universität wien

MASTERARBEIT

Titel der Masterarbeit

REMUS

-

A REstful Marketplace for Unified Services

Verfasser

Ralph Vigne, Bakk.rer.soc.oec. BA

angestrebter akademischer Grad

Diplom Ingenieur (Dipl.-Ing.)

Wien, 2011

Studienkennzahl lt. Studienblatt:	A 066 926
Studienrichtung lt. Studienbuchblatt:	Masterstudium Wirtschaftsinformatik
Betreuer:	Univ.-Prof. Dipl.-Ing. Dr. techn. Erich Schikuta

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Wien, den 15. Juni 2011

Unterschrift:

_____

Ralph VIGNE

Contents

Chapter 1

Introduction

1.1 Motivation

Over the last decades outsourcing of certain tasks within a business process has become a broadly used technique to allow companies to focus on their core businesses. Greaver [1, p. 4ff] extracted 20 reasons for this behavior by doing a survey over a broad sample of companies. Out of these twenty, we[1] name five principal reasons which were targeted in this thesis.

- Enhance effectiveness.
- Increase flexibility.
- Obtain new expertise and technology.
- Reduce investments and generate cash by transferring assets to provider.
- Turn fix costs into variable costs (pay-per-use).

Stricker et al. [2, p. 1] argued that efficient outsourcing and subcontracting still leads to higher profits by "[...] letting others do what they do better and/or cheaper". Schubert et al. [3] made a more IT/Cloud focused analysis and stated that outsourcing allows companies to ...

- ... gain more flexibility when reacting to external influences.
- ... reduce the amount of capital needed to explore new business areas.
- ... reduce the carbon footprint by using more efficient resources and improve their workload.

Among other things these are the reasons why companies are organizing their businesses into cross-organizational business processes and using workflow management systems (WfMS) (see e.g. [4, 5, 6, 7]). Beyond WfMS, outsourcing and distribution also influences new software development. Today's enterprise application software (EAS) also needs to consider distribution and interoperability aspects. Therefore a raising attention to the 'Software as a

[1] **Usage of Active Voice in this Thesis:** In this thesis I use the "we" to express the active cooperation within the research group "Workflow Systems and Technologies", however this thesis has been solely created by myself, and all unique contributions are clearly outlined in Section 9.1.

Service' (SaaS) and 'Service Oriented Architecture' (SOA) approaches can be observed over the last years.

Some companies developed business models focusing completely on providing services that other companies may outsource and therefore started a new market. This business concept origins in the idea of Portals (e.g. for booking a flight) and led to multi-layered cross-organizational business workflows [8, 9]. eBusiness Grids were created to act as a market place for such services. But latest investigations [3, p. 44ff] showed that these eBusiness Grids are going to be superseded by SaaS Clouds, and therefore reaching their next level. The main reasons for this according to Schubert et al. are:

- Cloud developments are driven by the industry instead of the scientific community and therefore have a sustainable business model (see also [10, 11, 3]).
- Clouds are designed to be very scalable, therefore are more suitable for throughput focused systems, and thus avoid availability problems.
- Participating service or resource vendors are allowed to keep their own interfaces because Clouds are designed to handle heterogeneity.

Jha et al. [11] further explain why it is necessary for a system, which is intended to act as a market place for services, to provide **two different kinds of services:** (1) high level services, which offer a more powerful service and aggregate the interfaces and combine the execution of a number of (2) low level services for simpler usage. They further state that multi-layered infrastructures can be realized better by using Clouds than Grids because of their higher abstraction level.

Because target platforms of these markets **shift from Grid to Cloud infrastructures** the need for new systems to utilize them is created. According to Schubert et al. [3, p. 45] the development of such systems is one of today's upcoming challenges. They identified one of Europe's main opportunities in the 'Cloud movement' in the further investigation of concepts and systems for a **'free market of IT services'** based on resources and services offered via Cloud infrastructures on a global market.

Schubert et al. further states that one of the main characteristics of these markets is that the **costs are shifted from Capital Expenditure (CAPEX) to Operational Expenditure (OPEX)** and resulting in lowered entry and exit barriers. By shifting the costs to OPEX the investment for entering a market is reduced which in turn lowers also possible losses if the business fails. This leads to a higher number of participants and therefore to more competition (see M. E. Porter [12] for economic details). Due to the lower enter and exit barriers and high competition these markets are also high innovative resulting in many newly developed ideas. An example for this relation is of course the many different 'App Stores' which emerged throughout the Web in the last few years. Two successful examples are the stores from Apple [13, 14] and Google [15, 16]. Both benefit from their easy-to-use and pay-on-demand strategy with low barriers.

Based on a literature study we identified that currently there is a big shift from Grids to Clouds taking place (see e.g. [3, 11]). Therefore new systems and concepts are needed to **offer (IT) services and resources on a global market** (e.g. [1, 2, 3, 8]). In order to fulfill market requirements these new systems exploit advantages and new features/concepts of Clouds.

While above we focused on outsourcing, which is a business-to-business (B2B) application, we map the concept on **business-to-consumer (B2C) applications**. As mentioned by Jha et al. [11] systems should provide services on different levels. In this thesis this concept is extended by building business cases around services and offering them within application domains. The vision is to create a system that provides different application domains (e.g. Cinema tickets, Computing resources, ...) and allow consumers to include them into their own processes. The application domains contain services which are selected by the user at runtime as they offer identical abstract functionality. The resulting interchangeability is equal to market transparency and thus enhances competition between vendors within an application domain and therefore leads to better service conditions for customers. It is further envisioned to **trade these application domains and business processes** like other web services. Similar to the mentioned App Stores (e.g. Apple, Google) anybody should be allowed to create application domains, extend them with own and external services and offer them to other users.

In order to proof the feasibility of the above described concepts a prototype implementation consisting of two main parts has been created. First, the **Repository** which manages and provides information about the application domains and services. It offers access via a simple RESTful-API. Second, an **Injection Service** which inserts information into the users process on-demand during its execution. In order to utilize the services in the Repository additionally a smart phone application was developed.

1.2 Problem Statement

Today a lot of similar but heterogeneous services are provided over the web. In order to use them in an interchangeable/arbitrary way and execute/provide them within Cloud infrastructures, the following problems have to be solved:

- Using services within a business process, without knowing the service interface, has to be compensated by **complex controlflows**, which makes it difficult to comprehend the original intend of the process. Further more maintenance (design-time) costs for processes are directly related to the complexity, which additionally increases costs.

- Whenever services evolve, business process using them, have to be also changed. This leads to very high business process maintenance costs for customers and to very **complex and expensive update strategies** for service vendors and/or customers.

- In order to support **vendor autonomy** and support the **reuse of existing services** when developing new business cases, it is beneficial to describe services in multiple-level (abstract/concrete) as proposed by Jha et al. [11].

- When a lot of similar services are offered, a way to **explore the differences** between them is needed. In order for the user to select particular services at run-time, services have to expose a common set of properties to support the decision-making.

1.3 Justification

A strong economic interest in using Cloud infrastructures for outsourcing is stated in many of today's publications (e.g. [1, 2, 3, 8]).

One reason is that companies want to **cooperate with other companies** (outsource) to be able to offer their products/services at predictable prices. But because cooperating with other companies means that their business processes become cross-organizational and therefore more complex, systems **assisting execution and maintenance** of theses processes are needed.

It may also be in the interest of many companies to keep parts of their business processes in private in order to **protect a valuable business asset**, i.e. process logic. Therefore it is important that each vendor may decide on its own how much information about the business process of a service is provided to its customers. Vendors can keep their **autonomy** by providing an abstract description for their services. Thus when changing the concrete service interface costumers are not affected as long as the abstract description is updated.

Because competition on a transparent market usually **increases quality and decreases prices** (market equilibrium) a common marketplace for such services is desirable.

In this marketplace each vendor describes the offered service according to a common abstract schema to **make the services comparable** for the customers. Based on this description customers decide which services/vendors fit best for them. The motivation for this is to allow customers process engines to **decide automatically at run-time which service to use**, using the properties schemas and service descriptions of each offered service.

Typically whenever a service is used by a customers process engine, additional components (i.e brokers, semantic matchers) are employed to match service calls to heterogeneous service interfaces. In order to avoid this middle layer the approach presented in this thesis proposes services to be described by a microflow, representing the protocol to interact with it, which can be injected directly into the controlflows of customers processes. Doing so provides the costumers process engine with all information needed to interact autonomously with the chosen service(s), and therefore does **not depend on any additional software components or service**.

The overall goal of this thesis is to show that the explicit use of **concepts provided by Cloud infrastructures** like e.g. scalability, heterogeneity can lead to higher resource efficiency, reliability and cost reduction.

1.4 Structure of the Thesis

So far we have argued why we have written this thesis by educing new trends and economic interests in this field (Section 1.1). We continued with an explanation of what we contribute to this field by naming problems addressed within this work (Section 1.2) and justify why we intend to solve them (Section 1.3). We close this chapter with a **Definition of Terms** (Section 1.5) to get a common understanding of the important concepts used within this thesis.

In the next Chapter (Chapter 2) we give an overview about today's publications and systems similar to our implementation. Because our system consists of different components we divided this chapter into different parts. The first part (Section 2.1) covers the Repository, the second part (Section 2.2) shows different approaches related to **Workflow Adaption** and the third part (Section 2.3) consists of a detailed discussion of two systems similar to ours (namely **ACE-flow** and **Adaptive Pegasus**). We discuss several aspects of today's research and systems by extracting their basic ideas and compare them to our approach for each component.

Chapter 3 focuses on educing the **Requirements** and the resulting **Architecture** for a system meeting the goals derived in the **Problem Statement** (Section 1.2) and **Related Work** (Chapter 2). We start with discussing our **Analysis** (Section 3.1) of each service/component included in our system and relate it to requested features and concepts from above. After the purpose of each service/component is cleared we combine them by defining the deployment of and the interaction between these services/components. At the end of this section the overall **Architecture** (Section 3.2) and all involved service/components of the system are defined.

Chapter 4 focus completely on the **Repository for Services**. At the beginning we further detail the architecture of our Repository by introducing the included **Resources and Levels**. In succession we show how they are interrelated. Next is the definition of an example (namely the **Cinemas Example**) which we will use as a running example through out this whole work (Section 4.1). Using always the same example allows us to explain each component with familiar parameters and operations and therefore makes it easier to understand. Using this example we start explaining what information is stored and provided at **Class Level** (see Section 4.2). We discuss in detail what Class Level Operations, Schemas for Input and Output Parameters, Templates, Properties and Service Schemas are and how they are defined. This leads directly to the **Instance Level** (see Section 4.3) where we show what information about a particular service is stored and how it is structured. An explanation of the **RESTful API** (see Section 4.4) showing how to find resources and to maintain the data finishes this chapter.

Chapter 5 is dedicated to the language we developed for our prototype implementation to describe the different microflows provided by the Repository. We start introducing it with an **example** for a class level microflow (Section 5.1.1) and an instance level microflow (Section 5.1.2), both related with the familiar Cinema example. **Context elements** are covered in Section 5.2. A comprehensive discussion about the **Call statement** is given in Section 5.3. An excursus covering the syntax of a **Call statement in CPEE syntax** is given in Section 5.3.4. The **Manipulate**, **Choose - Alternative**, **Groups and Conditions**, **Loops**, **Parallel - Branch** and **Critical** statements are explained in the Sections 5.4 - 5.9.

Chapter 6 introduces the **Injection Service** we developed for our system. We start with a step-by-step **Example** illustrating what happens during an injection in Section 6.1. After the desired overall functionality is cleared we continue with explaining the implemented **Interface** of the service in Section 6.2. A detailed discussion about the general **Algorithm** (Section 6.3) and how and why **Loops** must be treated differently (Section 6.3.2) finishes this chapter. For better understanding of the algorithm this chapter provides pseudo-code listings and XML snippets additional to a comprehensive description of the included operations and elements.

Chapter 7 focuses on the component called **Injection Handler**. This component is intended to coordinate injections for a WEE/CPEE instance. Whenever a call causes an injection it

registers itself at the Injection Handler using the **Interface** described in Section 7.1. We will explain this interface in detail and cover every message which is sent to or received from it. Next we introduction the **Algorithm** (Section 7.2) developed for this component to work the way we need it to fit in our overall system. Finally we discuss what **Adaptions to the CPEE Activity Handler** we made in Section 7.3 to support run-time service selection by **Injection Calls**.

Last thing we show is an implementation of a **Mobile Client** to illustrate our concept (Chapter 8). We start with describing the overall **Architecture and Deployment** of the involved components in Section 8.1. This section further explains how and when all components interact together. Next is to extend the familiar Cinemas example to make it more usable for the real world (Section 8.2). A detailed discussion of the implemented **Wallet** component is covered by Section 8.3. We explain how the user is able to define and store personal preferences inside it. We finish this chapter with an explanation of the **Worklist** (Section 8.4) we implemented for our client application. A description of its RESTful API and the protocol it follows are the main parts of this section.

We finish with a **Conclusion** (Chapter 9) over this work. We will discuss what we **Contributed** (Section 9.1) to this field by writing this thesis and what **Future Research** (Section 9.2) will be. At the very end we recap about what **(personal) lessons we have learned** over the time of writing this thesis (Section 9.3).

1.5 Definition of Terms

In this section we provide definitions about how we understand and use certain terms and concepts in the context of this work to get a common understanding.

Lewis et al. stated "We define **business processes** as a set of logically related tasks performed to achieve a defined business outcome." [17, p. 100] To the technical representation of such a business process we will further refer to with **workflow/controlflow (description)**.

A definition of **Cloud** we highly agree with is made by Schubert et al. [3]: "[...], we can define a 'Cloud' is an elastic execution environment of resources involving multiple stakeholders and providing a metered service at multiple granularities for a specified level of quality (of service)."

Heterogeneity is defined by the Oxford Dictionary [18] as "diverse in character or content" and therefore often used to describe the diversification of resources in Cloud infrastructures (e.g. [3, 19, 20, 11]). In the context of this thesis we understand heterogeneity the same way and use it to refer to diverse service interfaces (e.g. parameters and interaction protocols).

Fox et al. combined the understanding of business processes and services executed in a Cloud environment and defined **workflows in the Grid context** as follows: "The automation of the processes, which involves the orchestration of a set of Grid services, agents and actors that must be combined together to solve a problem or to define a new service." [21, p. 2]

According to Greaver [1, p. 3] "**Outsourcing** is the act of transferring some of an organization's recurring internal activities and decision rights to outside providers, as set forth in a contract. [...] As a matter of practice, not only are the activities transferred, but the factors of production and decisions right often are, too. **Factors of production** are the resources

that make the activities occur and include people, facilities, equipment technology, and other assets. **Decision rights** are the responsibilities for making decisions over certain elements of the activities transferred." During the execution time of this business process the consumer creates one virtual organization with each of the involved providers.

Virtual Organizations are a time-limited cooperation between one or more business partners. These cooperations are built and dismantled dynamically and a reaction of the companies to rapidly evolving markets. In a virtual organization each business partner offers one or more services (of his core business) to support the business process of its customer. We based this definition on the one given by Grefen et al. [6]. They further reused their definition of virtual organizations to define **workflow support in virtual organizations** by the means that the management systems in different organizations must be linked to manage cross-organizational processes.

Software as a Service (SaaS) Clouds, according to Schubert et al. [3] are identified by "[...], offering implementations of specific business functions and business processes that are provided with specific Cloud capabilities, i.e. they provide applications/services using a Cloud infrastructure or platform, rather than providing Cloud features themselves."

An **inter-organizational workflow**, according to Stricker et al. [2] is defined as a workflow where "[...] part of the work is executed outside the company's boundaries."

The term **Microflow** has its origin in the field of network protocols where Nichols et al. [22] defined it as follows: "A single instance of an application-to-application flow of packets which is identified by source address, destination address, protocol id, and source port, destination port (where applicable)." It was than redefined for Business Process management (BPM) by Lambros et al. [23, p. 24]: "Microflows can be used to implement operations of Web services that require coordination of a set of business components. The microflow would describe the sequence of actions to be taken to perform the business function provided by the operation." We differ from this definition insofar that we see a microflow as the way an offered service or resource of a particular vendor can be used, **allowing all kind of controlflow structures** instead of sequences only. Also is our definition of a microflow stated on a **higher abstraction level** because we describe a piece of business logic/use case instead of a service interface.

As **third-party/external services** we understand such services or resources that are offered from an external vendor, traded via a market and used within a workflow process. Vendors follow by the implementation of their service(s) the SaaS approach to make them easy-to-use for their costumers.

We define an **Injection** as the extension/adaption of an existing program with parts provided by third-party/external sources. Doing so at run-time allows to adapt the software dynamically depending on its actual state. Based on this we use the term **Microflow Injection** to describe the extension of a business process with microflows defining the interaction with third-party/external services. Because we do this at run-time, this technique is strongly related to **Adaptive Workflow Execution Engines** as they allow to change the controlflow during process execution.

Late Binding is defined by G. Joeris [24] as follows "The decision is taken at run-time, which process definition of a task definition is used to perform a task (late binding)." We therefore refer to this concept whenever the controlflow of a business process is extended at run-time with a microflow.

Representational State Transfer (REST) introduced by Fielding [25] is an architectural style that describes how a client and server communicate with each other and how information will be provided. Summarized it is a HTTP-based communication where the client initiates a *request* to a server which processes it and *responds* the result. Requests are generally built around the transfer of *representations* of *resources*.

As an **Application domain** we see a number of services or resources which are capable to achieve the same pre-defined goal. Each service or resource within an application domain must be completely interchangeable for the consumer.

Whenever we refer to **context elements/variable** we address objects placed in the execution context of a business process. They can be read and written by all activities in the controlflow and also from the outside using the RESTful API.

Chapter 2

Related Work

In this section we discuss research related with this field to get a deeper understanding of the concepts behind it. Further do we discuss already implemented systems and compare them to our approach.

The first part deals with different concepts of service repositories/marketplaces. We give a detailed discussion about the techniques used by these implementations and discuss them in relation to our approach. The next part focuses on the categorization of adaptions performed on workflow descriptions. We further discuss in which and why our approach fits into different categories and what defines them. In the last part we discuss our approach in relation to implementations using similar concepts as we do, namely ACE-flow [2] and *Adaptive Pegasus* [26]. We summarize this comparison in Table 2.2 at the end of this chapter.

2.1 Marketplace/Repository

This section is also published in Vigne et al. [27].

A repository always manages metadata related to services. As pointed out in [28, 29] metadata management covers the following topics:

1. Distribution, uniformity of access.
2. Metadata should be accessible using service-oriented protocols.
3. Management of the metadata life-cycle.
4. Granular and uniform access control to metadata.

Comparing the above mentioned concepts the marketplace covers the topics 1, 2 and 4. As described in this thesis, vendor maintained metadata (instance level) is available through the RESTful API. A common, reliable schema for the metadata is accessible at class level. The topic *Management of metadata life-cycle* is not covered by the marketplace. The client needs to take care about this by itself, e.g. using the techniques associated with ATOM feeds.

Bernstein et al. [30] pointed out that a repository should store information about the description of objects and the location of objects. They further stated that the most important

aspect of a repository is a simple interface to interact with it. We came to the same conclusion and therefore provide a resource oriented and RESTful API. It allows customers to request ATOM feeds, listing the provided resources and further to request the schemas and microflows describing each resource.

Additionally Jha et al. [11] claimed that a system, which is intended to act as a marketplace for services, must provide two different kinds of services: (1) high level services, which offer a more powerful service and aggregate the interfaces and combine the execution of a number of (2) low level services for simpler elaboration. The usage of high level services is typical for Cloud computing scenarios. Our service marketplace closely resembles this by providing class and instance level.

	UDDI	UDDI + OWL-S	Marketplace
Discovery			
Groups	Yes	Yes	Yes
Nested Groups	Implicit	Explicit	Explicit
Attributes	Yes	Yes	Yes
Browse Pattern	1	1	1
Drill-Down Pattern	n	n	m
Invocation Pattern	Yes	Yes	Yes
Selection			
Based on	Attributes	Semantics	Usage
Binding			
Description Language	WSDL	WSDL	abstract definition
Focused Item	Service	Service	Use-Case
Interface Unification Pattern	Client	Client/Server	Server
Knowledge About Item Usage	Client	Client	Server

Table 2.1: Comparison: UDDI vs. UDDI + OWL-S vs. The Repository

A common technique to explore services provided over the web is UDDI [31]. In UDDI it is possible to register services in different categories depending on the supported business. Customers searching for services can request three different types of information about them:

White Pages: include vendor data e.g. address, name, ...

Yellow Pages: represent the industrial categories using taxonomies like SIC [32], NAICS [33], UNSPSC [34], ...

Green Pages: include technical information about the service usage e.g. SOAP interface [35], WSDL [36], ...

But as stated by Blake et al. [37] most of the resources consumed by UDDI are used for message parsing and transmissions. UDDI's weakness is, that it is just a flat list of services. Finding similar services depends on semantic annotations on the actual services. A detailed discussion about the feasibility of UDDI in the context of service repositories is given in Vigne et al. [38].

Our approach also shares many goals with OWL-S [39]. OWL-S is used to bring semantic annotations to UDDI. Instead of simple querying for service parameters, it allows for sophisticated matchmaking. It furthermore provides a process model that describes how to use a service to achieve certain goals. It as well provides information for the actual execution. In Tab. 2.1 we compare the concepts of UDDI, UDDI extended with OWL-S and the Marketplace.

All three concepts allow for service groups, whereas nested groups (tree structure) are only implicit in UDDI (by creatively using additional parameters). All three techniques support

annotation of **attributes** to services. These attributes can be used when specifying a query. The **browse pattern** refers to returning a compact service list without all the details. The marketplace only differs from the other two solutions in the format of the returned data (ATOM). The **drill-down pattern** refers to returning detailed information for actual services. As the marketplace ensures the technical compatibility of all possible services, no services have to be excluded for technical reasons. Thus it returns potentially more services ($n \leqslant m$, see Tab. 2.1 for details). The invocation pattern is supported by all three solutions. **Service Selection** holds the biggest conceptual difference between the three techniques. The marketplace focuses on "usage" or "use cases" instead of services (functionality first, services later), all services associated with a functionality fully support the use case. **Interface unification** means that different services with slightly different syntactic properties are to be used. Whereas for UDDI this has to be ensured by the client, an OWL-S enabled repository delivers information how a client may transform and invoke a service. For an example implementation of OWL-S see Srinivasan et al. [40]. With the marketplace this is not necessary, as both the transformation and invocation is contained in the microflow. By **knowledge about the item usage** we express that for the first two approaches information is returned that has to be analyzed, before calling an actual service. As the marketplace is focused on use cases, this step is not necessary.

2.2 Categorization of Workflow Adaption

Han et al. [41] defined several different categories for workflow adaptions. They identified four different levels where adaptions can influence the workflow structure:

1. **Domain:** Adaption of a workflow system to changing business context.

2. **Process:** Model evolution and ad-hoc changes to model instances.

3. **Resource:** Adjustments at Components & Interfaces, Human resources, Data-related adaption.

4. **Infrastructure:** System re-configuration.

Our approach performs adaptions at the second and the third level. Because we change ad-hoc the binding of resources and tasks within a process (from a **logical perspective**), it is covered by the **third level (Resources)**. But because we see resources as a microflow describing how to interact with them, our approach is also covered by the **second level (Process)** because we ad-hoc change the controlflow description of an instance from a **technical perspective**.

They continue by identifying different mechanisms for adaptive workflow systems (see [41] for details). According to their definition our approach is clearly in line with what they called *Open-point approach*. They identify this type by the **explicit definition of points where changes may be performed**. Because we deal with activities which dynamically bind resources in form of microflows, resulting always in an adaption of the controlflow, they represent these points.

2.3 Similar Concepts

The first system we discuss is introduced by Stricker et al. and named **ACE-flow** [2]. Similar to our system ACE-flow is designed to allow the usage of external services/resources within a business process and supports the late binding approach to allow service selection at run-time.

When in ACE-flow the execution engine requests a service of a particular type the so called **Trader System** performs the negotiation and selection of a particular service without including the execution engine. To do so the execution engine must provide all parameters needed to select and execute the requested service into this request. After a service is selected the Trader System also initiates the execution of the selected service. When the vendor service finished its execution it returns the results to the Trader system which forwards it to the execution engine.

ACE-flow handles a number of things differently than we do. In the following we will explain the advantages/disadvantages of our approach compared to ACE-flow.

