator component validates the correctness of the extracted markup tags as introduced in annex A.3. Here, one primitive deals with the evaluation of the correctness with regard to the current XML standard, while the second primitive validates the correctness of the markup with regard to its own semantic restrictions; the respective primitives are listed in table 7.4. The algorithms for the validation of the markup language are introduced in appendix A.3.

Primitive	Description	Caller
checkXML(File): Boolean, ErrorCode	Validates whether the received file with regard to the XML standard.	Request processor (#2)
checkMarkup(File): Boolean, ErrorCode	Verifies the correctness of the included markup.	Request processor (#2)

Table 7.4: OSP component: Markup validator.

7.2.5 Underlay Creation Engine

The underlay creation engine supports the generation of an underlay system based on a given service request or list of abstract services and a related dataflow between them. Other methods for requesting underlay systems can be implemented by simply overriding the *createUnderlay()* function.

7.2.6 Late Binding Engine

The late binding engine provides the means to dynamically select and integrate a concrete service for an abstract service. Here, the late binding engine does not only support the dynamic selection of services based on additional knowledge, such as information on the quality of a service, but also the dynamic replacement of a service by an alternative one in the case that the current service fails or leaves the communication range of the user.

Thus, the late binding engine is supposed to support two major functionalities:

1. Lookup of appropriate services given an abstract description of the required functionality. Therefore, the late binding engine needs access to a service repository where the actual services are stored together with their mapping to concrete services.

Primitive	Description	Caller
createUnderlay (ServiceRequest, TimeLimit): Underlay, ErrorCode	Creates a underlay system based on the given service request by means of the algorithm presented in 5.3. TimeLimit denotes the upper time boundary that was given by the user for the creation of the underlay system. An error code specifies the failure of the creation process.	Request processor (#2)
createUnderlay (list<AbstactService>, Dataflow): Underlay, ErrorCode	Creates an underlay system for the given set of abstract services by means of the dataflow between die single services. An error code specifies the failure of the creation process.	Request processor (#2)

Table 7.5: OSP component: Underlay creation engine.

2. Selection of a service with the best fit in the case that multiple services are returned by the lookup procedure.

The primitives for looking up appropriate services, as well as for the selection between multiple concrete services for a given abstract service, are listed in Table 7.6. While the lookup primitives are used to access the service repository in order to match an abstract service against a (possible empty) set of concrete services, the selection primitives deal with the selection between the multiple concrete services that were found for the abstract one. Here, additional information on the services itself – information on the services’ quality for example –, or on the preferences of the user can support the decision making process. The user preferences can capture preferences of the user that might not require additional information on the services.

For instance, the user preferences may show that the user favours a high degree of offline capabilities in the resulting application, or that the user wants to reduce the communication required between the distributed parts of the underlay system. The allocation of services can then be made according to these preferences. In the former case, the allocation aims to include as many services as possible on the client side, while it aims to assign long chains of services to a single part of the underlay system in the latter case.

The selection primitives can also be invoked when the user preferences or the information of the single services are `null`; in that case, a service is selected randomly.

Primitive	Description	Caller
lookupService (list⟨AbstractService⟩): list⟨list⟨ServiceDescription⟩⟩	The late binding requests concrete services for multiple abstract services simultaneously and is thus returned a list of lists (henceforth referred to as matrix) of service descriptions.	Request processor (#4), Underlay Execution Engine (#15)
selectService (list⟨ServiceDescription⟩, list⟨ServiceQuality⟩): ServiceDescription	Selects a service from a set of services based on the given additional information on the single services.	local
selectService (list⟨ServiceDescription⟩, UserPreferences): ServiceDescription	Selects a service based on given user preferences.	local
mergeBindings (list⟨AbstractService⟩, list⟨list⟨ServiceDescription⟩⟩, UserPreferences): list⟨AbstractService⟩, list⟨ServiceDescription⟩, ErrorCode	Evaluates the bindings from multiple parties in order to derive exactly one concrete service for every abstract one.	local

Table 7.6: OSP component: Late binding engine.

7.2.7 Service Repository

The service repository holds the service descriptions of the available services as well as the mapping of their concrete functionality to abstract functions. It provides an interface for service providers to create, edit, and delete current service descriptions and mappings, and a second interface to accept requests for a service lookup from the late binding engine. The mapping may either be provided by the vendor of the proxy or the single service providers. In the latter case, the service provider has to be trusted to add only non-malicious mappings of services.

Assume that the proxy vendor defines an abstract service *center()* operating on a resource *map* and requires the attributes *lat* and *long* of a resource *location* as input. A service provider *X* now creates a service description for its service and includes it within the repository. It moreover maps its concrete function *xCenter()* to *center()* and the attributes *xLat* and *xLong* to *lat* and *long*, respectively. In the event that the late binding engine now

gets a request of the form $(center(), map)$, the repository returns the service description of the service included by the service provider X .

A set of primitives for the insertion, deletion and modification for concrete services as well as for mappings from abstract to concrete services is given in Table 7.7.

Primitive	Description	Caller
createService (ServiceDescription): Boolean, ErrorCode	Includes the description of a service within the service repository. The proxy vendor may also provide descriptions for 3rd party services.	Service provider, Proxy vendor
updateService (ServiceDescription): Boolean, ErrorCode	Updates the description of a service within the service repository. Only the service provider who has created the description can update it later on.	Service provider
deleteService (ServiceDescription, MappingDescription): Boolean, ErrorCode	Deletes the description of a service within the service repository. Only the service provider who has created the description or the proxy vendor can delete such a description.	Service Provider, Proxy Vendor
createMapping (ServiceDescription, MappingDescription): Boolean, ErrorCode	Includes a mapping of an already existing service description to an abstract service.	Service provider, Proxy vendor
updateMapping (ServiceDescription, MappingDescription): Boolean, ErrorCode	Updates a mapping of an already existing service description to an abstract service.	Service provider, Proxy vendor
deleteMapping (ServiceDescription, MappingDescription): Boolean, ErrorCode	Deletes a mapping of an already existing service description to an abstract service.	Service provider, Proxy vendor
getService (AbstractService): list<ServiceDescription>, ErrorCode	Matches the abstract service to the available mappings. The identified services are returned by their service description.	Late binding engine (#4)

Table 7.7: OSP component: Service repository.

7.2.8 QoS Evaluator and User Preferences Evaluator

The Quality of Service (QoS) evaluator measures the resource consumption of services during their execution, i.e. their memory consumption or response time. When multiple service descriptions have been returned from the service repository, the late binding engine may request QoS specific properties of the single services in order to make a selection decision. Here, the QoS evaluator may return complex QoS information (which has to be processed and evaluated by the late binding engine) or a simple value between 0 and 1. The latter approach enables an easy merging of the feedback from the QoS evaluator and the user preferences evaluator. A potential model for the evaluation of a service's quality with respect to given user preferences has been published in [146], but is not discussed in greater depth in this thesis. The primitives for deriving a service's quality are given in Table 7.8, while Table 7.9 lists primitives to match a QoS aspects of services to user preferences.

Primitive	Description	Caller
evalServiceQoS (list⟨ServiceDescription⟩): list⟨ServiceQuality⟩, ErrorCode	Looks up the QoS parameters of a set of services and returns them either as a list or a 2x2 matrix (in case multiple QoS parameters are available).	Late binding engine(#7.1)

Table 7.8: OSP component: QoS Evaluator.

Primitive	Description	Caller
evalServicePref (list⟨ServiceDescription⟩, UserPreferences): list⟨ServiceQuality⟩, ErrorCode	Evaluates the quality of a set of services with respect to the current context and preferences of the user. The values are returned as a list.	Late binding engine(#7.2)

Table 7.9: OSP component: User preferences evaluator.

7.2.9 Call Generator

The call generator component automatically creates a service call from a given service description that can be used to invoke the respective service.

This call is included within the workflow graph and invoked when the respective transition is passed. Table 7.10 specifies the primitives for the generation of service calls for one or more services.

Primitive	Description	Caller
createCall (list⟨ServiceDescription⟩): list⟨ServiceCall⟩, ErrorCode	Automatically creates and returns a set of service calls for the given set of service descriptions.	Request processor (#8)

Table 7.10: OSP component: Call generator.

Service calls for a given service description can be cached by underlay execution engines, which increases the chances of a local service recovery when a service fails during service execution. Section 7.3.5 addresses this issue in greater detail.

7.2.10 Partitioning Engine

The partitioning engine receives an underlay system from the request processor, where every abstract service has been replaced by a concrete service. This allocation also contains information on the location of the single services, so that the underlay system can be split up into multiple sub-underlay-systems as introduced in section 6.3; the according primitive is listed in Table 7.11.

Primitive	Description	Caller
partitionUnderlay (list⟨list⟨ServiceCall⟩⟩, Dataflow): list⟨Workflow⟩, ErrorCode	Automatically creates and returns a list of underlay systems for the given sets of services, taking into account the dataflow between all services within the lists.	Request processor (#9)

Table 7.11: OSP component: Partitioning engine.

7.2.11 Underlay Execution Engine

The underlay execution engine executes the workflow of services, ensures their correct data passage and handles the state management of the Web application. When the application is distributed between the client and the

proxy side, two workflows are executed in parallel while they ensure the correctness of the execution by means of the orchestration synchronization protocol introduced in section 7.3.

The primitives of the underlay execution engine listed in Table 7.12 can be divided into three groups. First, there are two basic primitives for the deployment and removal of an underlay system that can be called by the request processor to initiate or abort the execution of a concrete underlay system, respectively. Second, services can be replaced during runtime. Here, primitives are available to exchange a concrete service for another one in the case that both services are hosted on the same device, and to delete a service from an underlay system. The latter primitive is called if a failed service is replaced by a service from another device. In that case, the origin service has to be deleted while the substituting service is inserted into the underlay system hosted on the respective remote device. The primitive for communicating the insertion of a service into an underlay system is discussed in section 7.3. Third, there is a primitive that initiates the execution of a service. The output data generated by the service is passed to other services' inputs by means of the dataflow graph. In the case that the service's execution led to a modification of a resource, the resource state is updated accordingly. This modification is communicated to other sub-underlay systems as soon as a remote service requires access to the resource as described in section 6.3. The mapping of the resource's modification to an update of the resource's graphical presentation is realized through the view controller that is notified via an event as soon as a resource state is changed.

7.2.12 View Controller

When an underlay system is deployed within a client side underlay execution engine, a resource listener is instantiated that listens on the client's resource state. Every time a resource modification is detected, the *update()* primitive is invoked that passes the new state of the resource to the view controller, which adapts the presentation of the view accordingly; the primitive is listed in Table 7.13.

Primitive	Description	Caller
deploy(Underlay): Boolean, ErrorCode	Deploys an underlay system.	Request processor (#10)
remove(Underlay): Boolean, ErrorCode	The underlay system is removed from the engine such that another system can be deployed.	Request processor (#10)
replaceService (ServiceCall, ServiceCall)	Replaces a service within an underlay system, e.g. since the origin service failed and can be replaced by another service hosted on the same device.	Request processor (#10)
deleteService (ServiceCall)	Deletes a service from an underlay system, e.g. since the origin service has failed and was replaced by service from another device.	Request processor (#10)
executeAction(Action)	Initiates a service's execution.	local

Table 7.12: OSP component: Underlay execution engine.

Primitive	Description	Caller
update(list(ResState)): Boolean, ErrorCode	Updates the state of a resource.	Underlay Execution Engine (#13)

Table 7.13: OSP component: View controller.

7.3 THE ORCHESTRATION SYNCHRONIZATION PROTOCOL (OSP)

7.3.1 General Overview

The Orchestration Synchronization Protocol (OSP) enables the distribution and coordinated execution of underlay systems. In the following, it is assumed that only the underlay execution engine, the late binding engine, and the view controller are present at the client side. This assumption is in-line with the extended mashup architecture introduced in section 3.4. Here, the late binding engine constitutes the service integration engine, while the runtime environment consists of the underlay execution engine and a respective view controller. In the interest of improving readability, all client sided components are referred to as *client runtime* within the remainder of this section.

Figure 7.3 illustrates the embedding of the OSP into the extended server-side mashup architecture introduced in section 3.4.2.

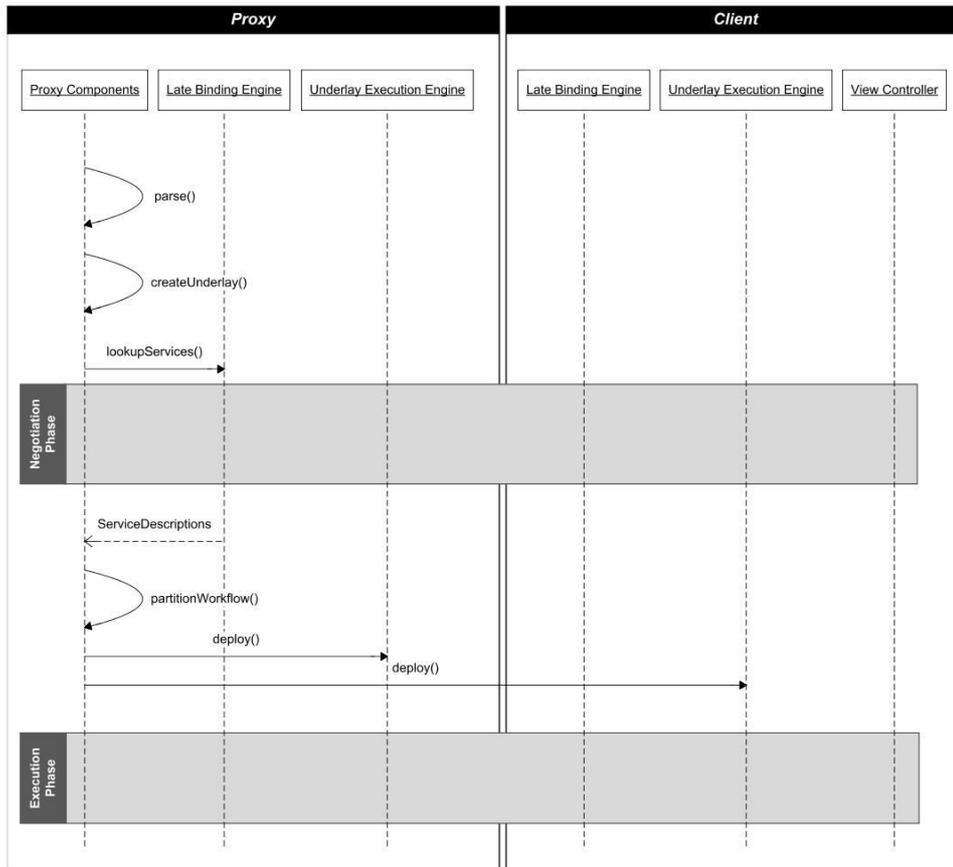


Figure 7.3: Embedding of the orchestration synchronization protocol (OSP).

Initially, the Web site or application requested by a user (identified via a URI) is inspected via the `parse()` primitive in order to check whether it contains a service request or markup tags. If this is the case, the respective information is extracted and transformed into an underlay system by the underlay creation engine. If markup tags were extracted they are validated beforehand. As a response to the user's request, the application's presentation and a client-side runtime environment is sent back to the user¹.

When applying the introduced underlay system to the domain of Web applications, the response contains an HTML page that is cleared from the additional service request or markup tags. The transmitted runtime is written in pure JavaScript, so that it can be executed by every common Web browser without any modifications.

¹The initial request of the user its related response has been dispensed with in favor of improving clarity in Figure 7.3.

By invoking the *lookupService()* primitive, the request processor component invokes the involvement of the OSP to handle the distribution and execution of the underlay system. In general, the OSP can be divided into two phases, namely the *negotiation* and the *execution phase*. The first phase starts after the client receives its runtime environment and is then followed by the second one. During the negotiation phase, the concrete services required within the scope of the underlay system are dynamically allocated to either the client or the proxy side. Therefore, the late binding engines on the proxy and the client side look up the services that they are potentially able to provide, while a selection of those services is performed based on additional knowledge or user preferences. The OSP's negotiation phase returns a set of concrete services that replace the abstract services within the previously created underlay system. This allocation of concrete services to abstract services is equal to the coloring of transitions of the underlay system's workflow graph, where all concrete services hosted on the same device are assigned the same color. The concrete underlay system is then passed to the partitioning engine, which splits up the underlay system into multiple sub-underlay systems as discussed in section 6.3. The single partitions, i.e. sub-underlay systems, that are derived from the partitioning engine, are deployed to the respective underlay execution engines at the client and proxy side. The deployment initiates the execution of the underlay systems, in which the execution phase handles the distributed execution of the single underlay systems by synchronizing their resource access and managing the passage of input and output data. Here, a resource that has been created by means of an execution of a service within the workflow graph is dynamically integrated into the presentation of the Mashup at the client side.

The following three sections deal with more detailed descriptions of the negotiation and execution phase, respectively, as well as their related communication primitives.

7.3.2 Negotiation Phase

The negotiation phase realizes the allocation of concrete services residing on multiple devices to abstract services of the given underlay system. This allocation serves as input for the underlay partitioning algorithm introduced in section 6.3. The general action flow of the OSP's negotiation phase is illustrated in Figure 7.4.

The negotiation phase of the OSP is initiated when

1. the user requests a document containing a service request or markup tags, or

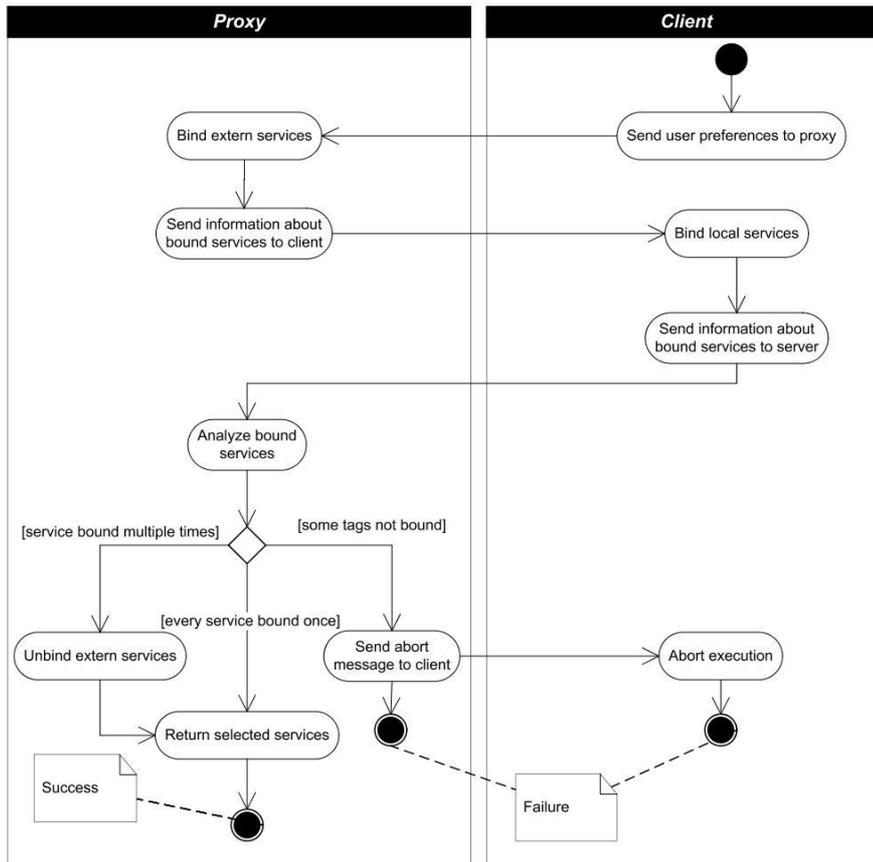


Figure 7.4: OSP Negotiation phase - Activity diagram.

- markup is dynamically generated on the client side, e.g. as a result of client-side script execution.

In both cases, the proxy selects appropriate concrete services for the abstract services that can create the requested resources. The binding decision may be improved by additional knowledge, e.g. in the case that user preferences are available. The following alternatives for preferences exist:

- **Private preferences:** Preferences are only available on the client and can be used during the binding of client side services.
- **Public dynamic preferences:** Preferences can be sent to the server during the negotiation phase.
- **Public static preferences:** Preferences are available on the server and are uploaded before the negotiation phase.

For the following diagrams (including Figure 7.4), the second alternative is chosen, thus, the user preferences are explicitly transmitted from the client to the proxy.

The proxy first requests the user's preferences and inspects them. Here, the preferences can contain general preferences with regard to the allocation style. For instance, the user may prefer to create an application with the largest possible offline capability, which leads to the favored integration of client side services as discussed in sections 6.3 and 7.2.6. In addition, the preferences may define a certain type of service quality preferred by the user, such as services with a short response time. When the preferences contain such additional information, the late binding engine can access the QoS evaluator and user preferences evaluator as introduced in section 7.2 to select between multiple possible concrete services for the same abstract service.

The proxy sends a message to the client, which contains information on the services that can be bound at the proxy side. The client then inspects the proxy's propositions for the replacement of the abstract services and itself looks up concrete services that can match the given abstract services. The proxy receives the client's selection of services and considers it against its own selection.

In general, three constellations can occur. First, all services may be allocated disjunctively to the client and the proxy; in this case, the binding process is completed. Second, there may be a service that has been allocated to neither the client nor the proxy. Here, the execution of the underlay system is aborted since the required functionality cannot be provided by means of the currently available services. Third, all services might be allocated to the proxy or the client, where some services have been allocated to both of them. In this case, the preferences of the user are employed to support the decision making process. For instance, when the user prefers an application with a high degree of offline capabilities, client-side services are favored and the respective proxy bindings are deallocated. The negotiation phase is concluded by returning either the final set of concrete services that match all abstract services of the given underlay system or an error code, which indicates that at least one service can neither be provided by the proxy, nor by the client.

The negotiation phase is finalized by returning the set of selected concrete services for a given underlay system. Given the disjoint allocation of services, the request processor component initiates the partitioning of the underlay system into a client side and a proxy side underlay system as introduced within section 6.3, such that both underlay systems can be executed in parallel while ensuring their coherent execution by means of synchronization

primitives. The deployment of the single underlay system's partition denotes the initialization of the OSP's execution phase.

7.3.3 Execution Phase

The OSP's execution phase deals with the execution of distributed underlay systems, where additional synchronization messages stemming from the partitioning algorithm introduced in section 6.3 ensure that the distributed underlay system exposes an equivalent behavior as the undistributed one.

The execution phase encompasses the message exchange between multiple underlay execution engines deployed on the involved devices. Figure 7.5 illustrates the general activities within one underlay execution engine.

At every point in time, the execution engine is within one of the following states:

RUNNING: while passing a transition and thus invoking a service,

SUSPENDED: when a failed service is recovered, or

IDLE: when there is no enabled transition.

The *RUNNING* state is considered as the execution engine's default state and thus not explicitly depicted in Figure 7.5. If the execution engine is neither in state *SUSPENDED* nor *IDLE*, it is in the *RUNNING* state until the execution terminates.

The execution engine traverses the underlay system's workflow beginning with the initial location. The actually considered location is called *current*, thus, the initial location is set as *current* when initiating the execution of an underlay system. When a new location is set as *current*, the guards of all outgoing transitions from *current* are evaluated; exactly one enabled transition is selected and the respective service is invoked. The invocation for a service starts a *thread* that waits for the finalization of a service's execution. This concept has already been introduced in section 4.3.4 and supports the parallel execution of services through asynchronous service invocations. More details on the respective realization are given in section 8.2.2.1. After the service's invocation, the transition's target location is set to *current*. The execution continues until a location is entered that does not have any outgoing transitions; these kind of locations are referred to as end locations and denote the successful finalization of the underlay system's execution.

Each time *current* is set to a new location, whether the engine is interrupted or suspended is checked. In the former case, all threads of the

execution, i.e. all current service executions, are interrupted and the workflow's execution fails. In the latter case, the *SUSPENDED* state is entered, denoting that the engine will be blocked until the execution is resumed. If the engine is neither suspended nor interrupted, the current location's outgoing transitions are inspected.

If *current* has no outgoing transition, the engine waits for all running threads to complete before terminating successfully. In contrast, if there are outgoing transitions, but none are enabled, i.e. all guards are evaluated as *false*, the execution engine enters the *IDLE* state. The execution engine waits in the *IDLE* state until an event occurs, which may change the values returned by the guards of *current*'s outgoing transitions. Otherwise, if at least one guard returns *true*, the respective enabled transition is selected. If there are multiple enabled transitions, one is selected randomly.

When a transition has been selected, it is checked whether the transition is labeled with a *receive* or *send* synchronization.

If the selected transition has a synchronization of the receive type, it is listened on the respective channel in order to receive data from a sending service. If no data can be received immediately, the synchronization is blocked while listening on the channel until the response is received or a time-out occurs. Multiple receive synchronizations can be active concurrently. If there are multiple enabled outgoing transitions with a receive synchronization, they start listening simultaneously until one of them returns data. The remaining transitions stop listening on their respective channels and the action of the transition whose receive synchronization was matched by an incoming signal is executed. When there is a send synchronization attached to the transition, the synchronization primitive is not executed until the action finished, while the setting of the new current location takes place after the synchronization signal is sent.

In order to remain responsive to service failures, a timer is started when a service is invoked. If the action does not finish until the timer expires, its execution is interrupted, the whole execution is suspended and a replacement for the unresponsive service is looked up. Section 7.3.5 discusses this procedure in greater detail. If the recovery process is successful, the action containing the new found service is restated. Otherwise the workflow's execution ends with an error message.

If the action finishes successfully, i.e. before the according timeout occurs, it is marked as executed (via its process ID as introduced in section 4.3.4 and the delivered outputs are sent to services awaiting them as defined in the underlay system's dataflow graph. If the transition has a send synchronization, the message is sent. After that the workflow's execution continues with transition's target as *current* location.

In addition to the mechanism described above that triggers the replacement of a potentially failed service, two more situations are equipped with a timeout mechanism. A recovery request can be considered as a service invocation itself, i.e. the invocation of a service that handles the replacement of another service. Therefore, the recovery process can also be stopped by a timeout mechanism. In addition, the listening on one or more channels to receive data can lead to a starvation of the overall process. Therefore, signals are only listened for a predefined amount of time, until a timeout occurs and the workflow's execution is interrupted.

7.3.4 *Communication and Synchronization*

In this section, the communication primitives of the OSP are introduced.

There are three general classes of messages within the scope of the OSP. Negotiation messages are applied during the negotiation phase in order to derive the allocation of concrete services to the abstract services of a given underlay system. These messages are exchanged between the single late binding engines deployed on the respective devices and listed in Table 7.14. Execution and synchronization messages are exchanged between the single underlay execution engines and ensure the coherent execution of distributed underlay systems with regard to resource access and execution order. The related primitives are listed in Table 7.15 and 7.16, respectively.

The negotiation phase encompasses two primitives; their embedding into the general sequence of action is depicted in Figure 7.6. The *getPreferences()* primitive is called by the late binding engine located on the proxy and requests the user's preferences, which contain information on the preferred allocation style of service as described in section 7.2. The returned preferences are used at the proxy side to look up services suited for the given preferences. For instance, when the user prefers services with a short response time, the proxy side late binding engine can select an appropriate service in the event that it found multiple concrete services suited for the replacement of an abstract service.

The *bindServices()* primitive is first invoked to transmit the proxy's selection of services to the client, which in turn invokes the same primitive to notify the proxy of the client's binding decision. The primitive carries a matrix as parameter, where the first column contains a list of abstract services α_i . The second column holds a list of concrete services for each service α_i , i.e. a list of m_i concrete services $d_i^0, \dots, d_i^{m_i}$ that can replace α_i . When a service cannot or should not be replaced, the list of concrete services is **null**. The parameter transmitted to the client thus defines which services can be replaced by the proxy side and which abstract services still need a replacement.

The proxy only transmits those abstract services to the client, which operate on resources within the client's scope as defined in section 6.2.