In ACE-flow the negotiation with the different **service vendors is kept outside the business process** and performed autonomously by the Trader System. This keeps the process structure more simple than ours, where the microflow covering the negotiation with a service/resource is injected into the business process for each of them separately. On the other hand, from the perspective of the execution engine and process mining, it is more comprehensive to have all these microflows at hand for later analysis. Same holds for service execution. When the Trader System of ACE-flow requests the execution of a service, no information about its execution or interface is included into the business process. Although there are log files, the information is not as easy accessible as it is when the information is included into the controlflow of the business process. An advantage of such a Trader System, if all information for the service call is included in the request, is, that it can **select services where the interfaces fit to the available information** (See also Martin et al. [39] for binding services with UDDI/OWL-S). For example, if one parameter is missing the Trader System can select services which will also work without this particular parameter. In our approach we require each service to have a similar interface or is at least capable of transforming the provided data to fit for its own interface. This makes our approach a little less flexible than ACE-flow. The major advantage of our system is that, in contrast to ACE-flow, we do not need **a component working in the middle** like ACE-flow's Trader System. When the workflow execution requests information from our Repository it receives all microflows of each fitting service. After this the workflow execution engine is able to negotiate with and execute external services on its own. We prefer this approach because of its simpler system architecture and the lower resource usage/dependency. In our loosely-coupled system each component is stateless, allowing to make best use of the principles coming with Cloud infrastructures. An other major difference is that in our system each instance of a business process is allowed to use its own service/component to select the **best service** at run-time. In ACE-flow these functionality is included directly into the Trader System which makes it (a) less flexible and/or (b) extends the parameter set with parameters related to the preferences-function. We argue that these parameters should not be included into an actual service request as they are not directly related to it. Also does using own services to perform this selection allows to react faster and more efficient to external influences and to adjust the preferences-function for each process instance individually.

The second system we discuss is **Pegasus**. This system is also intended to execute workflow processes and use dynamic resources during the execution. As Lee et al. [26] explains, it can use any kind of registry to locate available resources. In the example implementation they use the **Globus Replica Location Service** (RLS) [42] as a service/resource repository and the **Directed Acyclic Graph Manager** (DAGMan) [43] as execution engine. The concept of Pegasus is that before the execution of a workflow process starts, the **abstract workflow** (high-level description) is compiled into a **concrete workflow**. During this compilation requested services/resources in the abstract workflow are mapped to **virtual resources** from the RLS. After the compilation the concrete workflow is executed by the DAGMan. During the execution the DAGMan delegates tasks to defined virtual services/resources in RLS which are executed by fitting available services/resources at the time. Because the RLS includes a Jobmanager, it has the ability to allocate all available services/resources at a time and dynamically change them to provide the optimal execution of the delegated task. After the services/resources finished execution the results are given back to the RLS which passes them on to the DAGMan. What Lee et al. [26] did by **extending Pegasus** to *Adaptive Pegasus* is that they observe the **queuing time** of the different virtual resources in the RLS (using the Jobmanager), and if there is a significant change they request a new compilation from Pegasus. As Pegasus takes the updated information into account this leads to a better overall performance of the workflow. Using this technique they gain a similar behavior to native **late binding** approaches, without loosing the advantage of the workflow optimization.

Similar to ACE-flow, Pegasus does not include any information about the actual services/resources into the workflow description. What Pegasus makes different to ACE-flow and our approach is, that it **optimizes the description of the virtual resources and the concrete workflow** by analyzing the whole abstract workflow to gain a better overall performance. While our approach binds services/resources exactly at the time they are needed (late binding), Pegasus binds them at compile time (early binding), although they are virtual at this time. Its concept to by-pass the problems coming with **early binding** is to perform re-compilations if there are significant changes in the infrastructure observed. As Pegasus is **designed for scientific applications using Grid infrastructures**, which are usually more predictable (Jobmanager provides information about queuing time) and homogeneous (resource clusters) than services/resources offered via a market this approach aims for a different goal than ours in this specific domain. But similar to ACE-flow, an additional component (RLS/Jobmanager) to the execution engine is needed to execute the external service/resource calls. So Pegasus is, like ACE-flow, different at this point from our approach.

In Table 2.2 we summarize the differences and similarities of our approach to the two introduced systems.

	ACE-Flow	Adaptive Pegasus	Marketplace
Interface Unification (Interface Transformation)	NO (Trader System performs semantic match)	NO (RLS covers interface transformation)	YES (Interface transformation included in Microflow)
Parameter Complexity	Service Parameters + Selection Parameters	Service Parameters	Service Parameters
Service Selection (Scope)	Trader System (System)	Jobmanager (System)	Selection Service (Process/Instance)
Service Binding (Binding Approach)	Indirect (early)	Indirect (early, simulating late using virtual resources)	Direct (late)
Service Execution/Interaction	Trader System	RLS	Execution Engine
Intermediary Components during Service Execution	Trader System	RLS, Jobmanager	NONE
Workflow Optimization	NO	YES	NO
Target Environment (Main Application)	Clouds (Business Workflows)	Grids (Scientific Workflows)	Clouds (Business Workflows)

Table 2.2: Comparison: ACE-Flow - Adaptive Pegasus - Marketplace

Chapter 3

Requirements and Architecture

In this section we educe the requirements for our prototype implementation. To meet the goals and concepts defined so far (see Section 1.2 and 2 for details) we discuss (a) the considered functionality in the overall architecture and (b) what each involved software component/service provides. In the end we define how the components must interact to provide the requested overall functionality to proof our concept.

3.1 System Analysis

First: How to make use of **the concepts provided by Cloud infrastructures**? Scalability and reliability aspects are among others in the center of such systems. We try to meet **scalability aspects** by a Service Oriented Architecture (SOA) design. Loosely coupled components acting together as one system allow to distribute and replicate them easily over different resources. To build such a system we need to implement **handler components** which take care of the coordination and distribution of certain other components. Using such handler components allows additionally to switch to a different component if one fails. This fits also for **reliability issues** of our system. Because of the chosen SOA design we identify the following components our system must consist of:

1. a workflow execution engine (WEE/CPEE, see Section 3.1.1)
2. a repository of services (see Chapter 4)
3. a component coordinating and delegating workflow adaptions (see Chapter 7)
4. a component performing the actual workflow adaption (see Chapter 6)

A detailed description of the interaction between these components and how we are going to deploy them is given in Section 3.2.

3.1.1 The Workflow Execution Engine (WEE)

We decided to use the WEE [44, 45] as our target execution engine because of its lightweight design and good customizability. It was designed for Cloud infrastructures, which makes it suitable for our system. The capability to handle controlflow changes during run-time

allows us to inject the services microflows directly into the controlflow and therefore **decreases design-time process complexity** and **increases flexibility** for service vendors. To achieve our goal we needed to customize the Activity Handler of the WEE, which is in charge for the actual service requests, to enable a few things additionally to default supported REST calls.

- Enable SOAP (Simple Service Access Protocol) calls by creating the envelopes and extracting the result.
- Registers the WEE instance for workflow adaptions at the according component and stops process execution.
- Provide all necessary information about the execution instance to enable a proper injection of microflows.
- Extend its return value/object with a status code indicating how the execution of an activity finished. This is similar to the HTTP-status codes (see HTTP Protocol definition [46]).

Details about these adaptions are given in Section 7.3.

3.1.2 The Repository of Services

In our system the Repository is used to store and organize services for later use within business processes. To filter services depending on their properties, it additionally has to provide a way to annotate them. The heterogeneous environment of Clouds must be considered when the language used for service description is designed. It further supports easy client application development by providing additional necessary information for them via a RESTful API. Finally a way to maintain the stored information must be provided.

Organizational Aspects and Service Filtering

As already elaborated by Blake et al. [37] most resources are consumed by message parsing and transmission when using UDDI [31]. They also show a strong performance decrease for repositories with a high rate of changes because they need to find the differences since the last request. We use ATOM feeds [47] to provide organizational information in a RESTful way using the URI to differ between them. We assume server-side version control is not necessary, because ATOM feeds include information about the last update of entries, it is well supported to implement a version control on the client side.

According to Jha et al. [11] a multi-layered architecture is needed which we will achieve by **organizing the repository in different layers**. In respect to this demand and out of usability and flexibility reasons we define the following requirements:

- A layer representing an application domain, which we further refer to as **class level**, providing the domain interface is needed. The interface contains ...
  - ... a **properties schema** for the included services.
  - ... **at least one operation** that can be executed. Each is described by a microflow including calls to actual service operations or class level operations within the same

application domain. By supporting the aggregation of class level operations we can easily create operations with different granularity.

  - ... **service schemas** a vendor can validate his description against. Each of these schemas contains a list of properties needed to be filled and operations needed to be provided.

- A layer where the actual services are defined. We further refer to it as **instance level**. At this level each service defines ...

  - ... **service operations** defining all transitions defined in the Workflow Activity Model (WAMO) [48, 49] using microflows including actual service calls. Each operation used within any class level operation of the application domain has to be defined this way.

  - ... **values** for all defined properties in the schema provided at the according class level.

- For better granularity within an application domain we define one more layer placed between the other two. We are further referring to this layer as **subclass level**. Its purpose is purely organizational and provides no more technical information about its services than already provided by its class level. It allows vendors to narrow down their set of services and increase query accuracy for customers leading to less bandwidth consumption.

All member services must fulfill the scheme of the application domain in means of providing the operations (including parameters) used within the class level operations and defined properties. To avoid inconsistencies between the schema for the domain/services and to reduce the maintaining effort, a way to generate them automatically on demand is needed. Further must be guaranteed that each service is validated against the according schema when it is registered or updated. Doing so ensures that only valid data is provided by the Repository and all services within the same application domain can be used arbitrary.

A schema for **service filtering** is proposed by Sirin et al. [50, p. 44] which states three different levels:

1. The **service profile** describes what functionality a particular service provides (e.g. yellow-pages of UDDI). We realize this with the already introduced application domains and properties in the Repository.

2. The **process model** describes how this functionality is provided and how much control the customer of the service has in terms of granularity (e.g. the process model of BPEL4WS [51]). This is realized with the class and instance level microflows provided by the Repository.

3. The **grounding** defines the protocol an agent needs to support when interacting with the requested service (e.g. WSDL [52] mapping). As part of the microflow each call to a service provides information about the requested protocol, but it is up to the WEE to implement it. Therefore we came up with the already described extensions for the WEE's Activity Handler.

We further need to consider a way of **selecting services** depending on their properties. As we excluded SLA negotiation explicitly from this prototype implementation, we perform the

selection during the actual injection and/or within an explicit service. To do so, a way to define **service constraints** within the business process must be provided. These constraints should be capable to refer to fix-values or context elements and compare them using any supported operation with the properties (only leave nodes) given in a Xpath-styled syntax [53].

Heterogeneity Aspects: Parameter Transformations and Microflows

Because Cloud infrastructures include heterogeneous resources we must provide a way to describe services and how to interact with them in a unified way. First we split services into two parts: (1) their **interfaces** containing in- and output parameters and (2) the **microflows** describing the way to interact with them. This concept is also implemented by UDDI/OWL-S (see Martin et al. [39] and Section 2.1 for details).

Our idea is that similar services may need and produce similar data but they may have different names, types and formats and some may need less and other more information than provided by the according business process. We face this problem by defining one common interface for each application domain named **class interface**. This interface should include enough information to fit for most of the available services.

If the class interface is given:

- A service expects some parameters included in the class interface under different names. Therefore we must provide a way to **rename parameters**.
- A service expects some parameters included in the class interface but with a different data type. Therefore we must provide a way to **define and change types** of parameters.
- A service expects some parameters included in the class interface in a different format e.g. a date string in UTC format instead of ISO 8601. Therefore we must provide a way to **define and change parameter formats**.
- A service expects parameters that are not included in the class interface. Therefore we must support to define a service call using **different parameters than the class interface**. If a service expects less it must be possible to use only a subset of the class interface for the actual call. If a service expects the **parameters/information provided in a different form** e.g. a date string is provided but separate day, month and year strings are expected, it must be supported to split or aggregate them. Last if a service expects more information then provided, it must be possible to use fix (static) values or the value of a context element from its own scope for **undefined parameters**.

Providing this also allows to react to **changes of a particular service interface**. Service vendors can freely change their interfaces the way they want to as long as they provide a description how to transform it back to the class interface. A system providing this **parameter transformation** satisfies most of the heterogeneity aspects according to interfaces.

Further do we need to solve heterogeneity issues according to the **services microflow**. Because interacting with services may be very complex (microflows) and it could be that a vendor realizes a given functionality using more then one service we need ...

- ... a general way to describe how to use them.
- ... to consider that using or providing services and their microflows is always a **security issue**. As stated by Grefen et al. [6] protecting valuable knowledge of a business process is very important. Therefore a system must allow service providers to confine the granularity of their exposed microflows (e.g. offering only one proxy service instead of offering all necessary services and parameter transformations directly) and allow customers to decide individually if a provided controlflow meets their security policy (see e.g. Alam et al. [54] for process security issues and approaches).
- ... to make intense use of the capabilities provided by adaptive WEEs. The possibility to simply inject the services microflow into the original business process should be exploited.

Because our system targets highly innovative markets we further support the rapid creation of new class level operations. It therefore has to provide a way to reuse operations at different levels in more than one class level operation.

Injecting the services microflows directly into the original process meets a lot of the **flexibility** and some **security** aspects demanded by such a system.

Finally we provide a way for each vendor to define its own microflows for different transitions within a business process. As defined by Eder et al. [49] in their WAMO model, microflows for the following transitions are needed: (1) Execute, (2) Compensate, (3) Undo, (4) Redo, (5) Suspend and (6) Abort.

Even if **repair strategies are beyond the focus of this thesis** we want to consider it for our prototype implementation because they are tightly connected to this field.

Client Application Development

To allow easy usage of the Repository within a client application we need to provide ...

- ... support for annotating parameters with multi-lingual captions.
- ... a schema for input and output parameters, aggregated overall included activities, by the API.
- ... an API which is easy to understand and use.
- ... schemas to validate the generated data against.

Using this data allows client applications to **automatically create UIs** for different application domains. Further is it easier for developers to implement applications if the system sticks to widely supported standards e.g. ATOM-feeds [47], XML Schemas [55], XPath [53].

Maintenance

An API to maintain information stored in the Repository has to be implemented. Because we want the interface to be very simple we decided to implement a CRUD (Create, Read, Update and Delete) interface [25] which is often used together with RESTful applications. This further implies that only HTTP parameters, methods and status codes are used to

communicate with the different services/resources within the Repository. Providing this methods on all three levels of the repository allows to completely maintain the data stored in the Repository.

3.1.3 Injection Handler

To make full use of the scalability and reliability concepts provided by Clouds, our system needs a component that delegates the actual microflow injections to different Injection Services. Further is it possible that e.g. in a parallel branch of a business process two or more injections needed to be performed at the same time. The Injection Handler component handles all pending injections for a process instance and ensures that they are performed the right way. It will do this by delegating each pending injection including all needed information e.g. position, controlflow description to any available Injection Services. After all injections are performed correctly it has to update the controlflow description of the process instance and restart its execution. Additionally is the notification concept provided by the WEE used to begin its job only when it is in the right state e.g. the execution of the process must be properly *stopped* to perform any injections.

3.1.4 Injection Service

The component which is in charge for the actual injection must also be designed in a way to allow the use of Cloud concepts. Designing it as a RESTful service will have the desired effect as RESTful components are stateless and therefore easy redistributable (reliability and scalability demands) within Cloud infrastructures. Further does the targeted WEE also provide a RESTful API (see Stuermer [45] for details) and therefore fit together perfectly.

The Injection Service will perform the actual microflow injection when it is delegated to it by the Injection Handler. Whenever an injection is performed, the following things must be considered:

- The injection algorithm of the service needs to differ between class level and instance level injections.
- Whenever an injection is performed context elements and a microflow are included. Context elements are identified and referenced by their name and instructions included in the microflow by their ID. Therefore the algorithm must ensure that no naming/ID conflicts result of an injection.
- Context elements must be created before the injected controlflow begins and removed after it has ended.
- Class level injections have to take care about calls to external services which also manipulate context elements, e.g. the “Call - Manipulate Statement” described in Section 6.3.1.
- Instance level injections should inject each service in its own branch of a parallel statement to produce an efficient controlflow. Further has it to store results and properties of injected services for later use in a context element which allows to relate them to its original service.

- If constraints are defined within the original call statement they have to be evaluated during an instance level injection.
- It has to be able to resolve injections placed within a loop.

Because we allow each repository to have its own syntax/language, we need a way to **transform this syntax/language** into the WEEs controlflow description syntax. The broad support within different programming languages and platforms makes XSL stylesheet transformations [56] the right decision. Further is the block-based structure (XML) of the WEE/CPEE's controlflow description easy to parse using XSLT .

3.2 System Architecture

This section gives an explanation of our system architecture. By explaining stepwise what happens when and where during an injection (see Figure 3.1) we show how the components interact and what information is exchanged. Further will we discuss how we deploy the components and what goal we achieve by it. Details about the components interfaces and their algorithm are given in the according chapters, later in this thesis.

3.2.1 Excursus: The CPEE

While this system was implemented and the thesis written, the WEE (implemented by G. Stuermer) was extended with a RESTful interface and notification model. This extended version is named **Cloud Process Execution Engine** (CPEE). The CPEE is available at `http://www.pri.univie.ac.at/workgroups/wee/`. We give a short outline about the provided API and notification model for the purpose of understanding how our system utilizes it.

- The API includes, along other functions, a function to ...
  - ... read/update the controlflow of the process execution.
  - ... read/update the actual position of the process execution.
  - ... create/read/update/delete data elements and endpoints.
  - ... start/stop the execution of the process.
- Further does a notification model allow components, providing a RESTful interface, to subscribe for different events. Among others we name three events which are used by our system:
  - **Syncing After** is sent whenever the execution of an activity is finished. This event is part of a voting system provided by the CPEE which allows each registered component to vote if the execution of the process instance should continue by responding true or false.
  - **Stopped** is sent whenever the execution of a process instance is stopped.
  - **Description Changed** is sent whenever the process description of an instance is changed. This event is used by our client implementation described in Chapter 8.

Whenever an activity has to be executed or the WEE needs to interact with outside components it uses a so called Handlerwrapper. We focus only on the part used to execute activities and therefore refer to it as **Activity Handler**. In Chapter 7.3 we explain in details the way we adapted it to enable the here described behavior.

For further information about the CPEE check for upcoming publications from J. Mangler et al. about Cloud based process execution.

3.2.2 Deployment

As stated above **scalability and reliability** are two of the main concepts of Cloud infrastructures. We already made clear that our targeted WEE was designed to be distribute over such one. Therefore we do not give any details about how it is done and just assume that it is (for details see [44, 45]). Because our system focus only on one process instance at a time we see only one instance (expressed by a single cloud) of the **WEE/CPEE** in Figure 3.1. Further provides our prototype system only one instance of an **Injection Handler** which is defined within the call statement. Like for the WEE, that's why only one Injection Handler is shown. In an evolved version of our prototype implementation will be more than one Injection Handler for the whole system and be selected during the execution by the **Activity Handler**. We will do so to gain the requested reliability and scalability effects. In this version we did not implement this for the purpose of simplicity and the strong correlation to the target system [57]. All other components can be freely distributed and replicated (expressed by a bunch of clouds) within a Cloud infrastructure to fulfill scalability (throughput) demands. The Injection Handler can select whatever **Injection Service** it decides to use at runtime. Further can each Injection Service use any of the available **Repositories**. As we will explain later (see Section 5.3.3 for details) is the Repository already defined by an URI within the business process, but the Repository itself is because of its RESTful design easily deployable over any number of servers using e.g. Apache's Load Balancer [58].

With this SOA based architecture and the strict use of HTTP-based protocols our system allows to be distributed over many different resources. Also that's why many of today's distribution and load balancing strategies can be used. For a detailed discussion about different scaling and distribution strategies see Teo and Ayani [57]. Further is it possible to allocate resources in time of high workload to reach the desired throughput. Also the other way around by deallocating resources in times of low workload for cost reduction works too.

3.2.3 The Interaction Between the Components

As explained in the previous section our system consists of loosely-coupled services interacting together (SOA design). This section shows how the involved services/components interact together when an injection needs to be performed. To understand our system it is important to know when which service interacts with others and who they are.

We do so by giving a step by step example of what happens when an activity is supposed to use services from the Repository, causing an injection when executed by the CPEE instance (see Figure 3.1). As said above, whenever an activity needs to be executed, the CPEE instance uses the Activity Handler. That's why this is the origin of our example.

Step One: The Activity Handler registers the CPEE instance for an injection at the Injection Handler. For details see Figure 3.2.

Step Two: When the CPEE instance is in the right state the Injection Handler delegates each pending injection to any available Injection Service. For details see Figure 3.3.

Step Three: The Injection Service looks up the Repository to get the needed information to perform the requested injection. For details see Figure 3.4.

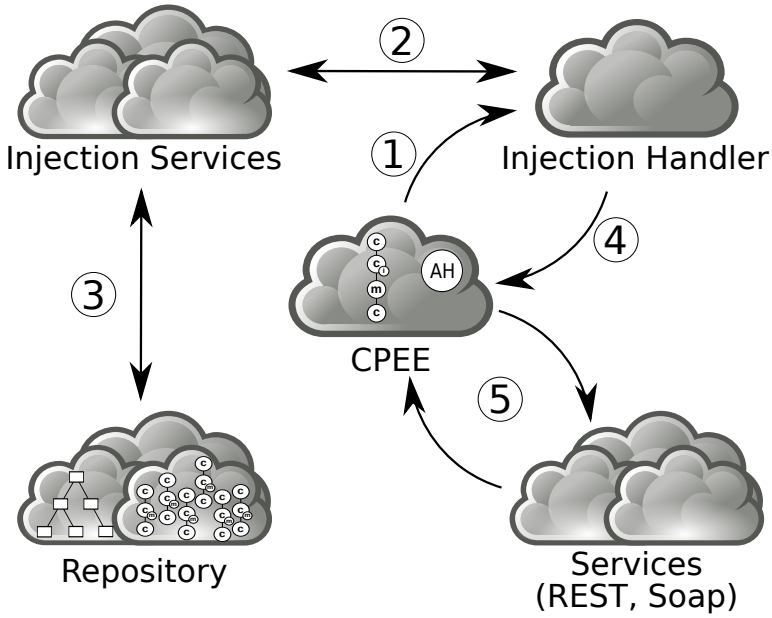

Figure 3.1: Architectural Outline

Step Four: When all injections are finished correctly the Injection Service updates the controlflow and execution positions of the CPEE instance. For details see Figure 3.5.

Step Five: When the CPEE instance continues with the execution of the process the newly injected microflows, describing how to interact with the former requested services, are executed.

Step One: Preparing an Injection

Whenever the Activity Handler (AH) executes an activity supposed to use services from a repository, the AH and the Injection Handler (IH) interact as follows (see Figure 3.2).

(1) The AH registers the CPEE instance for an injection at the actual execution position using the IH at the provided URI[2].

(1a) The IH registers itself for the **Syncing After event** at the requested CPEE instance (if it not already is). The CPEE instance responds a notification-key and a secret for message encryption if the registration is accepted.

After the CPEE instance accepted the registration the IH responds to the AH that it takes care of the requested injection. Now the AH interrupts the execution of this

[2]In our prototype implementation we provide the particular URI of the IH within each activity statement. In respect to Cloud concepts the particular URI should be chosen at run-time by the AH or some other resource managing service/component of the infrastructure. If there are more then one IH used for one CPEE instance, they must consist of merging strategies to guarantee that the changes from each of them are in the final version of the controlflow.

activity immediately. If a manipulate-block is defined by the activity it will not be executed here[3].

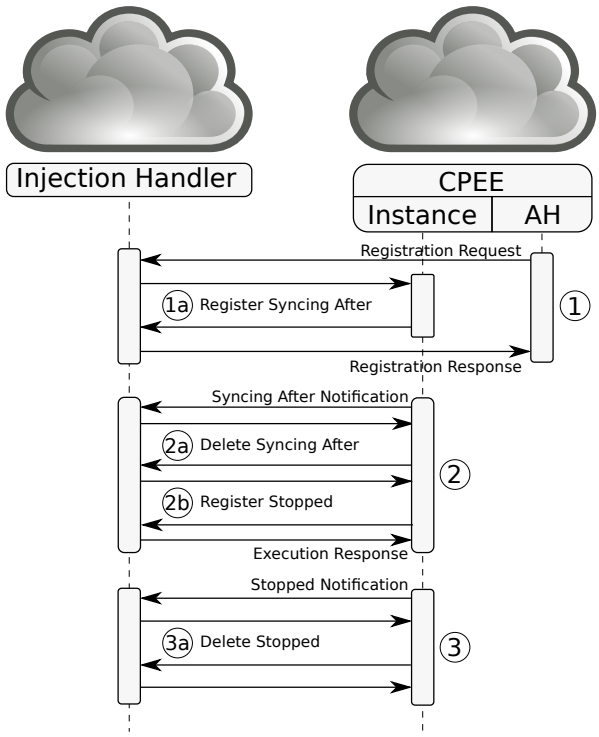

Figure 3.2: Step One: Interaction between Activity Handler and Injection Handler

(2) The CPEE instance informs the IH whenever an activity is finished. It awaits a response if it is allowed to continue because of the prior registration for this event.