The client-side late binding engine looks up concrete services for the abstract ones, where also abstract services that are already affiliates with one or more concrete services provided by the proxy can be matched. The list of service descriptions sent back to the proxy as a reply to the *bindService()* primitive is then merged with the list created by the proxy itself. When the merging process is successful, i.e. there is at least one concrete service for every abstract service as described within section 7.3.2, the negotiation phase

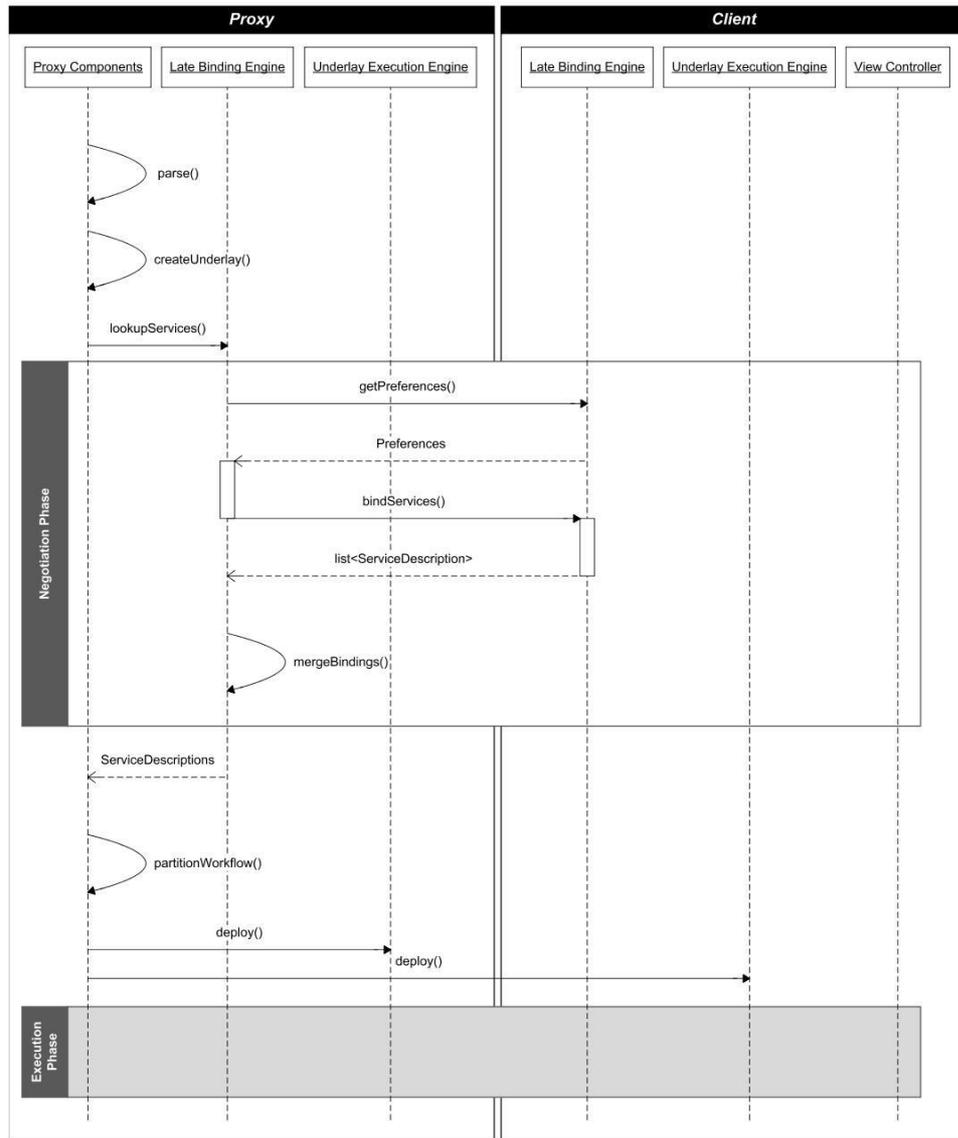


Figure 7.6: Embedding of the OSP's negotiation phase.

finishes by returning the derived list of service descriptions; otherwise, the process is aborted.

Primitive	Description	Caller
getPreferences(): UserPreferences	Gets user preferences.	Proxy
bindServices (list⟨AbstractService⟩, list⟨list⟨ServiceDescription⟩⟩): list⟨list⟨ServiceDescription⟩⟩	Communicates an initial allocation of concrete services to abstract services and requests possible bindings for the abstract services.	Proxy, Client

Table 7.14: OSP: Negotiation messages.

The execution phase handles the communication between underlay execution engines deployed on multiple devices; three general messages may be sent. First, a message can contain input data transferred between multiple underlay systems running in parallel. The transmission of this data is directed by the dataflow graph, specifying which output data is used as input for other services. Second, a message is available to insert a new service into the workflow; this may become necessary if a failed service from a device *A* is replaced by a service hosted on a device *B*. Third, an *abort()* primitive is available to notify the aborting of a workflow's execution. All communication primitives of the OSP's execution phase are listed in Table 7.15.

A fourth primitive *recoverService()* can only be invoked at the client-side underlay execution engine. When a service fails at the client side and cannot be replaced locally, this primitive is called to notify the proxy side execution engine that the OSP's negotiation phase has to be re-initiated to look up an appropriate replacement. Section 7.3.5 discusses this issue in greater detail.

The coherent execution of multiple underlay systems is ensured by a set of synchronization messages, which are divided into two groups. First, there are primitives to get the current state of a shared resource as well as to lock and unlock the access to a resource. Second, a synchronization message can be sent and received through a virtual channel with a specific ID. Receivers listening on a channel with the same ID may block until a message is received. Here, the type of the channel is given as a parameter, which specifies which kind of communication is desired; the different supported channel types have been introduced as part of the underlay system in section 4.4. All synchronization messages are listed in Table 7.16; they can be invoked by all underlay execution engines independent of their location, i.e. whether they are proxy or client side execution engines.

An example flow of messages during the OSP's execution phase is illustrated in Figure 7.7.

Primitive	Description
updateInput (list⟨InputID, Object⟩): Boolean, ErrorCode	Updates input parameters in a workflow. This is needed to resolve data dependencies between services that are part of different workflows.
insertService (Location, list⟨Transition⟩)	Inserts service into existing a workflow. A new transition T and a new location L is appended. T goes out of existing location given as first parameter and goes into L . Existing transitions given as second parameter go out of L . This primitive can be used together with <i>replaceService()</i> to move service between workflows e.g. if in the recovery case the replacement was found on another peer.
abort (MashupID)	Aborts mashup execution.
recoverService (AbstractService): ServiceCall	Forwards a replacement request to the proxy side late binding engine, such that the OSP' negotiation phase is started again via the <i>lookupService()</i> primitive.

Table 7.15: OSP: Execution messages.

Primitive	Description
getState(list⟨ResID⟩): list⟨ResID, ResState⟩	Gets the resources' current states.
lockState(list⟨ResID⟩): list⟨ResID, ResState⟩	Gets and locks the resources' current states.
unlockState(list⟨ResID⟩): Boolean, ErrorCode	Unlocks the resources' states.
sendSynchronization (ChannelID, ChannelType)	Sends synchronization signal on channel with ChannelID. Receivers behave according the ChannelType as defined in section 4.4.
receiveSynchronization (ChannelID, ChannelType)	Starts listening for a synchronization signal on a channel with the given ChannelID. Receivers behave according the ChannelType as defined in section 4.4. Listening on multiple different channels at the same time is possible; the first incoming signal releases the listening to other channels.

Table 7.16: OSP: Synchronization messages.

Before executing a service, the state of the corresponding resource is retrieved. States can either be accessed in a read-only or read-write mode. In the first case, the *getState()* primitive is used to access a resource's state; this primitive is not blocking so that the process can proceed with its processing,

but modifications on the returned object are not allowed. In the latter case, the state object must be locked by the *lockState()* primitive, which blocks the caller until the object is write-accessible. After issuing changes on the state object it is unlocked by the *unlockState()* primitive. Finally, dataflow information, i.e. the transmitting of content between services, is communicated via the *updateInput()* primitive.

Each transition in the workflow graph can have an additional synchronization action, which either listens for signals on a specified channel by means of the *receiveSynchronization()* primitive or initiates the transmission of a signal via the *sendSynchronization()* primitive. Within the sequence diagram shown in figure 7.7, the client blocks after it has sent the input update since it waits for a synchronization message sent by the proxy, for example, because it requires some data from the proxy that has yet to be produced by a proxy-side service execution. As soon as the proxy sends the synchronization message, the client- and the proxy-side workflow graphs pass their respective transitions synchronously and (possibly) exchange data. When a resource has been created by the execution of a service that possesses a graphical presentation, it is embedded at the client side.

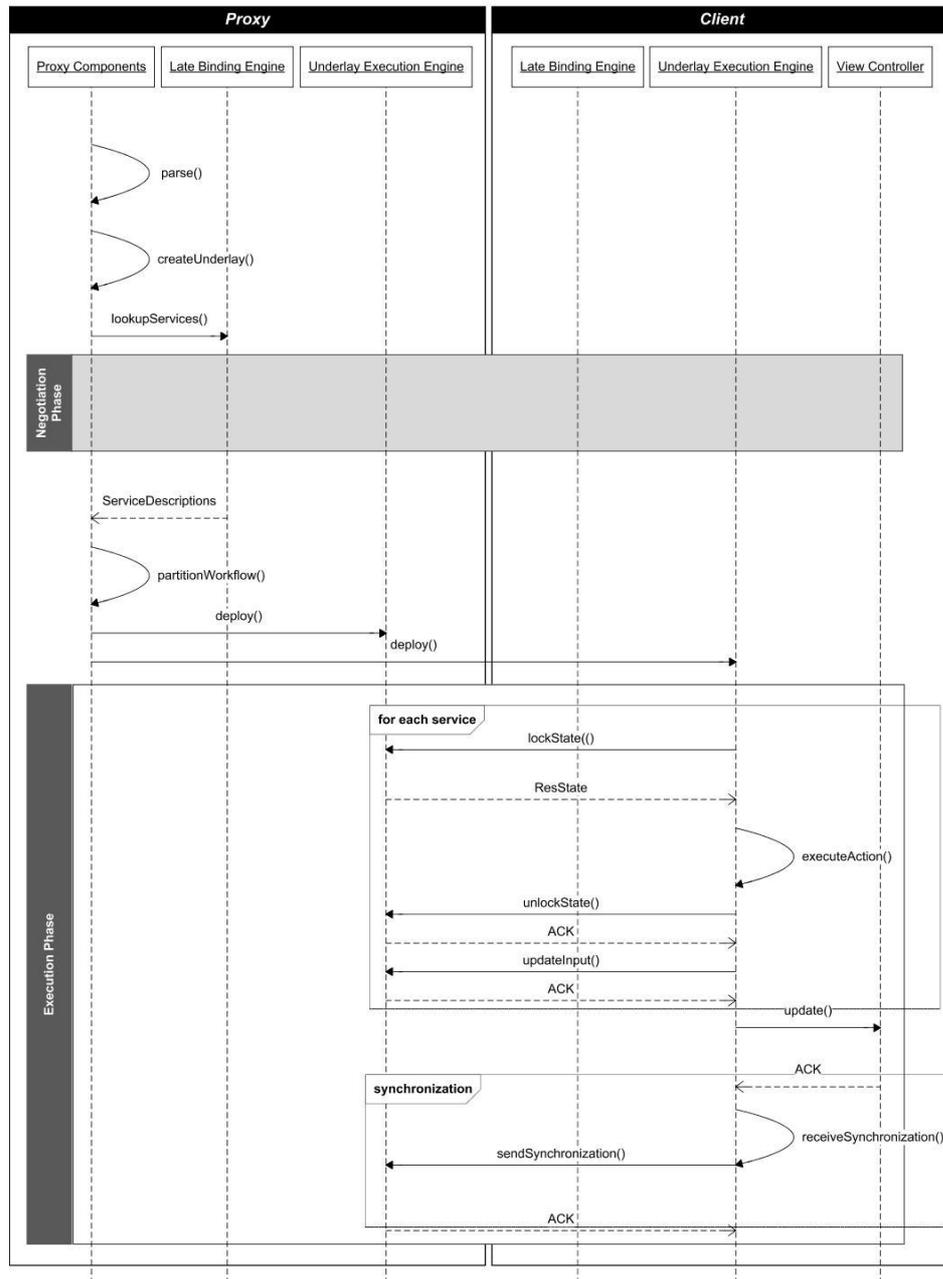


Figure 7.7: Embedding of the OSP's execution phase.

7.3.5 Recovery: Replacing Failed Services

Service recovery is realized by a multi-layered approach. It is aimed at a local replacement of services, so that the proxy does not necessarily have to be involved when a client-side service fails. Thereby, services may also be replaced during an offline mode of the application.

Three general cases can be distinguished.

1. A service replacement is found locally,
2. a service replacement is found on another device, or
3. no service replacement is found at all.

When a service failure is detected by a client-side underlay execution engine, it calls the *bindServices()* primitive of the local late binding engine directly to look up a potential replacement. Here, the abstract service related to the failed concrete service is passed as parameter. When the late binding engine is able to discover another concrete service that matches the respective abstract service, the underlay execution engine checks whether it has a service call cached for the discovered service description as introduced in section 7.2.9. If that is the case, the service can be directly replaced. Otherwise, service recovery is realized by the partial repetition of the OSP's negotiation phase.

If a service drops out at the client side that cannot be replaced locally as described above, the local underlay execution engine notifies the proxy-side execution engine by invoking the *recoverService()* primitive. The proxy-side execution engine then calls the late binding engine via the *lookupService()* primitive, which triggers the OSP's negotiation phase as described in section 7.3.2. Finally, a call is generated for the returned service description so that the service can be integrated at the client side.

Assume a service at the client side drops out as shown in Figure 7.8. If a replacement is found locally, the service can be substituted via the *replaceService()* primitive without notifying the proxy. If no replacement can be found, the proxy has to be requested for assistance, thus, the late binding engine of the proxy has to look for an appropriate replacement for the failed service. When a service is found, it is integrated into the underlay system via the *insertService()* primitive, while the client removes the failed service by means of the *deleteService()* primitive. If no replacement can be found on either the client or proxy side, the recovery fails and the underlay system's execution is aborted, i.e. the *abort()* primitive is executed on the proxy as well as on the client side.

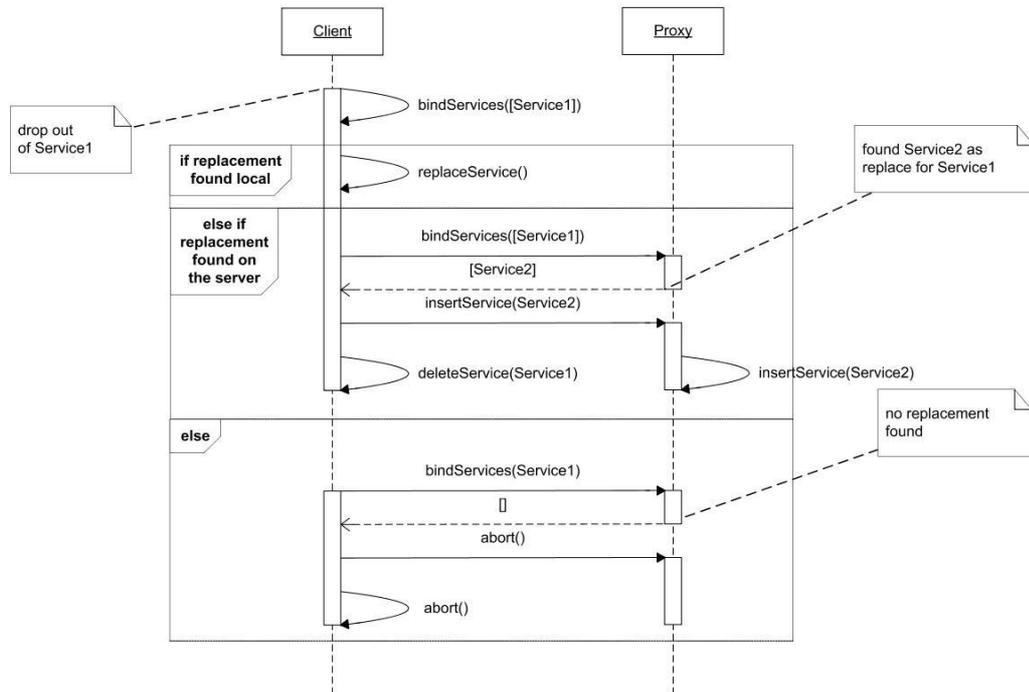


Figure 7.8: OSP: Service recovery - Sequence diagram

CHAPTER 8

CREATING DYNAMIC AND DISTRIBUTABLE WEB MASHUPS

In this chapter, the underlay system introduced in chapter 4 as well as the algorithms for its automatic creation and distribution presented in chapters 5 and 6, respectively, are embedded within the Web's architecture to demonstrate the instantiation of the theoretical results within the Web domain. Here, the implementation follows the specification of components and interfaces defined in chapter 7.

This chapter does not try to discuss the concrete implementation details, but rather to create a mapping from an abstract specification to the real world domain of the Web. Most of the related components are bundled under the name of *MashWeb* and made available at *myLab*¹, a laboratory for research of technologies for Web and Web2.0 driven by the Fraunhofer Institut FOKUS.

The following three sections 8.1, 8.2, and 8.3 deal with the realization of service integration, the execution of distributed underlay systems with respect to their required runtime environment and communication protocol, and their automatic creation. Finally, section 8.4 discusses three application scenarios built on top of these components².

8.1 DYNAMIC INTEGRATION OF RESOURCES

Resource integration is realized by a late binding engine that performs a mapping from abstract services to concrete services. Here, an abstract service encompasses the service's abstract name, its IOPE description, and the resource on which the service operates. Three different layers are distinguished. The abstract services constitute the late binding engine's request interface, i.e. an abstract service is considered as a request for the lookup of one or more concrete services. A concrete service is a service description that

¹<http://mylab.fokus.fraunhofer.de/service/mashweb/overview>

²These demos do not constitute the author's sole achievements, but were jointly developed at the Fraunhofer institut FOKUS in the scope of various projects and activities on top of *MashWeb*.

defines the service's interfaces. Both layers are mediated by a mapping layer, which relates the single parts of an abstract service to a concrete service.

An example is depicted in Figure 8.1.

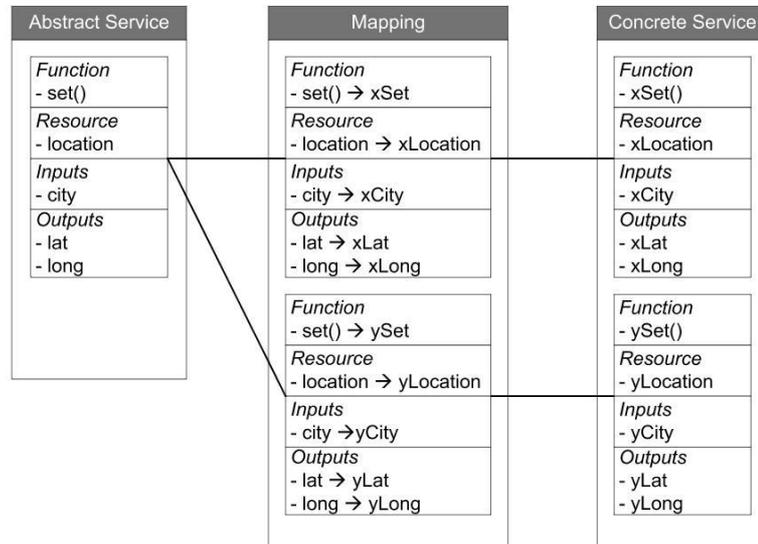


Figure 8.1: Mapping abstract services to concrete services.

Here, a service that derives the latitude and longitude of a city given as string, is specified as an abstract function *set()* operating on a resource *location*. It possesses an abstract input value *city* and two abstract output values *lat* and *long*. The mapping layer contains two maps, in which the abstract services are related to concrete services. The services themselves are given in the third layer. The separation of the mapping between abstract and concrete services by means of a mapping layer decouples the concrete service description from the abstract one, so that the mapping can be taken up although a different interface description language is used to model the concrete service.

Since most services within the Web are made available via REST or JavaScript interfaces, the Web Application Description Language (WADL) [73] has been chosen as the interface description language in the prototype. Thus, a mapping is provided that receives an abstract service description as input and can –in turn– respond with a set of matching concrete services identified via their respective WADL descriptions.

A simple implementation allows the generation of a REST service call from a corresponding WADL file [123]. A small extension was implemented in order to support also the addressing of services exposing a JavaScript API.

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <application xmlns:xs="http://www.w3.org/2001/XMLSchema"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5   xmlns="http://research.sun.com/wadl/2006/10">
6
7 <resources base="http://maps.google.com/">
8   <resource path="staticmap">
9     <method name="GET" mref="getMapImage">
10      <request>
11        <param style="query" required="true" name="key"
12          fixed="xxx"/>
13        <param style="template" required="true" name="lat"
14          type="xs:double"/>
15        <param style="template" required="true" name="lng"
16          type="xs:double"/>
17        <param style="query" required="true" name="center"
18          fixed="{lat},{lng}"/>
19        <param style="template" required="true" name="width"
20          type="xs:int"/>
21        <param style="template" required="true" name="height"
22          type="xs:int"/>
23        <param style="query" required="true" name="size"
24          fixed="{width}x{height}"/>
25        <param style="query" required="true" name="zoom"
26          type="xs:int"/>
27        <param style="query" required="false" name="maptype"/>
28        <param style="query" required="false" name="markers"/>
29      </request>
30
31      <response>
32        <param name="map_url" style="plain" fixed="{request.uri}"/>
33        <representation mediaType="application/octet-stream">
34          </representation>
35        </response>
36      </method>
37    </resource>
38  </resources>
39
40 </application>

```

Listing 8.1: WADL description for Google Maps (static).

Listing 8.1 shows the WADL description for the static Google Map used within the prototype implementation; it is included in the service repository to make it accessible during the binding procedure.

The mapping shown in Table 8.1 enables the access to this concrete service via an abstract service.

<i>Abstract functions</i>	<i>Google Static Map API</i>
<i>Action Mapping</i>	
create	getMapImage
<i>Request Parameter Mapping</i>	
lat	center(lat,lng)
lng	center(lat,lng)
width	size(widthxheight)
height	size(widthxheight)
type	maptype
ZoomLevel	zoom
maptypeControl	
panControl	
zoomControl	
draggableMap	
zoomOnDoubleClick	
<i>Response Parameter Mapping</i>	
uri	map_url

Table 8.1: Mapping for Google Maps (static).

Note that only the inputs and outputs of a service are required during the late binding mechanism. The services' preconditions and effects are part of their dependency description and are therefore only required during the automatic service composition process. Their usage is later discussed in section 8.3.

Abstract services can be derived from multiple kinds of inputs tailored for specific purposes. For instance, the markup language is considered as a special way to describe abstract services, as every tag encompasses a resource, an abstract function, and a set of inputs and outputs given as attributes.

In order to support the engineering approach for the creation of underlay systems, an abstract JavaScript API has been developed that enables Web developers to access resources via a unified JavaScript API. An example of the integration of a map is shown in Listing 8.2. Here, a resource *myMap* is generated by the *createMap()* function, which dynamically includes a map into a Web application via a service capable of creating the resource with the given parameters, i.e. that is draggable, can be zoomed, and so forth. Through the optional attribute *provider*, the dynamic binding procedure can

```
1 <script type="text/javascript">
2 mapManager.createMap({
3   id: "mymap",
4   provider: "google",
5   width: inputs.width,
6   height: inputs.height,
7   lat: inputs.lat,
8   lng: inputs.lng,
9   maptypeControl: true,
10  zoomOnDoubleClick: true,
11  draggableMap: true,
12  zoomControl: true
13 });
14 </script>
```

Listing 8.2: Creating a map with the MashWeb JavaScript API.

be circumvented, so a service can be explicitly be chosen instead of selecting one dynamically based on user preferences or other additional information.

8.2 EXECUTING UNDERLAY SYSTEMS

In this section, a Web conform realization of the underlay system is introduced. Section 8.2.1 discusses the realization of the workflow model in detail, highlighting the central aspects of resource orientation, state management, and inter-workflow communication. Section 8.2.2 deals with a respective runtime environment and describes the implementation of the OSP to realize communication between multiple underlay systems.

8.2.1 The Workflow Model

Figure 8.2 shows the abstract model of workflow graphs as defined in section 4.3. The root element of a workflow graph is represented by an instance of the class *TimedAutomaton*. This class declares a set of *locations* representing the locations of the workflow graph, in which *initialLocation* holds the initial location. The same applies to *transitions*, which defines a set of instances from the class *Transition* representing the transitions of the workflow graph. Furthermore, the property variables of the class *TimedAutomaton* hold all declared variables represented by the interface *Variable*. A clock (instance of class *ClockVariable*) is a special kind of variable which accepts only values from type integers representing the number of clock ticks. Finally, the property *actions* declares a set of actions (interface *Action*) used in the transitions

of the workflow graph. Assignments are represented by the class *Assignment* and constitute a simple kind of actions that only assign a new value to a variable. Class *Location* holds all outgoing and incoming transitions defined in the properties *outgoings* and *incomings* of this class and an *invariant* as instance of class *Constraint*. Similar, class *Transition* holds the *source* and *target* locations (bidirectional references between *Locations* and *Transitions*) of the transition, the action, a set of assignments, and a guard from type *Constraint*. The interface *Constraint* provides the single method *isValid()* which checks if the constraint is valid or not.

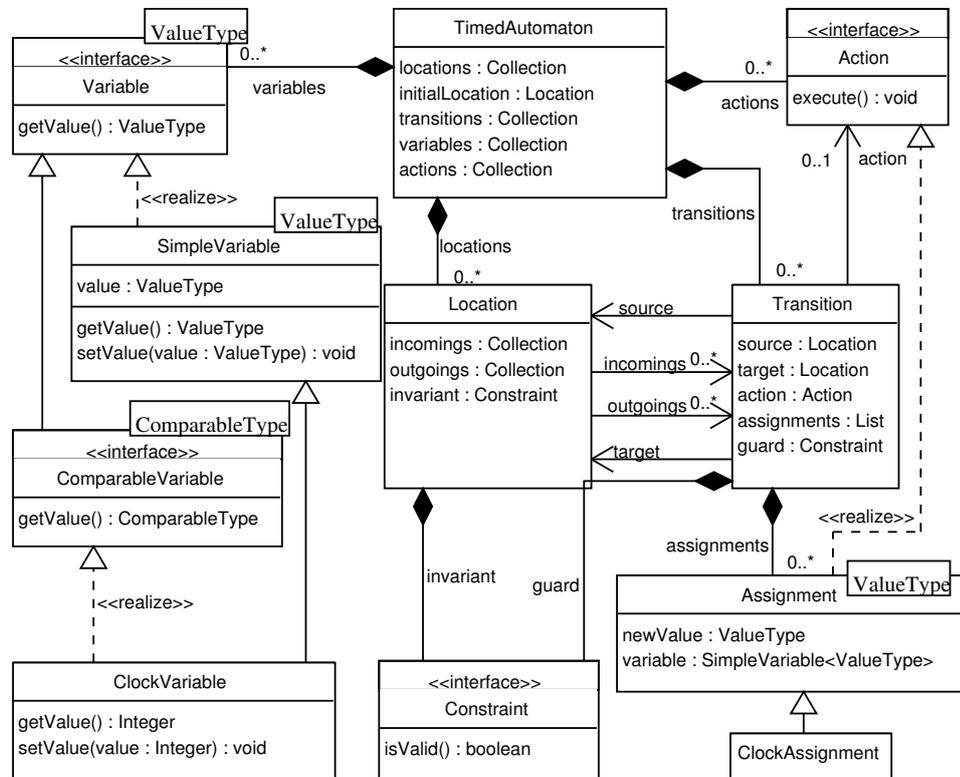


Figure 8.2: UML diagram of the workflow model.

Workflows are represented as XML files in which actions and variables of the underlying timed automaton are identified with WADL methods and parameters. The respective WADL files can be parsed during runtime; the respective REST calls are generated automatically based on the WADL descriptions and included in the timed automaton.

Listing 8.3 shows a partial skeleton of a XML file holding a timed automata based workflow with WADL descriptions, specifying the import of an external WADL file.

The root element `<element name="ta">` defines a new XML element with the name `ta` containing all sub-elements. An `import` element contains one (`minOccurs="1"`) or more (`maxOccurs="unbounded"`) `wadl` elements defining `<complexType name="wadl">` elements. These complex types specify the `wadl` elements with the attribute `location` containing the URL of the WADL document and one or more `method` elements. Here, `method` elements are defined by the attributes' `id` and `wadlName` required to locate the WADL method in the WADL document.

The timed automata based workflow description itself is mainly specified by two elements. A `<location>` element describes a location of a timed automaton; it possesses four attributes as listed in Table 8.2.

Attribute	Description
<code>initial</code>	Specifies whether the location is an initial location or not
<code>invariant</code>	Contains the invariants that have to be fulfilled in the location
<code>id</code>	Unique identifier of the location
<code>label</code>	Name of the location (optional)

Table 8.2: Attributes of locations.

A `<transition>` element describes the transition connecting two locations, featuring six attributes as enumerated in Table 8.3.

Attribute	Description
<code>initial</code>	Restricts the passage of the transition
<code>invariant</code>	Assigns values to clocks or variables
<code>source</code>	Contains the id of the source location
<code>target</code>	Contains the id of the target location
<code>action</code>	Contains the id of the action
<code>label</code>	Name of the transition (optional)

Table 8.3: Attributes of transitions.