(2a) The IH removes the registration for the **Syncing After event** if the causing activity was registered for an injection.

(2b) If the causing activity is registered at the IH, it registers itself for the **Stopped event** at the CPEE instance. Again the CPEE instance responds a key and a secret for message encryption.

After this two requests are done, it responds *false* to the notification causing the CPEE instance to stop the execution as soon as possible. If the causing activity is not registered at the IH it responds *true* allowing the CPEE instance to continue with its execution.

(3) The CPEE instance notifies the IH about the state change because of the prior registration for this event. If the new state is **Stopped** the IH starts performing the injections for the instance.

(3a) The IH removes the registration for the **Stopped event**.

Now the IH delegates the actual injection to any Injection Services as described in the next section.

Step Two: Collecting Instance Information and Delegating Injections

Now the instance is in the state **Stopped** which means that no activity is executed at the moment and changes to data elements, endpoints and the controlflow can be performed. The Injection Handler (IH) processes now the **injection queue**, where all registered injections for a CPEE instance are stored, by delegating them to Injection Services (IS) (see Figure 3.3).

[3]In Section 6.3 we give a detailed discussion why this behavior is important for our approach.

(1) The IH start with requesting the processes controlflow from the CPEE instance. This description is used to apply all changes caused by service injection.

(2) The IH requests the actual execution positions from the CPEE instance. Because of parallel branches there can be more than one execution position at a time within a CPEE instance. To ensure that no position is lost or wrong they also need to be considered during the injection.

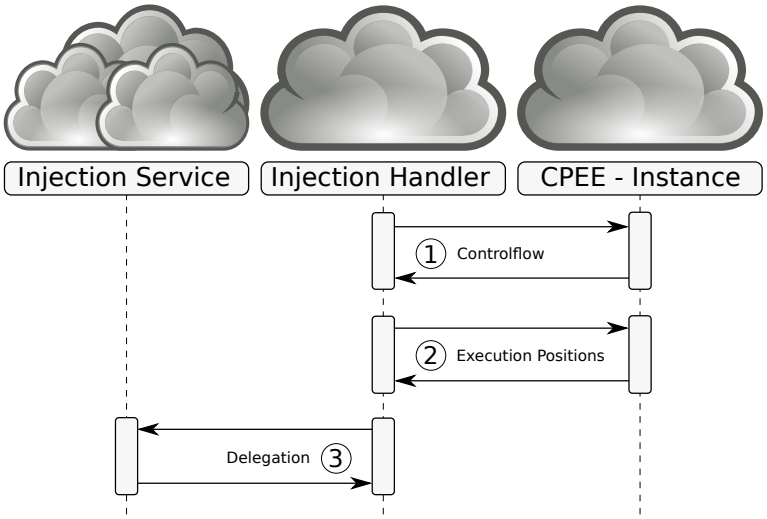

Figure 3.3: Step Two: Collecting Instance Information and Delegating Injections

(3) The IH processes one pending injection after the other by delegating them to an IS (Step Three - Section 3.2.3). The response consists of the adapted controlflow and updated execution positions after the particular injection was performed.

How the results are used and transferred into the CPEE instance is explained below in Section "Step Four: Updating and Restarting the Instance" (3.2.3). We continue with describing what components are involved during the actual injection.

Step Three: Injection

The Injection Service (IS) is in charge to perform the actual injection delegated to it at Step Two. To do this properly it needs information from the CPEE instance, the Repository and the Injection Handler (IH) (see Figure 3.4).

(1) First the **Endpoint** of the activity causing the injection is requested from the CPEE instance by the IS. Because in the controlflow only symbolic names for endpoints and data elements are provided, their actual value must be requested from the CPEE instance whenever such one is needed.

(2) The IS requests the microflows provided at the endpoint received in (1). If it is an injection at **class Level** the according microflow is requested. If it is supposed to inject instance level microflows the whole sub-tree, starting at the provided endpoint, is parsed and the description of each service is requested separately.

(3) If constraints, defined in the controlflow, are pointing to **context elements** instead of fix values the IS must additionally request them from the CPEE instance. Again every context element is requested separately.

(4) The IS requests the actual execution positions from the IH to ensure that each position is set correctly after the injection. It uses the IH instead of the CPEE instance

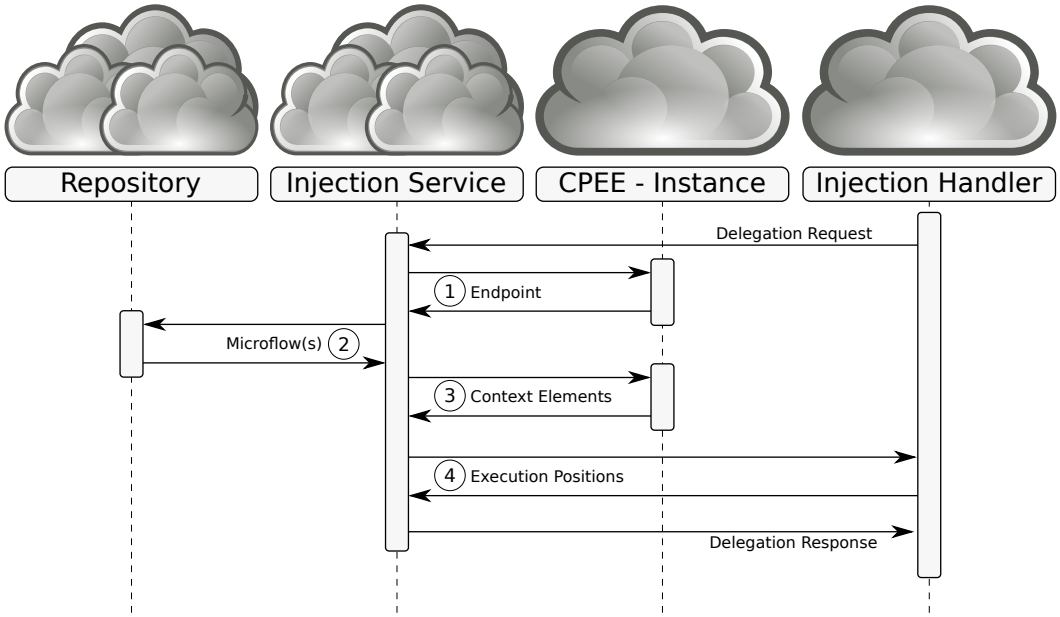

Figure 3.4: Step Three: Injection

because the execution positions provided by the CPEE instance may be outdated because of prior performed injections.

After the injection is finished the resulting controlflow and the updated execution positions are responded to the IH.

Step Four: Updating and Restarting the Instance

After the last injection in the queue was performed, the final version of the controlflow (including all injected microflows) is responded. Further is it now decided where the updated execution positions for the re-start of the CPEE instance are. Before the execution of the initial process can be continued by the CPEE instance the Injection Handler (IH) needs to update it (see Figure 3.5).

(1) The IH replaces the initial **controlflow** of the CPEE instance with the one derived by the injections.

(2) The IH replaces the initial **execution positions** within the controlflow of the CPEE instance with the updated ones for the updated controlflow.

(3) The IH sends a **Start** request to the CPEE instance triggering the resumption of the process execution.

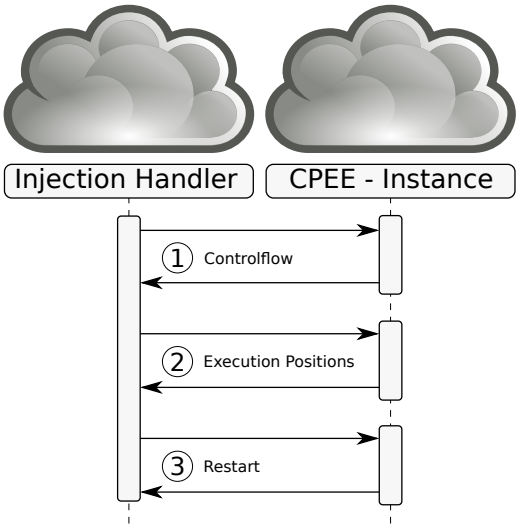

Figure 3.5: Step Four: Updating and Restarting the CPEE instance

Step Five: Using Injected Services

After all injections are done the original controlflow is now extended with information about how to interact with the external services requested from the Repository. Using this technique allows the CPEE instance to continue with the execution of the process without needing any further components than the external services them self. The Activity Handler of the CPEE instance takes care of the communication with the services in a RESTful or SOAP-based way and stores the results within context elements for further use.

Chapter 4

Repository

Jha et al. [11] proposed that a repository for services should consist of two layers. While they focused mainly on functional aspects of a repository we additionally considered organizational aspects resulting in the additional level(s) in the middle (they could be more than one). We use them to group services with common properties together, similar to the classification levels (Segment, Family, Class, ...) of the UNSPSC [34] code used by UDDI. Further do they allow to consider only a subtree of the Repository and therefore save bandwidth and computing time when exploring it. Further do we provide a RESTful API to use and maintain the Repository. We will explain how we implemented it in the following sections of this chapter.

An earlier version of this repository was already published by Vigne et al. [38]. Although our vision has evolved over time it still shares some main principles with this prior version. Further will some concepts and ideas of the Repository be published in Vigne et al. [27].

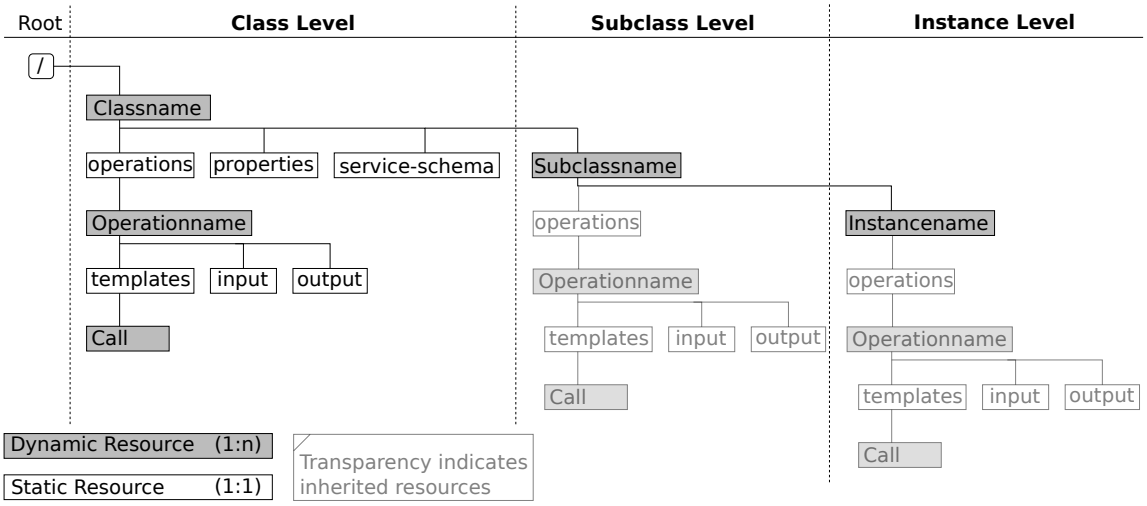

Figure 4.1: Structure of the Repository

Figure 4.1 shows the three levels and the resources provided by each of them. We further explain that the Repository is organized in a strictly hierarchical manner. The idea is that the **class level** defines how the interfaces of its **instance level** must look like. This concept allows us to use services within an application domain in an arbitrary and interchangeable

way. We will explain later why the **subclass level** is not relevant for the interfaces and its intention.

We implemented this structure using directories in a file system. Each resource is represented by a directory including its sub-resources which again are directories. Class level resources and instance level resources include also a file with definitions in their directory. The content of these two files is explained in the next two sections.

Our reasons for using the file system instead of a database are mainly **portability and performance**. Because the anatomy of data we are dealing with is very small, it can be handled quite well by a file system and a web server. Both were initially intended to serve files in an efficient way. Additionally, databases are not available at all platforms we target with our design (mobile devices) or will have different interfaces and therefore decreasing portability. As we explained when describing the architecture of our system and the benefits of the SOA design, are we intending to use built-in functionality of Web Servers [58] (e.g. load balancing).

In Figure 4.1 the following resources are illustrated semi-transparent to indicate that they are inherited from their parent resource:

A list of operations and the according **class level microflows** are provided on each of the three layers. We decided to do so in order to increase flexibility of the Repository. If the Repository is distributed over different servers with different URIs it may be hard to find the parent of a resource because our system provides only data about its child resources but not about its parent resources (for details see the ATOM definition [47]). But if each layer provides a list with the defined class level operations this is not a problem anymore.

Templates are the second data provided on more than one level. As they are part of the class level microflows they are provided as sub-resources of the operation they are defined in (see Figure 4.1). We use templates to support the possibility of defining multiple interfaces (target platforms and languages) to interact with the user. We only support their definition at class level to prevent service vendors to collect additional data from the user not included in the interface. We see instance level microflows as completely autonomous without the need of any additional data than provided by the interface or created/defined within the microflow.

As information about operations and templates are defined at class level we do not want to give a detailed explanation here and refer to Section 4.2 where the definition and structure of them is explained in detail.

The reason why we left out the **subclass level** in our above explanations is that this level has a purely organizational purpose and does not add any further definitions to its successors. Although properties can be defined (see Section 4.2.4) we wanted an additional possibility to divide services into separate partitions. We only show one subclass level but our system supports to define more than one. This way the granularity inside the Repository can be adjusted.

4.1 Demonstrator: Application Domain 'Cinema'

We define an application domain named **Cinemas**. This domain is intended to include services of cinemas providing operations to search for a movie and to book tickets for their shows. Further do we provide an additional operation named `search & book` at class level. It searches for a movie over all registered services, let the user select a particular show and finally book tickets for the selected show. Therefore we must aggregate the `search` operations of all cinemas within the provided path of the Repository, call an external resource for the selection and finally call the `book` operation of the selected cinema. Figure 4.2 illustrates this interrelation between the operations using BPMN [59].

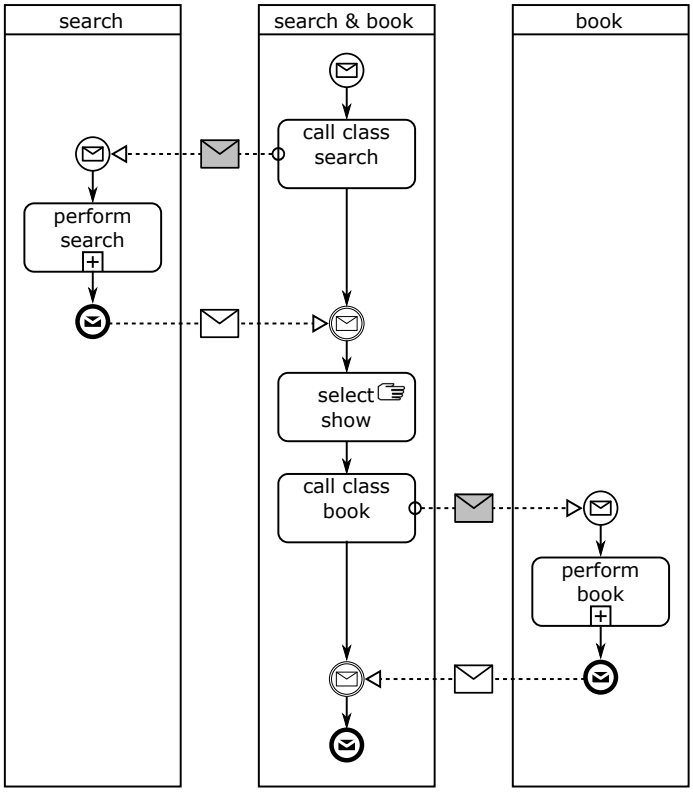

Figure 4.2: BPMN: Operations Defined at Application Domain

Further do we define that each cinema must provide a number of properties for their services. They have to include a block named address and an other one named vendor. Address must include data for street, zip-code, city and state. This allows costumers to search for a e.g. a movie in a particular city or a zip code range using property constraints. The vendor block include contact data about the vendor like its name, phone number email and homepage URI.

We explain how to define the application domain (class level) in order to achieve this functionality and provide snippets of its definition file over different Listings. We will also discuss what data (structure) a cinema must provide (instance level) to register for it.

4.2 Class Level

At class level we define **class level operations** (including **templates**) using microflows and the **properties schema** for the application domain. Based on these microflows and the properties schema a **schema for services** and **input and output parameter schemas** are created. Each service (instance level) within this class must fulfill the provided schemas.

4.2.1 Class Level Operations

We already cleared that our system provides the definition for various class level operations for each class. A **List of all Provided Operations** is provided for their further exploration. We decided to use the XML standard for this purpose.

Listing 4.1 shows an example of operations defined for a class. Each operation is represented by an element named `operation`. These elements provide the operation name using the attribute *name* which is used whenever the operation is referenced. This is (a) when the definition of the operation is requested and (b) when the operation is referenced by a call statement. The attribute *short* provides a short description of the operation.

LISTING 4.1: Generated List of All Provided Operations for Example 4.1

```
1<operations xmlns="http://rescue.org/ns/domain/0.2">
2  <operation name="search" short="search for avaliable shows"/>
3  <operation name="search_and_book" short="search for shows, select show and
       book tickets"/>
4  <operation name="book" short="book tickets for a given show and cinema"/>
5</operations>
```

Its intent is to allow customers to get a quick idea of the provided operations of an application domain with one request. It should be noted that this list is generated on demand by parsing the definition file of the application domain and therefore must not be maintained explicitly.

Each of these class level operations represent a microflow describing how the instance level operations are used. They make use of different operations provided by the registered services within the application domain. Using these service operations allows to create new functionality (based on the functionality provided by the services) and offer it as one service operation. Because Conallen defined that "Web applications implement business logic, and its use changes the state of the business (as captured by system)." [60, p. 1] we state that these microflows are capable to provide similar functionality like web applications in terms of **offering complex interrelated service functionality**.

Operations can only be defined within an `operations` element (see Listing 4.2 line 1) and are identified by an `operation` element. Each operation defines its *name* and a *short* description of its functionality using attributes. Because the operation name is also used as an ID it must be unique within an application domain. Further is these data used to generate the **List of all Provided Operations** (see Listing 4.1). Lines 2, 11 and 31 are examples for the definition of an operation.

Each operation may define its own context elements including `context-variables` (line 12) and `endpoints` (line 15). Context elements must be defined before the execute block. A detailed explanation about the definition of context elements is given in Section 5.2. The injection algorithm (see Section 6.3 about prefixing IDs) ensures that references to context elements within a microflow stay valid and do not conflict with other context elements.

The injection-ready **microflow** has to be defined within the `execute` element (see line 19). The microflow itself is defined using all the controlflow statements described in Chapter 5. Finally the *namespace* attribute of an operation element ensures that only valid controlflow is provided (details about XML Namespaces can be found at [61]).

LISTING 4.2: Definition of Class Level Operations for Example 4.1

```
1 <operations>
2   <operation name="search" short="search for avaliable shows" xmlns="...">
3     <execute>
4     ...
5     <call id="perform_search" endpoint="resource_path" service-operation="
          search">
6       ...
7     </call>
8     ...
9     </execute>
10  </operation>
11  <operation name="search_and_book" short="search for shows, select show and
        book tickets" xmlns="...">
12    <context-variables>
13      ...
14    </context-variables>
15    <endpoints>
16      ...
17    </endpoints>
18    ...
19    <execute>
20      <call id="call_find" endpoint="resource_path" service-operation="search
            ">
21        ...
22      </call>
23      <call id="perform_select" endpoint="selector_service" endpoint-type="
            outside" http-method="post" info="true" default-tpl-name="mobile"
            default-tpl-lang="en">
24        ...
25      </call>
26      <call id="call_book" endpoint="selected_cinema" service-operation="book
            ">
27        ...
28      </call>
29    </execute>
30  </operation>
31  <operation name="book" short="book tickets for a given show and cinema"
        xmlns="...">
32      <execute>
33        <call id="perform_book" endpoint="resource_path" service-operation="
              book">
34          ...
35        </call>
36      </execute>
37  </operation>
38 </operations>
```

For a better understanding of how the data within the Repository is interrelated we give a short explanation of the three different types of calls at this level. For a detailed discussion about these statement see Section 5.3.3.

1. A call to an **other class level operation** is shown at line 20 and 26. This allows to build even more class level operations by reusing already defined ones and combining them with other calls within a new microflow. Using this type of calls allows us to realize the method `search & book` from the cinema example. More details about how this kind of calls is handled during an injection is given in Section 6.3.

2. A call to an **instance level operation** (for details see Section 4.3) uses an operation defined at instance level (see line 5 and 33). These operations can be identified by defining a call referring to them self[4]. In our example these are the operations named `search` and `book`.

3. A call to an **external resource** is shown in line 23. These can address any resources providing a RESTful or SOAP interface. We use this one to realize the user selection in our example. These operations are identified by a different set of attributes than the other two. We will discuss them later in Section 5.3.

4.2.2 Input and Output Parameters Schema

Each operation defines a set of parameters and endpoints taken as input and a set of parameters it responds as output. These sets of parameters represent the exact data each service at instance level can expect to receive and hast to deliver. How this data is handled within microflows is explained in Chapter 5.

Because each parameter can be referenced more than one time, their names have to be unique within one class level microflow. This allows to use input parameters for different instance level operation calls and to assign output parameters in different branches of the microflow. The designer has to ensure that each output parameter has a valid value at the end of it by design. Further must the schema generation algorithm consider that we support more than one class and instance level operation calls within one microflow. This may lead to definitions where output parameters of an operation call are reused as input for successive operation calls and therefore the input parameter must not be given by the referring call statement and the output parameter is, in our definition, not part of the output schema.

We decided to use the Relax NG (RNG) schema standard [62] to describe these interfaces. We support XML Schema data types [55] for parameter type definitions. Further is the creation of complex data structure, by nesting elements as defined by RNG schemas, supported.

LISTING 4.3: Generated Input Parameter Schema for Example 4.1

```
1 <rng:grammar xmlns:datatypeLibrary="..." xmlns:rng="...">
2   <rng:start>
3     <rng:element xmlns:rng="..." name="additional_endpoints">
4       <rng:element xmlns:d="..." name="selector_service">
5         <d:caption xml:lang="en">Used at call(s): perform_select</d:caption>
6       </rng:element>
7     </rng:element>
8     <rng:element xmlns:rng="..." name="input-message">
9       <rng:element xmlns:rng="..." xmlns:d="..." name="title">
10        <d:caption xml:lang="en">Movietitle</d:caption>
11        <d:caption xml:lang="de">Filmtitel</d:caption>
12        <rng:data type="string"/>
13      </rng:element>
14      <rng:element xmlns:rng="..." xmlns:d="..." name="date">
15        <d:caption xml:lang="en">Date</d:caption>
16        <d:caption xml:lang="de">Datum</d:caption>
17        <rng:data type="date"/>
18      </rng:element>
```

[4]While we decided to do so, to gain better syntactical flexibility when defining new operations, we gave up the support for recursions within microflows at this level.

```
19      </rng:element>
20    </rng:start>
21 </rng:grammar>
```

Listing 4.3 gives an example how such a schema for input parameters may look like. It consists mainly of two parts:

Additional Endpoints defined within the microflow are listed first (lines 3 - 7: element `additional_endpoints`). As explained in Section 5.3 we allow the definition of endpoints referring to input parameters by adding the attribute *endpoint-type='outside'* to the statement. This allows the user to use his own service during the execution. An example for that could be that within a microflow a decision has to be made and therefore a call to a service of the user is defined e.g. selecting the particular show in our example. Providing the endpoint as a parameter allows to change it from call to call and therefore increases flexibility and re-usability of the operations. Of course this data is only generated for input schemas.

Parameters are the second part of the schema (lines 8 - 19). Depending on what it represents this element is named `input-message` or `output-message`. Each parameter consists of a data type defined within the element `data` (line 12 and 17) and supporting XML Schema data types and one or more `caption` elements (line 10,11 and 15,16). The caption elements are representing data for generic UI generation in different languages specified by the *lang* attribute. The schema can be used to (a) validate data sent to and received from the class level operations and therefore acting as a guideline how to use them in other applications and (b) generating UIs either for data input from the user or presenting the results to the user in client applications.

We derive this schema by parsing the microflow of the requested operation. If the **input schema** is requested all parameters and endpoints defined as input are collected. Also we need to collect all parameters defined as output to build the differential by removing each input that is also named as an output. We do not consider the order of the parameters because we already defined that their names must be unique within one class level operation. At last the resulting set of input parameters and additional endpoints is transformed into an RNG schema. It works similar when the **output schema** is requested. The difference is that additional endpoints are not considered this time and that the input parameters are stripped from the set of output parameters if their names match. The applied algorithm is illustrated in pseudo-code at Listing 4.4.