```

1 <element name="ta">
2 <element name="import" minOccurs="0">
3   <complexType>
4     <sequence>
5       <element name="wadl"
6         type="tns:wadl"
7         minOccurs="1"
8         maxOccurs="unbounded"/>
9     </sequence>
10  </complexType>
11 </element>
12 ...
13 Other Timed Automaton Elements
14 ...
15 <complexType name="wadl">
16   <sequence>
17     <element name="method"
18       minOccurs="1"
19       maxOccurs="unbounded">
20       <complexType>
21         <attribute name="id"
22           type="ID"
23           use="required"/>
24         <attribute name="wadlName"
25           type="string"
26           use="required"/>
27       </complexType>
28     </element>
29   </sequence>
30   <attribute name="location"
31     type="anyURI"
32     use="required"/>
33 </complexType>
34 ...
35 Other timed automaton type definitions
36 ...
37 </element>

```

Listing 8.3: Part of the timed automaton specification.

8.2.1.1 Realizing Inter-Workflow Communication

In order to enable a distributed execution of underlay systems, the workflow model is extended by the concept of synchronizations. The new classes are depicted in 8.3.

The *Synchronization* interface enables the exchange of primitives between distributed workflows. It has two subinterfaces *ReceiveSynchronization* and

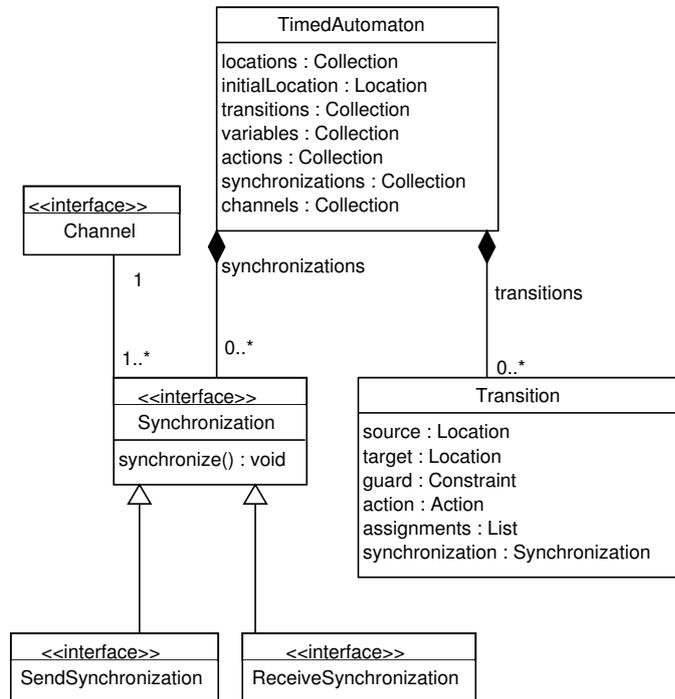


Figure 8.3: UML diagram of the workflow model’s extension towards inter-workflow communication.

SendSynchronization. The former is responsible for receiving messages, while the latter is used to send synchronization primitives.

Communication takes place over abstract channels specified by the interface *Channel*. Each synchronization has exactly one channel referenced by the *channel* attribute. The *TimedAutomaton* class contains a collection of *synchronizations* used within its execution. In addition to a source, a target, an action, assignments, and guards, a transition can encompass a *synchronization*, which is either of receive or of send type.

8.2.1.2 Considering the Notion of Resources

To realize a resource-oriented view on services as introduced in section 4.3.6, an action is identified with the service invocation and the respective resource it is executed upon. The corresponding class diagram is shown in Figure 8.4.

The resource class hierarchy is depicted in Figure 8.5. A resource is created by the *ResourceFactory* class. Simultaneously, it is placed inside of the *ResourceRepo* repository, which represents the resource space as introduced in section 6.2; here, resources can be looked up later on.

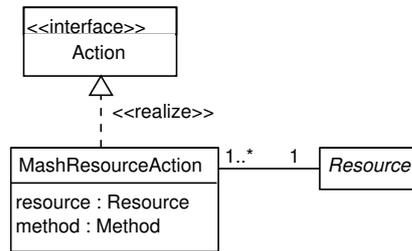


Figure 8.4: UML diagram for the consideration of resource-oriented actions.

Each resource possesses a *Semaphore* object which handles accesses to the resource's state. The semaphore has two methods *acquire* and *release*. The first one is called before acquiring access to the state. If the resource is used by another action, the method blocks until it is freed by calling the *release* method. Calling a resource's method always incorporates *acquire* and *release* calls, respectively, in order to guarantee a unique access to the current state of the resource.

Currently, there are sixteen resource types defined, i.e. classes deriving from *Resource* as depicted in Figure 8.5, where resources' methods and attributes were left out in order to accomplish a clearer arrangement of elements. A full overview of the current classification is listed in appendix B. The class structure can be extended if required.

8.2.1.3 State Management

Since a resource's state is identified by its attributes' values, it can be serialized to a JSON object [128], which is a key-value pair structure. This notation is more space efficient than XML, which is used in the standard SOA stack. It is advantageous as it minimizes the data amount that needs to be communicated by resource state synchronization. Moreover, the processing of JSON, particularly at the client side, is quicker and less expensive, as it is native to JavaScript. The implementation for Java facilitates the mapping between the native types and JSON. In addition, the content of two JSON objects can be easily compared. This has a significance for the state synchronization, since not all elements of a resource have to be exchanged when communicating a resource's new state, but only those that have changed since the last update.

The generation of resources' state updates that only contain the modified version of the resources' state is achieved by looking up the current holder of the lock at the proxy side. When a client-side service wants to gain access to

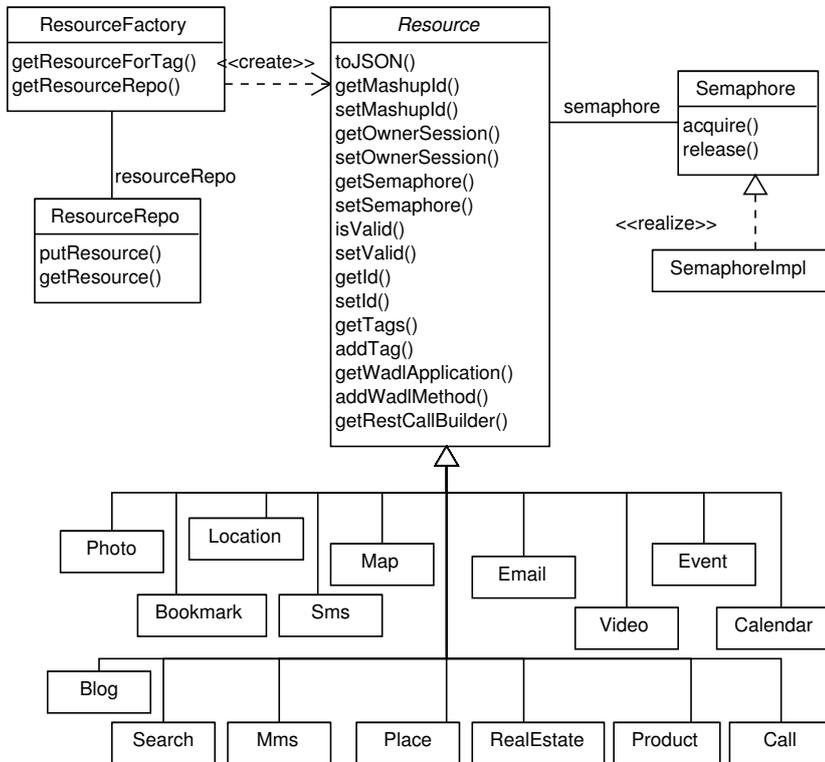


Figure 8.5: UML diagram: Resource hierarchies.

a specific resource, it checks whether it is already the holder of the resource's lock itself by calling the resource's monitor *acquire* method. If the client is the lock's owner, the service can directly access the resource. Otherwise, a *getState()* or *lockState()* request is sent to the proxy, where the former primitive requests a read-only access to the resource and the latter one a read-write access³.

In the following the process for the *lockState()* primitive is described. After receiving the request, the proxy looks up the lock locally. If the proxy itself is the lock's owner, the resource's state update is sent back to the requesting client as soon as the lock is released. The requesting client becomes the new lock owner. Due to its central position, the proxy serves as mediator for requesting a resource's lock. Thus, if the lock is kept by another client, the proxy forwards the *lockState()* request to the corresponding client, which responds with the actual resource's state as soon as the lock is freed. The *id* of the new lock owner is saved at the proxy in order to enable a quick

³See section 7.3.4 for details

retrieval of the next request. Moreover, a time stamp is recorded in order to facilitate the comparison in a state update. The lock owner's response is forwarded to the requesting client as soon as it is received at the proxy side.

The treatment of the *getLock()* primitive, i.e. the request for accessing a resource in a read-only mode, is similar. However, since there is no need to wait until the lock has been released by the current lock's owner, the state update is issued immediately. Note that the requester does not become the resource's lock owner in that case.

8.2.2 The Underlay Execution Engine

Within this section, the underlay execution engine's realization is discussed. Section 8.2.2.1 introduces the runtime environment for workflow and related dataflow graphs, while section 8.2.2.2 concentrates on the realization of the OSP.

8.2.2.1 Server and Client Side Runtime Environments

The underlay system's runtime environment⁴ is implemented in Java and can be either located on the Web (where it is accessible by a REST interface) or locally on the client side; the basic workflow execution engine has been published in [145]. For the latter case, a respective runtime has been implemented in pure JavaScript, so that it can be transferred to the client during the negotiation phase of the OSP as introduced in section 7.3. After compression, the client-side runtime encompasses less than 50KB code and can be rapidly transferred to client-side devices. Thereby, the client does not need to be modified to support the usage of the proposed components; a Web browser featuring a JavaScript engine is the only requirement.

A key advantage of the runtime's realization is its abstraction from concrete service implementations. Instead, appropriate 3rd party services can be dynamically integrated into the workflow during runtime based on the IOPE descriptions of the abstract services. Thus, the runtime engine is independent of the internal structures of services and relies on the restricted I/O behavior of services instead. By passing a transition within the workflow graph, postulating that all inputs of the action related to this transition are available, the workflow runtime engine delegates the input parameter values to an invoker component, which calls the real third party service and delegates the resulting output parameter values to the workflow runtime engine after

⁴As in the scope of section 7.3.1, the term *runtime* is abstractly used for all components required at either the client or server side to improve readability.

the service has been executed. Through this clear separation between the execution semantic of underlay systems and the execution semantic of third party services, the runtime engine is capable of integrating heterogeneous third party services, i.e. services with varying kinds of binding methods such as WSDL or WADL.

The UML statechart diagram depicted in Figure 8.6 models the components necessary for describing the semantics of the runtime engine.

The two main components are represented by the two parallel states (statechart states) *Executor* and *Timer*. Furthermore, there is a parallel state called *ActionThread_i* for each action α_i that is necessary to call the service bound to the action α_i . The parallel state *Receiver* is needed to notify the *Executor* process when data is available to read from a channel queue. In this case, transitions with receive synchronizations related to the same channel can read and remove the received data from the channel.

The *Timer* process acts as global clock that holds the time in the variable *time*. The value range of time is N and represents the count of clock ticks since the start of the system; the duration of a clock tick is configurable. In the current implementation, the duration of a clock tick is equal to the time between two following events (i.e. state charts events) *TimerInterrupt*. The *Timer* process consists of the two states ACTIVE and PAUSED and a final state that can be reached through the event *Terminate* triggered by calling the method *Terminate()* in the *Executor* process. Note that –within the scope of this state chart diagram– all methods trigger events with the same name, e.g. the event *Terminate* triggers the method *Terminate()*. The two processes run concurrently. If the *Timer* process is in the state ACTIVE, the *Executor* process is in the state WAITING. The same applies if the *Timer* process is in the state PAUSED, which implies that the *Executor* process is active (and not in state WAITING). These two cases represent delayed transitions and action transitions of the related transition system to the given timed automata as introduced in section 4.3.

After selecting the initial location l_0 from the timed automaton using *selectInitialLocation()*, the runtime engine moves into the state (START, PAUSED), which means that the *Executor* process moves into its initial state START and the *Timer* process in the state PAUSED. Therefore, the *Executor* process marks the initial location as current location $l = l_0$ and moves into the next state SELECT OUTS. Afterwards, all outgoing transitions of the current location l are selected. If the returned list *outs* is empty, the runtime terminates and the *Executor* process moves into its final state. If the current location has at least one outgoing transition, a random transition $t_i = (l_i, g_i, \alpha_i, s_i, \lambda_i, l'_i)$ is selected, where l_i is the current location l in step i and the guard g_i is *true*. The guard g_i is false if s_i is a receive synchroniza-

tion and the current queue of s_i 's channel is empty. If t_i is not empty, the *Executor* process sets the current location l to l'_i , resets all clocks in λ_i using *resetClocks()*, and calls action α_i . Here, α_i is started via *Start_i()*, which triggers the event *Start_i* of *ActionThread_i*, sends data to the channel of s_i using *sendData(s_i)* if s_i is a send synchronization, and moves into the state SELECT OUTS to begin a new iteration (SELECT OUTS \rightarrow CHECK END \rightarrow CHECK ENABLED). Otherwise, if t_i is empty, denoting that all guards of all outgoing transitions are false, the *Executor* process moves into the state WAITING after calling *Resume()*, which triggers the Event *Resume* and the *Timer* process proceeds into state ACTIVE.

The *Timer* process increments all clocks after T clock ticks using *UpdateClocks()*. In this case, T represents the time unit for incrementing clocks. Afterwards, the *Timer* process moves back into state STOPPED after calling *Notify()* triggering the event *Notify* in the *Executor* process to start a new iteration. In this case, the value of some guards may change from false to true after incrementing the clocks. Similarly, the value of a guard can change from false to true if the execution of an action α_i terminates. In this case the *ActionThread_i* calls *Notify()* such as within the *Timer* process.

The *Receiver* process continuously waits for a *Receive* event in state RECEIVING. When receiving data, the process updates all related channel queues and notifies the *Executor* process by triggering the event *Notify*. In this event, the *Executor* process starts a new iteration and checks if new transitions are enabled.

The UML class diagram depicted in Figure 8.7 shows the main classes of the workflow execution engine.

The entry class of this component is *TimedAutomatonExecutor* which provides the single method *execute()*. This method operates on a workflow graph from type *TimedAutomaton* and returns an object from class *RuntimeHandler* necessary to control the execution of the given workflow graph. Furthermore, it is possible to change some configuration parameters such as the time for a clock tick in milliseconds using an instance of class *ExecutionConfig*. The *RuntimeHandler* provides methods necessary to start, suspend, terminate, or resume the execution of a workflow graph. It is also possible through a *RuntimeHandler* instance to get information on the current state of the execution represented by class *RuntimeInfo*, which provides methods to get the current location and all active transitions of the workflow graph, i.e. all transitions whose actions are currently running. Furthermore, the *RuntimeHandler* provides methods to subscribe and unsubscribe to instances from the type *RuntimeListener* to notify registered applications when entering or exiting a location (methods *locationEntered()* and *locationExited()*) or when passing a transition (*transitionEntered()* or *tran-*

sitionExited()). The implementation class *RuntimeHandlerImpl* of the interface *RuntimeHandler* manages all threads of running actions represented by the class *ActionThread*; a timer thread is represented by the class *Timer*.

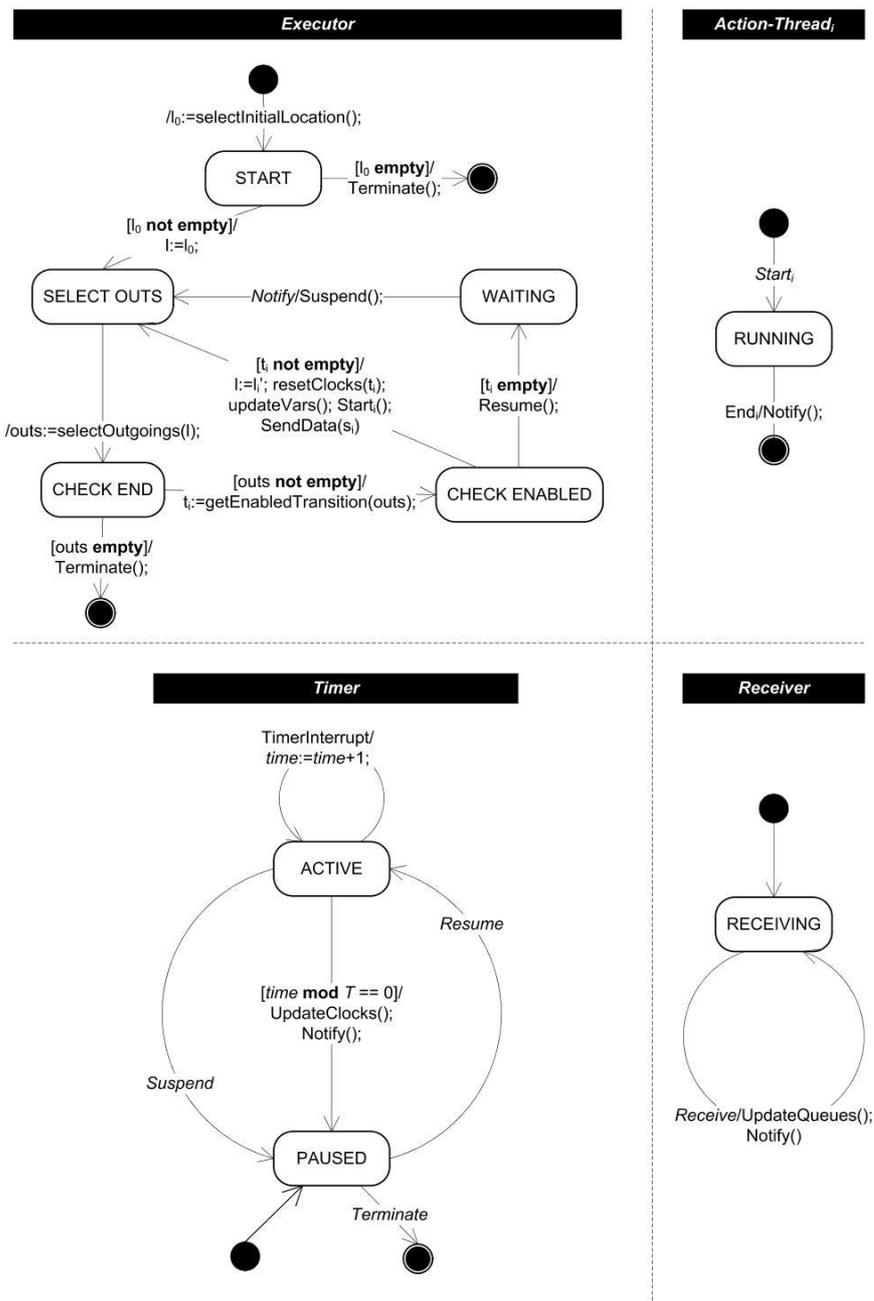


Figure 8.6: UML statechart diagram of the workflow execution engine.

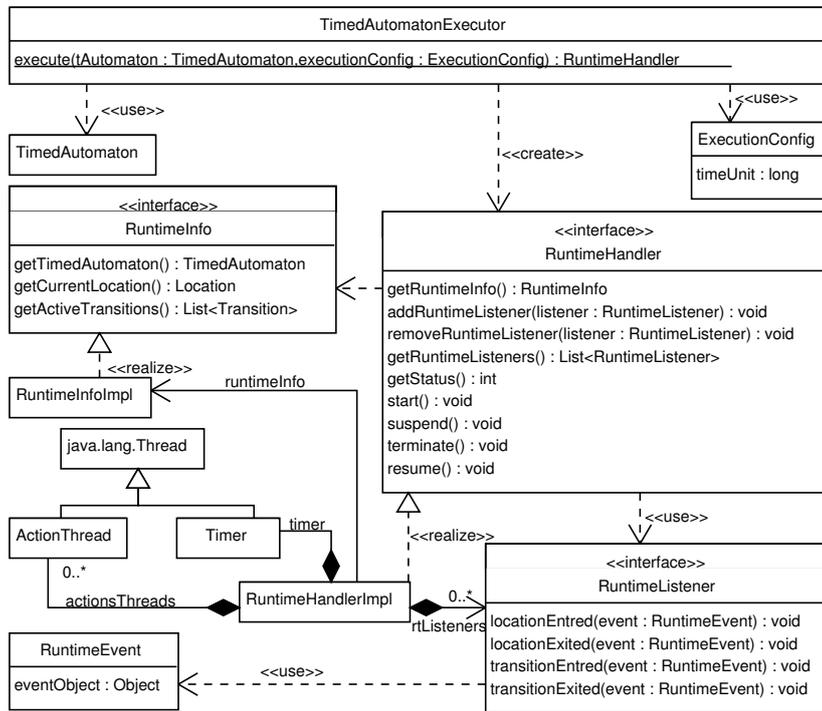


Figure 8.7: The workflow execution engine.

8.2.2.2 *jOSP: A Java-based Implementation of the Orchestration Synchronization Protocol (OSP)*

Rose distinguished three general approaches to the design of application protocols [158].

- Employing an already existing protocol
- Defining an exchange model on top of an already existing infrastructure
- Defining a novel protocol from scratch

To minimize implementation effort and avoid unnecessary protocol fragmentation, the options should be checked in the order given above.

There is no protocol in place that mediates the distributed execution of underlay systems on the Web, but the OSP can be specified on top of HTTP so that synchronization messages are transferred via HTTP. By relying on HTTP, the OSP inherits the whole HTTP infrastructure, e.g. authentication mechanism, proxies, MIME data encoding and so forth, and thereby ensures a smooth embedding of the OSP into today's Web architecture.

The proxy-side runtime environment is addressable via a REST interface and communicates with the client side runtime environment in accordance with the Web's architectural style by means of the request/response pattern. This classic pattern implies that communication is always initiated by a client's request that is –in turn– answered by a server's response. Thus, communication between Web service consumer and provider in SOA as well as for *RESTful* services is generally synchronous, since their messaging protocols depends on HTTP.

However, in order to support the distribution of Web applications that run on multiple devices, bidirectional communication between the underlay execution engines is required. Most notably, the proxy must be able to initialize primitive exchanges. There are multiple solutions in place to realize server initiated communication within the Web as discussed in section 2.2.4.1. Candidates are technologies for the realization of a *server push* such as Comet [41]. Many of those approaches fail to work across multiple different Web browsers. The Direct Web Remoting (DWR) framework is capable of providing the required functionality, as it constitutes a RPC library that makes it possible to call Java functions from JavaScript and to call JavaScript functions from Java [164]. It works also among different kinds of Web browsers.

In the following, the server-side implementation of the synchronization protocol is described, but the realization at the client also has a similar class structure.

There are two Java classes, namely the *Receiver* and *AsynchResponseListener*, that are provided by DWR and referenced within a Web browser.

Both classes inherit from *AbstractReceiver* and are responsible for handling the synchronization protocol's primitives received from a client. More precisely, the first handles interactions initiated by client, while the latter accepts responses to prior server's requests. The reason for such a realization is the simulation of a synchronous request/response message exchange pattern in an environment which does not allow such interaction types. The UML class diagram 8.8 shows the inheritance of the two classes.

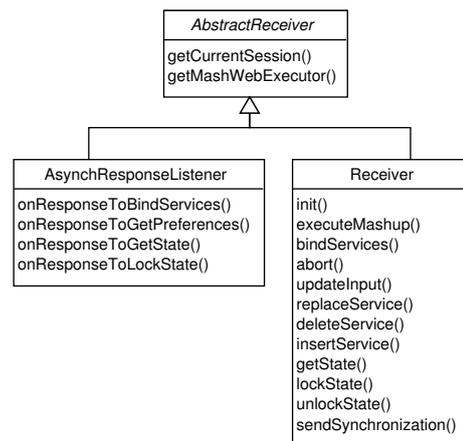


Figure 8.8: Receiving Messages: UML Class Diagram.

There are further classes defined by DWR employed to send protocol primitives. A *Sender*, a *ProxySender*, and *AsynchResponder* class are inherited from the abstract class *AbstractSender* as depicted in Figure 8.9. The super class implements common methods used to send a message. They employ DWR to generate a JavaScript tag dynamically and add it to the HTML site that is opened by the receiver. Methods in the *Sender* class are used when a server initiates primitive exchanges. The *AsynchResponder* responds to prior client's requests. The *ProxySender* class is used to forward messages that originate from a client or a server and are addressed to multiple clients.

8.2.2.3 Presentation Mapping

Resources are represented as JSON objects, where the single key-value pairs encompass the resource's attributes. Every resource defined within the scope of an underlay system can be uniquely addressed by its *id*. To provide a mapping from a resource to a respective presentation, a HTML `<div>` tag is included within the HTML page containing the mashup. This tag is linked to

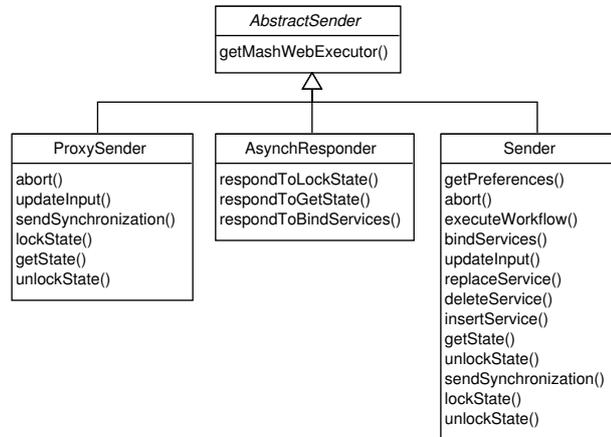


Figure 8.9: Sending Messages: UML Class Diagram.

a resource via its *id*, so that the resources presentation is included within the scope of the `<div>` tag. Here, a presentation mapping defines the exact way a resource is presented. A resource listener monitors operations on resources and updates the resource's presentation via a common Ajax request.

8.3 CREATING UNDERLAY SYSTEMS

In this section, the implementation of the mandatory components for the automatic creation of service compositions is described. Here, section 8.3.1 outlines the modeling and instantiation of semantic service descriptions, while section 8.3.2 encompasses the realization of the discovery and the composition module.

8.3.1 Semantic Description Specification and Discovery

Based on the lightweight model for semantic service descriptions introduced in section 5.2, the model's specification is given based on XML Schema (XSD) [200] so that the semantic service descriptions can be defined as XML documents with respect to the language specification.

There are a lot of tools which can parse XML documents and convert them to an internal representation based on the specification defined in the XML Schema document. A new Java API called Java Architecture for XML Binding (JAXB) [89] is one of these tools which generate a Java model from the XML Schema definition. Listing 8.4 shows a part of the XSD specification of services as defined in the semantic model. The `<service>` element

constitutes the XML document's root element and holds the semantic service description. It contains a set of `<operation>` elements as defined in the operation specification shown in Listing 8.5, and two attributes `id` and `name` used as identifier and label for the service, respectively.

```

1 <element name="service">
2   <complexType>
3     <sequence>
4       <element ref="tns:operation" minOccurs="0"
5         maxOccurs="unbounded"/>
6     </sequence>
7     <attribute name="id" type="int"/>
8     <attribute name="name" type="string"
9       use="required"/>
10  </complexType>
11
12  <!-- Restrictions definition -->
13  ...
14 </element>

```

Listing 8.4: Service specification.

An `<operation>` element as specified in Listing 8.5 contains the sub-elements `<inputs>`, `<outputs>`, `<preconditions>`, `<effects>`, `<rtPreconditions>`, and `<rtEffects>` representing the corresponding sets defined in the semantic model as introduced in section 5.2. Listing 8.5 shows the specification of the elements `<inputs>` and `<preconditions>`. The `<inputs>` element contains a set of sub-elements `<input>` specified in the XML data type `parameter`. The element `<preconditions>` contains a set of sub-elements `<precondition>` specified in the XML data type `statement`. The same applies to the other elements that were left out in the listings for reasons of clarity.

```

1 <element name="operation">
2   <complexType>
3     <sequence>
4       <element name="inputs" minOccurs="0">
5         <complexType>
6           <sequence minOccurs="1">
7             <element name="input" type="tns:parameter"
8               minOccurs="1" maxOccurs="unbounded"/>
9           </sequence>
10          </complexType>
11        </element>
12
13        <element name="preconditions" minOccurs="0">
14          <complexType>
15            <sequence>
16              <element name="precondition" type="tns:statement"
17                minOccurs="1" maxOccurs="unbounded"/>

```

```

18     </sequence>
19     </complexType>
20 </element>
21
22     <!-- Other element definitions -->
23     ...
24 </sequence>
25 <attribute name="id" type="int"/>
26 <attribute name="name" type="string"/>
27 </complexType>
28
29 <!-- Restrictions -->
30     ...
31 </element>
32
33 <!-- Complex types definitions -->
34 <complexType name="parameter">
35 <attribute name="id" type="ID"/>
36 <attribute name="name" type="string" use="required"/>
37 <attribute name="modelRef" type="anyURI" use="required"/>
38 <attribute name="internal" type="boolean" default="false"/>
39 </complexType>
40
41 <complexType name="statement">
42 <attribute name="id" type="ID"/>
43 <!-- The subject references a parameter element -->
44 <attribute name="subject" type="IDREF" use="required"/>
45 <attribute name="predicate" type="anyURI" use="required"/>
46 <!-- The object references a parameter element -->
47 <attribute name="object" type="IDREF"/>
48 <attribute name="correct" type="boolean"/>
49 </complexType>

```

Listing 8.5: Operation specification.

After converting a service description document to an internal representation –here, a Java model as depicted in Figure 8.10 was chosen– it can be easily saved in a database representing the service description repository using object-relational mapping (ORM) APIs, which convert data between object-oriented programming languages and relational databases. Hibernate [76], which is an ORM library for Java, is used in the implementation to decouple the algorithms from the underlying database. The discovery module operates on the database representing the semantic repository and selects operations using the Hibernate Query Language (HQL), which operates on classes instead of relational database tables. The implementation class *DiscoveryMediatorImpl* of the interface *DiscoveryMediator*, which builds the entry point for this module, shows that the discovery module operates on the

semantic repository using HQL. Furthermore, the Spring Framework [186] is used to provide a set of useful components like Spring-ORM and Inversion of Control (IoC) container that facilitate the usage of Hibernate.

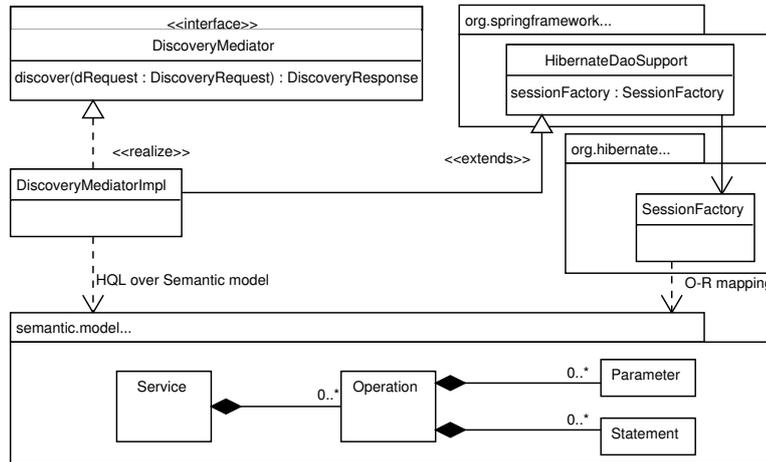


Figure 8.10: UML diagram of the discovery module.

8.3.2 Composition Module

This section outlines a part of the realization of the composition module. The UML diagram of this module is depicted in Figure 8.11 and gives an overview of the classes necessary for the realization of the composition algorithm. The interface *CompositionMediator* builds the entry point for the module and handles discovery requests represented by the class *CompositionRequest*. Furthermore, a configuration object of the class *CompositionConfig* is required to set configuration parameters such as *MAX_CAPACITY* for the capacity limit of the pool size and *MAX_NODES* for the maximal number of nodes in the composition response.

The output of the composition component interface is an instance of the class *CompositionResponse* representing the set of composition nodes generated by the composition algorithm. An implementation of the interface *CompositionMediator* is represented by the class *CompositionMediatorImpl*, which has a reference to the discovery module to find new service operations in each composition step. Other classes in the UML diagram like *CompositionNode* and *CompositionNodePool* represent data types defined in the previous sections with the same functionality. In another part of the composition module, the converter generates workflow and dataflow graphs from the

abstract compositions, i.e. composition nodes, returned by the composition component.

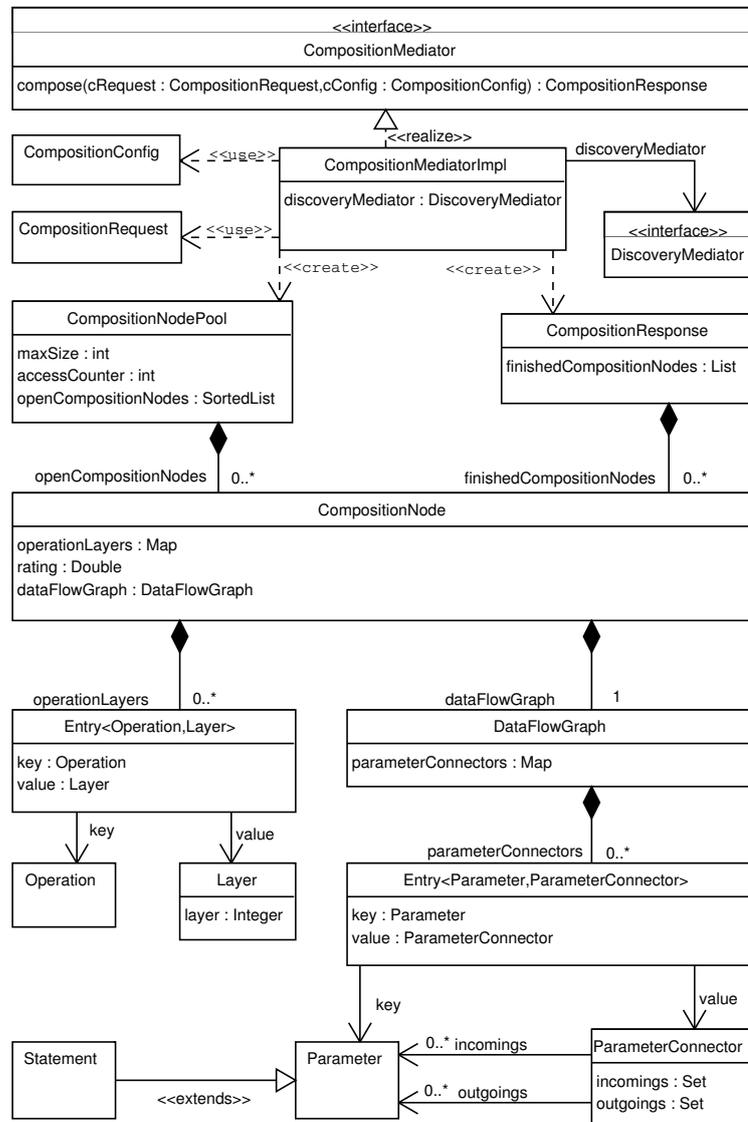


Figure 8.11: UML diagram of the composition module.

8.4 APPLICATION SCENARIOS

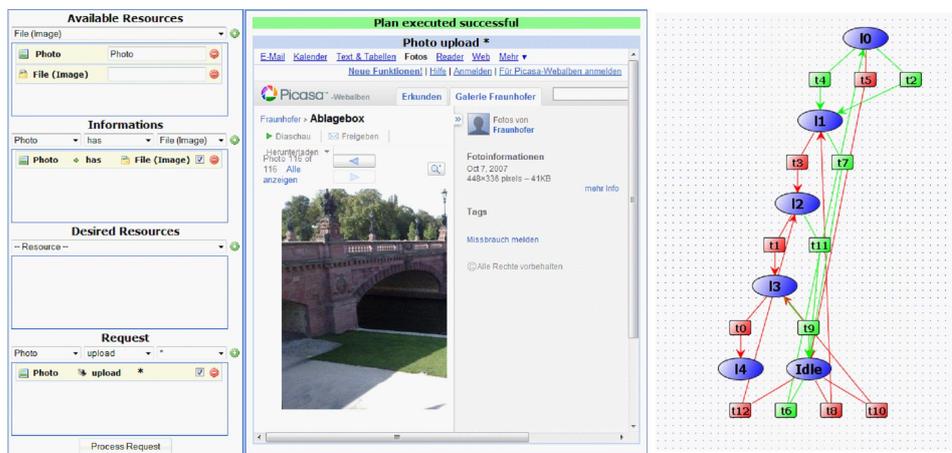
Within this section, three application scenarios for the components introduced above are sketched. The first scenario deals with the creation of a

request pane that can serve as user interface for the automatic creation of Web mashups. While the second scenario focuses on the integration of classic telecommunication services within Web mashups, the third application highlights private data handling, integration of heterogeneous devices, and inter-workflow communication.

8.4.1 A Web Client for Effect Driven Mashup Creation

The algorithm for the automatic creation of service compositions introduced in section 5.3 enables end-users to rapidly create custom mashups in an effect-driven way. Thus, users can specify which effect a mashups should have or generate; the creation algorithm then automatically builds a service composition that meets the users' request. That way, users do not have to specify how an application works, but only how it behaves.

Figure 8.12 shows a browser with an integrated request pane on the left hand side that allows for the specification of a user request and displays the resulting mashup in the remaining space at the righthand side of the Web browser.



In the small example shown in Figure 8.12, a user's photo should be uploaded somewhere so that other users can access it through the Web. On the right side of the figure, the workflow graph of the underlay system automatically created based on the request inserted in the request pane is shown.

8.4.2 *sendAround: Incorporating Telecommunication Services*

Mobile phones expose telecommunication services that generally cannot be found on other portable user-devices. For instance, a mobile phone enables users to set up a call to one or more other users, to send short text messages or messages containing images and videos. With current technologies, those features cannot be easily integrated into Web mashups, since they reside on different devices than the one executing and rendering the mashup itself, e.g. the user's netbook.

The *sendAround* application lets users search for photos, videos, events, and special events within a certain vicinity and communicate the results to other users through classic telecommunication services. A screenshot of the application is illustrated in Figure 8.13. Assume user looks for a restaurant in Berlin. With the *sendAround* application, the user may search for places and events in Berlin, which are then drawn on a map. By clicking on the markers on the map, a dialog appears that lets the user transmit the selected content by means of telecommunication services residing on its mobile phone to other users.

If *sendAround* was implemented with classic Web technologies, users would have to use their mobile phone to manually type a message containing the content that was retrieved from the map. For instance, when the user was interested in sending a photo along with the message to show another user an image of the restaurant he or she selected, the user would have to use his or her mobile phone's browser to search for the respective image and manually attach it to the text message.

However, *sendAround* was implemented with the markup language which incorporates services abstractly and thus works as follows. All resources within the mashup were described by abstract mashup tags, e.g. Mashup Markup, thus, there is a `<map>`, a `<photo>`, a `<video>`, a `<place>` and an `<event>` tag to integrate the respective resources via underlying services. When the user's profile contains information on the user's preferences, the application is automatically tailored accordingly. For instance, Google maps can be dynamically integrated instead of other alternatives for maps in the event that the user prefers the look-and-feel of Google. Note that different users may get a functionally equivalent application with different services

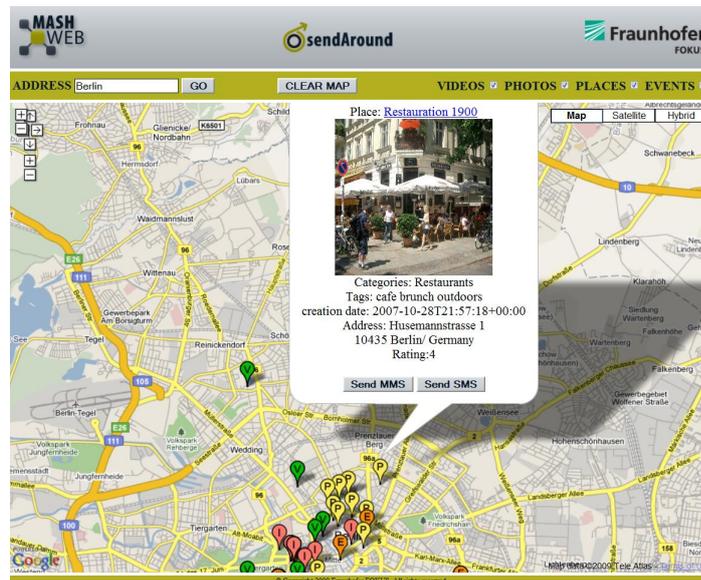


Figure 8.13: *sendAround* Demo: Sending Places around that surround you.

included, although the code written by the software developer was exactly the same.

To include a telecommunication service, the software developer has made use of the `<message>` tag to integrate a SMS and MMS service. When the user called the mashup, this tag was substituted dynamically by the SMS and MMS services running at the user's mobile phone. Note that this kind of service integration is impossible during the designtime of an application, since it is of utmost importance that the services are integrated with the user's mobile phone (which cannot be known at designtime).

After the successful integration of a SMS and MMS service, two buttons "Send SMS" and "Send MMS" are displayed in the balloon tip of the restaurant as shown in Figure 8.13. These buttons correspond to a selected presentation mapping of the respective resource. By clicking on the "Send MMS" button, a dialog opens where the user can specify the receiver of the message. The message body contains the description found in the balloon tip together with the image of the restaurant. When sent, the message is transmitted from the proxy to the user's mobile phone. Here, the client-side runtime engine running within the user's Web browser takes the data as input for the MMS service, which transmits the message, signals its transmission to the proxy-side runtime environment, and moves back into an idle state.

Note that this procedure not only allows the dynamic integration of telecommunication services, but also provides a mechanism to pass any kind

of content dynamically between multiple end-user devices and a mashup server. Modifications at the single user devices are not needed; a JavaScript engine within the devices' browser that executes the dynamically deployed runtime is sufficient.

8.4.3 ScatterPoker: A Distributed Interactive Community Web Application

ScatterPoker is a distributed Poker application based on several MashWeb components; a screenshot is given in Figure 8.14.



Figure 8.14: ScatterPoker Table

ScatterPoker allows multiple users who are possibly at different locations to play a digital poker game against each other. The setting is as follows. Assume that six users want to play poker against each other, where three friends are sitting together in a living room in Berlin while the others are sitting in New York. Information within poker games can be abstractly divided into two types, namely common information such as playing cards shared by all users, the money each user possesses, the current size of the pot and so forth. At the same time, there is some information that remains private, i.e. the cards each user holds. This setting requires the handling of private data as motivated in section 3.4, i.e. some data has to be processed by

local services from the single users. Here, each user has a personal device such as a smartphone, PDA or netbook which handles and displays the private data for each user. The large screen in the living room is then used to display shared content; Figure 8.14 shows the information viewable by all users. The application moreover integrates a location service that determines the current position of the users. Whether this location is provided by a Web service or by a local service on the user devices accessing a local GPS module is determined during runtime, i.e. when the abstract location service is replaced by a concrete service. Since the three users in Berlin cannot see the other players staying in New York, the local cameras of the user devices are integrated to show a video stream of the other users at the left hand side of the playing table.

The underlay system for ScatterPoker was directly engineered. Here, the graphical user interface of the UPPAAL model checker [191] was used to draw the workflow graphs required for the definition of the underlay system directly. A transformation from UPPAAL to JavaScript and JSON was implemented, so that the drawn graphs can be directly exported to underlay systems expressed as JSON objects with functions and guards implemented in JavaScript. In addition to the user-friendly interface, UPPAAL provides an easy way to formally verify the correctness of the developed applications, i.e. software developers can guarantee that the underlay system is working properly.

For ScatterPoker, two communicating workflow graphs have been designed. The graph controlling the functionality on the table is shown in Figure 8.15, while the graph running on the users' mobile devices is illustrated in Figure 8.16.

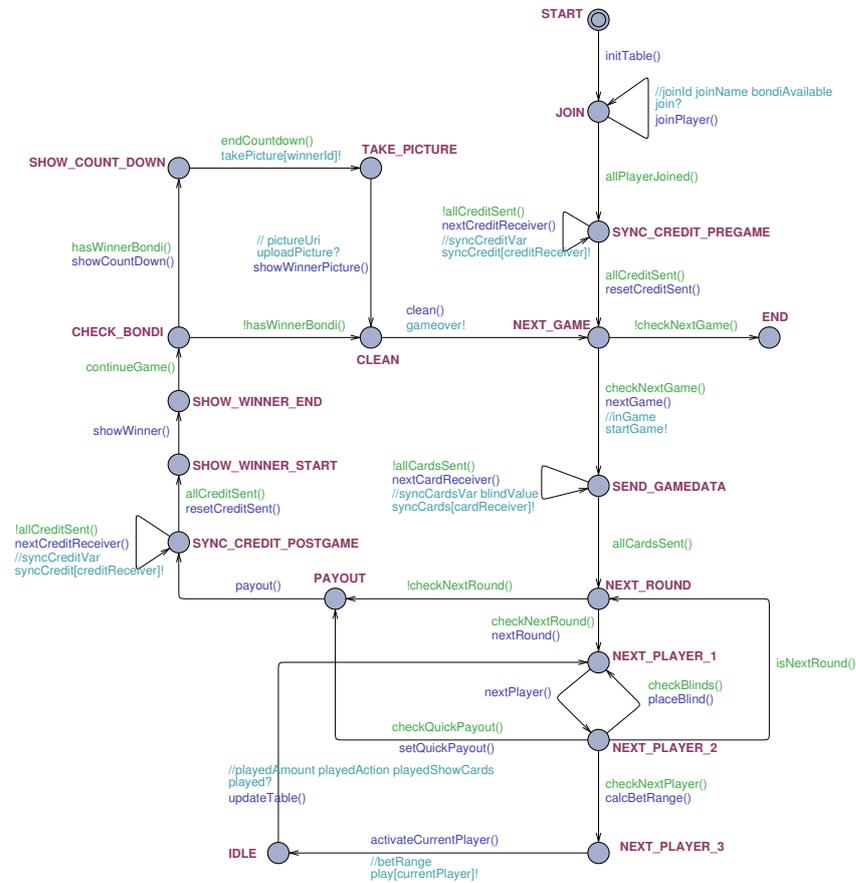


Figure 8.15: ScatterPoker: Workflow graph of the underlay system running on a community device such as a TV screen.

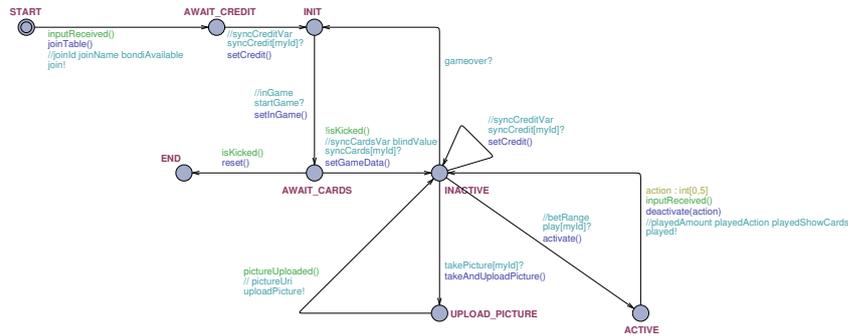


Figure 8.16: ScatterPoker: Workflow graph of the underlay system running on mobile user devices.

CHAPTER 9

CONCLUSION AND FUTURE PROSPECT

9.1 CONCLUSION

In the course of thesis, an underlay system for Web mashups has been presented that leverages access to services residing on multiple user devices, combining their capabilities within value-added Web applications. The underlay system bridges the gap between classic service composition languages and Web applications characterized by a high degree of user interactivity and a rich presentation. A resource-oriented view on services enables the mapping of the underlay's current state to a Web mashup's presentation.

The abstraction of services beyond resources also paves the way towards a dynamic integration of services based on their functionality rather than on their interface descriptions. This abstraction enables the dynamic integration of services residing on multiple possibly highly heterogeneous devices into an underlay system.

Two central algorithms have been discussed within the scope of underlay systems. First, an algorithm for the automatic creation of underlay systems has been presented that supports the creation of underlay systems based on a given user request. In this case, the request encompasses a set of effects and outputs that the resulting mashup should generate. Since the automatic creation of service compositions is complex and time consuming, special emphasis was put on the optimization of the composition algorithm with regard to the maximum amount of time the user is willing to wait. It has been shown that the development of multiple service compositions in parallel, in which the number of active partial service compositions is dynamically adapted over time, can considerably enhance the chances of finding a solution for a given request. Especially if no additional knowledge about the service domain is available, it has been proven that a logarithmic decrease in the number of service compositions developed in parallel decouples the algorithm's dependencies from the given service domain considerably.

Second, an algorithm was presented that distributes an underlay system among multiple devices, so that each device holds a part of the underlay system that is in control of the locally available services. During the partitioning

process, the single partitions are extended by communication structures that ensure the coherent execution of the underlay system independent from its distribution.

All components required for the realization of automatically producible and distributable underlay systems have been specified, together with their respective communication primitives. In that direction, the Orchestration Synchronization Protocol (OSP) has been introduced, which mediates between the single parts of a distributed underlay system. The OSP is responsible for both the negotiation between the available devices for the allocation of concrete services to abstract services and the coherent execution of the single partitions of the underlay system.

Finally, the specification of the underlay system has been grounded in the domain of the Web. Here, a substitution for the abstractly introduced concepts is given, paving the way towards the implementation of an underlay system for dynamic Web Mashups. Application scenarios demonstrate the feasibility of the realization.

9.2 FUTURE WORK

There are multiple opportunities for expanding the outcomes presented so far.

9.2.1 Establishment of the MashWeb JavaScript API

The MashWeb API has been made accessible to registered users at <http://www.mylab.fokus.fraunhofer.de/mashweb/demo> via API keys. To rate the acceptance of the API and evaluate the categorization and abstract functionalities in more real-world scenarios, the API is planned to be opened to Web developers for field trials and will then be adjusted based on the gathered feedback.

9.2.2 Client to Client Communication

The need to support unmodified clients led to a proxy centric communication between clients, i.e. client-to-client communication would always be mediated by the proxy, since the classic Web architecture does not support direct browser to browser communication. Thereby, the offline capabilities of applications are restricted to single devices.

The evaluation of methods for the communication between multiple user devices is part of ongoing work.

9.2.3 Integration with sophisticated Policy Mechanisms

For now, the OSP's negotiation phase is driven by the user's preferences with regard to the application's offline capabilities. A detailed discussion about employing additional knowledge to enhance the decision-making process of the OSP's negotiation phase was considered to be out of the scope of this thesis. Nevertheless, promising steps towards the consideration of services' quality were undertaken and are worth continuing [146, 100].

In addition, linking the allocation of services with policy systems can be beneficial, especially when integrating services from within managed networks such as the telecommunications domain, in which service access is commonly restricted by the service provider. Here, a policy decision point (PDP) holding policies for access to certain services can be used to grant access to such telecommunication services, where the OSP's negotiation phase would act as the corresponding Policy Enforcement Point (PEP). A possible policy model with a focus on the exposure of telecommunication services to the Web has been introduced by Blum et al. in [25].

9.2.4 Services Accessing Multiple Resources

For now, the service model assumes that the service operates on a single resource, i.e. only modifies the attributes of one resource. It should be evaluated whether an extension towards services operating on multiple resources would be advantageous. It certainly provides less restriction on services and thereby eases the classification of services. Thus, it would allow a more coarse-grained view on services. However, the classification of services can become too complex. A deeper evaluation would be required to identify the number of services suffering from the current restrictions.

9.2.5 Measuring the Web's Service Domain

In order to better adapt the automatic service composition algorithm to the domain of the Web, a measurement of the current service and API landscape should be undertaken. Here, `programmableweb.com` can serve as central repository for open Web APIs. At this point in time, there are approximately 1500 APIs listed. However, an API such as the Google Maps API encompasses more than 90 methods, which have to be considered as single services within the scope of the given underlay system. A script can be implemented that runs over all listed APIs and extracts the number of contained methods. For each method, the number of input parameters and number of directly

modified object can be derived. Moreover, a realistic number of different data types can be gathered. This measurement of the Web's service domain can serve as an input for the automatic underlay creation mechanism and compare the current user-centric algorithm's performance with a targeted algorithm's performance that builds upon realistic domain knowledge.

9.2.6 Offline Capabilities through Caching

Approaches to enhancing applications' offline capabilities by caching Web content at the client side already exist. The most prominent example is Google Gears [68], which is tailored for Google's Web application and is made available as browser plug-in. A promising research area encompasses the combination of the offline capabilities provided by local services and the estimation of data required by services within a certain future time span. That way, input data for local services can possibly be derived in advance and ensure a greater depth of the client's offline capabilities.

9.2.7 Instantiation of the Underlay System within the Domain of SOA

Enterprise mashups, i.e. mashups tailored to support enterprises and enterprise interoperability, are becoming more important recently [65]. Therefore, the underlay system is currently evaluated with regard to its instantiation within the scope of the SOA domain, where the underlay can facilitate access to business processes via automatically generated user interfaces. Since business processes commonly constitute large-scale processes, the application's view has to be restricted based on a user's role. For instance, users with different access rights to a certain part of the application should be provided with different views and other means to interact with the applications.

In 2007, a specification that aims at the integration of user interaction into BPEL in a reverse engineering style and is referred to as BPEL4People was published. However, BPEL4People only extends BPEL through a task for the processing of user inputs and does not cope with the actual presentation of the service composition [3, 2].

The mapping between the underlay system's workflow and dataflow to BPEL is a matter of engineering. Note that especially the workflow was defined in a very general and mathematical way to simplify its instantiation with multiple different workflow languages. Two key problems can be identified. First, the timeout mechanism has to be re-modeled through BPEL's *wait*, *terminate*, and *empty* activities, which trigger the replacement of a service. There is already some research on recovery processes for BPEL [119].

Second, the achievement of resource orientation is also challenging, since there are considerably more different resources available within the scope of business process than on the Web. Fortunately, the resources that require a presentation should remain limited, and similar to the ones identified within the scope of the Web. However, the mapping will become more complex. Especially with regard to business processes spanning multiple enterprises, challenges, in terms of semantic interoperability, will constitute the main obstacle [43]. While the problem of semantic interoperability is still discussed, most solutions focus on special closed application domains [162, 83, 20].

APPENDIX A

A RESOURCE-ORIENTED MARKUP LANGUAGE FOR WEB MASHUPS

In this section, an HTML-based mashup language is introduced and specified. Section A.1 outlines the motivation and discusses the objectives of the mashup language, while section A.2 deals with its specification. Section A.3 presents the validation of created markup code beyond the correctness of the XML structure itself.

A.1 MOTIVATION AND OBJECTIVES OF THE MASHUP LANGUAGE

The common method of creating Web pages is resource oriented. However, in the relatively new kind of composed Web applications, i.e. mashups, this notion cannot be followed. The remote services that mashups integrate into one application need to be embedded in a function-oriented fashion via custom application code.

This section introduces an abstract mashup language and a description of the necessary underlying architecture components. The mashup language abstracts from services and introduces a new set of resources to easily create mashups. While the resources are represented by HTML-like tags, the integration of data from multiple services is represented by the tree-structure of these tags. The resulting advantages are resource abstraction and rapid mashup creation.

Derived from the current architecture and the current approach to creating mashups, including the resulting complexity, two major objectives for a mashup language emerge.

First, the language should provide a way to create mashups in a resource-oriented manner. This essentially means that the language should be the foundation of abstraction from actual services. An important aspect that has to be considered during the specification is the different nature of APIs. While a large part of available APIs are accessible either by JavaScript or server-side code, several APIs are based solely on JavaScript. These are usually run client-sided and involve heavy user interaction. Respective APIs

include most map APIs such as Google Maps [70] or Yahoo! Maps. Eventually, the language needs to provide placeholders for services which developers can use to implement mashups independently of the nature of the underlying APIs.

The second key objective is providing a way to easily mash services. Mashups integrate data, i.e. the output of several services is mixed and reused [115]. The language needs to reflect this and needs to be kept simple at the same time. Although rather simple compared to service compositions within SOA, the integration of external services into mashups requires programming skills that go well beyond basic HTML knowledge. Therefore, the language should provide developers with low programming skills and the means to integrate at least basic functionality from external services. The language should not necessarily require scripts when used in combination with HTML. Return data of services should be automatically and seamlessly integrated into the Website, just like audio and video data is with the newly introduced `audio` and `video` tags of HTML 5 [198].

On the other hand, the language should integrate well with more advanced technologies and programming languages in order to serve the needs of more skilled Web developers. Thus, it should offer possibilities to access and manipulate the return data before it is integrated into the HTML passed to the client's browser.

A.2 LANGUAGE SPECIFICATION

An example and fragments of an extended Backus-Naur Form (EBNF) developed based on the XML specification [197] are used to explain the syntax and semantics of the language.

In the mashup language, services providing the same resources are grouped together. Currently, 16 categories are defined; a list of identified resources can be found in appendix B. The language provides a universal set of tags that represent these resources, their related actions and input data. The markup is embedded into valid HTML documents and therefore appears to be an extension of them¹. It provides the desired simplicity and ease of use since Web developers are familiar with its syntax without depending on any specific programming language. The presentation possibilities are not affected because the mashup markup is substituted with valid HTML, CSS

¹The mashup markup is an XML-based language that seeks to ease the development of mashups and was designed to extend HTML 5. Since XHTML 1.1 allows modular extensions [196, 208] the embedding of mashup markup in current XHTML is also possible. For reasons of simplicity, the term HTML is used to refer to all versions of HTML and XHTML that are in use in the Web today when no version is explicitly named.

and JavaScript that is then seamlessly integrated into the HTML document. Thus, the rich interaction possibilities that the combination of HTML, CSS and JavaScript provide remain possible to use.

In order to create a defined and self-contained part in an HTML document that contains mashup markup, each mashup part is embedded in a `<mashup>` tag. This forms the root-tag of the mashup as shown in the following EBNF:

```
MashupElement ::= '<mashup' [S 'id' Eq '"' (Letter)+ '"'] '>'
(MElement)* '</mashup>'
```

Since the syntax is XML-based, two basic elements exist: empty elements and elements that contain content, starting with a start tag and ending with an end tag. The EBNF below specifies empty tags and tags that may contain content.