LISTING 4.4: Algorithm for Parameter Generation in Pseudo-Code

```
1  # Collecting input data
2  ADD all inputs referring to a message-parameter to $inputs
3  ADD all variables referring to an input-parameter to $inputs
4
5  # Collecting output data
6  ADD all outputs referring to a message-parameter to $outputs
7  ADD all variables referring to an output-parameter to $outputs
8
9  IF requested_resource == 'input'
10   EACH $input IN $inputs
11     DELETE $input IF $input is in $output
12   END
13
14   # Collecting additional endpoints
```

```
15      ADD all endpoints with 'type = outside' to $inputs
16      RETURN $inputs
17    ELSE
18      EACH $output IN $outputs
19        DELETE $output IF $output is in $input
20      END
21      RETURN $outputs
22    END
```

4.2.3 Templates

We already explained that calls to external resources are allowed within class level microflows. We support two different kinds of such resources:

Data centered: Interfaces of such resources are implemented to compute the parameter definition (see Section 4.2.2) of the activity in the microflow.

UI centered: These resources only provide functionality to display pre-defined user interfaces and do not (necessarily) take any care about the parameters defined in the interface.

Data centered resources can operate using only the parameter definition of the call statement. These resources compute them either in an automated way to create the response like classical services do or build a customized UI for the user like e.g. rich workflow client implementations. All necessary information for UI building is included in the parameter definition in form of their type, occurrence and multilingual captions (details see Section 5.3.1). Implementing such resources includes the advantage that data and UI information can be used to built a rich and customized UI fitting perfectly into the existing environment. The disadvantage of these resources is that whenever the interface of a call statement is changed the implementation of the resource may need to be adapted too.

UI centered resources, as we understand them, know how to use the information provided within templates. They do not care about the parameter definition of the call statement because all this information is included in the provided (UI) template. One example for such a resource is a web browser using XSLT to transform the parameters into HTML. Because the resulting HTML-form also includes ranges for valid values of the response parameters there is no need for the browser to care about them. The advantage of using this implementation style is that there is no need to adapt the resource when an interface is changed, given that the templates are changed synchronous to the interface. Disadvantages of such resources are that the layout and design of the UI may not fit well into an existing software environment. Although our system provides a way to define more than one template for a call statement, differing between target device and language, it still may be suboptimal for a some environments. Also has the customer no influence on the definition of these templates because they are defined at class level (application domain). For an example of such an implementation see the Worklist introduced in Section 8.4.

LISTING 4.5: Structure of a Template Definition

```
1<templates>
2  <xslt name="mobile" xml:lang="en">
3    <xsl:stylesheet version="1.0" xmlns:xsl="...">
4      <xsl:output method="html"/>
```

```
5       <xsl:template match="/">
6         <html>
7           <head>
8             <script type="text/javascript" src="..."/>
9             <script type="text/javascript">
10              function send_selection(data, callback) {
11                ...
12              }
13            </script>
14          </head>
15          <body>
16            <table border="1">
17              <xsl:apply-templates select="//show"/>
18            </table>
19          </body>
20        </html>
21      </xsl:template>
22
23      <xsl:template match="//show">
24        <tr><td><table>
25        <tr><td>Cinema:</td><td><xsl:value-of select="cinema_uri"/></td></tr>
26        ...
27        <tr><td colspan="2"><xsl:element name="input">
28          <xsl:attribute name="type">button</xsl:attribute>
29          <xsl:attribute name="value">Select</xsl:attribute>
30          <xsl:attribute name="onClick">JavaScript:send_selection({
31            'show_id' : '<xsl:value-of select="show_id"/>',
32            ...
33          },'<xsl:value-of select="$instance-uri"/>/callbacks/<xsl:value-of
                select="$callback-id"/>');
34          </xsl:attribute>
35        </xsl:element></td></tr>
36        </table></td></tr>
37      </xsl:template>
38    </xsl:stylesheet>
39  </xslt>
40
41  <xslt name="mobile" xml:lang="de">
42    ...
43  </xslt>
44 </templates>
```

A template has always to be defined within the call statement it is used for (see Listing 4.5). Its declaration must be inside an `templates` element which is used to group many them together. Within this element more `template` elements are allowed. Each template element represents one template for this call statement. They define a *name* attribute and a *lang* attribute. The combination of these two must be unique within a call because it acts as an ID for the templates. Within a template element the whole instruction set of XSLT is allowed. The **input parameters** of the call statement are provided as XSLT variables and can therefore be used directly within the stylesheet.

In the example (Section 4.1) we define such templates in the operation `search & book` to let the user select one show, tickets should be booked for (see Listing 4.5). We use templates to interact with the user in our client implementation.

4.2.4 Properties Schema

Figure 4.1 also shows **properties** as a resource of the class level. Our idea when implementing them was to provide a way to query services within a class during the execution of the controlflow and allow the controlflow designer to define that only services which fulfill some properties should be considered in the execution. Therefore we needed a way to define which properties must be provided by all services. We faced this problem with an RNG schema which defines what properties are provided. This schema allows to define exactly what information can be used for constraint elements (details see Section 5.3.2) and what information vendors must provide. The definition of them must be within a `properties` element at the beginning of the definition file. An example for such an definition is given in Listing 4.6.

LISTING 4.6: Properties Schema Provided for Example 4.1

```
1 <rng:grammar xmlns:datatypeLibrary="..." xmlns:rng="...">
2   <rng:start>
3     <properties xmlns="..." xmlns:rng="..." xmlns:d="...">
4       <rng:element name="address">
5         <d:caption xml:lang="en">Address</d:caption>
6         <d:caption xml:lang="de">Adresse</d:caption>
7         <rng:element name="street">
8           <d:caption xml:lang="en">Street</d:caption>
9           <d:caption xml:lang="de">Strasze</d:caption>
10          <rng:data type="string"/>
11        </rng:element>
12        ...
13      </rng:element>
14    </properties>
15  </rng:start>
16 </rng:grammar>
```

Whenever a service registers for a class it is validated that the properties section of the service definition fulfills this schema. Because the whole set of instructions provided by RNG is allowed, complex structures can be defined too.

4.2.5 Service Schema

Each application domain provides a schema it validates all member services against. Second is this schema a guideline helping new vendors to describe their service the way it is expected by the Repository. We already mentioned before that this schema is generated by the Repository by transforming the class level definitions. The schema consists mainly of three parts.

A properties schema: This is the same schema as provided by the properties schema (Listing 4.6). The reason why this is provided here a second time is to have a complete scheme to validate service descriptions (see Section 4.3) against. How the properties schema is mixed into the service schema is shown in listing 4.7 at the lines 4 - 10.

List of service operations: This schema is generated by parsing all class level microflows and checking each call statement if it refers to an instance level operation. If it does it is added to the schema if it has not already been. Using this algorithm educes only the service (instance level) operations needed, independent from how often they are referenced or how complex the class level operations are interrelated. The result is a

list with operations which must at least be defined at instance level. To support the set of transitions defined by Eder et al. [49] we expect each service to give a separate microflow for each of them. All transitions are listed between line 13 - 18 in Listing 4.7. As expressed by the `ref` element, each microflow can use the whole set of controlflow statements. (see Chapter 5 for details).

Controlflow-code: In order to provide a complete schema to validate instance level description against, the controlflow schema needs to be included. We dedicated the whole Chapter 5 to it and will therefore not discuss it here.

LISTING 4.7: Generated Service Schema for Example 4.1

```
1 <grammar xmlns="..." xmlns:s="..." xmlns:flow="..." xmlns:rng="..." xmlns:
    domain="..." datatypeLibrary="...">
2   <start>
3     <element name="s:service-description">
4       <element name="s:properties">
5         <element name="s:address">
6           <element name="s:street">
7             <rng:data xmlns="..." type="string"/>
8           </element>
9           ...
10        </element>
11        <element name="s:operations">
12          <element name="s:search">
13            <element name="s:execute"><ref name="controlflow-code"/></element>
14            <element name="s:compensate"><ref name="controlflow-code"/></
                element>
15            <element name="s:undo"><ref name="controlflow-code"/></element>
16            <element name="s:redo"><ref name="controlflow-code"/></element>
17            <element name="s:suspend"><ref name="controlflow-code"/></element>
18            <element name="s:abort"><ref name="controlflow-code"/></element>
19          </element>
20          <element name="s:book">
21            ...
22          </element>
23        </element>
24      </element>
25    </start>
26    <define name="controlflow-code">
27      ...
28    </define>
29 </grammar>
```

4.3 Instance Level

Now that we have cleared what data is defined at class level and how the instance level schema is generated we discuss how instance level descriptions are defined. The intend of this layer is to describe the interaction with a particular service and its properties in a way to make it usable for our system.

As explained in the section above the instance level definitions must validate against a schema provided at class level (see Section 4.2.5 for details). This schema includes the instance level operations and properties each service must provide. Listing 4.8 shows the structure of such an instance level description generated for the example introduced in Section 4.1.

LISTING 4.8: Definition of a Service (Instance Level) for Example 4.1

```
1 <?xml version="1.0"?>
2
3 <service-description xmlns="http://rescue.org/ns/service/0.2"
4                      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
5   <properties xmlns="http://rescue.org/ns/properties/0.2">
6     <adress>
7       <street>Ignaz-Kooeck Strasse 1</street>
8       ...
9     </adress>
10    ...
11  </properties>
12  <operations xmlns="http://rescue.org/ns/controlflow/0.2">
13    <search>
14      <endpoints>
15        ...
16      </endpoints>
17      <context-variables>
18        ...
19      </context-variables>
20      <execute>
21        ...
22      </execute>
23      <compensate>
24        ...
25      <compensate/>
26      <undo>
27        ...
28      <undo/>
29      <redo>
30        ...
31      <redo/>
32      <suspend>
33        ...
34      <suspend/>
35      <abort>
36        ...
37      <abort/>
38    </search>
39    <book>
40      ...
41    </book>
42  </operations>
43 </service-description>
```

Lines 5 - 11 define the **properties of the service**. According to the provided schema they must be placed within an `properties` element. The structure (RNG) and its data (XML Schema data types) as defined at class level, are instanced here by each service. It is important that our system ensures that only valid data is provided because it may be later referenced by constraints in the customers business process (see Section 5.3.2).

The second block that must be provided are the **instance level operations** (lines 12 -43). It includes one operation block for each instance level operation referenced within any class level microflow (e.g. line 13 and 39). Within the definition of an instance level operation it is allowed to define context elements used within the following microflows. Context elements

include `endpoints` and `context-variables`[5]. They can be referenced by their name what is the reason why each name must be unique within its block. After the context elements, all **microflows used for different service transitions** are defined (lines 20 - 37). The transitions defined by Eder et al. [49] are namely: (1) execute, (2) compensate, (3) undo, (4) redo, (5) suspend and (6) abort. During our injection we use only the microflow defined within the `execute` element. Microflows defined within the other transition elements may be used by e.g. an external repair services to set the actual service back into a valid state again.

Each of this instance level microflows may use the whole set of controlflow instructions (see Chapter 5). At this level all calls are of the type *external resource* because here the WEE interacts directly with the resources/services provided by the vendor.

4.4 RESTful API

This section focuses on the API provided by the Repository. First the APIs implemented functionality to request data about application domains and services is explained. Second we discuss the APIs functionality to maintain these data. Both APIs are designed in a strictly RESTful way and therefore using only HTTP-methods, HTTP-parameters and HTTP-status codes. Resources are requested using URIs and query parameters. All interaction with it is designed in a completely stateless way meaning that no further request of a resource influences the following or is influenced by the prior one. Before we explain how to interact with the Repository in details we give a short overview about the techniques used by this implementation.

We already mentioned above that we sticked to the **CRUD-methods** (Create, Read, Update, Delete) for the design of the API. The HTTP-methods related to them are:

POST to create resources within the Repository.

GET to request resources from the Repository.

PUT to update resources within the Repository.

DELETE to delete resources from the Repository.

These methods are implemented by resources identified by **URIs**. As we already explained in Figure 4.1 we differ between dynamic and static resources. A **dynamic resource** has no structural predefined name. It is just defined that some resources at a given level represent a particular class. Further must each resource of this class provide the same predefined functionality. Only thing that is not predefined is the name which is why we refer to them as dynamic resources. The second type are the **static resources**. For this kind of resources everything is predefined. It is exactly defined where these resources are, how they are named and what functionality they provide. In the following section we will give examples referring to resources within the Repository.

To avoid misunderstandings we define our notation right here. We begin with defining `repo:uri` as a symbol for the URI of the Repositories **root resource**. Whenever we refer to a dynamic resource we will use curly-brackets to indicate that the name is only symbolic.

[5]The context elements are kind of global for the transitions. We argue that for an arbitrary use of the services, each transition must be executable with the same set of data.

E.g. when we refer to the *List of Operations* from any class we target a resources like this `repo:uri/{class}/operations`. `/{class}` is the symbol for any class within the Repository starting at `repo:uri`. Figure 4.1 shows that each class resource provides a sub-resource named `/operations` providing a list with operations defined for the particular resource targeted by `repo:uri/{class}`.

Out of all **HTTP-status codes** [46] the Repository implements only a subset. In the following we give a short explanation of the ones used within the implementation.

200 OK: Indicates that everything worked as expected. Our implementation provides no further information about what exactly happened. It could be the response for e.g. a data request or a resource update.

201 Created: Indicates that a new resource was created successfully. It is responded when a HTTP POST-request was successfully executed to create a resource e.g. a class, a subclass or an instance.

404 Resource not Found: Indicates that the requested resource was not found. This happens when (a) the URI points to resources not included in the resource-tree or (b) when a schema is requested but the data for generating it was not found e.g. properties schema or the input parameter schemas for a class level operation which is not defined.

409 Conflict: Indicates that the requested operation could not be executed because of a conflict with IDs. The Repository responds it when a new resource should be created or an existing one should be renamed with/to a name that is already used by an other resource at the requested level.

410 Gone: Indicates that the requested resource is no longer available at this Repository and that no re-direct information is defined. This HTTP-status code is often used by feed-libraries to keep their local data up-to-date. We use it to allow clients implementing a caching mechanism for a subset of data and to keep them up-to-date. The Repository responds it when the URI has the correct format but the resource data could not be found.

415 Unsupported Media Type: Indicates that the requested operation is not performed because the supplied data did not validate. The Repository responds this whenever a resource should be created or updated and the supplied data did not validate against the according schema.

500 Internal Server Error: Indicates that something went wrong. This should not be responded by the Repository but if it happens it is embarrassing because something went awfully wrong in our implementation.

HTTP supports two different types of parameters which can be either `query parameters` or `body parameters` (see *HTTP/1.1: Protocol Parameters* [63] for details). In our implementation we use query parameters if the data is short. Whenever a potential large piece of data must be sent to the Repository e.g. a class or instance description, body parameters of the mime-type *text/xml* are used.

Figure 4.3 gives an overview about the supported functionality by each resource of the Repository. It shows what HTTP-method and HTTP-parameter combination is needed for their execution and what response can be expected.

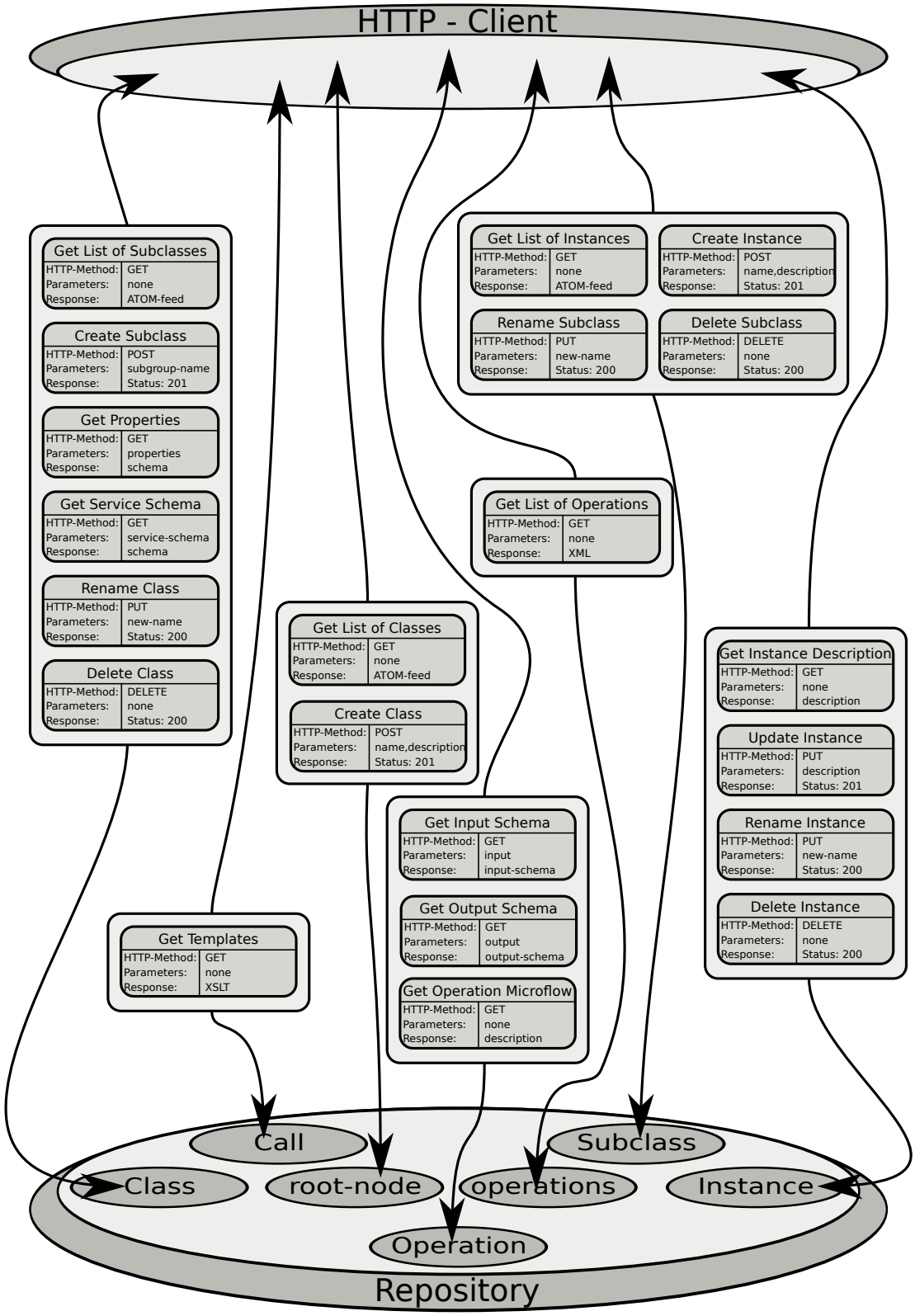

Figure 4.3: RESTful API of the Repository

4.4.1 Finding Resources

The Repository provides **lists of Contained Sub-Resources**. These are responded at the *root-node*, *Class* and *Subclass* resources (see Figure 4.3) to a HTTP GET request with no parameters.

We give some examples how the URIs, where this information is provided at, may look like. We assume that the Repository holds a class named *Cinemas* which itself holds three subclasses named *Arthouse*, *Multiplex* and *Drive-In*. Further is the *root-node* of our Repository accessible at `repo:uri`.

1. A list of all provided classes can be received by making an HTTP GET-request at `repo:uri`.
2. A list of all subclasses of the class *Cinemas* is responded to a HTTP GET-request at `repo:uri/Cinemas`.
3. A list of all services within the subclass *Arthouse* is the response to a HTTP GET-request at `repo:uri/Cinemas/Arthouse`.

ATOM-feeds allow a number of attributes and elements helping subscribers to filter and manage information (for details see *Atom: The standard in syndication* [47]) which come in handy when developing a client application for our Repository. Further do we find a wide support within today's programming languages and web browsers for these feeds what also benefits client development. These are the two main reasons why we decided to use ATOM-feeds for our implementation.

A stripped example how such an ATOM-feed, provided at class level by the Repository and therefore representing a list of subclasses, could look like is given in the listing below (Listing 4.9). We do not use all of the supported elements of an ATOM-feed and will therefore list only the one we actually used below.

title: In our implementation this element provides the URI of the requested resource.

updated: Provides the date when the data within this resource was last changed[6].

generator: Provides the root-node URI of the Repository.

id: As an unique ID for the provided feed we again use the Repository URI extended with the path to the requested resource.

link: Is used by browsers to identify the kind of data they are dealing with. This is similar to the mime-type attribute of HTTP-parameters. In this case it is *application/atom+xml*. See Listing 4.9 line 6 for an example.

entry: Each represents one child resource (lines 16 - 24 in Listing 4.9). Each one of them consists of the following elements:

title: This element represents the name of the referenced resource.

[6]Information about the last change of data can be very helpful to implement caching techniques within client applications. We explained above that we provide no version control within the Repository. Compared to UDDI we therefore save computing resources and bandwidth (as explained by Blake et al. [37]) but support similar functionality in client applications making use of this attribute.

author: Supports a lot of elements but we only use the `name` element to name the creator of the referenced element.

id: Again we use the URI of the referenced resource as an unique ID.

link: The URI where further information about the referenced resource can be requested and its name.

updated: The date when data within the referenced resource or the element itself was last changed.

LISTING 4.9: ATOM-feed Representing *List of Classes*

```
1 <feed xmlns="http://www.w3.org/2005/Atom">
2   <title>Resourcelist at repo:uri/Cinemas</title>
3   <updated>2010-09-12T22:16:33+02:00</updated>
4   <generator uri="repo:uri">RESCUE</generator>
5   <id>repo:uri/Cinemas/</id>
6   <link type="application/atom+xml" rel="self" href="repo:uri/Cinemas/"/>
7   <schema>
8     <operation name="search" href="repo:uri/Cinemas/operations/search">
9       <message type="input" href="repo:uri/Cinemas/operations/search?input"/>
10      <message type="output" href="repo:uri/Cinemas/operations/search?output
            "/>
11    </operation>
12    ...
13    <properties href="repo:uri/Cinemas?properties"/>
14    <service href="repo:uri/Cinemas?service-schema"/>
15  </schema>
16  <entry>
17    <title>Arthouse</title>
18    <author>
19      <name>RESCUEv0.2</name>
20    </author>
21    <id>repo:uri/Cinemas/Arthouse/</id>
22    <link href="repo:uri/Cinemas/Arthouse/">Arthouse</link>
23    <updated>2010-08-15T11:39:00+02:00</updated>
24  </entry>
25  ...
26 </feed>
```

We extended the ATOM-feed schema with a Repository specific element named `schema` (see Listing 4.9 lines 7 - 15). At class level and below information about the provided operations of a class is provided within this element.

Operation elements represents a supported operation of this particular class. There may be one or more of these elements for one class. The operation name is defined in the attribute *name*. The attribute *href* points to the URI providing the class level microflow of it. This elements have two child elements both named `message`.

Input-Message: Message elements with the *type* attributes value *input* provide an URI to the schema of the input parameters using the *href* attribute.

Output-Message: Message elements with the *type* attributes value *output* provide an URI to the schema of the output parameters using the *href* attribute.

Properties elements provide the URI of the properties schema of the class using the attribute *href*.

Service elements provide the URI where the service-schema for this class can be requested using the *href* attribute.

Our idea, when implementing this extension to the ATOM-feed schema, is to provide information about the class of the requested resource. This allows easy usage of the Repository within client applications and also to browse the Repository using a standard web browser. As long as a resource at class level or below is requested the Repository provides information where all according data of it can be found. It additionally allows a further distribution of the data in a Cloud infrastructure as explained in Section 3.2.2.

4.4.2 Accessing and Maintaining Resources

In this section we describe the functionality provided by our implementation to access and maintain the stored data using the RESTful API. Figure 4.1 shows the provided resources and how they are structured while Figure 4.3 shows methods and messages implemented by each of them.

Resource: Root-Node (/)

Represents the access point to the Repository. We assume that the Repository is reachable at `repo:uri` which is used as a symbol for e.g. `http://localhost:9290/groups`.

Get List of Classes - Message: Provides an ATOM-feed where all classes stored within the Repository are listed. See Listing 4.9 for further examples.

HTTP-Method: GET

Parameters: none

Response Parameter: ATOM-feed

HTTP-Status: 200 (OK)

Create Class - Message: Creates a new class within the Repository.

HTTP-Method: POST

Parameters:

1. `group-name:` defines the name of the new class
2. `domain-description:` defines the class level description as discussed in Section 4.2 for the new class

Response Parameter: none

HTTP-Status: 201 (Created), 409 (Naming Conflict), 415 (Invalid Description)

Resource: Class

Represents a particular class stored within the Repository. In our example these resources are reachable at `repo:uri/{class}`.

Get List of Subclasses - Message: Provides an ATOM-feed with all subclasses registered for the particular class. For further examples see Listing 4.9.

HTTP-Method: GET

Parameters: none

Response Parameter: ATOM-feed

HTTP-Status: 200 (OK), 410 (Gone)

Create Subclass - Message: Registers a new subclass within the class targeted by the URI.