```
MElement ::= MEmptyElement | MTag MContent MTag
```

The services are classified in distinct service categories, named resources. These are used in the mashup language to realize the abstraction from actual services. Each tag in the mashup markup represents one resource.

```
MEmptyElement ::= '<' MResourceS S?'/>'
MSTag ::= '<' MResourceS S? '>'
METag ::= '</' MResourceE S? '>'
MResourceS ::= Mmap | Mlocation | Mphoto
MResourceE ::= 'map' | 'location' | 'photo'
```

Each resource can have one of multiple methods, referred to as actions, and a set of attributes that may be mandatory or optional. Thus, every resource has an `action` attribute used to set the action that is defined as a placeholder in the mashup markup just as resources are. In addition to the `action` attribute, additional attributes can be defined for either mandatory or optional input. During a binding process, the triple of resource, action and attributes are dissolved into actual service calls.

Below, the EBNF for a map resource is shown with one example action `center`. `centerMapAtr` refers to attributes that are specifically defined for the resource map and its `center` action.

```
Mmap ::= 'map' (MapAction)? (IDAtr)?
MapAction ::= centerMap | unknownAction
centerMap ::= S 'action' Eq '"'center'"' centerMapAtr
```

```
centerMapAtr ::= S 'address' Eq ('''(Letter)+''') | subsAtr )
```

Every resource as well as the `< mashup >` tag can optionally define an `id` whose value identifies the element unambiguously. If used more than once in a document and for resources of the same type, it references the identical element. Thus, the `id` attribute permits the simple reuse of a resource in the document. If a resource is used more than once, the action and mandatory attributes can be omitted on the second occurrence.

In the following EBNF `IDAtr` refers to the `id` attribute.

```
IDAtr ::= S 'id' Eq ''' (Letter)+ '''
```

Up to now, the language has provided the means to abstract from actual services. Service calls are represented by a triple of resource, action and attributes, while these can be mandatory or optional.

Based on this, a simple example for the use of mashup markup can be introduced. In listing A.1, the inclusion of a map is shown in mashup markup. A `< mashup >` tag with the identifier `usecase` represents the start of the mashup markup part. Within this mashup, a map resource represented by the tag `< map >` is used.

```
1 < mashup id="usecase" >
2   < map action="create" id="map_canvas "
3     width="720" height="500" >
4   < /map >
5 < / mashup >
```

Listing A.1: Basis of a use-case that is implemented with Mashup Markup.

A.2.1 Integrating Hybrid Content

So far, the specification of the mashup markup does not address data integration, although this is major characteristic of mashups. In order to allow the development of mashups with mashup markup, the easy transformation of the output data of one resource as input data for another resource is a central construct of the language. For the integration of resources a tree structure provides an intuitive way to define the dataflow between multiple elements of the mashup language. This is defined with regards to the HTML paradigm to group-nested elements which are for example described in [198, section 4.9]. Just like the table element in HTML can contain a number of children that alter the table's structure or representation, each resource's children have effects on it.

In the mashup markup, the tree is dissolved from the leaf nodes to the root

node. The output data from the leaf node is used as input for their parent nodes, thus providing a direct mapping from output to input data.

As shown in the EBNF below, the content of each tag may be empty, plain text, other HTML elements or other mashup elements.

```
MContent ::= (element | PCData | Reference | CDsect | PI | Comment | MElement)*
```

The specifications for nested resources provide the means to truly create mashups with the language. In Listing A.2, the resources for photos and the user's location are nested into the `<map>` tag, implying that the resource "location" is used as input for the creation of the resource "map".

```
1 <mashup id="usecase">
2   <map action="create" id="map_canvas"
3     width="720" height="500">
4     <location action="getCurrentPosition"
5       id="user_position" />
6   </map>
7
8   <map action="addMarker" id="map_canvas"
9     html="You are here">
10    <location id="user_position" />
11  </map>
12
13  <map action="addMarker" id="map_canvas">
14    <photo action="searchPhotos"
15      id="photo_collection">
16      <location id="user_position" />
17    </photo>
18  </map>
19 </mashup>
```

Listing A.2: Use-case implemented with Mashup Markup.

A.2.2 From Resource Integration to Service Composition

To further ease the use of the mashup markup, attributes of a resource can be substituted with the output of embedded elements. This is realized as follows: the output of nested resources, which matches the compulsory attributes of a parent element but is omitted there, is automatically included. If output that corresponds to optional attributes should be included in the parent element, this has to be stated explicitly for each attribute in the outer resource with the special attribute value '###'. Output from embedded elements that is neither needed as input for compulsory attributes of their

parent element nor marked to be included is disregarded. The EBNF below introduces the special value '###':

```
subsAttr ::= '###'
```

In order to run the example, the mashup markup needs to be integrated into a valid HTML document. Evidently, modifications to the known mashup architecture must be made, and will be described in the next section. Eventually, the mashup markup is substituted with HTML, CSS and JavaScript during runtime and a valid document is passed to the client's browser.

A.2.3 *Dynamic Replacement of Services*

The workflow graphs of the underlay system introduced in section 4.3 are based on timed automata and thus provide a natural means for the expression of realtime behavior within the the underlay system itself. The markup language defines an optional attribute that allows the automatic generation of a workflow substructure, which deals with the dynamic replacement of services when they do not respond within a given timespan.

```
timeout ::= (Number)
```

The attribute *timeout* can be included within every resource tag. When a timeout attribute is detected during the parsing of the markup, a timeout construct is appended to the locations of the workflow graph that have outgoing transitions labeled with actions that consume at least one of the outputs of the service protected by the timeout attribute. The timeout construct itself has already been introduced in section 4.3.5.

For example, assume that the current location of the user may be provided by either a GPS module on the user's smartphone or a Web service that derives the user's location through his or her current WLAN cell. When the Web service is integrated within the current mashup and the user goes offline, the Web service is no longer able to respond. The GPS device should overtake the duty of the Web service seamlessly.

In such cases, the user may include the *timeout* attribute into the `<location>` tag, where the number defines the time span in milliseconds that the service is granted to finish its computation.

A.3 MASHUP MARKUP VALIDATION

In this section, the algorithm that is used in the mashup markup validator is shown and explained subsequently with the help of pseudocode.

First, as shown in Algorithm A.1, the complete document, at this time consisting potentially of HTML 5, XHTML 5, XHTML 1.x, client-side scripts, server-side scripts and mashup markup, is checked for XML syntax errors (line 2). This can be done by any available XML processor, e.g. Xerces [207], since the mashup markup is XML-based and a document that contains valid mashup markup is fully XML compliant.

The semantics, however, cannot be validated by common XML processors. Therefore, the mashup markup validator is necessary. For the validation, the mashup markup is extracted (line 3) and processed as a tree, which is common in XML processing [27, 210].

Algorithm A.1 Pseudocode for validation of XML compliance of code.

```

1: procedure VALIDATEMARKUP(Markup)
2:   if document.validXML then
3:     tree : markup = EXTRACTMASHUPMARKUP()
4:     array : validNodes
5:     array : invalidNodes
6:     POSTORDER(tree.rootnode)
7:   else
8:     ABORT()
9:   end if
10: end procedure

```

One of the central concepts of the mashup markup is the reflection of the data integration of different services. Therefore, the output of inner elements is mapped to the input of outer elements, i.e. elements in outer resources can be omitted. Moreover, optional attributes can be marked to be substituted by the output of inner elements as well. In order to check each resource for completeness, the tree of elements needs to be resolved from the innermost elements outwards. This traversal algorithm is well-known as post-order traversal [49]. In the post-order traversal algorithm, the subtrees are followed from left to right and eventually the root node is processed. In Algorithm A.2, the post-order traversal through the mashup markup is shown in pseudocode.

In line 2 and 3 of Algorithm A.2, the currently processed node is checked for children. If any children are found, the post-order traversal function is called recursively for the leftmost child node. If either all children are processed or none exist, i.e. the node is a leaf, the node itself is processed. The

Algorithm A.2 Pseudocode for post-order traversal through Mashup Markup tree.

```

1: procedure POSTORDER(node)
2:   if node.hasUncheckedChildren then POSTORDER(node.children.leftmost)
3:   else
4:     if node.resourceID.is('mashup') then RECHECKINVALID()
5:     else
6:       if node.resourceID.invalid then ABORT()
7:       else
8:         if node.action.inexistent then
9:           if node.ID.exists then
10:            invalidNodes << node
11:           else ABORT()
12:           end if
13:         else
14:           if action.invalid then ABORT()
15:           else
16:             if validateManInputData(node) ^
17:               validateOptInputData(node) then
18:                 validNodes << node
19:               else
20:                 invalidNodes << node
21:               end if POSTORDER(node.parent)
22:             end if
23:           end if
24:         end if
25:       end if
26:   end procedure

```

complete tree has been traversed once when the node itself is the `<mashup>` node. Nonetheless, as reflected in line 6, the tree has to be traversed several times due to the possible references within the code.

If the node is not the `<mashup>` element, its components are checked subsequently. First, the element's resource type is validated (line 9). This means that the mashup markup tag has to be defined. If the resource stated in the mashup markup does not match any defined tags, it is not supported and therefore invalid: the mashup validation fails (line 10, 11). If the resource is defined, the validation process continues.

Subsequently, the existence of an `action` attribute is checked (line 14). When it does not exist, the node is checked for the existence of an `id` (line 16). If

an `id` exists, it is possible that the node refers to another resource which is valid. In this case, the node is buffered (line 17). The buffering is necessary because the failed nodes need to be rechecked to be able to dissolve references after the complete XML tree has been traversed once. This is reflected in Listing A.5.

If no `id` is found, the validation is aborted (lines 18-20). Each resource needs either a valid triple of resource, action and attributes, or a valid tuple of resource and `id` that refers to another valid resource.

When an `action` attribute is identified, this is validated in line 24. Again, as for the resource type, a mashup markup definition is utilized. If the action is not available for the resource, the validation is aborted (lines 25, 26).

If the action is valid, the mandatory and optional input data is validated in two separate functions shown in Algorithms A.3, A.4 (lines 29 - 32).

If the input data is valid and thus the tag is valid, it needs to be buffered for the rechecking of referencing nodes (line 33). There are three reasons for doing this: first, to allow the data integration, the output of subtrees needs to be available to validate their root. Second, referencing resource instances need to be resolved eventually. Third, actions can depend on another action to be executed first. This is handled by the service execution engine. Here, however, just the possible existence of a workflow that provides all necessary input data needs to be ensured.

If the check of the input data is invalid, the node is buffered for later rechecking (line 35).

Independent of the check of the input data, the post-order traversal is recursively called for the parent node. Eventually, after the complete tree has been traversed once, it is rechecked for possible references or substituted input data (line 6). The function *recheckInvalid()* is shown in Algorithm A.5.

In Algorithm A.3, the validation of the input data is shown. Each input element defined as necessary (lines 2, 3) is handled according to its type. If the input element is an attribute, the node is searched for this attribute (lines 4 - 6). If it exists, the next input element is checked (line 7). If it is not found in the node itself, its children are checked (line 10) and if it does exist, the next input element is checked (line 11). If the input element is not found, the input data validation is aborted (line 14).

Input elements that are not attributes, but rather the tag's content, are handled in analogy (lines 17 - 28).

Algorithm A.4 shows the validation of optional input data. The flow to validate optional input data is fundamentally different from the flow to validate mandatory input data. Instead of searching each mandatory element defined in the mashup markup definition in the code at hand, the definition is searched for those attributes stated within the code (lines 3 - 7). If the

Algorithm A.3 Pseudocode for the validation of the mandatory input data.

```

1: procedure VALIDATEMANDATORYINPUTDATA(node)
2:   while TagDefinition.MandatoryInput != NULL do
3:     if MandatoryInput.isAttribute? then           ▷ Search atr in node
4:       if node.contains?(MandatoryInput) then NEXT()
5:     else
6:       if node.children.contains?(MandatoryInput) then NEXT()
7:       else ABORT("Mandatoryinputmissing")
8:       end if
9:     end if
10:    else           ▷ Check node for content
11:      if node.content.empty? then           ▷ Search subtree for content
12:        if subtree.empty? then ABORT("Mandatoryinputmissing")
13:        else NEXT()
14:        end if
15:      end if
16:    end if NEXT()
17:  end while return true
18: end procedure

```

Algorithm A.4 Pseudocode for the validation of the optional input data.

```

1: procedure VALIDATEOPTIONALINPUTDATA(node)
2:   while node.OptionalInput != NULL do
3:     if OptionalInput.isAttribute? then           ▷ Check if input is usable
4:       if TagDefinition.OptionalInput.isIn?OptionalInput then
NEXT()
5:       else WARNING("Dataofattributenotusable") NEXT()
6:       end if
7:     else           ▷ Check if tag content is usable
8:       if TagDefinition.OptionalInput.contentUsable? then NEXT()
9:       else WARNING("Tagcontentnotusable") NEXT()
10:      end if
11:    end if
12:  end while
13: end procedure

```

additional input is usable, the next element is checked (line 8). If not, a warning is thrown (line 9 - 11) and the next optional input element is checked (line 12). Again, tag content is handled analogous to attributes (line 15 - 23).

In Algorithm A.5, all resources that failed during the validation of their input data but have an *id* are re-considered. For every failed resource, the buffer with valid resources is checked (line 9). If a valid resource with the

Algorithm A.5 Pseudocode for rechecking of the tags.

```

1: procedure RECHECKVALID
2:   if invalidNodes.empty? then EXIT()
3:   else
4:     if invalidNodes.noChanges then ABORT()
5:     else
6:       if validNodes.contains?(invalidNodes.first) then
7:         if inputData.valid? then
8:           invalidNodes.delete(node)
9:           validNodes << node RECHECKINVALID()
10:        else
11:          node.updateInputdata RECHECKINVALID()
12:        end if
13:      else
14:        invalidNodes.pushFirstToLast RECHECKINVALID()
15:      end if
16:    end if
17:  end if
18: end procedure

```

same *id* exists, the output is considered as input data for the corresponding failed resource (line 12). Possibly, the dissolved triple provides the needed input data for another failed triple. Therefore, the tree has to be traversed once for every dissolved tuple of resource and *id*. Each dissolved tag is deleted from the list of invalid nodes and added to the one containing the valid ones (lines 13, 14).

The rechecking and thus the validation process ends when either all invalid nodes are dissolved or no more changes can be made (lines 2 - 7).

APPENDIX B

RESOURCE DEFINITIONS

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tagdef xmlns="http://www.fokus.fhg.de/mashweb/tagdef/v3"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xsi:schemaLocation="http://www.fokus.fhg.de/mashweb/tagdef
5 /v3 ../../spec/v31.xsd">
6 <tag name="map">
7 <action name="create" type="constructor">
8 <request>
9 <requestParam name="id"/>
10 <requestParam name="lat" type="float" required="true"/>
11 <requestParam name="lng" type="float" required="true"/>
12 <requestParam name="width" type="integer"
13 required="true"/>
14 <requestParam name="height" type="integer"
15 required="true"/>
16 <requestParam name="type" def="street"/>
17 <requestParam name="zoomlevel" type="integer"/>
18 <requestParam name="maptypeControl" type="boolean"
19 def="false"/>
20 <requestParam name="panControl" type="boolean"
21 def="false"/>
22 <requestParam name="zoomControl" type="boolean"
23 def="false"/>
24 <requestParam name="dragableMap" type="boolean"
25 def="false"/>
26 <requestParam name="zoomOnDoubleClick" type="boolean"
27 def="false"/>
28 <requestParam name="continousZoom" type="boolean"
29 def="false"/>
30 <requestParam name="scrollwheelZoom" type="boolean"
31 def="false"/>
32 </request>
33 <response>
34 <responseParam name="uri" required="true"/>
35 </response>
36 </action>
37 <action name="center">
38 <request>
39 <requestParam name="id" required="true"/>
```

```
40 <requestParam name="lat" type="float" required="true"/>
41 <requestParam name="lng" type="float" required="true"/>
42 <requestParam name="pan" type="boolean" def="false"/>
43 </request>
44 <response>
45 <responseParam name="uri" required="true"/>
46 </response>
47 </action>
48 <action name="setZoomlevel">
49 <request>
50 <requestParam name="id" required="true"/>
51 <requestParam name="zoomlevel" type="integer" def="12"
52 <requestParam name="zoomlevel" type="integer" def="12"
53 </request>
54 <response>
55 <responseParam name="uri" required="true"/>
56 </response>
57 </action>
58 <action name="addMarker">
59 <request>
60 <requestParam name="id" required="true"/>
61 <requestParam name="markerId"/>
62 <requestParam name="lat" type="float" required="true"/>
63 <requestParam name="lng" type="float" required="true"/>
64 <requestParam name="html"/>
65 <requestParam name="icon" type="uri"/>
66 </request>
67 <response>
68 <responseParam name="uri" required="true"/>
69 </response>
70 </action>
71 <action name="updateMarker">
72 <request>
73 <requestParam name="id" required="true"/>
74 <requestParam name="markerId" required="true"/>
75 <requestParam name="lat" type="float" required="true"/>
76 <requestParam name="lng" type="float" required="true"/>
77 <requestParam name="html"/>
78 <requestParam name="icon" type="uri"/>
79 </request>
80 <response>
81 <responseParam name="uri" required="true"/>
82 </response>
83 </action>
84 <action name="removeMarker">
85 <request>
86 <requestParam name="id" required="true"/>
87 <requestParam name="markerId" required="true"/>
88 </request>
```

```
89     <response>
90       <responseParam name="uri" required="true"/>
91     </response>
92   </action>
93   <action name="removeAllMarkers">
94     <request>
95       <requestParam name="id" required="true"/>
96     </request>
97     <response>
98       <responseParam name="uri" required="true"/>
99     </response>
100   </action>
101 </tag>
102 <tag name="location">
103   <action name="getPosition" type="constructor">
104     <request>
105       <requestParam name="address" required="true"/>
106     </request>
107     <response>
108       <responseParam name="lat" required="true" type="double"/>
109       <responseParam name="lng" required="true" type="double"/>
110       <responseParam name="accuracy" type="double"/>
111       <responseParam name="city"/>
112       <responseParam name="country"/>
113       <responseParam name="zipCode"/>
114       <responseParam name="street"/>
115       <responseParam name="nr"/>
116     </response>
117   </action>
118   <action name="getAddress" type="constructor">
119     <request>
120       <requestParam name="lat" required="true" type="float"/>
121       <requestParam name="lng" required="true" type="float"/>
122     </request>
123     <response>
124       <responseParam name="city" required="true"/>
125       <responseParam name="country" required="true"/>
126       <responseParam name="zipCode" required="true"/>
127       <responseParam name="street" required="true"/>
128       <responseParam name="nr" required="true"/>
129     </response>
130   </action>
131   <action name="getPositionByIP" type="constructor">
132     <request>
133       <requestParam name="id"/>
134       <requestParam name="ip"/>
135     </request>
136     <response>
137       <responseParam name="lat" required="true" type="double"/>
```

```
138 <responseParam name="lng" required="true" type="double"/>
139 <responseParam name="accuracy" type="double"/>
140 <responseParam name="city"/>
141 <responseParam name="country"/>
142 <responseParam name="zipCode"/>
143 <responseParam name="street"/>
144 <responseParam name="nr"/>
145 </response>
146 </action>
147 <action name="getCurrentPosition" type="constructor">
148 <request>
149 <requestParam name="id"/>
150 </request>
151 <response>
152 <responseParam name="lat" required="true" type="double"/>
153 <responseParam name="lng" required="true" type="double"/>
154 <responseParam name="accuracy" type="double"/>
155 <responseParam name="city"/>
156 <responseParam name="country"/>
157 <responseParam name="zipCode"/>
158 <responseParam name="street"/>
159 <responseParam name="nr"/>
160 </response>
161 </action>
162 </tag>
163 <tag name="photo">
164 <action name="searchPhotos" type="constructor">
165 <request>
166 <requestParam name="query"/>
167 <requestParam name="tags"/>
168 <requestParam name="uploadDate" type="date"/>
169 <requestParam name="uploadDateRange" type="integer"/>
170 <requestParam name="takenDate" type="date"/>
171 <requestParam name="takenDateRange" type="integer"/>
172 <requestParam name="lat" type="float"/>
173 <requestParam name="lng" type="float"/>
174 <requestParam name="radius" type="float" def="0.5"/>
175 <requestParam name="geocoded" type="boolean"/>
176 <requestParam name="width" type="integer"/>
177 <requestParam name="height" type="integer"/>
178 <requestParam name="album"/>
179 <requestParam name="owner"/>
180 <requestParam name="sort" def="Taken Date Desc"/>
181 <requestParam name="resultSet" type="integer" def="10"/>
182 <requestParam name="offset" type="integer" def="0"/>
183 </request>
184 <response>
185 <responseParam name="photoList">
186 <listElem name="photo" minOccurs="0">
```

```
187     maxOccurs="unbounded">
188     <attribute name="provider"/>
189     <attribute name="title"/>
190     <attribute name="photoId"/>
191     <attribute name="uri" type="uri"/>
192     <attribute name="lat" type="double"/>
193     <attribute name="lng" type="double"/>
194     <attribute name="width" type="integer"/>
195     <attribute name="height" type="integer"/>
196     <attribute name="uploadDate" type="date"/>
197     <attribute name="takenDate" type="date"/>
198     <attribute name="album"/>
199     <attribute name="owner"/>
200 </listElem>
201 </responseParam>
202 </response>
203 </action>
204 <action name="searchGroup" type="constructor">
205 <request>
206 <requestParam name="query"/>
207 </request>
208 <response>
209 <responseParam name="groupId" required="true"/>
210 </response>
211 </action>
212 <action name="getPhotosFromGroup" type="constructor">
213 <request>
214 <requestParam name="groupId"/>
215 </request>
216 <response>
217 <responseParam name="uri" required="true"/>
218 </response>
219 </action>
220 <action name="getPhotoSizes" type="constructor">
221 <request>
222 <requestParam name="photoId"/>
223 </request>
224 <response>
225 <responseParam name="width" required="true">
226 </responseParam>
227 <responseParam name="height" required="false"/>
228 <responseParam name="uri" required="true"/>
229 </response>
230 </action>
231 <action name="getPhoto" type="constructor">
232 <request>
233 <requestParam name="id"/>
234 <requestParam name="photoId" required="true"/>
235 <requestParam name="provider" required="true"/>
```

```
236 </request>
237 <response>
238   <responseParam name="provider" />
239   <responseParam name="title" />
240   <responseParam name="photoId" />
241   <responseParam name="uri" type="uri" />
242   <responseParam name="lat" type="double" />
243   <responseParam name="lng" type="double" />
244   <responseParam name="width" type="integer" />
245   <responseParam name="height" type="integer" />
246   <responseParam name="uploadDate" type="date" />
247   <responseParam name="takenDate" type="date" />
248   <responseParam name="album" />
249   <responseParam name="owner" />
250 </response>
251 </action>
252 <action name="uploadPhoto" type="constructor">
253   <request>
254     <requestParam name="id" />
255     <requestParam name="file" required="true" />
256     <requestParam name="provider" required="true" />
257     <requestParam name="title" />
258     <requestParam name="desc" />
259     <requestParam name="tags" />
260     <requestParam name="permissions" def="private" />
261     <requestParam name="lat" type="float" />
262     <requestParam name="lng" type="float" />
263   </request>
264   <response>
265     <responseParam name="uri" type="uri" />
266   </response>
267 </action>
268 <action name="deletePhoto">
269   <request>
270     <requestParam name="id" />
271     <requestParam name="photoId" required="true" />
272     <requestParam name="provider" required="true" />
273   </request>
274   <response>
275     <responseParam name="dummy" />
276   </response>
277 </action>
278 <action name="updatePhoto" type="constructor">
279   <request>
280     <requestParam name="id" />
281     <requestParam name="photoId" required="true" />
282     <requestParam name="provider" required="true" />
283     <requestParam name="title" />
284     <requestParam name="desc" />
```

```
285     <requestParam name="tags" />
286     <requestParam name="permissions" def="private" />
287     <requestParam name="lat" type="float" />
288     <requestParam name="lng" type="float" />
289 </request>
290 <response>
291     <responseParam name="uri" type="uri" />
292 </response>
293 </action>
294 <action name="commentPhoto" type="constructor">
295     <request>
296         <requestParam name="id" />
297         <requestParam name="photoId" required="true" />
298         <requestParam name="provider" required="true" />
299         <requestParam name="author" required="true" />
300     </request>
301     <response>
302         <responseParam name="uri" type="uri" />
303         <responseParam name="comment" type="string" />
304     </response>
305 </action>
306 <action name="getComments" type="constructor">
307     <request>
308         <requestParam name="id" />
309         <requestParam name="photoId" required="true" />
310         <requestParam name="provider" required="true" />
311         <requestParam name="resultSet" type="integer" def="10" />
312         <requestParam name="offset" type="integer" def="0" />
313         <requestParam name="sort" def="Date Desc" />
314     </request>
315     <response>
316         <responseParam name="commentList">
317             <listElem name="comment" minOccurs="0"
318                 maxOccurs="unbounded">
319                 <attribute name="commentId" />
320                 <attribute name="user" />
321                 <attribute name="email" />
322                 <attribute name="dateCreated" type="date" />
323                 <attribute name="permalink" type="uri" />
324                 <attribute name="comment" type="string" />
325             </listElem>
326         </responseParam>
327     </response>
328 </action>
329 </tag>
330 <tag name="video">
331     <action name="searchVideos" type="constructor">
332         <request>
333             <requestParam name="query" />
```

```
334 <requestParam name="tags"/>
335 <requestParam name="uploadDate" type="date"/>
336 <requestParam name="uploadDateRange" type="integer"/>
337 <requestParam name="takenDate" type="date"/>
338 <requestParam name="takenDateRange" type="integer"/>
339 <requestParam name="lat" type="float"/>
340 <requestParam name="lng" type="float"/>
341 <requestParam name="radius" type="float" def="0.5"/>
342 <requestParam name="geocoded" type="integer"/>
343 <requestParam name="channel"/>
344 <requestParam name="sort" def="Taken Date Desc"/>
345 <requestParam name="resultSet" type="integer" def="10"/>
346 <requestParam name="offset" type="integer" def="0"/>
347 </request>
348 <response>
349 <responseParam name="videoList">
350 <listElem name="video" minOccurs="0"
351 <maxOccurs="unbounded">
352 <attribute name="provider"/>
353 <attribute name="title"/>
354 <attribute name="videoId"/>
355 <attribute name="uri" type="uri"/>
356 <attribute name="lat" type="double"/>
357 <attribute name="lng" type="double"/>
358 <attribute name="width" type="integer"/>
359 <attribute name="height" type="integer"/>
360 <attribute name="uploadDate" type="date"/>
361 <attribute name="takenDate" type="date"/>
362 <attribute name="channel"/>
363 <attribute name="owner"/>
364 <attribute name="length" type="time"/>
365 </listElem>
366 </responseParam>
367 </response>
368 </action>
369 <action name="getVideo" type="constructor">
370 <request>
371 <requestParam name="id"/>
372 <requestParam name="videoId" required="true"/>
373 <requestParam name="provider" required="true"/>
374 <requestParam name="content" def="active"/>
375 </request>
376 <response>
377 <responseParam name="provider"/>
378 <responseParam name="title"/>
379 <responseParam name="videoId"/>
380 <responseParam name="uri" type="uri"/>
381 <responseParam name="lat" type="double"/>
382 <responseParam name="lng" type="double"/>
```

```
383     <responseParam name="width" type="integer"/>
384     <responseParam name="height" type="integer"/>
385     <responseParam name="uploadDate" type="date"/>
386     <responseParam name="takenDate" type="date"/>
387     <responseParam name="channel"/>
388     <responseParam name="owner"/>
389     <responseParam name="length" type="time"/>
390 </response>
391 </action>
392 <action name="uploadVideo" type="constructor">
393     <request>
394         <requestParam name="id"/>
395         <requestParam name="file" required="true"/>
396         <requestParam name="provider" required="true"/>
397         <requestParam name="title"/>
398         <requestParam name="desc"/>
399         <requestParam name="tags"/>
400         <requestParam name="permissions" def="private"/>
401         <requestParam name="lat" type="float"/>
402         <requestParam name="lng" type="float"/>
403     </request>
404     <response>
405         <responseParam name="uri" type="uri"/>
406     </response>
407 </action>
408 <action name="deleteVideo">
409     <request>
410         <requestParam name="id"/>
411         <requestParam name="videoId" required="true"/>
412         <requestParam name="provider" required="true"/>
413     </request>
414     <response>
415         <responseParam name="dummy"/>
416     </response>
417 </action>
418 <action name="updateVideo" type="constructor">
419     <request>
420         <requestParam name="id"/>
421         <requestParam name="videoId" required="true"/>
422         <requestParam name="provider" required="true"/>
423         <requestParam name="title"/>
424         <requestParam name="desc"/>
425         <requestParam name="tags"/>
426         <requestParam name="permissions" def="private"/>
427         <requestParam name="lat" type="float"/>
428         <requestParam name="lng" type="float"/>
429     </request>
430     <response>
431         <responseParam name="uri" type="uri"/>
```

```
432     </response>
433 </action>
434 <action name="commentVideo" type="constructor">
435   <request>
436     <requestParam name="id"/>
437     <requestParam name="videoId" required="true"/>
438     <requestParam name="provider" required="true"/>
439     <requestParam name="author" required="true"/>
440   </request>
441   <response>
442     <responseParam name="uri" type="uri"/>
443   </response>
444 </action>
445 <action name="getComments" type="constructor">
446   <request>
447     <requestParam name="id"/>
448     <requestParam name="videoId" required="true"/>
449     <requestParam name="provider" required="true"/>
450     <requestParam name="resultSet" type="integer" def="10"/>
451     <requestParam name="offset" type="integer" def="0"/>
452     <requestParam name="sort" def="Date Desc"/>
453   </request>
454   <response>
455     <responseParam name="commentList">
456       <listElem name="comment" minOccurs="0"
457         maxOccurs="unbounded">
458         <attribute name="commentId"/>
459         <attribute name="user"/>
460         <attribute name="email"/>
461         <attribute name="dateCreated" type="date"/>
462         <attribute name="permalink" type="uri"/>
463         <attribute name="comment" type="string"/>
464       </listElem>
465     </responseParam>
466   </response>
467 </action>
468 </tag>
469 <tag name="blog">
470   <action name="createPost" type="constructor">
471     <request>
472       <requestParam name="id"/>
473       <requestParam name="provider" required="true"/>
474       <requestParam name="blogId" required="true"/>
475       <requestParam name="title" required="true"/>
476       <requestParam name="content"/>
477       <requestParam name="date" type="date"/>
478       <requestParam name="category"/>
479       <requestParam name="tags"/>
480       <requestParam name="draft" type="boolean" def="false"/>
```

```
481 </request>
482 <response>
483   <responseParam name="uri" required="true"/>
484 </response>
485 </action>
486 <action name="getPosts" type="constructor">
487   <request>
488     <requestParam name="blogId" required="true"/>
489     <requestParam name="provider" required="true"/>
490     <requestParam name="date" type="date"/>
491     <requestParam name="dateRange" type="integer"/>
492     <requestParam name="resultSet" type="integer" def="10"/>
493     <requestParam name="offSet" type="integer" def="0"/>
494   </request>
495   <response>
496     <responseParam name="postList">
497       <listElem name="post" minOccurs="0"
498         maxOccurs="unbounded">
499         <attribute name="postId"/>
500         <attribute name="user"/>
501         <attribute name="email"/>
502         <attribute name="category"/>
503         <attribute name="tags"/>
504         <attribute name="draft" type="boolean"/>
505         <attribute name="dateCreated" type="date"/>
506         <attribute name="permalink" type="uri"/>
507       </listElem>
508     </responseParam>
509   </response>
510 </action>
511 <action name="updatePost" type="constructor">
512   <request>
513     <requestParam name="id"/>
514     <requestParam name="provider" required="true"/>
515     <requestParam name="blogId" required="true"/>
516     <requestParam name="postId" required="true"/>
517     <requestParam name="title"/>
518     <requestParam name="date" type="date"/>
519     <requestParam name="category"/>
520     <requestParam name="tags"/>
521     <requestParam name="draft" type="boolean" def="false"/>
522   </request>
523   <response>
524     <responseParam name="postId"/>
525     <responseParam name="user"/>
526     <responseParam name="email"/>
527     <responseParam name="category"/>
528     <responseParam name="tags"/>
529     <responseParam name="draft" type="boolean"/>
```

```
530     <responseParam name="dateCreated" type="date"/>
531     <responseParam name="permalink" type="uri"/>
532   </response>
533 </action>
534 <action name="deletePost">
535   <request>
536     <requestParam name="id"/>
537     <requestParam name="provider" required="true"/>
538     <requestParam name="blogId" required="true"/>
539     <requestParam name="postId" required="true"/>
540   </request>
541   <response>
542     <responseParam name="dummy"/>
543   </response>
544 </action>
545 <action name="commentPost" type="constructor">
546   <request>
547     <requestParam name="id"/>
548     <requestParam name="provider" required="true"/>
549     <requestParam name="blogId" required="true"/>
550     <requestParam name="postId" required="true"/>
551     <requestParam name="author" required="true"/>
552   </request>
553   <response>
554     <responseParam name="uri" type="uri"/>
555   </response>
556 </action>
557 <action name="getComments" type="constructor">
558   <request>
559     <requestParam name="id"/>
560     <requestParam name="provider" required="true"/>
561     <requestParam name="blogId" required="true"/>
562     <requestParam name="postId" required="true"/>
563     <requestParam name="resultSet" type="integer" def="10"/>
564     <requestParam name="offset" type="integer" def="0"/>
565     <requestParam name="sort" def="Date Desc"/>
566   </request>
567   <response>
568     <responseParam name="commentList">
569       <listElem name="comment" minOccurs="0"
570         maxOccurs="unbounded">
571         <attribute name="commentId"/>
572         <attribute name="user"/>
573         <attribute name="email"/>
574         <attribute name="dateCreated" type="date"/>
575         <attribute name="permalink" type="uri"/>
576         <attribute name="comment" type="string"/>
577       </listElem>
578     </responseParam>
```

```
579     </response>
580   </action>
581 </tag>
582 <tag name="bookmark">
583   <action name="searchBookmarks" type="constructor">
584     <request>
585       <requestParam name="query"/>
586       <requestParam name="tags"/>
587       <requestParam name="date" type="date"/>
588       <requestParam name="dateRange" type="integer"/>
589       <requestParam name="sort" def="Date Desc"/>
590       <requestParam name="resultSet" type="integer" def="10"/>
591       <requestParam name="offset" type="integer" def="0"/>
592     </request>
593     <response>
594       <responseParam name="bookmarkList">
595         <listElem name="bookmark" minOccurs="0"
596           maxOccurs="unbounded">
597           <attribute name="user"/>
598           <attribute name="uri" type="uri"/>
599           <attribute name="screenshot" type="uri"/>
600           <attribute name="description"/>
601           <attribute name="tags"/>
602           <attribute name="dateTimeCreated" type="datetime"/>
603           <attribute name="comment" type="string"/>
604           <attribute name="noOfUsers" type="integer"/>
605         </listElem>
606       </responseParam>
607     </response>
608   </action>
609   <action name="getBookmark" type="constructor">
610     <request>
611       <requestParam name="id"/>
612       <requestParam name="provider" required="true"/>
613       <requestParam name="bookmarkId" required="true"/>
614     </request>
615     <response>
616       <responseParam name="user"/>
617       <responseParam name="uri" type="uri"/>
618       <responseParam name="screenshot" type="uri"/>
619       <responseParam name="description"/>
620       <responseParam name="tags"/>
621       <responseParam name="dateTimeCreated" type="datetime"/>
622       <responseParam name="comment" type="string"/>
623       <responseParam name="noOfUsers" type="integer"/>
624     </response>
625   </action>
626   <action name="createBookmark" type="constructor">
627     <request>
```

```
628 <requestParam name="id"/>
629 <requestParam name="uri" type="uri" required="true"/>
630 <requestParam name="provider" required="true"/>
631 <requestParam name="title"/>
632 <requestParam name="desc"/>
633 <requestParam name="tags"/>
634 <requestParam name="date" type="date"/>
635 </request>
636 <response>
637 <responseParam name="user"/>
638 <responseParam name="uri" type="uri"/>
639 <responseParam name="screenshot" type="uri"/>
640 <responseParam name="description"/>
641 <responseParam name="tags"/>
642 <responseParam name="dateTimeCreated" type="datetime"/>
643 <responseParam name="comment" type="string"/>
644 <responseParam name="noOfUsers" type="integer"/>
645 </response>
646 </action>
647 <action name="deleteBookmark">
648 <request>
649 <requestParam name="id"/>
650 <requestParam name="provider" required="true"/>
651 <requestParam name="bookmarkId" required="true"/>
652 </request>
653 <response>
654 <responseParam name="dummy"/>
655 </response>
656 </action>
657 <action name="updateBookmark" type="constructor">
658 <request>
659 <requestParam name="id"/>
660 <requestParam name="bookmarkId" required="true"/>
661 <requestParam name="uri" type="uri" required="true"/>
662 <requestParam name="provider" required="true"/>
663 <requestParam name="title"/>
664 <requestParam name="desc"/>
665 <requestParam name="tags"/>
666 <requestParam name="date" type="date"/>
667 </request>
668 <response>
669 <responseParam name="user"/>
670 <responseParam name="uri" type="uri"/>
671 <responseParam name="screenshot" type="uri"/>
672 <responseParam name="description"/>
673 <responseParam name="tags"/>
674 <responseParam name="dateTimeCreated" type="datetime"/>
675 <responseParam name="comment" type="string"/>
676 <responseParam name="noOfUsers" type="integer"/>
```

```
677     </response>
678   </action>
679 </tag>
680 <tag name="event">
681   <action name="searchEvents" type="constructor">
682     <request>
683       <requestParam name="query"/>
684       <requestParam name="tags"/>
685       <requestParam name="Date" type="date"/>
686       <requestParam name="DateRange" type="integer"/>
687       <requestParam name="lat" type="float"/>
688       <requestParam name="lng" type="float"/>
689       <requestParam name="radius" type="float" def="0.5"/>
690       <requestParam name="category"/>
691       <requestParam name="venue"/>
692       <requestParam name="sort" def="Date Desc"/>
693       <requestParam name="resultSet" type="integer" def="10"/>
694       <requestParam name="offset" type="integer" def="0"/>
695       <requestParam name="provider"/>
696     </request>
697     <response>
698       <responseParam name="eventList">
699         <listElem name="event" minOccurs="0"
700           maxOccurs="unbounded">
701           <attribute name="eventId"/>
702           <attribute name="uri" type="uri"/>
703           <attribute name="title"/>
704           <attribute name="description"/>
705           <attribute name="tags"/>
706           <attribute name="startTime" type="datetime"/>
707           <attribute name="endTime" type="datetime"/>
708           <attribute name="city"/>
709           <attribute name="country"/>
710           <attribute name="address"/>
711           <attribute name="zipCode"/>
712           <attribute name="lat" type="double"/>
713           <attribute name="lng" type="double"/>
714           <attribute name="created" type="datetime"/>
715           <attribute name="user"/>
716           <attribute name="price" type="double"/>
717           <attribute name="image" type="uri"/>
718           <attribute name="category"/>
719           <attribute name="venue"/>
720           <attribute name="venueId"/>
721         </listElem>
722       </responseParam>
723     </response>
724   </action>
725 <action name="getEvent" type="constructor">
```

```
726 <request>
727   <requestParam name="id"/>
728   <requestParam name="eventId" required="true"/>
729   <requestParam name="provider" required="true"/>
730 </request>
731 <response>
732   <responseParam name="eventId"/>
733   <responseParam name="uri" type="uri"/>
734   <responseParam name="title"/>
735   <responseParam name="description"/>
736   <responseParam name="tags"/>
737   <responseParam name="startTime" type="datetime"/>
738   <responseParam name="endTime" type="datetime"/>
739   <responseParam name="city"/>
740   <responseParam name="country"/>
741   <responseParam name="address"/>
742   <responseParam name="zipCode"/>
743   <responseParam name="lat" type="double"/>
744   <responseParam name="lng" type="double"/>
745   <responseParam name="created" type="datetime"/>
746   <responseParam name="user"/>
747   <responseParam name="price" type="double"/>
748   <responseParam name="image" type="uri"/>
749   <responseParam name="category"/>
750   <responseParam name="venue"/>
751   <responseParam name="venueId"/>
752 </response>
753 </action>
754 <action name="createEvent" type="constructor">
755   <request>
756     <requestParam name="id"/>
757     <requestParam name="title" required="true"/>
758     <requestParam name="startTime" type="datetime"
759       required="true"/>
760     <requestParam name="stopTime" type="datetime"
761       required="true"/>
762     <requestParam name="provider" required="true"/>
763     <requestParam name="desc"/>
764     <requestParam name="permissions" def="public"/>
765     <requestParam name="tags"/>
766     <requestParam name="price"/>
767     <requestParam name="venue"/>
768     <requestParam name="venueId"/>
769   </request>
770   <response>
771     <responseParam name="status"/>
772   </response>
773 </action>
774 <action name="updateEvent" type="constructor">
```

```
775 <request>
776   <requestParam name="id"/>
777   <requestParam name="eventId" required="true"/>
778   <requestParam name="provider" required="true"/>
779   <requestParam name="title"/>
780   <requestParam name="startTime" type="datetime"
781     required="true"/>
782   <requestParam name="stopTime" type="datetime"
783     required="true"/>
784   <requestParam name="desc"/>
785   <requestParam name="permissions" def="public"/>
786   <requestParam name="tags"/>
787   <requestParam name="price"/>
788   <requestParam name="venue"/>
789   <requestParam name="venueId"/>
790 </request>
791 <response>
792   <responseParam name="status"/>
793 </response>
794 </action>
795 <action name="deleteEvent" type="constructor">
796   <request>
797     <requestParam name="id"/>
798     <requestParam name="eventId" required="true"/>
799     <requestParam name="provider" required="true"/>
800   </request>
801   <response>
802     <responseParam name="status"/>
803   </response>
804 </action>
805 <action name="searchVenues" type="constructor">
806   <request>
807     <requestParam name="id"/>
808     <requestParam name="query"/>
809     <requestParam name="provider" required="true"/>
810     <requestParam name="resultSet" type="integer" def="10"/>
811     <requestParam name="offset" type="integer" def="0"/>
812     <requestParam name="sort" def="Name Desc"/>
813   </request>
814   <response>
815     <responseParam name="venueList">
816       <listElem name="venue" minOccurs="0"
817         maxOccurs="unbounded">
818         <attribute name="venueId"/>
819         <attribute name="uri" type="uri"/>
820         <attribute name="name"/>
821         <attribute name="desc"/>
822         <attribute name="venueType"/>
823         <attribute name="city"/>
```

```
824     <attribute name="country"/>
825     <attribute name="address"/>
826     <attribute name="zipCode"/>
827     <attribute name="lat" type="float"/>
828     <attribute name="lng" type="float"/>
829     <attribute name="createdAt" type="datetime"/>
830     <attribute name="user"/>
831     <attribute name="price" type="double"/>
832     <attribute name="image" type="uri"/>
833     <attribute name="categories"/>
834     <attribute name="tags"/>
835   </listElem>
836 </responseParam>
837 </response>
838 </action>
839 </tag>
840 <tag name="sms">
841   <action name="sendMsg" type="constructor">
842     <request>
843       <requestParam name="id"/>
844       <requestParam name="provider" required="true"/>
845       <requestParam name="to" required="true"/>
846       <requestParam name="message" required="false" def=""/>
847       <requestParam name="from"/>
848     </request>
849     <response>
850       <responseParam name="status"/>
851     </response>
852   </action>
853 </tag>
854 <tag name="mms">
855   <action name="sendMsg" type="constructor">
856     <request>
857       <requestParam name="id"/>
858       <requestParam name="provider" required="true"/>
859       <requestParam name="from"/>
860       <requestParam name="to" required="true"/>
861       <requestParam name="subject"/>
862       <requestParam name="message" def=""/>
863       <requestParam name="attachmentUrl"/>
864     </request>
865     <response>
866       <responseParam name="status"/>
867     </response>
868   </action>
869 </tag>
870 <tag name="place">
871   <action name="searchPlaces" type="constructor">
872     <request>
```

```
873 <requestParam name="query"/>
874 <requestParam name="category"/>
875 <requestParam name="lat" type="float"/>
876 <requestParam name="lng" type="float"/>
877 <requestParam name="radius" type="float" def="0.5"/>
878 <requestParam name="resultSet" type="integer" def="10"/>
879 <requestParam name="offset" type="integer" def="0"/>
880 <requestParam name="sort" def="Name Desc"/>
881 </request>
882 <response>
883 <responseParam name="placesList">
884 <listElem name="place" minOccurs="0"
885     maxOccurs="unbounded">
886 <attribute name="placeId"/>
887 <attribute name="website" type="uri"/>
888 <attribute name="name"/>
889 <attribute name="description"/>
890 <attribute name="categories"/>
891 <attribute name="tags"/>
892 <attribute name="city"/>
893 <attribute name="country"/>
894 <attribute name="address"/>
895 <attribute name="zipCode"/>
896 <attribute name="lat" type="double"/>
897 <attribute name="lng" type="double"/>
898 <attribute name="created" type="datetime"/>
899 <attribute name="phone"/>
900 <attribute name="rating"/>
901 <attribute name="image" type="uri"/>
902 </listElem>
903 </responseParam>
904 </response>
905 </action>
906 <action name="getPlaceDetails" type="constructor">
907 <request>
908 <requestParam name="id"/>
909 <requestParam name="placeId" required="true"/>
910 <requestParam name="provider" required="true"/>
911 </request>
912 <response>
913 <responseParam name="placeId"/>
914 <responseParam name="website" type="uri"/>
915 <responseParam name="name"/>
916 <responseParam name="description"/>
917 <responseParam name="categories"/>
918 <responseParam name="tags"/>
919 <responseParam name="city"/>
920 <responseParam name="country"/>
921 <responseParam name="address"/>
```

```
922 <responseParam name="zipCode"/>
923 <responseParam name="lat" type="double"/>
924 <responseParam name="lng" type="double"/>
925 <responseParam name="created" type="datetime"/>
926 <responseParam name="phone"/>
927 <responseParam name="rating"/>
928 <responseParam name="image" type="uri"/>
929 </response>
930 </action>
931 </tag>
932 <tag name="search">
933 <action name="webSearch" type="constructor">
934 <request>
935 <requestParam name="id"/>
936 <requestParam name="query" required="true"/>
937 <requestParam name="region" type="country" def="us"/>
938 <!-- The regional search engine to use -->
939 <requestParam name="country" type="country" def="us"/>
940 <!-- The country to restrict the search results to -->
941 <requestParam name="language" type="language"
942     def="en"/>
943 <requestParam name="childSafe" type="boolean"
944     def="false"/>
945 <requestParam name="resultSet" type="integer" def="10"/>
946 <requestParam name="offset" type="integer" def="0"/>
947 <requestParam name="sort" def="Name Desc"/>
948 </request>
949 <response>
950 <responseParam name="resultList">
951 <listElem name="result" minOccurs="0"
952     maxOccurs="unbounded">
953 <attribute name="title"/>
954 <attribute name="resultPosition" type="integer"/>
955 <attribute name="summary"/>
956 <attribute name="mimeType"/>
957 <attribute name="modDate" type="date"/>
958 <attribute name="website" type="uri"/>
959 <attribute name="cache" type="uri"/>
960 </listElem>
961 </responseParam>
962 </response>
963 </action>
964 <action name="spellingSuggestions" type="constructor">
965 <request>
966 <requestParam name="id"/>
967 <requestParam name="query" required="true"/>
968 </request>
969 <response>
970 <responseParam name="resultList">
```

```
971     <listElem name="result" minOccurs="0"
972         maxOccurs="unbounded">
973         <attribute name="result"/>
974         <attribute name="resultPosition" type="integer"/>
975     </listElem>
976 </responseParam>
977 </response>
978 </action>
979 </tag>
980 <tag name="product">
981     <action name="searchProduct" type="constructor">
982     <request>
983     <requestParam name="id"/>
984     <requestParam name="query" required="true"/>
985     <requestParam name="category"/>
986     <requestParam name="language" type="language" def="en"/>
987     <requestParam name="resultSet" type="integer" def="10"/>
988     <requestParam name="offset" type="integer" def="0"/>
989     <requestParam name="sort" def="Name Desc"/>
990 </request>
991 <response>
992     <responseParam name="productList">
993     <listElem name="product" minOccurs="0"
994         maxOccurs="unbounded">
995         <attribute name="productId"/>
996         <attribute name="isbn"/>
997         <attribute name="upc"/>
998         <attribute name="author"/>
999         <attribute name="title"/>
1000        <attribute name="subtitle"/>
1001        <attribute name="resultPosition" type="integer"/>
1002        <attribute name="category"/>
1003        <attribute name="producer"/>
1004        <attribute name="vendor"/>
1005        <attribute name="netPrice" type="double"/>
1006        <attribute name="currency"/>
1007        <attribute name="thumbnail" type="uri"/>
1008        <attribute name="image" type="uri"/>
1009    </listElem>
1010 </responseParam>
1011 </response>
1012 </action>
1013 <action name="getProductDetails" type="constructor">
1014 <request>
1015 <requestParam name="id"/>
1016 <requestParam name="productId" required="true"/>
1017 <requestParam name="provider" required="true"/>
1018 </request>
1019 <response>
```

```
1020 <responseParam name="productId"/>
1021 <responseParam name="isbn"/>
1022 <responseParam name="upc"/>
1023 <responseParam name="author"/>
1024 <responseParam name="title"/>
1025 <responseParam name="subtitle"/>
1026 <responseParam name="resultPosition" type="integer"/>
1027 <responseParam name="category"/>
1028 <responseParam name="producer"/>
1029 <responseParam name="vendor"/>
1030 <responseParam name="netPrice" type="double"/>
1031 <responseParam name="currency"/>
1032 <responseParam name="thumbnail" type="uri"/>
1033 <responseParam name="image" type="uri"/>
1034 </response>
1035 </action>
1036 </tag>
1037 <tag name="call">
1038 <action name="createCallback" type="constructor">
1039 <request>
1040 <requestParam name="id"/>
1041 <requestParam name="fromNumber" required="true"/>
1042 <requestParam name="toNumber" required="true"/>
1043 <requestParam name="provider"/>
1044 </request>
1045 <response>
1046 <responseParam name="status"/>
1047 </response>
1048 </action>
1049 </tag>
1050 <tag name="email">
1051 <action name="countNewMail" type="constructor">
1052 <request>
1053 <requestParam name="id"/>
1054 <requestParam name="userId" required="true"/>
1055 <requestParam name="provider" required="true"/>
1056 <requestParam name="folder" def="inbox"/>
1057 </request>
1058 <response>
1059 <responseParam name="newMailCount" type="integer"/>
1060 </response>
1061 </action>
1062 <action name="getNewMail" type="constructor">
1063 <request>
1064 <requestParam name="id"/>
1065 <requestParam name="userId" required="true"/>
1066 <requestParam name="provider" required="true"/>
1067 <requestParam name="folder" def="inbox"/>
1068 <requestParam name="resultSet" type="integer" def="10"/>
```

```
1069     <requestParam name="offset" type="integer" def="0"/>
1070     <requestParam name="sort" def="Date Desc"/>
1071 </request>
1072 <response>
1073     <responseParam name="newMailList">
1074         <listElem name="mail" minOccurs="0"
1075             maxOccurs="unbounded">
1076             <attribute name="mailId"/>
1077             <attribute name="from"/>
1078             <attribute name="cc"/>
1079             <attribute name="bcc"/>
1080             <attribute name="replyTo"/>
1081             <attribute name="subject"/>
1082         </listElem>
1083     </responseParam>
1084 </response>
1085 </action>
1086 <action name="sendMail" type="constructor">
1087     <request>
1088         <requestParam name="id"/>
1089         <requestParam name="from" required="true"/>
1090         <requestParam name="to" required="true"/>
1091         <requestParam name="provider" required="true"/>
1092         <requestParam name="cc"/>
1093         <requestParam name="bcc"/>
1094         <requestParam name="replyTo"/>
1095         <requestParam name="subject"/>
1096         <requestParam name="body"/>
1097     </request>
1098     <response>
1099         <responseParam name="status"/>
1100     </response>
1101 </action>
1102 <action name="getMailList" type="constructor">
1103     <request>
1104         <requestParam name="id"/>
1105         <requestParam name="userId" required="true"/>
1106         <requestParam name="provider" required="true"/>
1107         <requestParam name="folder" def="inbox"/>
1108         <requestParam name="resultSet" type="integer" def="10"/>
1109         <requestParam name="offset" type="integer" def="0"/>
1110         <requestParam name="sort" def="Date Desc"/>
1111     </request>
1112     <response>
1113         <responseParam name="mailList">
1114             <listElem name="mail" minOccurs="0"
1115                 maxOccurs="unbounded">
1116                 <attribute name="mailId"/>
1117                 <attribute name="from"/>
```

```

1118     <attribute name="cc"/>
1119     <attribute name="bcc"/>
1120     <attribute name="replyTo"/>
1121     <attribute name="subject"/>
1122   </listElem>
1123 </responseParam>
1124 </response>
1125 </action>
1126 <action name="getMail" type="constructor">
1127   <request>
1128     <requestParam name="id"/>
1129     <requestParam name="userId" required="true"/>
1130     <requestParam name="provider" required="true"/>
1131     <requestParam name="mailId" required="true"/>
1132   </request>
1133   <response>
1134     <responseParam name="mailId"/>
1135     <responseParam name="from"/>
1136     <responseParam name="cc"/>
1137     <responseParam name="bcc"/>
1138     <responseParam name="replyTo"/>
1139     <responseParam name="subject"/>
1140     <responseParam name="body"/>
1141   </response>
1142 </action>
1143 </tag>
1144 <tag name="calendar">
1145   <action name="getEvents" type="constructor">
1146     <request>
1147       <requestParam name="id"/>
1148       <requestParam name="userId" required="true"/>
1149       <requestParam name="provider" required="true"/>
1150       <requestParam name="tags"/>
1151       <requestParam name="date" type="date"/>
1152       <requestParam name="dateRange" type="integer"/>
1153       <requestParam name="resultSet" type="integer" def="10"/>
1154       <requestParam name="offset" type="integer" def="0"/>
1155       <requestParam name="sort" def="Date Desc"/>
1156       <requestParam name="updatedOnly" type="boolean"
1157         def="false"/>
1158       <requestParam name="newOnly" type="boolean" def="false"/>
1159     </request>
1160     <response>
1161       <responseParam name="eventList">
1162         <listElem name="event" minOccurs="0"
1163           maxOccurs="unbounded">
1164           <attribute name="title"/>
1165           <attribute name="description"/>
1166           <attribute name="startTime" type="datetime"/>