HTTP-Method: POST

Parameters:

1. `subclass-name:` the name of the new subclass

Response Parameter: none

HTTP-Status: 201 (Created), 409[7] (Naming Conflict)

Get Properties - Message: Requests the RNG schema for properties of the class targeted by the URI[8]. See Section 4.2.4 for details.

HTTP-Method: GET

Parameters:

1. `properties:` NIL-value

Response Parameter:

1. `schema:` representing the requested RNG schema

HTTP-Status: 200 (OK), 410 (Gone)

Get Service Schema - Message: Provides the schema each new service (instance) description will be validated against[9]. For details see Section 4.2.5.

HTTP-Method: GET

[7]Status code 409 is also used if the class to register is not found. We see this as a conflict in the provided name and therefore use 409 instead of 410 (Gone) or 404 (Not Found).

[8]In Figure 4.1 we show these resources as part of the resource tree which is not completely accurate. Because a query parameter is needed to request them using the class resources they are technically not explicit resources. But they provide a discrete set of data (see Section 4.2.4) and because of their complexity we argue that they are independent resources even if they are not accessible by their own URI. Our reason why we implemented it as it is, is that this way we avoid naming restrictions for resources, keep the responded data narrow and provide an easy understandable API.

[9]We argue the same way as we did before (see *Get Properties* - Message) why in Figure 4.1 these are shown as explicit resources.

Parameters:

1. `service-schema`: NIL-value

Response Parameter:

1. `schema`: representing the requested RNG schema

HTTP-Status: 200 (OK), 410 (Gone)

Rename Class - Message: Renames the class targeted by the URI.

HTTP-Method: PUT

Parameters:

1. `new-name`: the new name of the class

Response Parameter: none

HTTP-Status: 200 (OK), 409 (Naming Conflict), 410 (Gone)

Delete Class - Message: Deletes the class and all its sub-resources targeted by the URI.

HTTP-Method: DELETE

Parameters: none

Response Parameter: none

HTTP-Status: 200 (OK), 410 (Gone)

Resource: Operations

As Figure 4.1 shows, this resource is provided beyond all three levels of the Repository and represents the set of supported class level operations of the referenced resource. It is accessible at `repo:uri/{class}/operations`. See Section 4.2.1 for details.

Get List of Operations - Message: Provides XML data representing a list of all implemented operations for the targeted class. Listing 4.1 gives a full example of such an XML.

HTTP-Method: GET

Parameters: none

Response Parameter:

1. `xml`: representing the XML list of all supported operations

HTTP-Status: 200 (OK), 410 (Gone)

Resource: Operation

Each of this resources represents one particular operation implemented for the class provided in the URI. They can always be found as an sub-resource of the operation

resource within the Repository and therefore are provided at all three levels of it (see Figure 4.1). A detailed explanation of these resources is given in Section 4.2.1. They are accessible at `uri:repo/{class}/operation/{operation}`.

Get Operation Workflow - Message: Requests the microflow definition of the operation targeted by the URI. In Chapter 5 and Section 4.2.1 we will discuss of what elements such microflows are composed.

HTTP-Method: GET

Parameters: none

Response Parameter:

1. `class-level-workflow:` representing the description of the class level microflow

HTTP-Status: 200 (OK), 410 (Gone)

Get Input Schema - Message: Requests the RNG schema of the input parameters[10]. For details about this schema see Section 4.2.2. An example of such a schema is given in Listing 4.3.

HTTP-Method: GET

Parameters:

1. `input:` NIL-value

Response Parameter:

1. `schema:` representing the requested RNG schema

HTTP-Status: 200 (OK), 404[11] (Not found), 410 (Gone)

Get Output Schema - Message: Requests the output parameter RNG schema[12]. Details about this schema are given in Section 4.2.2.

HTTP-Method: GET

Parameters:

1. `output:` NIL-value

Response Parameter:

1. `schema:` representing the requested RNG schema

HTTP-Status: 200 (OK), 404[13] (Not found), 410 (Gone)

[10] In Figure 4.1 we show these resources as part of the resource tree what is technically wrong. Like above (see *Get Properties* - Message) we argue that this is still a proper definition.

[11] We also use 404 to indicate that for the requested class level operation no input parameters were found.

[12] In Figure 4.1 we show these resources as part of the resource tree what is technically wrong. Like above (see *Get Properties* - Message) we argue that this is still a proper definition.

[13] We also use 404 to indicate that for the requested class level operation no output parameters were found.

Resource: Call

This resources represent the templates defined for a particular call in a particular class level operation. We discussed this kind of resources in Section 4.2.3 using an example. As shown in Figure 4.1 these resources are provided as sub-resources of **templates**. They can therefore be accessed at `repo:uri/{class}/operations/{operation}/templates/{call}`, `repo:uri/{class}/{subclass}/operations/{operation}/templates/{call}` and at `repo:uri/{class}/{subclass}/{instance}/operations/{operation}/templates/{call}`. All three URIs target the same resource.

Get Templates - Message: Requesting the defined XSL stylesheet for a call within a class level operation. The class level operation, the call ID and the class are named within the URI.

HTTP-Method: GET

Parameters: none

Response Parameter:

1. `templates:`representing all XSL templates defined for the requested call

HTTP-Status: 200 (OK), 404[14] (Not Found), 410 (Gone)

Resource: Subclass

These resources are representing particular subclasses. As shown in Figure 4.1 they can always be found as sub-resources of `class` elements and therefore are accessible at `repo:uri/{class}/{subclass}`.

Get List of Instances - Message: Provides a list with all registered instances for the targeted subclass of the URI.

HTTP-Method: GET

Parameters: none

Response Parameter:

1. `ATOM-feed:` representing information about the class and a list with all registered services (instances) beyond this URI

HTTP-Status: 200 (OK), 410 (Gone)

Create Instance - Message: Defines a new instance within the subclass targeted by the URI. We discussed instances in details in Section 4.3.

HTTP-Method: POST

[14]We also use 404 to indicate that although the requested class level operation was found, no call with the provided ID was found and therefore no templates could be provided. If the call was found but no templates were defined our implementation responds 200 with an empty templates-element. See Listing 4.5 for details.

Parameters:

1. `service-name:` defines the name of the new service (instance)
2. `service-description:` defines the instance level description as described in Section 4.3 for the new service

Response Parameter: none

HTTP-Status: 201 (Created), 409[15] (Naming Conflict), 415 (Invalid Description)

Rename Subclass - Message: Renames the subclass targeted by the URI.

HTTP-Method: PUT

Parameters:

1. `new-name:` the new name of the subclass

Response Parameter: none

HTTP-Status: 200 (OK), 409 (Naming Conflict), 410 (Gone)

Delete Subclass - Message: Deletes the targeted subclass and all its included services (instances).

HTTP-Method: DELETE

Parameters: none

Response Parameter: none

HTTP-Status: 200 (OK), 410 (Gone)

Resource: Instance

These resources are representing particular instances (services) within a class and subclass. What data is provided at this level was discussed in Section 4.3. These resources are accessible at the URI `repo:uri/{class}/{subclass}/{instance}`.

Get Instance Description - Message: Requests the microflow definition and the properties of the service targeted by the URI. In Chapter 5 we will discuss what elements are allowed within these microflows.

HTTP-Method: GET

Parameters: none

Response Parameter:

1. `instance-level-workflow:` representing the description of the instance level microflow and the properties of of the requested service

HTTP-Status: 200 (OK), 410 (Gone)

[15]Status code 409 is also used if the class or subclass to register is not found. We see this as a conflict in the provided name and therefore use 409 instead of 410 (Gone) or 404 (Not Found).

Update Instance - Message: Updates the definition of the instance targeted by the URI. It replaces its instance level microflow and its properties definition with the new ones provided with the request.

HTTP-Method: PUT

Parameters:

1. `service-description`: representing the new description of the instance

Response Parameter: none

HTTP-Status: 200 (OK), 410 (Gone), 415 (Invalid Description)

Rename Instance - Message:

HTTP-Method: PUT

Parameters:

1. `new-name`: defining the new name of the service (instance)

Response Parameter: none

HTTP-Status: 200 (OK), 409 (Naming Conflict), 410 (Gone)

Delete Instance - Message: Deletes the targeted instance.

HTTP-Method: DELETE

Parameters: none

Response Parameter: none

HTTP-Status: 200 (OK), 410 (Gone)

Chapter 5

Controlflow Description

This chapter focuses on the elements allowed within microflows. The defined instructions are inspired by the CPEE syntax (see Stuermer [45]) but are represented in XML and extended with composition specific attributes. Using BPEL [51] would also be possible, but to work with our approach it need to be extended with composition specific concepts too. We will discuss in detail what possibilities workflow designer have to define the different microflows at class level and at instance level. For better understanding we will illustrate the use of different statements using the Cinemas example (see Section 5.1). We start with explaining how data elements and endpoints can be defined (see Section 5.2). After we cleared this we will focus on how to call actual services (see Section 5.3). We explain how the input parameters are defined and how the execution results are provided in Section 5.3.1. We further show how the properties, which each service must define, are used to select a subset out of them using constraints in Section 5.3.2. Next thing we discuss are the different kinds of calls that emerge out of our design. They are discussed in detail in Section 5.3.3. At this point we give a short excursus about how calls are defined using CPEE syntax (see Section 5.3.4). How the already mentioned UI templates can be defined is explained in Section 5.3.5. What the purpose of a manipulate element is, is in the focus of Section 5.4. Decisions in the controlflow can be made using choose - alternative - otherwise statements as explained in Section 5.5. How conditions can be defined and grouped together is explained in Section 5.6. Loops are supported within microflows and therefore explained in Section 5.7. To allow the parallel execution of different branches we provide the parallel - branch statement which is explained in Section 5.8. Last statement explained is the critical statement (see Section 5.9). It is used to ensure that, in a parallel - branch at a time only statements included in the critical block are executed.

5.1 Demonstrator: Class and Instance Level Operations

We use the example defined above in Section 4.1 to illustrate how to use the controlflow statements explained in this section. We defined that each service within this application domain provides at least the instance level operations `search` to search for shows of a particular movie and the operation `book` to book tickets for a particular show. To realize the defined class level operation named `search_and_book` we show how to re-use the class level operation associated to the instance level operations search and book. Figure 4.2 illustrates the interrelation of the defined class level operations for the application domain Cinemas.

5.1.1 Class Level Operation: Search and Book

With this operation we provide the possibility for customers to search over all registered cinemas (within the requested URI) for a particular movie[16], select a show and book tickets for it.

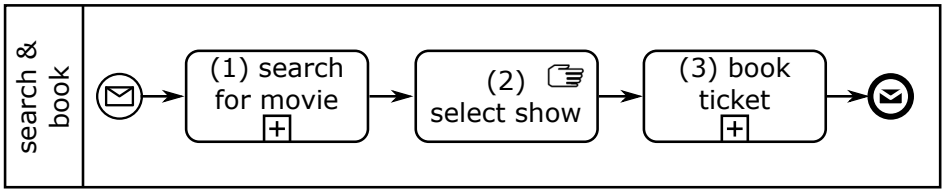

Figure 5.1: BPMN: Operation `Search & Book`

The controlflow we used to implement this operation is illustrated in Figure 5.1 and explained below:

1. Call the `search` operation from each cinema fulfilling the defined constraints.
2. Let the user **select a show** to book. To support data centered and UI centered clients we additionally provide templates for this activity.
3. Book tickets for the selected show using the operation `book` from the selected cinema.

5.1.2 Instance Level Operation: Search

Each service in this class provides at least the two operations `search` and `book`. To keep the example simple we only describe the operation `search` and how it is implemented by one of our real world cinemas we elaborated during the implementation of the prototype. Its microflow is illustrated in Figure 5.2. Here we give a step-wise explanation of it.

1. Transform input parameters to fit for the services interface.
2. Request FilmInfo using the provided `title` and `date` parameters.
3. Parse result of FilmInfo request and generate FimlmInfo - Array (transform FilmInfo).
4. For each Film from FilmInfo.
   (a) Get FilmID from FlimInfo - Array and request information according to the ID (request FilmInfo).
   (b) Extract information about each show from the response and add the information to the output data (add data to `output` parameter).

[16]BPMN uses a **Hand** sign in the upper right corner to indicate a so called *manual task*. Because in our example the task `(2) select show` has to be performed by a user we decided to use this symbol for this task.

5.2 Context Elements

In this section we show how workflow designers can define context variables to store information for later use and endpoints (resource URIs) used by call statements within a controlflow.

These definitions must be placed first behind the `declaration` element of the according operation. (e.g. operation `search` in Listing 5.1). First thing defined are the context variables. They are placed within an `context-variables` element (e.g. Listing 5.1 line 2). We differ between three declaration types:

Empty Variable: When a variable is declared for later use without defining anything about its type or content (e.g. Listing 5.1 line 3).

Initialized Variable: When a variable with an initialization value is defined. The designer does not define anything about the type of the variable. In our implementation this means that the type of the variable is assigned according to the Java Script Object Notation (JSON) [64] standard. The value of the variable is defined between its opening and closing tags. An example is given in Listing 5.1 line 4.

Typed Variable: This variables are defined directly by the designer using the *class* attribute. The value of the class attribute represents the exact definition of an variable in means of type and value. This kind of declaration bypasses the JSON interpretation and therefore allows the designer to specify a certain class (e.g. Listing 5.1 line 5). Further is it possible to use class operations provided by the underlying programming language. For example the operation `today` provided by the class `Date` (Listing 5.1 line 6) or the definition of the New Year's Eve (line 7) using the factory operation `civil`.

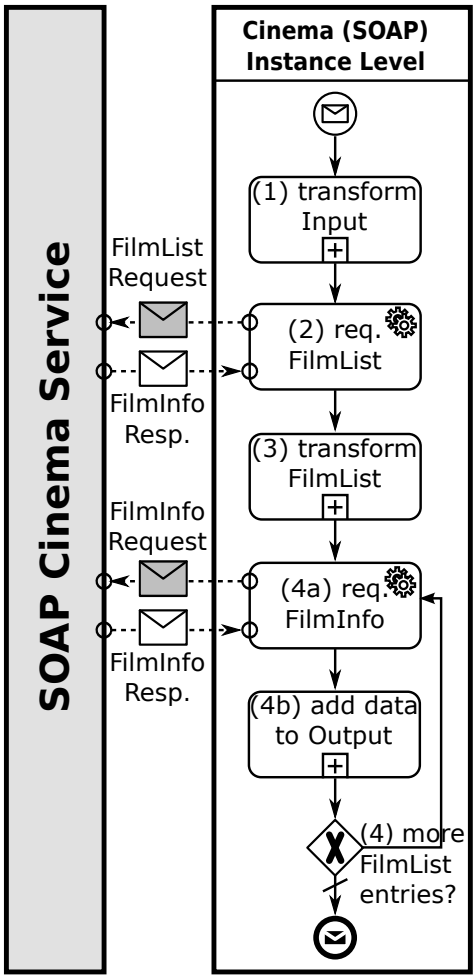

Figure 5.2: BPMN: An Example SOAP Service of a Cinema

LISTING 5.1: Definition of Context Elements

```
1 <search>
2   <context-variables>
3     <film_id/>
4     <hall_number>not available</hall_number>
5     <film_ids class="Array.new"/>
6     <today class="Date.today"/>
7     <new_years_eve class="Date.civil('2010','12','31',ITALY)"/>
8   </context-variables>
9   <endpoints>
10    <selected_cinema/>
11    <wsdl>http://some.cinema.org/online-tickets?WSDL</wsdl>
```

```
12      <soap_service>http://some.cinema.org/online-tickets</soap_service>
13    </endpoints>
14    <execute>
15      ...
16    </execute>
17    ...
18 </search>
```

Second thing that is declared are **endpoints**. In our understanding each service has a particular endpoint representing its URI. Designer can define an empty endpoint (Listing 5.1 line 10) if the particular service is not known at design-time but referenced within the controlflow. An example use case for empty endpoints is the operation `Search & Book`, where the particular cinema is selected at run-time but the call of the operation `book` is defined at design-time. Doing so allows the designer to fulfill semantic constraints of the controlflow (e.g. each call must have an endpoint) and allows to set the particular value at run-time. If the service is already known at design-time (e.g. most calls at instance level) the URI is written between the opening and closing tags of the element (lines 11 and 12). Only calls to SOAP services need additionally the URI of the according WSDL (see *Web Service Definition Language* [52] for details). Because it represents an URI it is defined as an endpoint even if it does not directly address a service.

5.3 Call Statement

Whenever a resource is used within a controlflow it is referenced to it by a call statement. Every call statement mainly consists of three parts: (1) Parameters defining the input and the results of a call (see Section 5.3.1), (2) Service Constraints allowing to define a query for service selection at design-time (see Section 5.3.2) and (3) Templates allowing for UI definition within the service description (see Section 5.3.5). Further do we differ between three different types of calls which are discussed in Section 5.3.3.

5.3.1 Parameters

Whenever an external resource or an other service operation in the repository is called it is very likely that some values (input parameters) are needed by it to provide the expected functionality. Further may it provide some information after the execution (output parameters / result object) to the caller.

In Listing 5.2 we show how this information is defined during the design-time. We differ parameters at two dimensions: first is the **direction** defining if it is an input (line 4 - 13) for or output (lines 16 - 23) of the call. Second is the **scope** of the parameter. Here the designer defines if the parameter is a pass-through from or to an other operation and therefore **part of a message** (lines 4 and 16) or if it is an intermediate value and therefore only **locally** stored and accessible (lines 12 and 23).

LISTING 5.2: Definition of Input and Output Parameters

```
1<call id="..." ... >
2
3   <!-- Input parameter (message) -->
4   <input name="..." message-parameter="...">
```

```
5     <rng:element name="..." >
6       <domain:caption xml:lang="en" >...</domain:caption>
7       ...
8       <rng:data type="..."/>
9     </rng:element>
10  </input>
11
12  <!-- Input parameter (local) -->
13  <input name="..." variable="..."/>
14
15  <!-- Output parameter (message) -->
16  <output name="..." message-parameter="..." >
17    <rng:element name="..." >
18      <rng:data type="..."/>
19    </rng:element>
20  </output>
21
22  <!-- Output parameter (local) -->
23  <output name="..." variable="..."/>
24
25 </call>
```

The listing also shows that each parameter defines its structure and type of the elements. This way it is possible to implement parameters of a complex type. The structure is describe using RNG schema language and supports the whole instruction set (e.g. optional, zeroOrMore, ...). Exposing this definitions through the class level schemas allows developers to predict exactly how the input or output of a call looks like and therefore can easily implement the interface into their own application or service.

Input Parameters

Each defined input parameter is transmitted to the targeted resource under the name defined in the attribute *name*. Depending on the type of the call (see Section 5.3.4 for details) the parameters are either part of the HTTP-message or packed into an SOAP envelope. The needed functionality to build these SOAP envelopes is implemented in the Activity Handler described in Section 7.3.1.

The source of the parameter is defined in the second attribute. If the attribute is of the type *message-parameter* its origin is in the incoming message of the operation. When parameters are declared this way no local copy of them is used and also no transformation is performed on their value. The second type of input declaration uses the attribute *variable* to define the source of the input. This type is used when the value is provided directly within the scope of the operation. This could either be as an result of a prior call or the outcome of a transformation (see Section 5.4 for details).

Additional Endpoints

As we already explained, some operations may need endpoints provided by the caller (user/designer or other operation) to work properly. In our example the operation `Serach & Book` needs to know which resource/service is used to select the particular show (provided by

the user/designer) and which cinema was selected (provided by this operation) to book the tickets.

LISTING 5.3: Definition of Additional Endpoints as Input Parameter

```
1<call id="..." endpoint="..." endpoint-type="outside" http-method="..." info
    ="..." ...>
2   ...
3</call>
```

We define these endpoints by using the *endpoint-type* attribute in the call statement. Valid values for this attribute are **outside** which defines that the endpoint is found in the scope of the calling controlflow. If the attribute is not set or the value of it is **inside** the referenced endpoint is declared in the actual scope of the operation.

Output Parameters

Similar to the input parameters of a call statement there maybe output parameters too. We already showed in Listing 5.2 how they are defined. The name and the type of each parameter is declared the same way as input parameters. Because these parameters may also be of a complex type (similar to input parameters) they are again described using RNG schemas. Additionally to the declared variables and message-parameters each call adds one special variable to the result called `status`. If the HTTP-status of a call to an external service/resource or an instance level call should be directly included into the `status` parameter of the output-message, a message-parameter named `status`, defining the attribute *type = 'status'*, must be defined. If the status code should not be the result of a HTTP-call (REST or SOAP) it is possible to set its value using manipulate statements referring the message-parameter `status`. See Listing 5.11 for an example.

5.3.2 Service Constraints

We already mentioned that we support the definition of constraints related to the service selection within an operation. It gives the process designer the possibility to narrow the set of services used within an operation and helps to save bandwidth and computing resources. Further can it be used to avoid legal issues e.g. restrictions of countries where an operation is provided.

LISTING 5.4: Definition of Service Constraints

```
1<call id="..." endpoint="..." ...>
2  <constraints>
3    <constraint xpath="adress/city" comparator="==" variable="city"/>
4    <constraint xpath="adress/zip" comparator="&lt;" value="1900"/>
5  </constraints>
6  ...
7</call>
```

Listing 5.4 shows that constraints must be defined within a `constraints` element. Within this element `constraint` elements can be combined as logic expression using `group` elements (see Section 5.6 for details). Within a constraint element the following attributes are used to specify it:

1. The *xpath* attribute is used to specify the element within the services properties to be validated. Our system supports all functions provided by the XML Path Language (XPATH) specification [53].
2. The *comparator* attribute allows to specify how the referenced values are compared against each other. At this point all symbols defined in the underlying programming language are supported. For our prototype implementation this is the Ruby programming language [65] which allows to use the methods defined for the referenced properties after it is parsed by JSON.
3. The *variable/value* attributes define the value the property is compared with. Our implementation allows the designer to define a static value using the *value* attribute or refer to a variable within the local scope using the *variable* attribute. Using the variable attribute allows the designer to define a dynamic service selection depending on former user input or results of other calls. E.g. the actual workload of different resources provided within a Cloud infrastructure like in *Adaptive Pegasus* [26].

If constraint elements are not defined within a group element they are automatically connected by an logical **and**. They behave similar to the conditions described in Section 5.6.

5.3.3 Types

.

When a controlflow is executed three different types of calls may occur. First there may be calls referring to an other operation provided by the same resource like we do in the operation `Search & Book` when we use the operation `search` at the beginning. This causes our system to inject the `search` microflow into the actual one. Second will be a call referring to an resource outside the scope of the repository (external). This may be resources provided by the user like the call to the resource selecting a particular show in our example (see Figure 5.1: (2) select show). These two types may occur within class level microflows. Last are calls referring to actual services of the vendors like the search operation of a particular cinema in our example. These type of calls is only allowed within instance level microflows and defined by the vendor of the service.

Class Level: Injection Call

An injection call is performed whenever a call results in the injection of a microflow. This happens when the execution of an operation from a certain application domain within the Repository is requested.

LISTING 5.5: Definition of an Injection Call

```
1<call id="exec_search" endpoint="resource_path" service-operation="serach">
2  <input name="..." ... />
3  <output name="..." ... />
4</call>
```

An injection call is identified by the attribute *service-operation* which refers to the name of the requested operation in the repository located at the URI defined in the attribute *endpoint*. Listing 5.5 shows how the operation `Search & Book` includes the operation `search`

provided by the same application domain. The endpoint **resource_path** indicates that the same resources as referenced by the parent operation should be used e.g. `repo:url/Cinemas/Arthouse`. We use this keyword to pass endpoints automatically from the initial call statement to its succeeding microflows in a transparent way. Using this allows the designer of the class level microflow referring to the initial value of the defined endpoint.

5.3.4 Excursus: Injection Call in CPEE Syntax

Because our prototype is intended to cooperate with the already mentioned CPEE execution engine there must also be an pendant to the injection call in CPEE syntax. We do not want to explain this syntax in details but an example of such one should be enough to get a basic understanding of these type of calls.

LISTING 5.6: Definition of an Injection Call using CPEE Syntax

```
1<call id="exec_search" endpoint="services">
2  <constraints>
3    ...
4  </constraints>
5  <parameters>
6    <info>true</info>
7    <service>
8      <serviceoperation>"search"</serviceoperation>
9      <injection_handler>endpoints.injection_handler</injection_handler>
10     <count>0</count>
11   </service>
12   <additional_endpoints>
13     <selector_service>"select"</selector_service>
14   </additional_endpoints>
15   <parameters>
16     <title>context.title</title>
17     <date>context.date</date>
18   </parameters>
19 </parameters>
20 <manipulate output="result">
21   if result[0]['status'] == 200
22     context.reservation_number = result[0].value('reservation_id')
23     context.selected_cinema = CGI::unescapeHTML("#{properties.value('
           perform_book')[0].value('name')}")
24   end
25 </manipulate>
26 </call>
```

Constraints: They are defined and work the same way as they do when defined within the repository. See Section 5.6 for details.