```

```
1167     <attribute name="endTime" type="datetime"/>
1168     <attribute name="tags"/>
1169     <attribute name="allday" type="boolean"/>
1170     <attribute name="repeat" type="boolean"/>
1171     <attribute name="repeatEnd" type="date"/>
1172     <attribute name="reminder" type="integer"/>
1173   </listElem>
1174 </responseParam>
1175 </response>
1176 </action>
1177 <action name="getEvent" type="constructor">
1178   <request>
1179     <requestParam name="id"/>
1180     <requestParam name="userId" required="true"/>
1181     <requestParam name="provider" required="true"/>
1182     <requestParam name="eventId" required="true"/>
1183   </request>
1184   <response>
1185     <responseParam name="title"/>
1186     <responseParam name="description"/>
1187     <responseParam name="startTime" type="datetime"/>
1188     <responseParam name="endTime" type="datetime"/>
1189     <responseParam name="tags"/>
1190     <responseParam name="allday" type="boolean"/>
1191     <responseParam name="repeat" type="boolean"/>
1192     <responseParam name="repeatEnd" type="date"/>
1193     <responseParam name="reminder" type="integer"/>
1194   </response>
1195 </action>
1196 <action name="createEvent" type="constructor">
1197   <request>
1198     <requestParam name="id"/>
1199     <requestParam name="userId" required="true"/>
1200     <requestParam name="provider" required="true"/>
1201     <requestParam name="title" required="true"/>
1202     <requestParam name="desc"/>
1203     <requestParam name="startTime" type="datetime"
1204       required="true"/>
1205     <requestParam name="endTime" type="datetime"
1206       required="true"/>
1207     <requestParam name="tags"/>
1208     <requestParam name="allDay" type="boolean" def="false"/>
1209     <requestParam name="repeat"/>
1210     <requestParam name="repeatEnd" type="date"/>
1211     <requestParam name="reminder" type="integer"/>
1212     <!-- Time in minutes before event to send reminder -->
1213   </request>
1214   <response>
1215     <responseParam name="status"/>
```

```
1216     </response>
1217 </action>
1218 <action name="updateEvent" type="constructor">
1219   <request>
1220     <requestParam name="id"/>
1221     <requestParam name="userId" required="true"/>
1222     <requestParam name="provider" required="true"/>
1223     <requestParam name="title" required="true"/>
1224     <requestParam name="desc"/>
1225     <requestParam name="startTime" type="datetime"
1226       required="true"/>
1227     <requestParam name="endTime" type="datetime"
1228       required="true"/>
1229     <requestParam name="tags"/>
1230     <requestParam name="allDay" type="boolean" def="false"/>
1231     <requestParam name="repeat"/>
1232     <requestParam name="repeatEnd" type="date"/>
1233     <requestParam name="reminder" type="integer"/>
1234   </request>
1235   <response>
1236     <responseParam name="status"/>
1237   </response>
1238 </action>
1239 <action name="deleteEvent" type="constructor">
1240   <request>
1241     <requestParam name="id"/>
1242     <requestParam name="userId" required="true"/>
1243     <requestParam name="provider" required="true"/>
1244     <requestParam name="eventId" required="true"/>
1245   </request>
1246   <response>
1247     <responseParam name="status"/>
1248   </response>
1249 </action>
1250 </tag>
1251 <tag name="realEstate">
1252   <action name="searchProperties" type="constructor">
1253     <request>
1254       <requestParam name="id"/>
1255       <requestParam name="type" required="true"
1256         def="apartment"/>
1257       <requestParam name="rentOrBuy" required="true"
1258         def="rent"/>
1259       <requestParam name="lat" type="float"/>
1260       <requestParam name="lng" type="float"/>
1261       <requestParam name="radius" type="double"/>
1262       <requestParam name="netPrice" type="integer"/>
1263       <requestParam name="pricePerSqrMtr" type="integer"/>
1264       <requestParam name="noOfRooms" type="double"/>
```

```
1265 <requestParam name="noOfBathrooms" type="integer"/>
1266 <requestParam name="netArea" type="double"/>
1267 <requestParam name="netRent" type="double"/>
1268 <requestParam name="rentPerSqMtr" type="double"/>
1269 <requestParam name="heating" type="string" def="all"/>
1270 <requestParam name="constructionYear" type="integer"/>
1271 <requestParam name="renovationYear" type="integer"/>
1272 <requestParam name="balcony" type="boolean"/>
1273 <requestParam name="garden" type="boolean"/>
1274 <requestParam name="kitchen" type="boolean"/>
1275 <requestParam name="elevator" type="boolean"/>
1276 <requestParam name="parking" type="boolean"/>
1277 <requestParam name="basement" type="boolean"/>
1278 <requestParam name="barrierFree" type="boolean"/>
1279 <requestParam name="terrace" type="boolean"/>
1280 <requestParam name="newBuilding" type="boolean"/>
1281 <requestParam name="furnished" type="boolean"/>
1282 <requestParam name="excludeWithoutPhoto" type="boolean"/>
1283 <requestParam name="latestUpdate" type="date"/>
1284 <requestParam name="resultSet" type="integer" def="10"/>
1285 <requestParam name="offset" type="integer" def="0"/>
1286 <requestParam name="sort" def="Price Desc"/>
1287 </request>
1288 <response>
1289 <responseParam name="propertyList">
1290 <listElem name="property" minOccurs="0"
1291 maxOccurs="unbounded" content="html">
1292 <attribute name="propertyId"/>
1293 <attribute name="type"/>
1294 <attribute name="rentOrBuy"/>
1295 <attribute name="lat" type="float"/>
1296 <attribute name="lng" type="float"/>
1297 <attribute name="radius" type="double"/>
1298 <attribute name="netPrice" type="integer"/>
1299 <attribute name="pricePerSqrMtr" type="integer"/>
1300 <attribute name="noOfRooms" type="double"/>
1301 <attribute name="noOfBathrooms" type="integer"/>
1302 <attribute name="netArea" type="double"/>
1303 <attribute name="netRent" type="double"/>
1304 <attribute name="rentPerSqMtr" type="double"/>
1305 <attribute name="heating" type="string"/>
1306 <attribute name="constructionYear" type="integer"/>
1307 <attribute name="renovationYear" type="integer"/>
1308 <attribute name="balcony" type="boolean"/>
1309 <attribute name="garden" type="boolean"/>
1310 <attribute name="kitchen" type="boolean"/>
1311 <attribute name="elevator" type="boolean"/>
1312 <attribute name="parking" type="boolean"/>
1313 <attribute name="basement" type="boolean"/>
```

```
1314     <attribute name="barrierFree" type="boolean"/>
1315     <attribute name="terrace" type="boolean"/>
1316     <attribute name="newBuilding" type="boolean"/>
1317     <attribute name="furnished" type="boolean"/>
1318     <attribute name="image" type="uri"/>
1319     <attribute name="latestUpdate" type="date"/>
1320   </listElem>
1321 </responseParam>
1322 </response>
1323 </action>
1324 <action name="getProperty" type="constructor">
1325   <request>
1326     <requestParam name="id"/>
1327     <requestParam name="provider" required="true"/>
1328     <requestParam name="propertyId" required="true"/>
1329   </request>
1330   <response>
1331     <responseParam name="propertyId"/>
1332     <responseParam name="type"/>
1333     <responseParam name="rentOrBuy"/>
1334     <responseParam name="lat" type="float"/>
1335     <responseParam name="lng" type="float"/>
1336     <responseParam name="radius" type="double"/>
1337     <responseParam name="netPrice" type="integer"/>
1338     <responseParam name="pricePerSqrMtr" type="integer"/>
1339     <responseParam name="noOfRooms" type="double"/>
1340     <responseParam name="noOfBathrooms" type="integer"/>
1341     <responseParam name="netArea" type="double"/>
1342     <responseParam name="netRent" type="double"/>
1343     <responseParam name="rentPerSqMtr" type="double"/>
1344     <responseParam name="heating" type="string"/>
1345     <responseParam name="constructionYear" type="integer"/>
1346     <responseParam name="renovationYear" type="integer"/>
1347     <responseParam name="balcony" type="boolean"/>
1348     <responseParam name="garden" type="boolean"/>
1349     <responseParam name="kitchen" type="boolean"/>
1350     <responseParam name="elevator" type="boolean"/>
1351     <responseParam name="parking" type="boolean"/>
1352     <responseParam name="basement" type="boolean"/>
1353     <responseParam name="barrierFree" type="boolean"/>
1354     <responseParam name="terrace" type="boolean"/>
1355     <responseParam name="newBuilding" type="boolean"/>
1356     <responseParam name="furnished" type="boolean"/>
1357     <responseParam name="image" type="uri"/>
1358     <responseParam name="latestUpdate" type="date"/>
1359   </response>
1360 </action>
1361 </tag>
1362 </tagdef>
```

BIBLIOGRAPHY

- [1] *Resource Description Framework RDF*. Online: <http://www.w3.org/RDF/>. [106]
- [2] D. AGRAWAL, M. AMEND, M. DAS, M. FORD, C. KELLER, M. KLOPPMANN, D. KÖNIG, F. LEYMAN, R. MÜLLER, G. PFAU, K. PLÖSSER, R. RANGASWAMY, A. RICKAYZEN, M. ROWLEY, P. SCHMIDT, I. TRICKOVIC, A. YIU, AND M. ZELLER, *Web Services Human Task (WS-HumanTask)*, tech. report, Active Endpoints Inc. and Adobe Systems Inc. and BEA Systems Inc. and International Business Machines Corporation and Oracle Inc. and SAP AG, 2007. [230]
- [3] D. AGRAWAL, M. AMEND, M. DAS, M. FORD, C. KELLER, M. KLOPPMANN, D. KÖNIG, F. LEYMAN, R. MÜLLER, G. PFAU, K. PLÖSSER, R. RANGASWAMY, A. RICKAYZEN, M. ROWLEY, P. SCHMIDT, I. TRICKOVIC, A. YIU, AND M. ZELLER, *WS-BPEL Extension for People (BPEL4People), Version 1.0*, tech. report, Active Endpoints Inc. and Adobe Systems Inc. and BEA Systems Inc. and International Business Machines Corporation and Oracle Inc. and SAP AG, 2007. [42, 230]
- [4] R. AKKIRAJU, J. FARRELL, J. MILLER, M. NAGARAJAN, M.-T. SCHMIDT, A. SHETH, AND K. VERMA, *Web Service Semantics - WSDL-S*, November 2005. [Online]. Available: <http://www.w3.org/Submission/WSDL-S/>. [Accessed: November, 2007]. [36, 106]
- [5] J. ALMEIDA, A. BARAVAGLIO, M. BELAUNDE, P. FALCARIN, AND E. KOVACS, *Service Creation in the SPICE Service Platform*, in Proceedings of the 17th Wireless World Research Forum Meeting (WWRF17), 2006. [58]
- [6] R. ALUR, C. COURCOUBETIS, AND D. DILL, *Model-checking for real-time systems*, in Proceedings of of the 5th Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, 1990, pp. 414–425. [81]
- [7] R. ALUR AND D. L. DILL, *A Theory of Timed Automata*, Theoretical Computer Science **126** (1994), pp. 183–235. [68]
- [8] L. B. AND. B. TEKINERDOGAN, M. GLANDRUP, AND M. AKSIT, *On Composing Separated Concerns, Composability and Composition Anomalies*, in Electronic proceedings at ACM OOPSLA'2000 workshop on Advanced Separation of Concerns, Minneapolis, USA, 2000. [5]

- [9] G. R. ANDREWS, *Synchronizing Resources*, ACM Trans. Program. Lang. Syst. **3**, no. 4 (1981), pp. 405–430. [62, 94]
- [10] T. ANDREWS, F. CURBERA, H. DHOLAKIA, Y. GOLAND, J. KLEIN, F. LEYMAN, K. LIU, D. ROLLER, D. SMITH, S. THATTE, I. TRICKOVIC, AND S. WEERAWARANA, *Business Process Execution Language for Web Services, Version 1.1*, May 2003. [Online]. Available: <http://www.ibm.com/developerworks/library/specification/ws-bpel/>. [Accessed: November, 2007]. [39]
- [11] A. ANKOLEKAR, M. BURSTEIN, J. R. HOBBS, O. LASSILA, D. MARTIN, D. MCDERMOTT, S. A. MCILRAITH, S. NARAYANAN, M. PAOLUCCI, T. PAYNE, AND K. SYCARA, *DAML-S: Web Service Description for the Semantic Web*, in The Semantic Web - ISWC 2002: First International Semantic Web Conference, Sardinia, Italy, June 2002. [35, 57]
- [12] T. BAECK, *On the behavior of evolutionary algorithms in dynamic environments*, in Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on, 1998, pp. 446–451. [125, 128]
- [13] P. BALBIANI, F. CHEIKH, AND G. FEUILLADE, *Algorithms and Complexity of Automata Synthesis by Asynchronous Orchestration With Applications to Web Services Composition*, Electron. Notes Theor. Comput. Sci. **229**, no. 3 (2008), pp. 3–18. [59, 61, 105]
- [14] R. BARRETT AND C. PAHL, *Model Driven Design of Distribution Patterns for Web Service Compositions* (2006), p. 23. [69, 71]
- [15] A. P. BARROS, M. DUMAS, AND A. H. M. TER HOFSTEDÉ, *Service Interaction Patterns*, in Business Process Management, 2005, pp. 302–318. [69]
- [16] S. BECHHOFFER, F. VAN HARMELEN, J. HENDLER, I. HORROCKS, D. L. MCGUINNESS, P. F. PATEL-SCHNEIDER, L. TECHNOLOGIES, AND L. A. STEIN, *OWL Web Ontology Language and Reference*. W3C, February 2004. Online: <http://www.w3.org/TR/2004/REC-owl-ref-20040210>. [35]
- [17] O. BEN-KIKI, C. EVANS, AND B. INGERSON, *YAML Ain't Markup Language (YAML) Version 1.1*, tech. report, Oren Ben-Kiki, Clark Evans, Brian Ingerson. [16]
- [18] D. BERARDI, D. CALVANESE, G. DE GIACOMO, M. LENZERINI, AND M. MECELLA, *Automatic Service Composition Based on Behavioral Descriptions*, Int. J. of Cooperative Information Systems **14**, no. 4 (2005), pp. 333–376. [79]

- [19] D. BERARDI, F. CHEIKH, G. D. GIACOMO, AND F. PATRIZI, *Automatic Service Composition via Simulation*, Int. J. Found. Comput. Sci. **19**, no. 2 (2008), pp. 429–451. [58]
- [20] L. BERNARD, S. HAUBROCK, S. HÜBNER, W. KUHN, R. LESSING, M. LUTZ, AND U. VISSER, *Semantic Interoperability by means of Geoservices - Semantic Problems in three Use Cases and Approaches for Potential Solutions*, tech. report, Hannover, 2003. [231]
- [21] R. F. T. BERNERS-LEE AND L. MASINTER, *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax*, tech. report, IETF, 1998. [8]
- [22] J. BERSTEL, *Transductions and Context-free Languages*, B.G. Teubner, Stuttgart, 1979. [80, 81]
- [23] B. BERTHOMIEU, F. PERES, AND F. VERNADAT, *Bridging the Gap Between Timed Automata and Bounded Time Petri Nets*, Lecture Notes in Computer Science : Formal Modeling and Analysis of Timed Systems, Volume 4202, 2006 (2006), pp. 82–97. [81]
- [24] B. BERTHOMIEU AND F. VERNADAT, *Time Petri Nets Analysis with TINA*, in Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on, 2006, pp. 123–124. [81]
- [25] N. BLUM, T. MAGEDANZ, AND F. SCHREINER, *Definition of a Service Delivery Platform for Service Exposure and Service Orchestration in Next Generation Networks*, UbiCC Journal **3** (2008). [229]
- [26] B. BORGES, K. HOLLEY, AND A. ARSANJANI, *SOA News: Service-oriented architecture*, tech. report, IBM, 2004. [Online]. Available: http://searchsoa.techtarget.com/news/article/0,289142,sid26_gci1006206,00.html. [Accessed: August 19, 2009]. [25]
- [27] B. BOS AND L. QUIN, *The XML Data Model*, 2005. [Online]. Available: <http://www.w3.org/XML/Datamodel.html>. [Accessed: August 19, 2009]. [239]
- [28] A. BUCCHIARONE AND S. GNESI, *A Survey on Services Composition Languages and Models*, in in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006), A. Bertolino and A. Polini, eds., Palermo, Sicily, ITALY, June 9th 2006, pp. 51–63. [106]
- [29] M. CACERES, *Widgets 1.0: Packaging and Configuration*, W3C working draft, W3C, Apr. 2008. <http://www.w3.org/TR/2008/WD-widgets-20080414/>. [66]
- [30] M. CACERES, *Widgets 1.0: Requirements*, a WD in last call, W3C, June 2008. <http://www.w3.org/TR/2008/WD-widgets-reqs-20080625/>. [71]

- [31] *CakePHP: The Rapid Development PHP Framework*, 2008. [Online]. Available: <http://cakephp.org/>. [Accessed: August 19, 2009]. [49]
- [32] D. CALVANESE, G. DE GIACOMO, M. LENZERINI, M. MECELLA, AND F. PATRIZI, *Automatic Service Composition and Synthesis: the Roman Model*, Bull. of the IEEE Computer Society Technical Committee on Data Engineering **31**, no. 3 (2008), pp. 18–22. [58, 61]
- [33] D. CARL, J. CLAUSEN, M. HASSLER, AND A. ZUND, *Mashups programmieren - Grundlagen, Konzepte, Beispiele*, O'Reilly, Beijing, Cambridge, Farnham, Cologne, Paris, Sebastopol, 1 ed., 2008. [9, 11]
- [34] M. CARMAN, L. SERAFINI, AND P. TRAVERSO, *Web Service Composition as Planning*, in Workshop on Planning for Web Services, Trento, Italy, June 2003. [57]
- [35] F. CASSEZ AND O. H. ROUX, *Structural Translation from Time Petri Nets to Timed Automata*, Journal of Software and Systems **79** (2006), pp. 1456–1468. [81]
- [36] R. CERNY, *Topincs - A RESTful Web Service Interface for Topic Maps*, in Leveraging the Semantics of Topic Maps, Lecture Notes in Computer Science 4438/2007, Springer Verlag, 2007, pp. 175 – 183. [16]
- [37] G. B. CHAFLE, S. CHANDRA, V. MANN, AND M. G. NANDA, *Decentralized orchestration of composite web services* (2004), pp. 134–143. [70, 71]
- [38] E. CHRISTENSEN, F. CURBERA, G. MEREDITH, AND S. WEERAWARANA, *Web Services Description Language (WSDL) 1.1*, tech. report, W3C, 2001. [32]
- [39] E. M. J. CLARKE, O. GRUMBERG, AND D. A. PELED, *Model Checking*, The MIT Press, Cambridge, Massachusetts and London, England, 1999. [58, 59, 67, 81, 82]
- [40] G. F. COULOURIS, J. DOLLIMORE, AND T. KINDBERG, *Distributed systems: concepts and design*, Addison-Wesley, Wokingham; Sydney, 2nd ed. ed., 1994. [95]
- [41] D. CRANE AND P. MCCARTHY, *Comet and Reverse Ajax: The Next-Generation Ajax 2.0*, Apress, Berkely, CA, USA, 2008. [214]
- [42] I. CRNKOVIC, *Building Reliable Component-Based Software Systems*, Artech House, Inc., Norwood, MA, USA, 2002. [25]
- [43] P. CUDRE MAUROUX AND K. ABERER, *A Necessary Condition for Semantic Interoperability in the Large* (2004), pp. 859–872. [231]

- [44] F. CURBERA, *Unraveling the Web Services Web: An Introduction to SOAP, WSDL and UDDI*, IEEE Internet Computing **6**, no. 2 (2002), pp. 86 – 93. [32, 33]
- [45] E. M. G. DA SILVA, L. F. PIRES, AND M. J. VAN SINDEREN, *An Algorithm for Automatic Service Composition*, in 1st International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, ICSoft 2007, Barcelona, Spain, E. M. G. da Silva, L. F. Pires, and M. J. van Sinderen, eds., Portugal, July 2007, INSTICC Press, pp. 65–74. [58]
- [46] J. DE BRUIJN, C. BUSSLER, J. DOMINGUE, D. FENSEL, M. HEPP, U. KELLER, M. KIFER, B. KOENIG-RIES, J. KOPECKY, R. LARA, H. LAUSEN, E. OREN, A. POLLERES, D. ROMAN, J. SCICLUNA, AND M. STOLLBERG, *Web Service Modeling Ontology (WSMO)*, tech. report, W3C, 2005. [79]
- [47] *DIANE (Dienste In Ad-hoc NETzwerken) - Project*, 2008. [Online]. Available: <http://hnsp.inf-bb.uni-jena.de/>. [Accessed: August 19, 2009]. [49]
- [48] M. DIBERNARDO, R. POTTINGER, AND M. WILKINSON, *Semi-automatic web service composition for the life sciences using the BioMoby semantic web framework*, Journal of Biomedical Informatics **41**, no. 5 (2008), pp. 837–847. [56]
- [49] P. F. DIETZ, *Maintaining Order in a Linked List*, in STOC '82: Proceedings of the 14th Annual ACM Symposium on Theory of Computing, New York, NY, 1982, ACM, pp. 122 – 127. [239]
- [50] E. W. DIJKSTRA, *Guarded commands, nondeterminacy and formal derivation of programs*, Commun. ACM **18**, no. 8 (1975), pp. 453–457. [63]
- [51] D. DILL, *Timing assumptions and verification of finite-state concurrent systems*, in Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, J. Sifakis, ed., no. 407 in LNCS, Springer, 1989, pp. 197–212. [81]
- [52] L. DOLDI, *UML2 illustrated, Developing Real-Time & Communication Systems*, TMSO, 2003. [81]
- [53] W. DOSCH, P. MUENCHASRI, W. RUANTHONG, AND A. STÜMPPEL, *Model checking for input/output properties of a black-box model*, in ACST'07: Proceedings of the third conference on IASTED International Conference, Anaheim, CA, USA, 2007, ACTA Press, pp. 120–127. [79]
- [54] S. DUSTDAR AND W. SCHREINER, *A survey on web services composition*, IJWGS **1**, no. 1 (2005), pp. 1–30. [57]

- [55] S. EDELKAMP, *Automated Planning: Theory and Practice*, KI **21**, no. 1 (2007), pp. 42–43. [56]
- [56] J. EDER AND M. LEHMANN, *Workflow Data Guards*, in *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, Agia Napa, Cyprus, 10 2005, Springer Verlag, pp. 502–519. [64, 71]
- [57] A. E. EIBEN AND J. E. SMITH, *Introduction to Evolutionary Computing*, Natural Computing, Springer, October 2008. [128]
- [58] S. EILENBERG, *Automata, Languages, and Machines*, Academic Press, New York, 1974. [80, 81]
- [59] M. M. ELGAYYAR, S. J. ALDA, AND A. B. CREMERS, *Towards a user-oriented environment for web services composition*, in *WEUSE '08: Proceedings of the 4th international workshop on End-user software engineering*, New York, NY, USA, 2008, ACM, pp. 81–85. [48]
- [60] Z. FAN-ZI AND Q. ZHENG-DING, *A survey of classification learning algorithm*, in *Signal Processing, 2004. Proceedings. ICSP '04. 2004 7th International Conference on*, Aug.-4 Sept. 2004, pp. 1500–1504 vol.2. [60]
- [61] R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH, AND T. B. LEE, *RFC 2616 - HTTP/1.1, the hypertext transfer protocol*. <http://w3.org/Protocols/rfc2616/rfc2616.html>, 1999. [8]
- [62] R. T. FIELDING, *Architectural Styles and the Design of Network-based Software Architectures*, PhD thesis, University of California, Irvine, 2000. [8, 50, 78]
- [63] R. T. FIELDING AND R. N. TAYLOR, *Principled Design of the Modern Web Architecture*, in *Proceedings of the 2000 International Conference on Software Engineering*, 2000, pp. 407 – 416. [9]
- [64] C. M. FONSECA AND P. J. FLEMING, *An Overview of Evolutionary Algorithms in Multiobjective Optimization*, *Evolutionary Computation* **3**, no. 1 (1995), pp. 1–16. [125]
- [65] GARTNER RESEARCH, *Gartner Says Worldwide Application Infrastructure and Middleware Market Revenue Increased 13 Percent in 2007*. Press release. [230]
- [66] I. GAVRAN, *End-User Programming Language for Service-Oriented Integration*, in *Proceedings of the 6th WDAS*, November 2005, pp. 1 – 6. [54]
- [67] D. GEORGAKOPOULOS, M. F. HORNICK, AND A. P. SHETH, *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*, *Distributed and Parallel Databases* **3**, no. 2 (1995), pp. 119–153. [38]

- [68] GOOGLE, *Gears*. [Online]. Available: <http://gears.google.com/>. [Accessed: August 19, 2009]. [65, 71, 230]
- [69] *Google Data APIs Overview*, 2008. [Online]. Available: <http://code.google.com/intl/en/apis/gdata/overview.html>. [Accessed: August 19, 2009]. [17]
- [70] *Google Maps API - Google Code*, 2008. [Online]. Available: <http://code.google.com/apis/maps/>. [Accessed: August 19, 2009]. [234]
- [71] D. N. GRAY, J. HOTCHKISS, S. LAFORGE, A. SHALIT, AND T. WEINBERG, *Modern languages and microsoft's component object model*, Commun. ACM **41**, no. 5 (1998), pp. 55–65. [25]
- [72] S. HAAR, F. SIMONOT-LION, L. KAISER, AND J. TOUSSAINT, *Equivalence of Timed State Machines and safe Time Petri Nets*, in Proceedings of WODES 2002, Zaragoza, 2002, pp. 119–126. [81]
- [73] M. J. HADLEY, *Web Application Description Language (WADL)*, tech. report, Sun Microsystems Inc., November 2006. [34, 198]
- [74] P. B. HANSEN, *Distributed processes: a concurrent programming concept*, Commun. ACM **21**, no. 11 (1978), pp. 934–941. [63]
- [75] J. HARTMANIS, *Algebraic structure theory of sequential machines (Prentice-Hall international series in applied mathematics)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1966. [67, 68, 71]
- [76] *Hibernate: Relational Persistence for Java and .NET*. [Online]. Available: <https://www.hibernate.org/>. [Accessed: August 19, 2009]. [218]
- [77] D. HINCHCLIFFE, *The quest for enterprise mashup tools*, ZDNet (2006). [12]
- [78] C. A. R. HOARE, *Communicating Sequential Processes*, Commun. ACM **21**, no. 8 (1978), pp. 666–677. [63]
- [79] S. HOLZNER, *The Dojo Toolkit: Visual QuickStart Guide*, Peachpit Press, Berkeley, CA, USA, 2008. [49]
- [80] J. HOWELL, C. JACKSON, H. J. WANG, AND X. FAN, *Mashupos: operating system abstractions for client mashups*, in HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems, Berkeley, CA, USA, 2007, USENIX Association, pp. 1–7. [66]
- [81] M. N. HUHN AND M. P. SINGH, *Service-Oriented Computing: Key Concepts and Principles*, Internet Computing **9**, no. 1 (2005), pp. 75 – 81. [24, 27, 28, 29, 37]

- [82] R. HULL AND J. SU, *Tools for Design of Composite Web Services*, in SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, New York, NY, USA, 2004, ACM, pp. 958 – 961. [37, 39]
- [83] J. HUNTER, *Enhancing the Semantic Interoperability of Multimedia Through a Core Ontology*, IEEE Transactions on Circuits and Systems for Video Technology **13**, no. 1 (2003), pp. 49–58. [231]
- [84] M. HUTH AND M. RYAN, *Logic in Computer Science - Modeling and reasoning about systems*, Cambridge University Press, 2004. [82]
- [85] *i-Technology Viewpoint: Is Web 2.0 the Global SOA?* [Online]. Available: <http://webservices.sys-con.com/read/164532.htm>. [Accessed: August 19, 2009]. [10, 11, 22]
- [86] C. JACKSON AND H. J. WANG, *Subspace: Secure Cross-Domain Communication for Web Mashups*, in WWW 2007, Banff, Alberta, Canada, May 2007. [66]
- [87] I. JACOBS AND N. WALSH, *Architecture of the World Wide Web, Volume One*, tech. report, W3C, 2004. [8]
- [88] M. JAEGER, L. ENGEL, AND K. GEIHS, *A Methodology for Developing OWL-S Descriptions*, in First International Conference on Interoperability of Enterprise Software and Applications Workshop on Web Services and Interoperability (INTEROP-ESA '05), Springer, 2005. [36, 79, 106]
- [89] *Java Architecture for XML Binding (JAXB)*. [Online]. Available: <https://jaxb.dev.java.net/>. [Accessed: August 19, 2009]. [216]
- [90] H.-J. JIN AND H.-C. LEE, *Web Services Development Methodology Using the Mashup Technology*, in International Conference on Smart Manufacturing Application (ICSMA 2008), Goyang-Si, South Korea, April 2008, pp. 559 – 562. [9, 11, 12]
- [91] S.-M. JUN, D.-H. YU, Y.-H. KIM, AND S.-Y. SEONG, *A Time Synchronization Method for NTP*, Real-Time Computing Systems and Applications, International Workshop on **0** (1999), p. 466. [164]
- [92] M. B. JURIC, B. MATHEW, AND S. P., *Business Process Execution Language for Web Services*, Packt Publishing, January 2006. [39]
- [93] *Kapow Technologies*, 2008. [Online]. Available: <http://www.kapowtech.com/>. [Accessed: August 19, 2009]. [52]

- [94] G. KARAGIANNPOULOS, N. GEORGOPOULOS, AND K. NIKOLOPOULOS, *Fathoming Porter's Fice Forces Model in the Internet Era*, Info - The journal of policy, regulation and strategy for telecommunications **7**, no. 6 (2005), pp. 66 – 76. [42]
- [95] U. KELLER, R. LARA, A. POLLERES, I. TOMA, M. KIFER, AND D. FENSEL, *WSMO Web Service Discovery*, Tech. Report D5.1v0.1, DERI, November 2004. [79]
- [96] R. KHALAF, N. MUKHI, AND S. WEERAWARANA, *Service-Oriented Composition in BPEL4WS*, in Proceedings of the 2003 WWW Conference, Web Services Track, 2003. [39, 41]
- [97] H. KIL, W. NAM, AND D. LEE, *Automatic web service composition with abstraction and refinement*, in WWW, J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, eds., ACM, 2009, pp. 1121–1122. [59]
- [98] M. KLEIN, *Handbuch zur DIANE Service Description*, tech. report, Universität Karlsruhe, Faculty of Informatics, 2004. [49]
- [99] K. KROHN, R. MATEOSIAN, AND J. RHODES, *Methods of the Algebraic Theory of Machines. I: Decomposition Theorem for Generalized Machines; Properties Preserved under Series and Parallel Compositions of Machines*, J. Comput. Syst. Sci. **1**, no. 1 (1967), pp. 55–85. [67]
- [100] S. KRUESSEL, H. PFEFFER, AND S. STEGLICH, *Fuzzy Modeling of Resource Consumption for Service Composition Evaluation*, in In Proceedings of the Second International Conference on Advances in Human-oriented and Personalized Mechanisms, Technologies, and Services, 2009. [140, 229]
- [101] U. KÜSTER, B. KÖNIG, M. KLEIN, AND M. STERN, *DIANE - a matchmaking-centered framework for automated service discovery, composition, binding, and invocation on the web*, International Journal of Electronic Commerce (IJEC) **12**, no. 2 (2007), pp. 41 – 68. [49]
- [102] K. LARSON, P. PETTERSSON, AND W. YI, *UPPAAL in a Nutshell*, International Journal on Software Tools for Technology Transfer **1** (1997), pp. 134–152. [64, 82, 103]
- [103] A. LEFF AND J. T. RAYFIELD, *Web-Application Development Using the Model/View/Controller Design Pattern*, Enterprise Distributed Object Computing Conference, IEEE International **0** (2001), p. 0118. [141]
- [104] F. LEYMAN, *Web Service Flow Language (WSFL 1.0)*, in IBM, May 2001. [39]