Service - Block: These block includes the requested *service-operation* which is similar to the attribute *service-operation* explained in Class Level Injection Calls. Because an URI of an Injection Handler is only needed here and not within the Repository there is no pendant to the `injection_handler` element there. This endpoint refers to the Injection Handler that will be used when this injection is performed. For details on the Injection Handler see Section 7. The `count` element indicates the number of services that must at least have finished before the execution continues when the actual services are executed in parallel. This value will be used for the *wait* attribute of an `parallel`

element (see Section 5.8 for details). Because this restriction should only be used when designing actual applications (not class level microflows) it exclusively available for the CPEE call statement. It will be used if e.g. the execution should continue if two out of all matching services, regardless which or how many they were, finished their execution.

Additional Endpoints - Block: This block provides URI for the endpoints referenced within the according microflow description. It is important that each of these provides the URI of an feasible service. For details on the topic of additional endpoints see Section 5.3.1. How a list of the required endpoints can be requested is explained in Section 4.2.2.

Parameters - Block: In CPEE syntax this indicates that all elements included in this block are provided as parameters to the service call. Within this block each parameter named in the input schema (see Section 4.2.2) must be defined as element with the name from the schema. Its value is represented in the context of the element. As the CPEE supports Ruby code at this level context variables are accessed using the prefix `context.` (see line 15 and 16 in Listing 5.6) and endpoints are accessed by prefixing `endpoints.` (see Listing 5.6 line 9). If the value of the parameter is directly written into the call statement it allows all instructions that Ruby allows too e.g. String, Integer, Class instantiation, ...

Manipulate Block: This block allows to access the output message of the call using Ruby syntax. The *output* attribute specifies the name representing the result object. Again the CPEE uses arrays for this purpose. Each element of this array represents a key/value Class (e.g. Hash Class) implementing the structure defined for the output-message. We further extended this class with the function `value(String name)` to allow to access the first entry with the provided name within all sub resources of the Hash (see Sections 4.2.2 and 5.3.1 for further details). An example how to use this function is given in Listing 5.6 line 22 where the parameter `reservation_id` is requested from the result object / output-message. Additional to the result object there is an properties object provided. This object provides all properties defined for this application domain (see Section 4.2.4 for details) of all executed services. Services that were not included in the execution because they are not within the resource path or their properties did not fit are not included. Again this object is type of our extended Hash Class. An example for using the properties object is given in Listing 5.6 line 23 where the property *vendor/name* of the cinema used for the call `perform_book` is requested.

Class Level: External Call

An external call is used to perform calls to resources unknown (external) by the Repository. Usually this type of calls is used for user specific resources during the execution. That's why these type of calls can be identified in class level by defining the attribute *endpoint-type='outside'* meaning that its value is defined in the input-message of the initial call.

LISTING 5.7: Definition of an External Call

```
1<call id="exec_select" endpoint="selector" endpoint-type="outside" http-
     method="post" info="true" default-tpl-name="mobile" default-tpl-lang="en">
2  <input name="..." ... />
3  <output name="..." variable="..."/>
4  ...
5</call>
```

As shown in Listing 5.7 the definition further includes the following attributes:

1. *http-method:* This attribute defines which HTTP-method is used for the request. Valid values are: GET, POST, PUT, DELETE
2. *info:* This attribute defines if information about the execution engine and its state are provided as parameters. This information is of use when user specific resources want to interact with the execution engine. Valid values are true and false. If the attribute is set to true the following parameters are added by the Activity Handler automatically:
   (a) *call-instance-uri:* Provides the URI of the associated execution instance.
   (b) *call-activity:* Provides the ID of the associated call.
   (c) *call-endpoint:* Provides the value of the *endpoint* attribute defined for the call.
   (d) *call-lay*[17]*:* Provides the number of the lay defined for the actual call.
   (e) *call-oid:* This is the original ID of the associated call. It is only provided if the call has been transformed or injected at least one time before its execution.
3. *default-tpl-name/default-tpl-lang:* Defines which of the UI templates, defined within this call, is used as default if no other template is defined by the caller using the `tpl-name` and `tpl-lang` parameters.

Instance Level: Native Call

Native calls are only defined within instance level microflows and refer strictly to resources provided by the vendor responsible for it. Because each vendor service must work with only the information defined in the according injection call we do not allow the definition of UI templates here. Therefore a native call consists mainly of input and output parameters. These parameters differ in their definition from the already introduced ones in Section 5.3.1 as they define more technical details like the parameter type of an HTTP-response.

So far we assumed that each call is based on plain HTTP (REST) as protocol because they were all addressed to our/from the Repository but at this level we additionally support SOAP calls. We explain in the next two paragraphs how each of them is defined.

REST Calls: These type of call provides an attribute named *http-method* additionally to the attributes *ID*, which has to be unique within the microflow, and *endpoint*, which again provides the URI of the referenced resource. An example definition of such a call is given in Listing 5.8.

[17] Lays are used to differ between calls executed in a parallel - loop - branch construction. See CPEE papers for details.

If the referenced HTTP-method is `GET` the input parameters are treated as **query parameters**. Any other valid HTTP-method has its input parameters treated as **body parameters** within the request. As stated by Fielding [25] do RESTful resources (supporting the CRUD paradigm) work this way as they usually do not expect large input data by GET-requests. The *name* attribute specifies the name of the parameter. The order of them is the same as the **order of their definition**. As second attribute the already known *message-parameter* or *value* attributes are valid. An example of a parameter definition is given in Listing 5.8 line 2.

Output parameters define the target they are stored in (*message-parameter* or *variable*) as explained in Section 5.3.1. But if they are part of a native call they define additionally a *type* attribute which can be **simple** (for query parameters) or **complex** (for body parameters) (see Listing 5.8 line 3) and therefore over-ruling the defaults described above. If the value of this attribute is **status** the HTTP-status code of the response is assigned to the parameter (see Listing 5.8 line 4). Last the name of the referenced parameter has to be given in the *name* attribute. As requesting resources like HTML pages often results in one response parameter without a name it is allowed to define one output parameter without a name. If this is the case the first parameter of the response parameter set will be assigned.

LISTING 5.8: Definition of a Native REST Call

```
1<call id="Programm" endpoint="service" http-method="get">
2  <input name=".." .../>
3  <output variable="response" name="" type="complex"/>
4  <output message-parameter="status" type="status"/>
5</call>
```

SOAP Calls: This type of calls can be identified by the *soap-operation* and *wsdl* attribute in their definition. The soap-operation attribute refers to the requested operation in the WSDL (see *Web Service Definition Language* [36] for details). The location of the WSDL is defined within the endpoint referenced in the *wsdl* attribute. Listing 5.9 shows an example definition of such a call.

Input parameters are defined in the usual way. Packing the parameters into an SOAP envelope, as it is necessary for SOAP calls, happens in the Activity Handler (see Section 7.3.1 for details).

Output parameters define their target as already explained by naming either a message-parameter or a variable in the according attributes. Additionally the parameters define either the *type* attribute or a pair of *name / namespace* attributes. The *type* attribute for SOAP parameters only allows the value **status** to refer to the return status of the operation. Because SOAP provides a numeric status only within error-messages (see `faultcode` in [35] for details) it is set to NIL if the execution was without any error. Other values are not needed because of the way SOAP handles parameters (see SOAP Specifications [35] for further information). The *name* attribute of an SOAP parameter represents an XPATH query referring to the desired data in the response message. Because this response message usually consists of different namespaces, defining different parts of the SOAP message, the *namespace* attribute defines the **target namespace** (tns) which is used to identify the result parameters of the SOAP service within the message. Listing 5.9 line 3 gives an example for the definition of a SOAP output parameter.

LISTING 5.9: Definition of a Native Soap Call

```
1<call id="FilmInfo" endpoint="soap_service" soap-operation="FilmInfo" wsdl="
    wsdl">
2  <input name="..." .../>
3  <output variable="soap_response" name="descendant::tns:FilmInfoResult"
      namespace="http://sitec.at/Service"/>
4  <output message-parameter="status" type="status"/>
5</call>
```

5.3.5 UI Templates

User Interface (UI) Templates are used to define an UI within a call statement. We implemented this to provide a way for designers of class level microflows to request some user interaction. Applications using such a class level microflow can either display the UI directly (like our Worklist described in Section 8.4) or parse the data and use it as a basis to generate their own interface. The definition of them is provided by the repository as explicit resources and can be requested at `repo:uri/{class}/{subclass}/{instance}/operations/{operation}/templates/{call}`.

The XML data that should be transformed by the XSL transformation must be provided by the input parameter called `data`. For an example how to access data provided by this parameter from within the template see Listing 5.10. Every other input parameter is added to the XSLT stylesheet as a variable. They can be referenced using its name with a leading $-sign (as defined in the XSLT standard [56], e.g. $instance-uri). Additionally we add a few parameters to the XSLT which are of use most of the time (see also the description of the *info* attribute in Section 5.3.4).

instance-uri: In our implementation each CPEE instance can be addressed as RESTful resource using this URI. We provide this to let the user of the template know which instance is affected by it. See Listing 5.10 line 33 for an usage example.

oid: OID stands for *original ID* and holds the value originally given in the *ID* attribute of the call statement providing the template.

activity: This variable provides the actual ID of the call responsible for the execution of the template. As we explain in Chapter 7 this must not be the same as initially defined in the Repository.

callback-id: When the interaction with the user is finished the CPEE has to be informed. Therefore we provide by this variable an URI to post the data to. See List. 5.10 line 33 for an usage example.

LISTING 5.10: Definition of an UI Template

```
1<templates>
2  <xslt name="mobile" xml:lang="en">
3    <xsl:stylesheet version="1.0" xmlns:xsl="...">
4      <xsl:output method="html"/>
5      <xsl:template match="/">
6        <html>
7          <head>
8            <script type="text/javascript" src="..."/>
9            <script type="text/javascript">
10             function send_selection(data, callback) {
```

```
11                  ...
12                }
13              </script>
14            </head>
15            <body>
16              <table border="1">
17                <xsl:apply-templates select="//show"/>
18              </table>
19            </body>
20          </html>
21        </xsl:template>
22
23        <xsl:template match="//show">
24          <tr><td><table>
25          <tr><td>Cinema:</td><td><xsl:value-of select="cinema_uri"/></td></tr>
26          ...
27          <tr><td colspan="2"><xsl:element name="input">
28            <xsl:attribute name="type">button</xsl:attribute>
29            <xsl:attribute name="value">Select</xsl:attribute>
30            <xsl:attribute name="onClick">JavaScript:send_selection({
31              'show_id' : '<xsl:value-of select="show_id"/>',
32              ...
33            },'<xsl:value-of select="$instance-uri"/>/callbacks/<xsl:value-of
                  select="$callback-id"/>');
34            </xsl:attribute>
35          </xsl:element></td></tr>
36          </table></td></tr>
37        </xsl:template>
38      </xsl:stylesheet>
39    </xslt>
40
41    <xslt name="mobile" xml:lang="de">
42      ...
43    </xslt>
44  </templates>
```

In Section 8.4 we given detailed examples how UI templates are properly used within a business process or class level microflow.

5.4 Manipulate Statement

Each manipulate statement consists of two major parts. First there is the declaration of context elements (Listing 5.11 lines 2 - 13). Second is some programming code in a language the execution engine is able to execute. In the case of our prototype implementation (using the CPEE) this is Ruby code [65]. Declaring `variable` elements within a manipulate statement ensures that, when the programming code is executed, the defined variables have the same value as the referenced context elements and when the execution is finished, the value of the variables is copied back to the context elements. Doing so allows us to have transparent access to the context elements of the execution environment within a programming language and therefore allows way more complex and efficient code.

Generally we differ between four types of context elements. Each of them has its own declaration:

Local variables: This links the referenced context variable (*context* attribute) to the programming variable referenced in the attribute *local*. (See Listing 5.11 line 3)

Local Endpoints: This links the endpoint referenced in the attribute *endpoint* to the programming variable referenced in the attribute *local*. (See Listing 5.11 line 5)

Input-Message Parameters: This refers to a parameter in the input message of the operation. Because input message-parameters are read-only the changes in the programming variable are not copied back to it. The attribute *input-parameter* defines which parameter of the message is linked to the programming variable referenced in the attribute *local*. See Listing 5.11 line 7 for an example.

Output-Message Parameters: This allows to generate parameters for the operations output-message. To be in line with the general definition of output parameters it is necessary to define also multi-lingual captions and the type of the parameter. Such statements link the parameter referenced in the attribute *output-parameter* to the local variable referenced in the attribute *local*. See List. 5.11 lines 9 - 13 for an example.

LISTING 5.11: Example of a Manipulate Statement

```
1 <manipulate id="merge_result">
2   <!-- Referring to a local variable -->
3   <variable context="list_of_data" local="lst"/>
4   <!-- Referring to a local enpoint -->
5   <variable endpoint="selected_cinema" local="sel_ep"/>
6   <!-- Referring to an input-message parameter -->
7   <variable input-parameter="title" local="movietitle"/>
8   <!-- Defining an output-message parameter -->
9   <variable output-parameter="status" local="stat">
10    <rng:element name="status">
11      <rng:data type="positiveInteger"/>
12    </rng:element>
13  </variable>
14
15  # Some Programing Code is defined here
16 </manipulate>
```

5.5 Choose - Alternative - Otherwise Statement

The Choose - Alternative - Otherwise Statement provides a way to define conditional paths within the controlflow. Supporting these statements allows controlflow designers to define that some parts of the microflow are only executed if some conditions are fulfilled. How these conditions are defined is discussed in Section 5.6. Listing 5.12 shows an example with one alternative path and a default path. The statement begins with the opening `choose` element. Child elements of it can only be one or more `alternative` elements and one `otherwise` element. When the execution comes to a choose it continues all paths where the conditions are fulfilled. If no alternative matches it chooses the otherwise path. The execution continues after every chosen path has finished its execution.

LISTING 5.12: Example for a Choose - Alternative - Otherwise Statement

```
1 <choose>
2   <alternative>
3     <!-- Some condtions are defined here -->
4     ...
5   </alternative>
6   <!-- Here could be more alternative elements -->
7   <otherwise>
8     ...
9   </otherwise>
10 </choose>
```

5.6 Groups and Conditions

As already mentioned is it possible to connect two or more conditions (similar to service constraints explained in Section 5.3.2) together as logical expressions using `group` elements. Listing 5.13 shows in the second example (lines 5 - 10) that it is possible to nest group elements. The first group element defines that the conditions are connected by a *logical or*, meaning that the result of the first condition or the overall result of the second group must be true. The second group element uses a *logical and* as connector what means that both conditions must result to true to make the whole group true. If two or more conditions are defined outside of any group element they are handled like they were within an and-connected group element.

Each group element consists of two or more `condition` elements. These elements defines actual logical expressions which are resolved at runtime. As shown in Listing 5.13 each condition element consists of three attributes:

1. The *test* attribute (left-hand operator) defines against which context variable the test should be executed. Because only context variables are allowed here any other object (e.g. endpoints, message-parameters, ...) must be assigned to a context variable in a prior manipulate statement.

2. The *comparator* attribute defines which method[18] of the referenced object in the *test* attribute is executed during the test. We use this concept also in our implementation and therefore gain high flexibility when it comes to comparison. Every comparison method supported by the referenced object can be used for this attribute. As shown in the second example we use the `includes` method of a String object to test if the movie title contains the strings 'IMAX' and '3D'. (see Listing 5.13 line 8 and 9)

3. The *variable/value* attribute (right-hand operator) declares against what value or variable the test will be executed. It is important to ensure that the defined comparison method of the left-hand operator supports the value of the right-hand operator otherwise the test result may be wrong or the execution fails.

[18] In Ruby every comparison results in the execution of an object-method, e.g. a == b is resolved in a.==(b).

LISTING 5.13: Example for Conditions and Groups

```
1 <!-- Check if there was at least one show found -->
2 <condition test="number_of_shows" comparator="&gt;" value="0"/>
3
4 <!-- Check if price is low enough or the show is in 3D and IMAX -->
5 <group connector="or">
6   <condition test="price" comparator="&lt;" variable="max_spending"/>
7   <group connector="and">
8     <condition test="title" comparator="includes" value="3D"/>
9     <condition test="title" comparator="includes" value="IMAX"/>
10  </group>
11 </group>
```

5.7 Loop Statement

This statement allows the designer to implement a while-loop[19] within the microflow. In our controlflow language we do not support do-while-loops. A loop is indicated by a `loop` element followed by `condition` and `group` elements (see Section 5.6 for details).

This kind of loops is also known as head-driven loop. Conditions can be defined as already explained in Section 5.6.

LISTING 5.14: Example for a Loop Statement

```
1 <loop>
2   <!-- Conditions and Groups are defined here -->
3   <condition test="loop_count" comparator="&lt;" value="5"/>
4   ...
5 </loop>
```

In the example above the controlflow inside the loop element will be executed as long as the context variable `loop_count` is lower than five.

5.8 Parallel - Branch Statement

This statement allows designers to define paths that are executed in parallel. It is defined by a `parallel` element which includes at least one `parallel_branch` element as child. The parallel element supports one attribute named *wait*. The value of this attribute defines how many branches have at least to be finished before the execution continues after the parallel element. If this attribute is not present or its value is zero the execution does not continue before every branch is finished. Inside these `parallel_branch` elements the different paths are defined allowing all controlflow statements.

[19] In while-loops the condition is resolved before the path inside the loop is executed in contrast to the do-while loop where it is resolved after the execution. Do-While loops are only supported by the CPEE using the post_test - attribute instead of the *pre_test* attribute.

LISTING 5.15: Example for a Parallel - Branch Statement

```
1 <parallel wait="1">
2   <parallel_branch>
3     <!-- Path One -->
4   </parallel_branch>
5   <parallel_branch>
6     <!-- Path two -->
7   </parallel_branch>
8 </parallel>
9
10 <call id="after" ... />
```

In the example above the call statement with the id `after` is not executed until at least one branch finished its execution as indicated by *wait='1'*.

5.9 Critical Statement

This statement allows the designer to define groups of statements that are executed exclusively at a time. To interconnect different `critical` elements we provide the attribute named *ID*. The value of this attribute indicates which elements are interconnected and therefore not executed simultaneously.

LISTING 5.16: Example for a Critical Statement

```
1 <parallel wait="1">
2   <parallel_branch>
3     <!-- Some statements -->
4     <critical id="myCritical">
5       <!-- Exclusive statements -->
6     </critical>
7     <!-- Some statements -->
8   </parallel_branch>
9   <parallel_branch>
10    <!-- Some other statements -->
11    <critical id="myCritical">
12      <!-- Other exclusive statements -->
13    </critical>
14    <!-- Some other statements -->
15  </parallel_branch>
16 </parallel>
17
18 <critical id="myCritical">
19   <!-- More exclusive statements -->
20 </critical>
```

The example above shows how it can be ensured that a group of interrelated variables is always consistent even if they are written in two different branches. As the *wait* attribute of the `parallel` element indicates the execution continues after the first branch has finished. We further assume that after the parallel statement the context variables, set in one of the branches, are used. By grouping them also into a `critical` element with the same ID as the other two we ensure that even if the remaining branch access the variables at the time the controlflow reads them, the execution waits till the branch as left the critical block and therefore guarantee that the variables are always consistent in the means of being all set in the same branch.

Chapter 6

Injection Service

This chapter focuses on the component which is in charge of injecting class and instance level microflows provided by the Repository (see Chapter 4) into a business process without interfering with the original intend of the process. We start this chapter with a detailed step-by-step example (Section 6.1) illustrating what happens during an injection using the Cinema example (see Section 4.1 for details about the example). Section 6.2 gives a detailed discussion about the provided RESTful interface of this service. It explains what HTTP-messages are computed and how possible responses may look like. In the next section (Section 6.3) we discuss the algorithm developed to inject class level and instance level microflows in respect to correctness criteria and without interfering with the original intent of the process. We introduce our implementation (Section 6.3.1) using pseudo-code listings and XML snippets. With a detailed discussion about the operations and elements, used by our algorithm, we give a comprehensive explanation of the overall functionality. This section ends with a discussion about why and how loops are treated in a special way (Section 6.3.2).

6.1 Demonstrator: Service Injection

In this section we use the Cinemas example (see Section 4.1) to illustrate how our Injection Service works. As described by the example the operation `search & book` is provided within the *Cinema* domain. We will show in the following step-by-step introduction how the controlflow is changed when this operation is referenced within a controlflow named *Book Movie Tickets*.

Figures 6.1 - 6.4 use BPMN notation. As BPMN was not designed for adaptive workflow systems it lacks of some concepts. Therefore we extend/refine the BPMN notation:

1. **Injection Calls:** Activities including a rescue-ring in the lower right corner represent injection calls. Activities of this type cause always a change of the controlflow.

2. **Generated Activities:** Activities that are derived by the injection algorithm in respect to correctness criteria are marked with a script roll in the lower right corner. In BPMN these activities are called *Scripts* meaning that they are autonomous from any external service and usually for maintenance of the instance. This is very similar to our

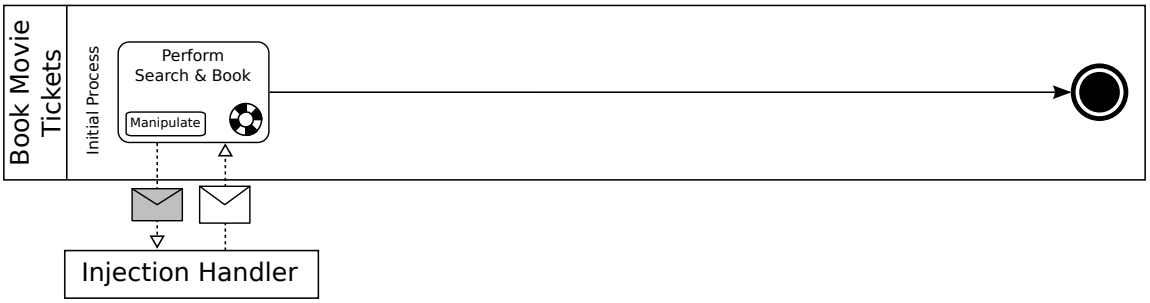

Figure 6.1: Step 1: Initial Process Using the Operation `Search & Book`

usage with the difference that we additionally indicate that these activities were not define by the designer or the user.

3. **'Call - Manipulate' Statements:** The CPEE supports call activities with an included manipulate statement to process the result object of a service call. We represent such statements by activities including a sub-activity called *Manipulate* in the lower left corner. Sub-activities are a concept which is included in the BPMN notation but we want to explicit explain it here to avoid misunderstandings.

Step One: Figure 6.1 shows the initial controlflow *Book Movie Tickets*[20]. The process designer only defined that the operation `search & book` should be executed whenever this process is instantiated and executed by a user.

As the message notation indicates this activity communicates only with the Injection Handler. When it is executed it registers the execution instance by the Injection Handler by requesting the injection of the controlflow represented by the operation `search & book` from the Repository.

Step Two: Figure 6.2 shows how the controlflow looks like after the class level injection of the `search & book` microflow has finished. The newly injected controlflow starts directly behind the derived *Create Objects* activity. In the top lane[21] can also be seen that the *Manipulate* statement, derived from the initial `Perform Search & Book` activity, is placed directly after the newly injected microflow and the clean-up statement of it at the very end. The execution position after the injection was performed is *after Perform Search & Book*.

When the execution comes to the activity *Call Find* it again requests the injection of a microflow provided by the Repository from an Injection Handler. This time the requested microflow is represented by the class level operation `Find` in the domain *Cinema*. As we explained already does our system supports the definition of nested class level operations which is shown here.

[20]To avoid unnecessary complexity within the diagrams we left out activities related to input data collection our result presentation in our example. See Section 8.2 for a more real world example definition.

[21]In BPMN lanes are a way to visualize different aspects/views of one process instance within one diagram. They are not to be confounded with pools which indicate an other process instance.