- [105] G. LI, H. HAI, AND Z. HU, *A flexible framework for semi-automatic web services composition*, Asia-Pacific Conference on Services Computing. 2006 IEEE **0** (2008), pp. 1258–1262. [56]
- [106] B. LIU, S. WU, AND T. LV, *A distributed workflow model based on web service*, in International Conference on Intelligent Computation Technology and Automation, 2008. [68, 70]
- [107] X. LIU, G. HUANG, AND H. MEI, *Towards End User Service Composition*, 31st Annual International Computer Software and Applications Conference, 2007. COMPSAC 2007. **1** (2007), pp. 676 – 678. [37]
- [108] M. LUTZ, *Programming Python*, OReilly, October 1996. [16]
- [109] O. MALER AND A. PNUELI, *On the Cascaded Decomposition of Automata, its Complexity and its Application to Logic*, ACTS Mobile Communication **48** (1994), p. pages. [67, 71]
- [110] Z. MANNA AND A. PNUELI, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992. [82]
- [111] A. MARCONI, M. PISTORE, AND P. POCCIANI, *Automated Web Service Composition at Work: the Amazon/MPS Case Study*, ICWS 2007. IEEE International Conference on Web Services, 2007. (2007), pp. 767 – 774. [29, 37, 56]
- [112] D. MARTIN, M. BURSTEIN, E. HOBBS, O. LASSILA, D. MCDERMOTT, S. MCILRAITH, S. NARAYANAN, B. PARSIA, T. PAYNE, E. SIRIN, N. SRINIVASAN, AND K. SYCARA, *OWL-S: Semantic Markup for Web Services*, tech. report, November 2004. [35, 36, 79]
- [113] K. MASON AND P. KRISHNAN, *Decomposition of Timed Automata*, J. UCS **5**, no. 9 (1999), pp. 574–587. [67, 68]
- [114] E. M. MAXIMILIEN, H. WILKINSON, N. DESAI, AND S. TAI, *A Domain-Specific Language for Web APIs and Services Mashups*, in ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 13–26. [12, 49]
- [115] D. MERRILL, *Mashups: The new breed of Web app*, tech. report, IBM Corp., 2006. [Online]. Available: <http://www.ibm.com/developerworks/xml/library/x-mashups.html>. [Accessed: August 19, 2009]. [9, 11, 12, 13, 16, 21, 22, 34, 234]
- [116] A. MICHLMAYR, F. ROSENBERG, C. PLATZER, M. TREIBER, AND S. DUSTDAR, *Towards Recovering the Broken SOA Triangle: A Software Engineering Perspective*, in IW-SOSWE '07: 2nd international workshop on

- Service oriented software engineering, New York, NY, 2007, ACM, pp. 22–28. [28]
- [117] N. MILANOVIC AND M. MALEK, *Current Solutions for Web Service Composition*, IEEE Internet Computing **8**, no. 6 (2004), pp. 51 – 59. [37, 38, 40]
- [118] D. L. MILLS, *Internet Time Synchronization: the Network Time Protocol*, IEEE Transactions on Communications **39** (1991), pp. 1482–1493. [164]
- [119] S. MODAFFERI AND E. CONFORTI, *Methods for Enabling Recovery Actions in WS-BPEL*, in OTM Conferences (1), 2006, pp. 219–236. [230]
- [120] N. MOHAMMED, B. C. M. FUNG, K. WANG, AND P. C. K. HUNG, *Privacy-preserving data mashup*, in EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology, New York, NY, USA, 2009, ACM, pp. 228–239. [66]
- [121] T. MURATA, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE **77**, no. 4 (1989), pp. 541–580. [80]
- [122] C. S. D. G. S. MURER, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley Longman, Amsterdam, 2nd ed. (15. november 2002) ed., 2002. ISBN-10: 0201745720 ISBN-13: 978-0201745726. [24]
- [123] J. MUSSER, *REST Compile*, 2007. [Online]. Available: <http://blog.programmableweb.com/2007/03/22/wadlandgooglerest-compile/>. [Accessed: September 15, 2009]. [198]
- [124] K.-H. NAM, K. S. BANG, AND W. CHOI, *A Method for Distributing Web Applications*, in ICACT 2008, February 2008. [65, 71]
- [125] M. G. NANDA, S. CHANDRA, AND V. SARKAR, *Decentralizing execution of composite web services*, SIGPLAN Not. **39**, no. 10 (2004), pp. 170–187. [63, 68, 70]
- [126] M. G. NANDA AND N. KARNIK, *Synchronization analysis for decentralizing composite Web services*, in SAC '03: Proceedings of the 2003 ACM symposium on Applied computing, New York, NY, USA, 2003, ACM, pp. 407–414. [63, 70]
- [127] D. S. NAU, H. MUÑOZ-AVILA, Y. CAO, A. LOTEM, AND S. MITCHELL, *Total-Order Planning with Partially Ordered Subtasks*, in IJCAI, 2001, pp. 425–430. [57]
- [128] NETWORK WORKING GROUP, *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627, 2006. [206]

- [129] OASIS, *Oasis SOA Reference Model TC*, August 2008. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm. [Accessed: September 15, 2009]. [23]
- [130] M. OGRINZ, *Mashup Patterns: Designs and Examples for the Modern Enterprise*, Addison Wesley Professional, 2009 (est.). [11, 24]
- [131] OMG, *CORBA Component Model 4.0 Specification*, April 2006. [25]
- [132] OPEN MOBILE TERMINAL PLATFORM (OMTP), *BONDI*. Online: <http://bondi.omtp.org>, 2009. Accessed: June 2009. [17]
- [133] T. O'REILLY, *What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software*, Communications & Strategies **1st Quarter 2007**, no. 65 (2007), pp. 17–37. [10, 11, 12]
- [134] E. ORT, *Service-Oriented Architecture and Web Services: Concepts, Technologies and Tools*, tech. report, Sun Developer Network (SDN), April 2005. [29, 30, 31]
- [135] E. ORT, S. BRYDON, AND M. BASLER, *Mashup Styles, Part 1: Server-Side Mashups*, tech. report, Sun Developer Network (SDN), May 2007. [9, 12]
- [136] E. ORT, S. BRYDON, AND M. BASLER, *Mashup Styles, Part 2: Client-Side Mashups*, tech. report, Sun Developer Network (SDN), August 2007. [9, 12, 22, 23]
- [137] D. PANDA, R. RAHMAN, AND D. LANE, *EJB 3 in Action*, Manning Publications Co., Greenwich, CT, USA, 2007. [25]
- [138] M. PAPAZOGLU AND D. GEORGAKOPOULOS, *Service-Oriented Computing*, Communications of the ACM **46**, no. 10 (2003), pp. 24 – 28. [26, 27, 30]
- [139] M. P. PAPAZOGLU, P. TRAVERSO, S. DUSTDAR, F. LEYMAN, AND B. KRÄMER, *Service-Oriented Computing: A Research Roadmap*, in Service Oriented Computing (SOC), F. Cubera, J. Bernd Krämer, and M. P. Papazoglou, eds., no. 05462, 2006. [24, 26, 28, 30, 37]
- [140] C. PAUTASSO, *BPEL for REST*, in BPM, 2008, pp. 278–293. [47]
- [141] J. PEER, *Web Service Composition as AI Planning-a Survey*, tech. report, University of St. Gallen, 2005. [57]
- [142] C. PELTZ, *Web Services Orchestration and Choreography*, Computer **36**, no. 10 (2003), pp. 46 – 52. [38, 39, 42]
- [143] C. PETTEY AND L. GOASDUFF, *Gartner's 2006 emerging technologies hype cycle highlights key technology themes*. <http://www.gartner.com/it/page.jsp?id=495475>, 2006. Online. Accessed April 2009. [9, 12, 22]

- [144] H. PFEFFER, *UPPAAL Model Checking as Performance Evaluation Technique*, master's thesis, Rheinische Friedrich-Whilhelms-Universität Bonn, 2005. [81]
- [145] H. PFEFFER, L. BASSBOUSS, AND S. STEGLICH, *Structured Service Composition Execution for Mobile Web Applications*, in Proceedings of the 12th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2008), Kunming, China, 2008, IEEE Computer Society Press, pp. 112–118. [208]
- [146] H. PFEFFER, S. KRUESSEL, AND S. STEGLICH, *Fuzzy Service Composition Evaluation In Distributed Environments*, in In Proceedings of I-CENTRIC 2008, 2008. [140, 177, 229]
- [147] H. PFEFFER, D. LINNER, AND S. STEGLICH, *Modeling and controlling dynamic service compositions*, in Proceedings of the Third International Multi-Conference on Computing in the Global Information Technology 2008 (ICCGI '08), Washington D.C., 2008, IEEE Computer Society, pp. 210 – 216. [28]
- [148] *Pipes: Rewire the Web*, 2008. [Online]. Available: <http://pipes.yahoo.com/>. [Accessed: Semptember 15, 2009]. [52]
- [149] L. PRECHELT, *Are scripting languages any good? A validation of Perl, Python, Rexx, and Tcl against C, C++, and Java*, *Advances in Computers* **57** (2003), pp. 207–271. [16]
- [150] *Programmable Internet Environment (PIE)*. [Online]. Available: <http://www.pie.fer.hr/Menu/index.htm>. [Accessed: September, 2009]. [54]
- [151] *Project Dynvocation*, 2008. [Online]. Available: <http://dynvocation.selfip.net/>. [Accessed: August 19, 2009]. [53, 54]
- [152] J. RAO AND X. SU, *A survey of automated web service composition methods*, in *Semantic Web Services and Web Process Composition*, Springer, Berlin, Heidelberg, 2005, pp. 43 – 54. [57]
- [153] RDF CORE WORKING GROUP, *RDF/XML Syntax Specification (Revised)*, tech. report, W3C, 2004. [34]
- [154] E. RECHTIN AND M. W. MAIER, *The art of systems architecting*, CRC Press, Inc., Boca Raton, FL, USA, 1997. [5]
- [155] J. RESIG, *jquery*. [Online]. Available: <http://jquery.com/>. [Accessed: September 10, 2009]. [49]

-
- [156] R. RICHARDS, *Pro PHP XML and Web Services (Pro)*, Apress, Berkely, CA, USA, 2006. [32]
- [157] L. RICHARDSON AND S. RUBY, *RESTful Web Services*, O'Reilly, Sebastopol, CA, 2007. [9, 31]
- [158] M. ROSE, *On the Design of Application Protocols*. RFC 3117 (Informational), Nov. 2001. [214]
- [159] W. ROSHEN, *Services-based enterprise integration patterns made easy, part 1: The evolution of basic concepts*, 2008. [28]
- [160] *Ruby On Rails*, 2008. [Online]. Available: <http://www.rubyonrails.org/>. [Accessed: Semptember 15, 2009]. [49]
- [161] N. RUSSELL, A. TER HOFSTEDDE, W. VAN DER AALST, AND N. MULYAR, *Workflow Control-Flow Patterns: A Revised View*, tech. report, BPMcenter.org, 2006. [69]
- [162] A. RYAN, *Towards Semantic Interoperability in Healthcare: Ontology Mapping from SNOMED-CT to HL7 version 3*, in Second Australasian Ontology Workshop (AOW 2006), M. A. Orgun and T. Meyer, eds., CRPIT 72, Hobart, Australia, 2006, ACS, pp. 69–74. [231]
- [163] M. SABBOUH, J. HIGGINSON, S. SEMY, AND D. GAGNE, *Web Mashup Scripting Language*, 2007. [50]
- [164] S. SALKOSUO, *DWR Java AJAX Applications*, Packt Publishing, 2008. [214]
- [165] V. SASSONE, M. NIELSEN, AND G. WINSKEL, *Models for Concurrency: Towards a Classification*, *Theoretical Computer Science* **170** (1996), pp. 297–348. [80]
- [166] C. SCHROTH AND O. CHRIST, *Brave New Web: Emerging Design Principles and Technologies as Enablers of a Global SOA*, in Proceedings of the IEEE International Conference on Services Computing (SCC 2007), 2007, pp. 597 – 604. [52, 54]
- [167] C. SCHROTH AND T. JANNER, *Web 2.0 and SOA: Converging Concepts Enabling the Internet of Services*, *IT Professional* **9**, no. 3 (2007), pp. 36 – 41. [54]
- [168] M. SHESHAGIRI, M. DESJARDINS, AND T. FININ, *A Planner for Composing Services Described in DAML-S*, in Proceedings of the AAMAS Workshop on Web Services and Agent-based Engineering,, June 2003. [57]

- [169] E. SIRIN, J. HENDLER, AND B. PARSIA, *Semi-automatic Composition of Web Services using Semantic Descriptions*, in In Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2002, 2002, pp. 17–24. [56]
- [170] D. SKROBO, *HUSKY: A Spreadsheet for End-User Service Composition*, PhD thesis, University of Zagreb - Faculty of Electrical Engineering and Computing, Croatia, April 2008. [54]
- [171] D. SKROBO, A. MILANOVIC, AND S. SRBLJIC, *Distributed Program Interpretation in Service-Oriented Architectures*, in Proceedings of the 9th World Multi-Conference of Systemics, Cybernetics and Informatics (WMSCI '05), Orlando, FL, 2005, IIIS, pp. 193 – 197. [54]
- [172] D. SKVORC, *Uncoupling Service Composition Architecture*, tech. report, School of Electrical Engineering and Computing - Department of Electronics, Microelectronics, Computer and Intelligent Systems, 2008. [54]
- [173] D. K. SMETTERS, *Building secure mashups*, Workshop on Web 2.0 Security and Privacy (W2SP 2008; part of the IEEE 2008 Security and Privacy Conference) (2008). [66]
- [174] J. SPILLNER, I. BRAUN, AND A. SCHILL, *WSInterConnect: Dynamic Composition of Web Services Through Web Services*, in DAIS 2006, 2006, pp. 181 – 186. [53]
- [175] J. SPILLNER, I. BRAUN, AND A. SCHILL, *Flexible Human Service Interfaces*, in ICEIS 2007, 2007, pp. 79 – 85. [53]
- [176] L. SRINIVASAN AND J. TREADWELL, *An Overview of Service-Oriented Architecture, Web Services and Grid Computing*, tech. report, HP Software Global Business Unit, November 2005. [28, 29, 30, 32]
- [177] B. SRIVASTAVA AND J. KOEHLER, *Web Service Composition - Current Solutions and Open Problems*, in ICAPS 2003, 2003. [37]
- [178] S. STEGLICH, *Next generation communication systems - outlook*, in Grundlagen Offener Kommunikationssysteme, 2008. [31]
- [179] L. D. STEIN, C. WONG, AND S. GUNDAVARAM, *Scripting Languages: Automating the Web*, O'Reilly & Associates, Sebastopol, California, May 1997. [16]
- [180] T. STEINER, *Automatic Multi Language Program Library Generation for REST APIs*, master's thesis, University of Karlsruhe (TH), Institute for Algorithms and Cognitive Systems, 2007. [34]

- [181] J. TATEMURA, A. SAWIRES, O. PO, S. CHEN, K. S. CANDAN, D. AGRAWAL, AND M. GOVEAS, *Mashup feeds: Continuous queries over web services*, in Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07), 2007, pp. 1128 – 1130. [48]
- [182] M. TER BEEK, A. BUCCIARONE, AND S. GNESI, *A survey on Service Composition Approaches: From Industrial Standards to Formal Methods*, Tech. Report 2006-TR-15, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 2006. [55, 57]
- [183] S. THATTE, *XLANG - Web Services for Business Process Design*, 2001. [39]
- [184] *The ActiveBPEL Community Edition Engine*, 2008. [Online]. Available: <http://www.activebpel.org>. [Accessed: September 15, 2009]. [48]
- [185] THE OWL SERVICES COALITION, *OWL-S: Semantic Markup for Web Services*, November 2004. [Online]. Available: <http://www.daml.org/services/owls/1.1/>. [Accessed: February 26, 2008]. [107]
- [186] *The Spring Framework - Reference Documentation*. [Online]. Available: <http://static.springframework.org/spring/docs/2.5.x/reference/index.html>. [Accessed: August 19, 2009]. [219]
- [187] I. TOMA, D. FOXVOG, AND M. C. JAEGER, *Modeling QoS characteristics in WSMO*, in MW4SOC '06: Proceedings of the 1st workshop on Middleware for Service Oriented Computing (MW4SOC 2006), New York, NY, USA, 2006, ACM, pp. 42–47. [79]
- [188] UDDI SPEC TECHNICAL COMMITTEE, *UDDI Version 3.0.2*, tech. report, OASIS, 2004. [33]
- [189] J. UDELL, *The browser as orchestrator*, February 2006. [Online]. Available: http://www.infoworld.com/article/06/02/08/74979_07OPstrategic_1.html. [Accessed: August 19, 2009]. [12]
- [190] UNCEFACT AND OASIS, *ebXML Business Process Specification Schema Version 1.0.1*, 2001. [39]
- [191] *Uppaal*. [Online]. Available: <http://www.uppaal.com/>. [Accessed: August 21, 2009]. [64, 82, 225]
- [192] W. VAN DER AALST, *Don't go with the flow: Web services composition standards exposed*, IEEE Intelligent Systems (2003). [38, 80]
- [193] W. VAN DER AALST, M. DUMAS, AND A. TER HOFSTEDÉ, *Web service composition languages: old wine in New bottles?*, in Euromicro Conference, 2003. Proceedings. 29th, Sept. 2003, pp. 298–305. [38]

- [194] W. M. P. VAN DER AALST AND K. VAN HEE, *Workflow Management: Models, Methods, and Systems*, MIT Press, 2002. [38]
- [195] S. VINOSKI, *RPC and its Offspring: Convenient, Yet Fundamentally Flawed*, March 2009. QCon. [95]
- [196] W3C HTML WORKING GROUP, *XHTML 1.1 - Module-based XHTML - Second Edition*, tech. report, W3C, 2007. [234]
- [197] W3C HTML WORKING GROUP, *Extensible markup language (xml) 1.0 (fifth edition) - w3c recommendation 26 november 2008*, tech. report, W3C, 2008. [234]
- [198] W3C HTML WORKING GROUP, *HTML 5 - A vocabulary and associated APIs for HTML and XHTML - Editor's Draft*. Editor's Draft, November 2008. [234, 236]
- [199] W3C WEB SERVICES ARCHITECTURE WORKING GROUP, *Web Services Glossary*, tech. report, W3C, 2004. [31]
- [200] *W3C XML Schema Definition Language (XSD)*. [Online]. Available: <http://www.w3.org/TR/xmlschema11-2/>. [Accessed: August 19, 2009]. [216]
- [201] H. WANG, P. TANG, AND P. C. K. HUNG, *Rlpla: A reinforcement learning algorithm of web service composition with preference consideration*, Services Part II, IEEE Congress on 0 (2008), pp. 163–170. [60]
- [202] WEB ONTOLOGY WORKING GROUP, *OWL Web Ontology Language - Overview*, tech. report, W3C, 2004. [35]
- [203] *Web Services Activity*, 2008. [Online]. Available: <http://www.w3.org/2002/ws/>. [Accessed: September 15, 2009]. [32]
- [204] M. WESKE, *Formal foundation and conceptual design of dynamic adaptations in a workflow management system*, in System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on System Sciences, Jan. 2001, pp. 10 pp.–. [60, 61]
- [205] S. J. WOODMAN, D. J. PALMER, S. K. SHRIVASTAVA, AND S. M. WHEATER, *A system for distributed enactment of composite web services*, in Work in progress report, Int. Conf. on Service Oriented Computing, 2003. [69]
- [206] D. WU, E. SIRIN, J. HENDLER, D. NAU, AND B. PARSIA, *Automatic Web Services Composition Using SHOP2*, in In Workshop on Planning for Web Services, 2003. [57]

-
- [207] *Xerces*, 2005. [Online]. Available: <http://xerces.apache.org/>. [Accessed: September 15, 2009]. [239]
- [208] XHTML 2 WORKING GROUP, *XHTML Modularization 1.1*, tech. report, W3C, 2008. [234]
- [209] XML PROTOCOL WORKING GROUP, *SOAP Version 1.2*, tech. report, W3C, 2007. [32]
- [210] XML QUERY WORKING GROUP, *Xquery 1.0 and xpath 2.0 data model (xdm)*, tech. report, W3C, 2007. [239]
- [211] L. XUANZHE, H. YI, AND L. HAIQI, *Towards Service Composition Based on Mashup*, in IEEE Congress on Services, Salt Lake City, UT, USA, July 2007, pp. 332 – 339. [9, 10, 12, 22, 29, 37, 52]
- [212] Z. YANG, J.-B. ZHANG, J. TAO, AND R. K. L. GAY, *Characterizing Services Composeability and OWL-S Based Services Composition*, in GCC, 2005, pp. 244–249. [79]
- [213] L. ZHANG, W. YUAN, AND W. WANG, *Towards a framework for automatic service composition in manufacturing grid*, in GCC, 2005, pp. 238–243. [57]
- [214] M. ZUR MUEHLEN, J. V. NICKERSON, AND K. D. SWENSON, *Developing web services choreography standards - the case REST vs. SOAP*, Decision Support Systems **40**, no. 1 (2005), pp. 9 – 29. [9]

LIST OF FIGURES

1.1	Relation between the thesis's main chapters.	4
2.1	Late binding: The difference between abstract services (AS) and concrete services (CS).	7
2.2	Request/response pattern as communication principle for provider and requestor.	8
2.3	General roles of mashups.	13
2.4	General Ajax model.	14
2.5	Cross site scripting: Violating the same origin policy.	15
2.6	Cross site scripting: Usage of a proxy server.	15
2.7	Server-side mashups: Architecture.	18
2.8	Client-side mashups: Architecture.	19
2.9	Classic roles of a SOA.	30
2.10	Top level of the service ontology according to OWL-S.	36
2.11	A correlation set in BPEL.	41
2.12	Relation between choreographies and orchestrations.	43
3.1	Classification of service composition approaches.	55
3.2	Different classes of service composition (CS) algorithms based on available knowledge.	60
3.3	Requirements for the underlay system.	72
3.4	Extending the mashup architecture.	74
4.1	Abstract depiction of a possible service composition.	85
4.2	(a) An according workflow graph. (b) A dataflow graph representing the I/O passing.	86
4.3	Basic control structures of automata semantics. (a) Nondeterministic selection. (b) Choice. (c) Looping.	87
4.4	Timed automaton structure ensuring parallel execution.	88
4.5	Extended timed automaton structure ensuring parallel execution.	89
4.6	(a) Origin part of the workflow graph. (b) Modification of the workflow graph for the protected transition. (c) Timeout construct.	92

4.7	Synchronized unicast message.	95
4.8	Passing data via synchronous channels.	96
4.9	Multicast message with queued signals.	97
4.10	Multicast message with updated signals.	98
4.11	Multicast message with temporally blocked sender.	100
4.12	Multicast message with discarded signals.	100
5.1	Three layers of service descriptions.	106
5.2	Example for semantic service descriptions.	109
5.3	Cyclic dependencies between I/O parameters.	118
5.4	Elements of the simulation setting.	127
5.5	Comparison of the success rates for blind forward chaining (BF), blind backward chaining (BB), heuristic forward chaining (HF), heuristic backward chaining (HB), and the novel algorithm for the automatic service composition creation (ASC) introduced in this chapter.	129
5.6	Abstract overview of the simulation setting.	130
5.7	Functions operating on the composition algorithm's pool size.	132
5.8	Comparison of the algorithms' success rate when the service repository's size is known.	133
5.9	Duration of the single functions in average, where the initial pool size is optimally chosen in dependence of the service repository's size.	134
5.10	Different optimal pool sizes depending on the service settings.	135
5.11	Success rates for all algorithms based on their initial pool size.	135
5.12	Deviation of the algorithms' success rates depending on their initial pool size.	136
5.13	Average deviation of the algorithms' success rate depending on their initial pool size.	136
5.14	Deviation of the algorithms' duration depending on the initial pool size.	137
5.15	Average deviation of the algorithms' duration depending on the initial pool size.	137
6.1	Visualization of a possible resource scope for a multi-client geo-caching game.	142
6.2	Colored transitions. A , C , D and H build the first partition (blue dashed arrows), B , E , F and G the second one (orange bold arrows).	144
6.3	The two partitions of the example workflow shown in 6.2 after every step of the partitioning algorithm.	148

6.4	Example of a colored workflow graph.	151
6.5	Two partitions derived from the workflow depicted in Figure 6.4.	157
6.6	Exemplary workflow as depicted in Figure 6.4, but with different coloring.	158
6.7	Two partitions derived from the workflow graph depicted in Figure 6.6.	158
6.8	Parallel execution block. (a) Exemplary colored parallel execution block. (b) First partition of the parallel execution block. (c) Second partition of the parallel execution block. . .	159
6.9	Modified partitions of the parallel execution block shown in Figure 6.8 (a). (a) Shows the modification of the partition depicted in Figure 6.8 (b). (b) Shows the modification of the partition depicted in Figure 6.8 (c).	160
6.10	Resolving nondeterministic choices. (a) A nondeterministic choice between three transitions. (b)-(d) Three possible partitions.	161
6.11	Resolving nondeterministic choices with a master automaton. (a) The master partition derived from the nondeterministic automaton shown in Figure 6.10(a). (b)-(c) The two slave partitions derived from the nondeterministic automaton shown in Figure 6.10(a).	162
7.1	Distributing Web applications: General overview.	166
7.2	Component overview.	168
7.3	Embedding of the orchestration synchronization protocol (OSP).	181
7.4	OSP Negotiation phase - Activity diagram.	183
7.5	OSP Execution phase - Activity diagram.	188
7.6	Embedding of the OSP's negotiation phase.	190
7.7	Embedding of the OSP's execution phase.	194
7.8	OSP: Service recovery - Sequence diagram	196
8.1	Mapping abstract services to concrete services.	198
8.2	UML diagram of the workflow model.	202
8.3	UML diagram of the workflow model's extension towards inter-workflow communication.	205
8.4	UML diagram for the consideration of resource-oriented actions.	206
8.5	UML diagram: Resource hierarchies.	207
8.6	UML statechart diagram of the workflow execution engine.	212
8.7	The workflow execution engine.	213
8.8	Receiving Messages: UML Class Diagram.	215

8.9	Sending Messages: UML Class Diagram.	216
8.10	UML diagram of the discovery module.	219
8.11	UML diagram of the composition module.	220
8.12	A Web client that enables users to request a mashup in an effect-driven way (left) and the relating workflow graph.	221
8.13	<i>sendAround</i> Demo: Sending Places around that surround you.	223
8.14	ScatterPoker Table	224
8.15	ScatterPoker: Workflow graph of the underlay system running on a community device such as a TV screen.	226
8.16	ScatterPoker: Workflow graph of the underlay system running on mobile user devices.	226

LIST OF TABLES

2.1	Comparison of proxy mashups (PM) and client-side mashups (CM).	21
2.2	BPEL: Basic activities.	40
2.3	BPEL: Structured activities.	40
2.4	WS-CDL: Types.	43
2.5	WS-CDL: Activities.	44
2.6	Business processes vs. mashups.	45
5.1	Methods for the creation of underlay systems.	104
5.2	Simulation settings for request difficulties.	130
5.3	Simulation settings for the generation of services.	131
5.4	Functions operation on the algorithm's pool size.	132
6.1	Partitioning Algorithm: Content of the key structures.	149
6.2	Partitioning Algorithm: Calling the getNext() method.	149
7.1	Parameters and data types.	170
7.2	Error codes.	171
7.3	OSP component: Request processor.	172
7.4	OSP component: Markup validator.	173
7.5	OSP component: Underlay creation engine.	174
7.6	OSP component: Late binding engine.	175
7.7	OSP component: Service repository.	176
7.8	OSP component: QoS Evaluator.	177
7.9	OSP component: User preferences evaluator.	177
7.10	OSP component: Call generator.	178
7.11	OSP component: Partitioning engine.	178
7.12	OSP component: Underlay execution engine.	180
7.13	OSP component: View controller.	180
7.14	OSP: Negotiation messages.	191
7.15	OSP: Execution messages.	192
7.16	OSP: Synchronization messages.	192
8.1	Mapping for Google Maps (static).	200
8.2	Attributes of locations.	203

8.3	Attributes of transitions.	203
-----	------------------------------------	-----