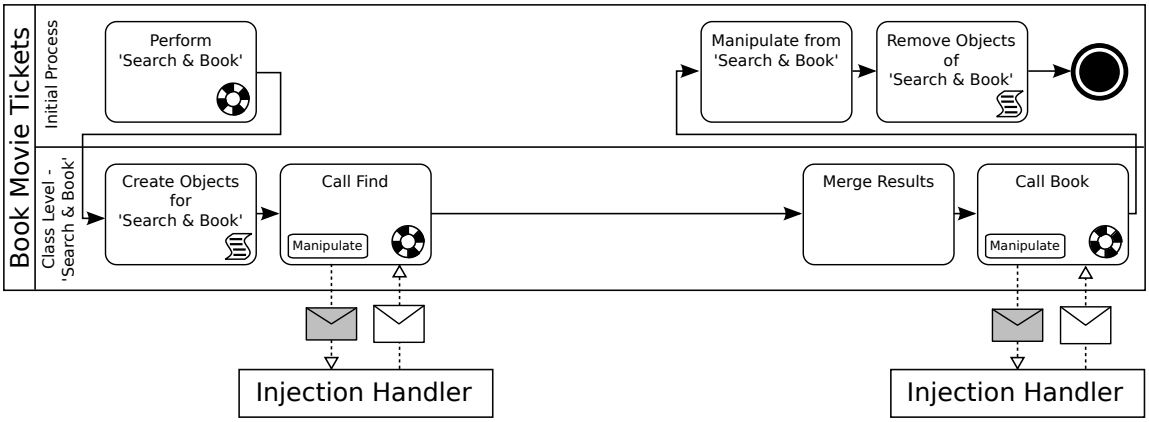

Figure 6.2: Step 2: Class Level Microflow of the Operation `Search` was Injected

Step Three: Figure 6.3 shows how the controlflow looks after the class level microflow of the operation `Find` was injected. Again, the new microflow is presented in its own new lane starting with the according *Create Objects* activity. Also are the *Manipulate* statement and the clean-up statement of it positioned directly after the new microflow. After the injection is finished the actual execution position is set to *after Call Find*.

When the execution reaches the activity *Perform Search* it again requests an injection from the Injection Handler. This time the instance level microflows of the operation `Find` from all matching services at the given endpoint will be injected.

Step Four: In Figure 6.4 the instance level microflows of two matching services are added (represented by an activity with a + at the bottom denoting a sub-process). Again they are in an own lane. As defined, each service is represented by a separate branch within a parallel statement[22] starting with the *Create Objects* statement and closed with the clean-up statement. In contrast to class level microflows, all manipulate statements needed to compute the results of the service call must be placed directly into the microflow allowing the clean-up to happen directly at the end of it. The actual execution position is set *after Perform Search* when the injection is finished. Further, as indicated by the message flow, do instance level microflows only communicate with their associated services and do not include the Repository anymore.

6.2 Interface

In our architecture only the Injection Handler (see Chapter 7) starts an interaction with an Injection Service. Whenever the Injection Handler requests an Injection Service it delegates a particular injection to it (in Figure 3.1 this is represented by *Step Two*) and expects the adapted controlflow description as result. To do so, the Injection Service expects the following input-message included in an HTTP POST-request:

[22]The parallel branches are merged using a **Complex Merge** symbol as defined by BPMN, because not every branch must necessarily be finished before the execution continues. See the *count* attribute of the call statement in Section 5.3.4 for details.

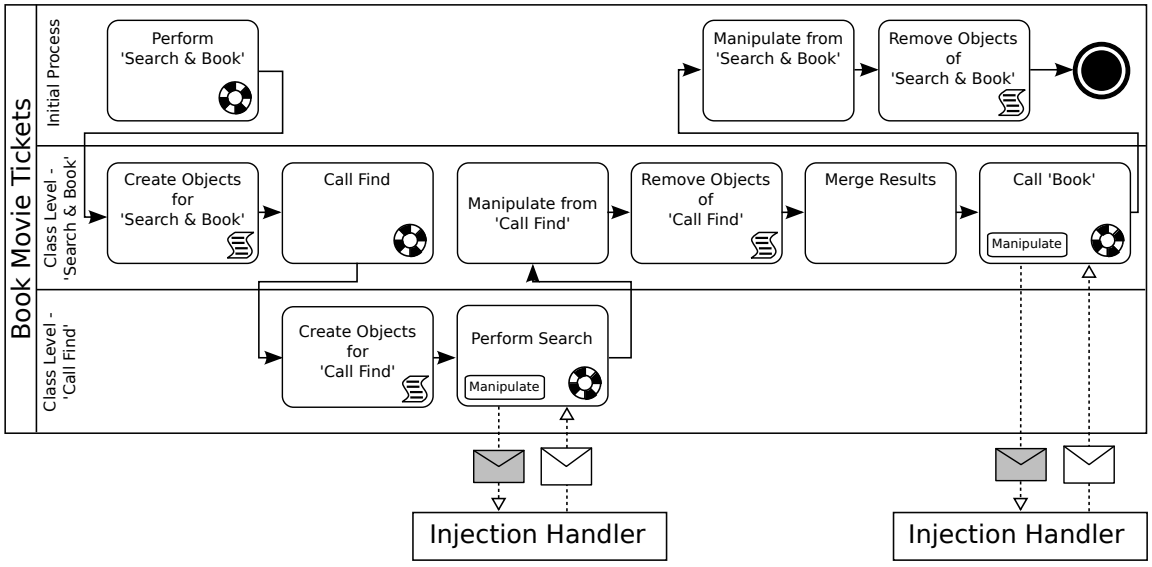

Figure 6.3: Step 3: Class Level Microflow of the Operation `Find` was Injected

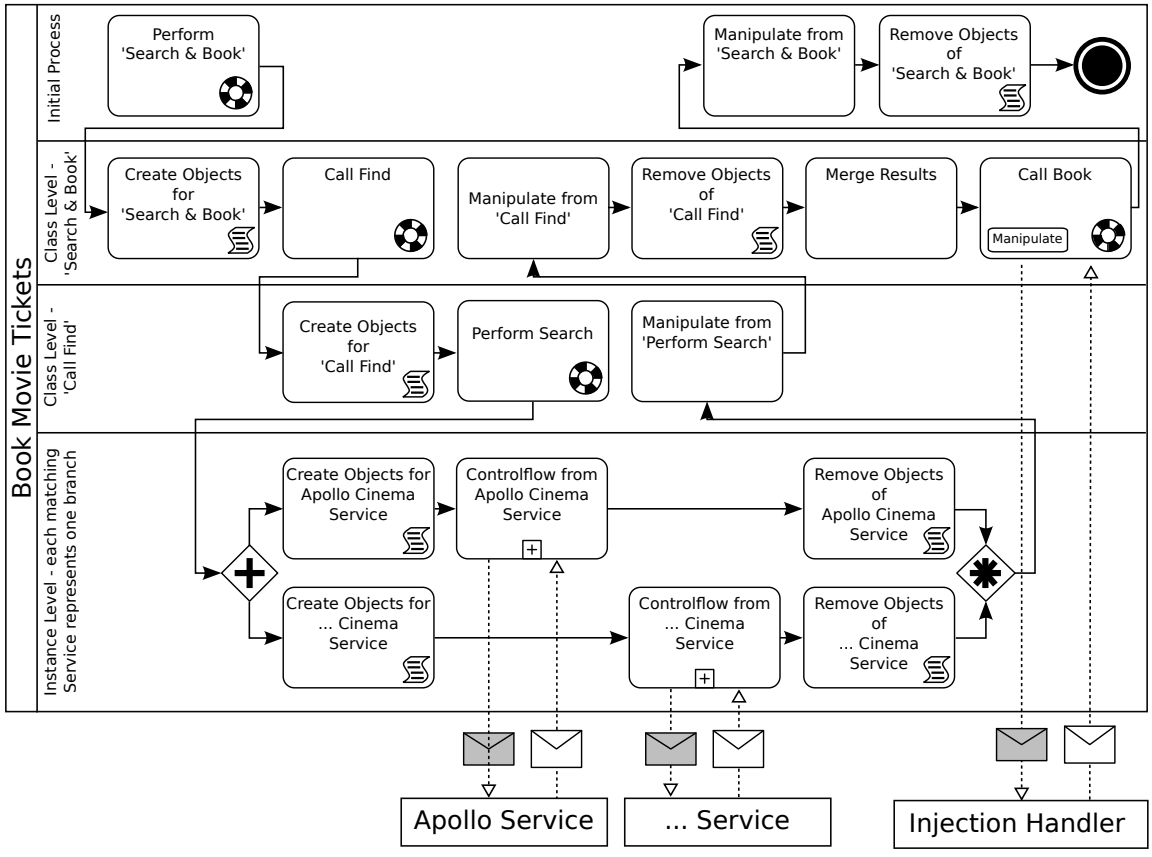

Figure 6.4: Step 4: Instance Level Microflow of Two Cinemas was Injected

- `position` represents the ID of the call statement causing the actual injection.
- `instance` represents the URI of the CPEE instance related to this injection.
- `handler` represents the URI of the Injection Handler responsible for this/these injection(s).
- `description` represents the XML representation of the whole controlflow in CPEE syntax.

If all this information is provided the Injection Service is able to perform the requested injection and to request some additional data, which may be needed during the injection, from the CPEE instance. If the injection was done without errors it responds the HTTP-status code 200 and a message consisting of two parameters:

`Positions:` This parameter provides all new positions (IDs of the controlflow statements) where the execution of the controlflow must continue. Because the CPEE supports parallel execution and therefore more than one branch could be affected by an injection, there may be more than one position included in the result. That's why we provide the new positions as a XML.

LISTING 6.1: Example of XML Positions Responded by an Injection Service

```
1 <positions>
2   <a1 new="a1_1">after</a1>
3   <a2 new="a2_1">at</a2>
4 </positions>
```

As shown in Listing 6.1 each position is provided as a separate element. The `name` of the element is the ID of the original position in the controlflow and the attribute *new* provides the new ID for it. The content of the element is either *after* or *at* and indicating if the execution should start **after** the referenced element or **at** the referenced element. It is explained in Section 6.3 why and how IDs of controlflow statements are changed and how it is decided if the execution should be continued *at* the new ID or *after* it.

`Description:` This parameter provides the actual controlflow description in CPEE syntax. This description represents the result of the injection algorithm after the delegated injection was performed.

If the injection was not performed as expected it will return an empty message and the HTTP-status code 501 to indicate to the Injection Handler that something unexpected happened and the injection was not performed at all or incomplete.

6.3 Injection Algorithm

As we already described in Section 4.2 and 4.3 our Repository provides information at two different levels. Each level represents different aspects of a service and therefore have to be computed accurate within the algorithm of the Injection Service. Injecting microflows into the controlflow of a CPEE instance is our approach to allow process designer to use services provided by the Repository in a transparent and arbitrary way. We argue that injecting the according microflows directly into the controlflow during run-time creates the benefit of providing all information about its execution within one instance. This further

allows for simpler algorithm when external operations are performed on business processes e.g. synchronization or repair because no other instances must be considered. It further enables run-time optimization of the controlflow as described by Schikuta et al. [66] using the A* algorithm.

Whenever an injection is performed it is important ...

1. ... not to interfere with the original intend of the process.
2. ... that changes are only performed in parts of the process that are not already executed. A more comprehensive definition is included in Rinderle et al. [67]. Within *loop* statements (for details see Section 6.3.2) these criteria is discussed in detail.
3. ... that all information about the injection is added into the controlflow to allow analyzing it afterwards.
4. ... to guarantee that each ID (controlflow statement, context variable, endpoint) is unique and all references to them are kept in sync.
5. ... that all matching services are executed in parallel. We argue that a process designer, not defining one particular service, wants each matching service to be executed in parallel. If not all services need to be finished the designer can use the *count* element when defining the call. For details about this attribute see Section 5.3.4 and Listing 5.6.

6.3.1 Implementation

In the following we explain the injection algorithm. We explain how we meet the so far elaborated requirements by (1) giving a listing in pseudo-code showing how it is implemented, (2) a detailed explanation of the major operations and elements of the algorithm and (3) an XML example of the controlflow description before and after an injection was performed. The message flow during an injection is presented in Figure 3.4.

Pseudo-Code Listings

LISTING 6.2: Main Method of the Injection Algorithm in Pseudo-Code

```
1 DEFINE injection($position, $instanceURI, $handler, $description, $positions)
2   GET endpoint from $instanceURI for $position into $endpoint
3   $group = CREATE group element and set attributes
4
5   IF injection is on Class Level
6     $microflow = GET class level microflow from $endpoint
7     SET according URIs within $microflow
8     IF NOT $position within a loop statement
9       $createNode, $removeNode = inject_class_level($microflow, $position,
            $group)
10      SET $position to 'after' within $positions
11    END
12  ELSE
13    $parallel = CREATE parallel statement
14    IF NOT $position within a loop statement
15      instance_level_injection($parallel, $position, $endpoint)
```

```
16      SET $position to 'after' within $positions
17    END
18  END
19
20  # Special treatment of injections performed within a loop (Section 6.3.2)
21  #=============================================================
22  IF $position within a loop statement
23    $loopCopy = Copy elements within loop statement
24    UPDATE $position and check if other $positions are affected
25    UPDATE IDs in $loopCopy
26    IF $position still within a loop statement
27      ADD $loopCopy before loop element in $description
28      RETURN injection($position, $instanceURI, $handler, $description,
            $positions)
29    END
30
31    $create, $remove = class_level_injection($microflow, $position, $group)
          IF injection is on Class Level
32    instance_level_injection($parallel, $endpoint, $position, $description)
          IF NOT injection is on Class Level
33    ADD $loopCopy before loop element in $description
34  END
35  #=============================================================
36
37  IF $position has manipulate block
38    $manipulateBlock = SPLIT manipulate block from $position
39    ADD $manipulateBlock and set attributes after $group
40  END
41
42  IF injection is on Class Level
43    ADD $create at first position into $group
44    ADD $remove after $manipulateBlock or $group
45  END
46
47  $description = TRANSFORM($description)
48  RETURN $positions, $description
49 END
```

In Listing 6.2 we show the main method of the injection algorithm. Lines 5 -11 cover most of the behavior when a class level injection - outside any *loop* statement - is performed. Lines 12 - 18 do the same only for an instance level injection outside of any loop. As both cases are implemented in their own method (see Listing 6.3 and 6.4) they can be handled very short here. Lines 22 - 36 cover the specifics of injections performed within a *loop* statement. We discuss the special treatment of such an injection in detail in Section 6.3.2. In short, the desired behavior is to enroll (see line 28) all *loop* statements if they include an injection. We implemented this by using a recursive call of the main method after (a) the actual loop was enrolled and (b) there is still (at least) one more outer loop present. Lines 38 - 41 perform the splitting of a *Call - Manipulate* statement into two separate nodes. To do so it adds the manipulate block into the controlflow description after the injected microflow. Lines 43 - 46 are performed if a class level injection happens and inject the *Create Objects* and the *Remove Objects* derived from the class level microflow at their proper positions. At the end (line 48) the newly created controlflow description is transformed into the syntax of the targeted execution system. After the transformation is done the updated execution positions and the new (transformed) controlflow description are returned to the Injection Handler (line 48).

LISTING 6.3: Class Level Method of the Injection Algorithm in Pseudo-Code

```
1 DEFINE class_level_injection($microflow, $position, $group)
2   UPDATE attributes e.g. IDs, references in $microflow
3   INJECT $microflow after $position into $description
4   RESOLVE message-parameters
5   $create = CREATE 'Create Objects' statement
6   $remove = CREATE 'Remove Objects' statement
7   RETURN $create, $remove
8 END
```

Listing 6.3 illustrates how a class level microflow is prepared for an injection and actually injected. At the beginning the nodes *Create Object* and *Remove Objected* are created. These two statements are automatically generated on the basis of the class level microflow by searching for the definition of context elements. When the algorithm resolves message-parameters and `outside endpoints` it replaces these references with the names of the actual context elements derived from the *call* statements of the actual controlflow description. Because at this point the algorithm can not identify the actual position of the *Create Objects* and *Remove Objects* statements, it returns them back to the main method which injects them later when their proper position can be defined.

LISTING 6.4: Instance Level Method of the Injection Algorithm in Pseudo-Code

```
1 DEFINE instance_level_injection($parallel, $endpoint, $position, $desc)
2   $services = GET all services from $endpoint
3   FILTER OUT $services of services not matching the constraints
4   $parallel = CREATE parallel statement
5   EACH $service IN $services
6     $create = CREATE 'Create Objects' statement
7     $remove = CREATE 'Remove Objects' statement
8     $branch = CREATE branch element
9     ADD $create first to $branch
10    ADD microflow of $service second to $branch
11    ADD $remove last to $branch
12    INJECT $branch into $parallel
13  END
14  ADD $branch into $desc
15 END
```

Listing 6.4 represents the algorithm used for instance level injections. It starts by requesting all services from the Repository offered at the provided URI ($endpoint). How this is done and what messages are exchanged is described in Section 4.4. In the next step, all services not matching the defined constraints are filtered out. As we already explained, each instance level microflow is injected into its own branch of a *Parallel - Branch* statement. The *Create Objects* and *Remove Objects* statements are derived similar as they are at class level. But because at instance level it is defined that they are at the beginning/end of the injected microflow they are already added at this point.

Operations and Elements

Split 'Call - Manipulate' Statements: The CPEE support so called *Call - Manipulate* Statements. These statements are, as the name already indicates, a *call* statement with an embedded manipulate block. These are necessary in the CPEE controlflow description to compute the result object of a *call* statement as they are only accessible within these manipulate blocks (see Stuermer [45] for details). Because in our system each call referring

to a service stores the explicit defined output parameters in an intermediated context variable (see Section 5.3.1 for details) we do not have this restriction. But as we inject into controlflows represented in CPEE syntax we must consider them in our algorithm. To inject the referenced microflow of these statements we need to split them. First is the original *call* statement without the manipulate block. Second is the manipulate block using the *context* attribute to access the result object of the injected microflow and the *properties* attribute to access the properties of the used services. Both values are defined in the `group` element described below. To identify these *manipulate* statements they are additionally marked with an *output* attribute providing the value *result*. An example for such a *manipulate* statement is given in Listing 6.6 at line 18. Splitting *Call - Manipulate* Statements this way allows our Injection Service (a) to inject microflows between the call and the newly created *manipulate* statement (which is its intended position as it logically represents a service call) and (b) access the result of the microflow as it was a native service call.

Group Element: This element is generated each time an injection changes something in the process. It is added directly **after** the *call* statement (if it is not within a loop - details see Section 6.3.2) causing the injection (see Section 6.2 input parameter `position`). The `group` element holds a number of attributes to store information about its actual injection. See Listing 6.6 line 11 for an example of a class level injection and Listing 6.7 line 3 for an instance level example.

Type: This attribute indicates during what situation it was created. If this represents the value *injection* it was created because of an injection outside a loop. If it represents *loop* as value its creation was caused by an injection inside a loop. See Section 6.3.2 for details to this case.

Cycle: This attribute is only set in combination with the attribute *type = 'loop'*. It indicates the iteration number of the according *loop* statement it represents[23].

Serviceoperation: Represents the name of the injected service operation within this group element. This string must have leading and ending quotes.

Source: Provides the ID of the *call* statement causing this injection.

Result: Provides the name of the context variable representing the result object.

Properties: Provides the name of the context variable representing the properties of each selected service.

Constraints: These elements represent an exact copy of the constraints defined within the *call* statement causing the injection. We copy them first after the group element to have all information for this injection within one element. This way the injection algorithm only has to parse its ancestors for considerable information and not the whole controlflow during follow-up injections. Compare Listings 6.5 and 6.6 for an example.

Data Elements: In Section 5.2 we introduced the possibilities provided to define **local** variables and endpoints within a microflow. During the injection it must be guaranteed that these names are unique within the whole controlflow. We do so by prefixing all names with the

[23]We use index-origin zero for counting the loop iterations.

ID of the causing *call* statement. Renaming these elements implies further that all references to them must be updated too to stay in sync. Also references to message-parameters, which were unknown until now, must be updated to refer to the proper context elements. Same is for endpoint references which are defined using the attribute *type = 'outside'*.

'Parallel - Branch' Statement: As we argued above, we assume that each service in the Repository, fulfilling the defined constraints, should be used in parallel. This is the reason why we create a `parallel` element whenever an instance level injection is going to be performed (see Listing 6.7 line 5). Each matching service is injected in its own `parallel_branch` element.

Injecting the Microflow: At this point the controlflow description is extended with the referenced microflow(s). It has to be considered that all IDs must be unique. That's why we prefix each ID with the ID of the causing *call* statement.

'Create Objects' Activity: To support context elements defined within the to be injected microflow(s) our algorithm creates a *manipulate* statement first after the `group` element. This statement covers all instructions necessary to create the defined context variables and endpoints for the injected microflow. Doing so allows to provide each microflow its own local scope of context elements. To make it easier to identify automatically generated statements they are marked with the attribute *generated = 'true'*. See Listing 6.6 line 15 for an example. If the injection is on instance level the properties of each service are added to the properties object. The properties object is referenced by the *properties* attribute of the according element (see 6.7 line 11) together with the result object referenced by the *context* attribute. Our system creates a Hash object where each property of the included services is stored. They are structured as defined in the properties schema (see Section 4.2.4 for details) using nested Hashes. Doing so allows the designer to know exactly what properties each involved service had at the time of execution and use them for further processing.

'Remove Objects' Activity: This is the clean-up of the microflows execution context. Each context element created by the *Create Objects* statement will be removed from the execution context because they are, by definition, not longer accessible. As we explained above, **local** context elements are only known and accessible by the microflow who defines it and all information intended to be used by subsequent controlflow statements outside the microflow is stored within the result object (output-message). To make it easier to identify automatically generated statements they are marked with the attribute *generated = 'true'*. See Listing 6.6 line 19 for an example.

Depending on the causing *call* statement (with or without a manipulate block) it is the first or the second statement after the `group` element is closed. If the injection was caused by a *Call - Manipulate* Statement the clean-up can only be performed after the according *manipulate* statement was executed. As this *manipulate* statement is consistently not included in the `group` element the *clean-up* statement is also placed outside the group element. That's why the relative position of the *clean-up* statement depends on the causing *call* statement.

Transformation Into Target Systems Syntax: As we already mentioned does the Injection Service transform the resulted controlflow description into the syntax of targeted workflow execution engine. Doing so allows that one repository can be used by different workflow execution engines. Each engine only needs to provide a XSLT stylesheet that can be used by the Injection Service to perform the transformation. By using XSLT stylesheets we provide a very flexible and easy-to-use way to integrate workflow execution engines supporting different syntax styles into our architecture. This fits well into the heterogeneity demands of Cloud based systems. Our prototype implementation provides one of these XSLT stylesheets to transform microflows provided by the Repository into CPEE syntax.

XML Representation of the Controlflow

Listing 6.5 represents a controlflow description causing an injection by referring to the class level operation named `search_and_book` (line 11) of the application domain *Cinemas*. See also Figure 4.2 for a BPMN diagram of the controlflow descriptions designed for this operation. All three XML listings are presented in CPEE syntax.

LISTING 6.5: Controlflow Description Causing an Injection

```
1 <call id="a01" endpoint="services">
2   <constraints>
3       <constraint xpath="adress/city" comparator="==" variable="city"/>
4   </constraints>
5   <parameters>
6     <service>
7       <serviceoperation>"search_and_book"</serviceoperation>
8       <injection_handler>endpoints.injection_handler</injection_handler>
9       <wait>0</wait>
10    </service>
11    <title>...</title>
12    ...
13  </parameters>
14  <manipulate output="result">...</manipulate>
15 </call>
```

Listing 6.6 represents the controlflow after the injection of the operation `search_and_book` was performed. The *Create Object* (line 15) and *Remove Object* (line 19) are placed at their proper positions and the split of the *Call - Manipulate* Statement in Listing 6.5 resulted in the *call* statement in line 1 and the *manipulate* statement in line 18. The group element in line 11 defines the result and properties object used by all further injected microflows within it. It further includes a copy of the the constraint elements (line 12 -14) defined by the initial call (line 2 -4).

LISTING 6.6: Controlflow Description After a Class Level Injection was Performed

```
1 <call id="a01" endpoint="services">
2   <constraints>
3     <constraint xpath="adress/city" comparator="==" variable="city"/>
4   </constraints>
5   <parameters>...</parameters>
6 </call>
7
8 <!-- After the injection, execution continues here -->
9 <!-- Group element caused by call "a01" -->
10
```

```
11 <group type="injection" serviceoperation="'search_and_book'" source="a01"
       result="context.result_a01" properties="context.result_a01['properties']">
12   <constraints>
13     <constraint xpath="adress/city" comparator="==" variable="city"/>
14   </constraints>
15   <manipulate generated="true" id="create_objects_for_a01">...</manipulate>
16   <!-- Controlflow of the class level operation 'search_and_book' -->
17 </group>
18 <manipulate output="result" id="manipulate_from_a01" context="context.
       result_a01" properties="context.result_a01['properties']">...</manipulate>
19 <manipulate generated="true" id="remove_objects_of_a01">...</manipulate>
```

Listing 6.7 shows how the controlflow looks like after the injection of the instance level operation `search`, caused by the class level operation `perform_search` of the *Cinemas* domain, happened. Line 5 shows the parallel element including the branch elements for each matching service. Line 8 and line 10 represent the *Create Object* and *Remove Object* statements for instance level microflows of the Apollo service. The manipulate block of the initial call (line 1) is placed in line 15 and the *clean-up* statement at the very end of the controlflow in line 16.

LISTING 6.7: Controlflow Description After an Instance Level Injection was Performed

```
1 <call id="a01__call_find__perform_search" endpoint="services" oid="
      perform_search">...</call>
2 <!-- Group element caused by call "a01__call_find__perform_search" -->
3 <group type="injection" serviceoperation="'search'" source="
      a01__call_find__perform_search" properties="context.result_a01['properties
      ']['call_find']['perform_search']">
4   <!-- All matching services from the repository are executed in parallel -->
5   <parallel generated="true">
6     <!-- Branch for Apollo Cinema service -->
7     <parallel_branch generated="true">
8       <manipulate generated="true" id="create_objects_for_a01__[...]_Apollo
            ">...</manipulate>
9       <!-- Controlflow of the instance level operation 'search' from the
            Apollo Cinema service -->
10      <manipulate generated="true" id="remove_objects_of_a01__[...]_Apollo
            ">...</manipulate>
11    </parallel_branch>
12    <!-- Branches for all other matching services -->
13  </parallel>
14 </group>
15 <manipulate output="result" id="
      manipulate_from_a01__call_find__perform_search" context="context.
      result_a01__call_find" properties="context.result_a01['properties']['
      call_find']">...</manipulate>
16 <manipulate generated="true" id="remove_objects_of_a01__call_find__perform
      ">...</manipulate>
```

6.3.2 Exception: Loop

As we already explained in Section 5.7 the controlflow description supports *loop* statements. In contrast to all other statements, where the injection is done by changing the original controlflow only in the area within the *call* statement, injection calls within *loop* statements affect a wider range of the controlflow.

For example: Within a *loop* statement an injection is performed and the controlflow is changed as described above, meaning that the *call* statement is directly resolved using the information provided by the Repository. This results in a situation where successive iterations of this loop will not execute the original controlflow what interferes with its original intend.

As these example illustrates, a special strategy for controlflow changes/injections within *loop* statements is needed. The aim of this strategy is that **each iteration of a loop can be changed for its own** without interfering with its original controlflow. If this is guaranteed all injections can be handled like there is no loop at all.

Rinderle et al. [68] states that their approach to handle loops in adaptive WfMS is "[...] to logically treat loop structures as being equivalent to respective linear sequences." Based on this idea we came up with using **loop unrolling** techniques for controlflows. Loop unrolling origins in compiler building as part of the so called **Loop transformations** applied by compilers on program code. Bacon et al. [69] describes in section 6.1.3 that "Unrolling replicates the body of a loop some number of times called the unrolling factor (u) and iterates step u instead of step 1." This means that compilers are allowed to copy the body of the loop one after another (unrolling) as many times as the loop would have been iterated (compilers must of course be able to predict this value to do so). While compilers do this at compile-time to speed up the execution time of the resulting program because sequences are faster to execute than loops, we do this at run-time to have separated controlflow parts representing the loop body of each iteration. Into this loop bodies we are allowed to inject as described above because we are no longer interfering with the next iteration of the loop.

As mentioned before our algorithm respects the correctness criteria educed by Rinderle et al. [67]. One of these criteria, namely the **Comprehensive compliance criterion (7)**, states that changes within loops are allowed, as long as they are traceable and do not lead to a situation conflicting with other criteria or ends up executing controlflow that has been executed before (within the same iteration). The algorithm described here fulfills all of this criteria.

To achieve this behavior our run-time strategy for loop unrolling looks like this:

1. Change *Do-While-Loop* statements into *While-Loop* statements by transforming its condition and changing the attribute from *post_test* to *pre_test*.

2. Copy the controlflow inside the *loop* statement directly before the loop statement. This way the copy of the actual iteration is always positioned directly before the *loop* statement.

3. Update the execution position to the *call* statement in the replicated loop body before the loop.

4. If the actual position is still within a *loop* statement (nested loop construction) start again at (1), otherwise continue with (5).

5. Perform injection.

A pseudo-code description of this strategy is given in Listing 6.2 lines 23 - 36. Line 29 represent the fourth step (start again) using a recursion to resolve nested loop constructions.

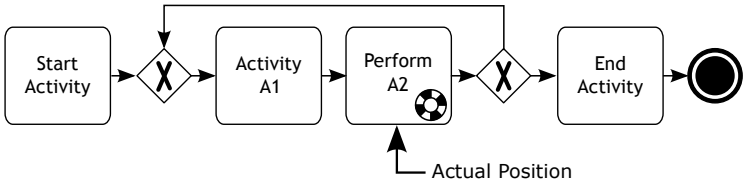

Figure 6.5: Loop Before Injection

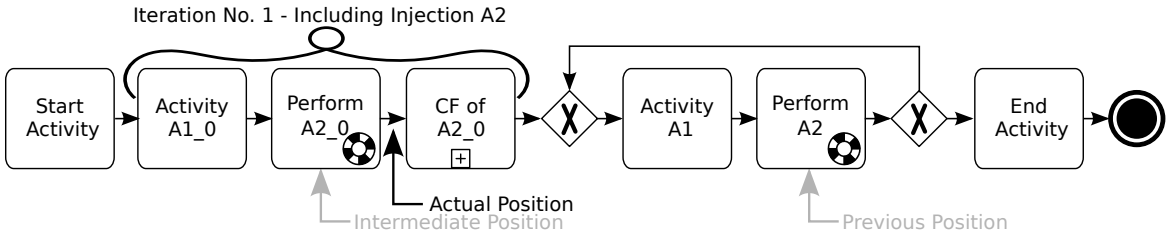

Figure 6.6: Loop After Injection

Example of an Injection Within a Loop Statement

Figure 6.5 shows a controlflow including a loop. Within this loop one activity (Activity A1) and one injection (Perform A2) are defined.

When the activity *Perform A2* is activated it starts the injection procedure, which according to our algorithm does the following:

1. Create the group element (see Listing 6.8 for details) first before the loop and insert A1 and A2 into it. Change the IDs of the inserted activities using the iteration number of the loop (represented by the *cycle* attribute) as postfix.
2. Set the actual execution position to *at A2_0* (Intermediate Position).
3. No more *loop* statements are found at the **Intermediate Position** and the injection of A2_0 can be performed (see Listing 6.2 line 29 for details), resulting in the injection of Activity *CF of A2_0* and the change of the execution position to *after A2_0* (Actual Position).

Although we change the **Actual Position** (after A2_0) in Figure 6.6 technically **before** the **Previous Position** (at A2) we argue that this is still in line with the correctness criteria defined by Rinderle et al. because no controlflow that has already been executed before is executed again. If we enter the loop again (depending on the defined condition) it will be an other iteration of it and therefore has not been executed before. Also did we not need to take care about any context elements because no elements only accessible (scoping) within the loop are allowed (in contrast to many programming languages). Also is it intended that changes within context elements of prior iterations influence successive iterations.

As already mentioned above unrolling a loop also uses a `group` element to mark the according changes in the controlflow. The group element includes the attributes *type = 'loop'* and *cycle* representing the number of the according iteration starting at zero. In XML it looks something like this:

LISTING 6.8: CPEE Syntax of the Controlflow Shown in Figure 6.6

```
1<call id="Start Activity" ...>...</call>
2<group type="loop" cycle="0" source="Perform A2">
3  <call id="Activity A1_0" ...>...</call>
4  <call id="Perform A2_0" ...>...</call>
5  <!-- Controlflow represented by 'CF of A2_0' -->
6</group>
7<loop pre_test="...">
8  <call id="Activity A1" ..>...</call>
9  <call id="Perform A2" ...>...</call>
10</loop>
11<call id="End Activity" ...>...</call>
```

Chapter 7

Injection Handler

The Injection Handler is used to distribute injections defined in business processes executed by the CPEE. As shown in Figure 3.2 and Figure 3.4 this component links CPEE instances with Injection Services. We start this chapter with a discussion about the provided interface and the supported messages. We additionally give a short explanation of the CPEE's notification model as it is strongly related to the implementation of this component. Afterwards we discuss the functionality this component has to provide to fit into our overall system architecture. Because this component is strongly related to the CPEE's Activity Handler we discuss how we adapted it to cooperate with the Injection Handler at the end of this Section.

7.1 Interface

Like all the other components involved in our system the Injection Handler provides a RESTful interface too. In the following section we will explain what messages are exchanged when an injection is requested by a business process. Because this component is intended to link CPEE instances to Injection Services it has to provide two interfaces: (1) The interface used to interact with the CPEE including the Activity Handler component of the CPEE (for details about the Activity Handler see Chapter 7.3). (2) The interface used to interact with Injection Services which is described in detail in Section 6.2.

The following messages are exchanged between the CPEE - Activity Handler and the Injection Handler when an injection is requested (see also Figure 3.2 for an illustration of this protocol).

Registration Request/Response: Whenever the CPEE executes an injection call the Activity Handler sends this message to the Injection Handler. Each call provides the URI of the Injection Handler it wants use. The URI is provided by the `injection_handler` parameter. (see Listing 5.6 for details).

HTTP-Method: POST

Parameters:

1. `activity:` provides the ID of the call statement causing the injection
2. `instance:` provides the URI of the CPEE instance (`http://cpee:uri`) the activity is part of.

Response Parameter: none

HTTP-Status: 200 (OK), 501 (Internal Error)

Subscribe 'Syncing After': As we mentioned in Section 3.2.1 the CPEE provides functionality to subscribe for certain events of an instance. This message is used to subscribe the Injection Handler for `Syncing After` events of an instance by sending it to `http://cpee:uri/notifications/subscriptions`. If a service is registered for such an event it will be notified by an HTTP-message each time these events occurs.

HTTP-Method: POST

Parameters[24]**:**

1. `uri:` provides the URI where the notification should be sent to. Providing this parameter explicit allows services to subscribe other services then them self for notifications about certain events. Second is the parameter called
2. `topic:` indicates a group of events provided by the CPEE. Because we want to subscribe for the `Syncing After` event its value is *running*.
3. `votes:` indicates what particular event, out of the group indicated by the parameter named topic, is requested. For this subscription its value is *syncing_after*.

Response Parameter:

1. `key:` identifies the new subscription and must be provided again when the Injection Handler wants to delete it.

HTTP-Status: 200 (OK), 501 (Internal Error)

'Syncing After' Notification: As the names suggest, this message is sent to all subscribed services by the CPEE if a `Syncing After` event occurred. The `Syncing After` event indicates that the execution of an activity has finished and the CPEE awaits a reply if it is allowed to continue with the execution (replying *true*) or not (replying *false*). How this technique is used by the Injection Handler is explained in the next Section.

HTTP-Method: POST

Parameters:

1. `vote:` represents the event, in our case *syncing_after*.
2. `topic:` represents the topic of the event. For *syncing_after* it is *running*.
3. `notification:` provides detailed information about the event like e.g. the ID of the activity causing the notification or the URI of the according CPEE instance.

[24] According to the CPEE security mechanism there are some more parameters. As this security system is beyond the focus of this thesis they are only mentioned to give a comprehensive description of the message.

Response Parameter:

1. `continue:` indicates if the execution should continue (value is *true*) or if it should stop the execution (value is *false*).

HTTP-Status: 200 (OK), 501 (Internal Error)

Delete 'Syncing After' Subscription: This message requests the deletion of the referenced subscription from the CPEE. After this message was computed no more event notifications, belonging to this particular subscription, will be sent. This message is sent to `http://cpee:uri/notifications/subscriptions/{key}` where {key} represents the key of the particular subscription received during its creation.

HTTP-Method: DELETE

Parameters[25]**:** none

Response Parameter: none

HTTP-Status: 200 (OK), 501 (Internal Error)

Subscribe 'Stop' Notification: Similar to the way services can subscribe for `Syncing After` notifications they can also subscribe for `Stop` events. This event informs subscribers that the according instance is now in the state *stopped*[26].

HTTP-Method: POST

Parameters[27]**:**

1. `url:` provides the URI of the subscriber. Again this must not be the same service initiating the subscription. Second parameter is named
2. `topic:` indicates the group of events the subscription belongs to. In this situation its value is *properties/state*.
3. `events:` indicates the particular event for this subscription. In this situation is the value *change*.

Response Parameter:

1. `key:` identifies the new subscription and must be provided again when the Injection Handler wants to delete it.

HTTP-Status: 200 (OK), 501 (Internal Error)

'Stop' Notification: This message informs the subscriber that the according instance has properly stopped and therefore changes from the outside can be applied. Therefore the Injection Handler can start with delegating pending injections to the Injection Services and applying the resulting controlflow changes on the instance.

[25] According to the CPEE security mechanism there are some more parameters. As this security system is beyond the focus of the thesis they are only mentioned to give a comprehensive description of the message.

[26] Values of an instance (e.g. controlflow description, context elements, ...) can only be changed from outside when it is in the state *stopped*.

[27] According to the CPEE security mechanism there are some more parameters. As this security system is beyond the focus of the thesis they are only mentioned to give a comprehensive description of the message.

HTTP-Method: POST

Parameters:

1. `event:` indicates the event related to this notification. In this situation its value is *change*.
2. `topic:` provides the topic associated to the event which in our case is *properties/state*.
3. `notification:` provides information about the instance causing the notification like e.g. the URI of the instance or the new state of it (in this situation it must provide the value *stopped*).

Response Parameter: none

HTTP-Status: 200 (OK), 501 (Internal Error)

Delete 'Stop' Subscription: This message looks the same like the *Delete 'Syncing After' Subscription message*. Again the `key` used in the URI must belong to the proper subscription.

The messages the Injection Handler exchanges with the CPEE after the injection was performed successfully are (see Figure 3.5 for illustration of this protocol):

Set Controlflow: This message is used to set a new/changed controlflow description for the CPEE instance. We use it to update the controlflow description with the one where the microflows of the registered injections are included.

HTTP-Method: POST

Parameters:

1. `description:` providing the new controlflow. Its XML description (CPEE syntax) must be inside a `description` element. The target URI of the message is the URI of the CPEE instance (`http://cpee:uri`) followed by `/properties/values/description`.

Response Parameter: none

HTTP-Status: 200 (OK), 501 (Internal Error)

Set Positions: This message is used to set/update the execution positions where the CPEE will continue after the restart of the instance. The XML structure consists of a root element named `positions` where each child element represents the `ID` of the statement where the position is set to and the value of the element represent either *at* if the execution should continue with the referenced statement or *after* if the execution should continue after it. The target URI of the message is the URI of the CPEE instance (`http://cpee:uri`) followed by `properties/values/positions`.

LISTING 7.1: Positions in the WEE/CPEE XML Structure

```
1 <positions>
2   <a1>after</a1>
3   <a2_0>at</a2_0>
4 </positions>
```

HTTP-Method: POST

Parameters:

1. `content:` represents the actual execution positions in XML format.

Response Parameter: none

HTTP-Status: 200 (OK), 501 (Internal Error)

Restart Execution: This message is used to restart the referenced CPEE instance. Its target URI is the URI of the CPEE instance (`http://cpee:uri`) followed by `properties/values/state`.

HTTP-Method: POST

Parameters:

1. `value:` indicating the state it should go to. In this case the value of the parameter must be *running* to request the instance to change its state to running.

Response Parameter: none

HTTP-Status: 200 (OK), 501 (Internal Error)

All communication the Injection Handler has with the CPEE instances is covered by these messages. In the next section we will show how and when the messages are exchanged what the desired behavior is.

7.2 Algorithm

In this section we explain how the Injection Handler actually works. We already described the interface in the section above and explain now how they are intended to use. As we explained in Section 3.2.3 the Injection Handler interacts with the Activity Handler of the CPEE and the Injection Services. As Figures 3.2 - 3.5 illustrate, the algorithm of the Injection Handler consists of the following major parts:

1. Receiving an injection request.
2. Receiving a `Syncing After` notification.
3. Receiving a `Stopped` notification.
4. Delegating the injections.
5. Restarting the CPEE instance.

In the following Listings (Listing 7.2 - 7.4) we give a more technical description of the implementation of our algorithm.

LISTING 7.2: Injection Handler: Receiving Injection Request (Pseudo-Code)

```
1 # Receiving an injection request from $instance for $injectionCall
2
3 IF $instance NOT in $pendingInjections
4   CREATE $mutex FOR $instance
5   USE $mutex
6     $key = SUBSCRIBE for 'Syncing_After' at $instance
7     ADD $instance, $activity, $mutex, $key, $state='start' TO
          $pendingInjections
8   END
9 END
10
11 RETURN injection response
```

Listing 7.2 creates a Mutex in line 4. A Mutex is an implementation of a semaphore and is used to handle exclusive access to shared resources. This access control is needed because the CPEE allows only to subscribe for events at the level of instances and therefore the Injection Handler may receive also notifications about activities not related to an injection. Further is the Injection Handler implemented as a web service and therefore designed to handle multiple requests at a time. To protect the Injection Handler from multiple requests from the same instance we use a Mutex to 'lock' access to the injection queue (in the following Listings this variable is called $pendingInjections). Doing so allows the Injection Handler to compute one request of an instance at a time without changes in the injection queue and therefore guarantees a consistent state of it.

For example, when an instance processes multiple branches each branch throws an event when an activity is finished. If the injection queue would not be locked by a semaphore the Injection Handler would, if the timing is right, registers itself more than one time for `Syncing After` or `Stopped` resulting in inconsistent behavior of the whole system.

LISTING 7.3: Injection Handler: Receiving 'Syncing After' Notification (Pseudo-Code)

```
1 # Receiving a 'Syncing After' notification from $instance
2 $mutex = $pendingInjections[$instance][:mutex]
3
4 USE $mutex
5   IF in $pendingInjections[$instance] is $activity included
6     IF $pendingInjections[$instance][state] == 'start'
7       UN SUBSCRIBE 'Syncing After' using $pendingInjections[$instance][key]
8       $key = SUBSCRIBE 'Stopped' at $instance
9       $pendingInjections[$instance][key] = $key
10    END
11    RETURN 'continue=false' # Forces execution to stop as soon as possible
12  END
13  RETURN 'continue=true' # Allow execution to continue
14 END
```

In Listing 7.3 line 4 we use the Mutex created in Listing 7.2 for the according instance. Line 6 introduces the `state` property which is used to indicate in which state (from the perspective of the Injection Handler) the instance is. The following states are used in our implementation:

1. **Start:** Indicates that at least one injection was requested for this instance and that the subscription for `Syncing After` was already done successfully.
2. **Active:** Indicates that at least one activity, registered for an injection, has finished and the instance was already informed to stop as soon as possible and the subscription

for the `Stopped` event is already done.

3. **Finished:** Indicates that the registration phase for new injections for this instance is finished.

We use this different states to coordinate all subscriptions and registrations of events and activities for a CPEE instance. For that reason they are also included in the Mutex.

LISTING 7.4: Injection Handler: Receiving 'Stopped' Notification (Pseudo-Code)

```
1 # Receiving 'Stopped' notification from $instance
2
3 UN SUBSCRIBE 'Stopped' using $pendingInjections[$instance][key]
4 $pendingInjections[$instance][state] == 'finished'
5
6 $positions = GET positions from $instance
7 $description = GET controlflow description from $instance
8
9 FOR EACH $position IN $pendingInjections[$instance][activities] DO
10   $description, $newPositions DELEGATE INJECTION to Injection Service
11   UPDATE $positions IF $newPositions included in $positions
12 END
13
14 DELETE $instance FROM $pendingInjections
15
16 UPDATE description with $description at $instance
17 UPDATE positions with $positions at $instance
18 RESTART $instance
```

Listing 7.4 does not use the Mutex anymore because at this point in execution the instance is completely stopped and therefore incapable of sending any notifications. While the injection for each position is delegated it is important to know that the iterated list of positions is always updated with the results of the Injection Service. This covers the case if an injection changes the position of an other pending injection. By doing so it would not be forgotten and performed at the proper position (e.g. during *Loop unrolling* - see Section 6.3.2 for details).

7.3 Adapting the CPEE Activity Handler

As explained in Section 3.2 and discussed in details by G. Stuermer's [45] the WEE/CPEE implements a component called Handler Wrapper/Activity Handler to interact with services referenced by call statements. The original implementation focuses on using RESTful services which are already known at design-time and therefore restricts it in these two aspects. To use our Repository the way it is intended to be used these two restrictions had to be abrogated as we additionally support ...

1. ... SOAP calls.
2. ... dynamic service selection at run-time.

The next two sections give an explanation how we extended the Activity Handler component of the WEE/CPEE to meet our requests[28].

[28] There is one more extension we implemented into the Activity Handler which focuses on providing user interfaces within CPEE call statements. But as this extension is strongly related to our client implementation we decided to explain it there. See Section 8 for details about this extension

7.3.1 SOAP Calls

As described in Section 5.3.4 we can define calls to SOAP services using the *soap-operation* and *wsdl* attributes (see Listing 5.9 for an example). As it is the job of the Activity Handler to perform the actual service call it must be able to handle SOAP calls and all related parameter transformations. We did this by extending the implementation of the Activity Handler to behave differently if the *soap-operation* attribute is present in the parameter set of the call statement. In Listing 7.5 we show the implementation of the part responsible for SOAP calls in pseudo-code.

LISTING 7.5: Implementation of SOAP Call Extension in Pseudo-Code

```
1 DEFINE activity_handle($passthrough, $parameters)
2
3   ... # Here is the original implementation of the Activity Handler
4
5   IF $parameters include 'soap-operation'
6     GET wsdl from $parameters['wsdl'] into $wsdl
7     $envelope = CREATE SOAP envelope using $parameters['parameters']
8     $response = CALL $endpoint using $envelope          # Perform SOAP call
9     IF $response includes SOAP:FAULT                    # Call failed
10      RETURN message from $response['SOAP:FAULT']
11    ELSE                                                # Call successful
12      RETURN $response['SOAP:BODY']
13    END
14  END
15  ... # Here is the original implementation of the Activity Handler
16 END
```

At line 7 in Listing 7.5 a SOAP envelope, according to the SOAP specifications [35], is created. According to this specification each parameter is provided inside its own element where the name of the element represents the parameter name and the text of the element is the value of the parameter e.g. <numberOfSeats>5</numberOfSeats>. At line 8 the actual service call is performed. The HTTP-method POST is used for it as it is the standard method for SOAP services. In line 10 the error message received from the service is returned if an error happened during the execution.

Usually SOAP services return expressive error messages including a detailed description of the reason why the execution failed. In our implementation this messages is forwarded to the result object provided by the instance level microflow. Doing so allows the service provider to define proper reactions depending on the error and using the `status` parameter provided by it. Because SOAP services, unlike RESTful services, do not use the HTTP-status code it is necessary to set it inside the instance level microflow manually. If the call was successfully executed the body part of the response message is included in the result object.

As in our implementation, output parameters must be referenced explicit in the microflow, the assignment of the according body elements to the context variables can be generated automatically during the transformation of the controlflow description. Details about this parameter definition are given in Section 5.3.4. The actual assignment of their values to the context elements is done, like for any other call, in an automatically generated manipulate block.

7.3.2 Injection Calls

While we implement the functionality needed to handle injection calls, we first need to weaken the concept that the endpoint provided by the call statement is the endpoint of the requested service. As we already discussed in Chapter 7 is - in the case of an injection call - the Injection Handler requested instead of the Repository, as the endpoint would suggest.

We do so by using the parameter named `injection_handler` instead of the endpoint if the parameter set includes a parameter named `service`. After the registration for an injection has been performed successfully, the Activity Handler raises an `Wee::Signal::SkipManipulate` error to prevent the WEE/CPEE from - if present - executing the manipulate block. Because after each activity, subscribed resources are allowed to vote if the execution should continue, this guarantees that the split of the *Call - Manipulate* statements (see Section 6.3.1 for details) do not lead to a situation where controlflow code is executed twice. Further is the attribute *info* of the call statement (see Section 5.3.4 for details) resolved here because the required information e.g. call-endpoint, call-lay, ... is best accessible at this point.

Listing 7.6 shows the relevant parts of the implementation in pseudo-code.

LISTING 7.6: Implementation of Injection Call Extension in Pseudo-Code

```
1 DEFINE activity_handle($passthrough, $parameters)
2
3   ... # Here is the original implementation of the Activity Handler
4
5   IF $parameters['info'] == 'true'
6     ADD info abut the execution instance and its parameters to $parameters
7   END
8
9   IF $parameters include 'service'
10    REGISTER for an injection at the URI provided by $parameters['service']['
          injection_handler']
11    RAISE Wee::Signal::SkipManipulate error
12  END
13
14  ... # Here is the original implementation of the Activity Handler
15 END
```

Chapter 8

The Mobile Client - An Example Implementation

In this chapter we focus on the client implementation we developed to illustrate the concepts described so far. To gain high flexibility in terms of platforms we decided to implement a web application. Using HTML5 technologies (i.e. WebSockets [70], Local Storage [36], ...) combined with JavaScript allows to build rich web applications running in state-of-the-art web browsers. These techniques are also broadly supported on operating systems for mobile devices like e.g. Apple's iOS or Google's Android. We further used on the jQTouch Framework [71] to make the web application look like a native one e.g. page transitions and user interface design. Although our concept is based on web technologies we argue that it can still be seen as a native one because all implemented components can be executed directly on the device. There are two possible ways to install the software on a mobile device:

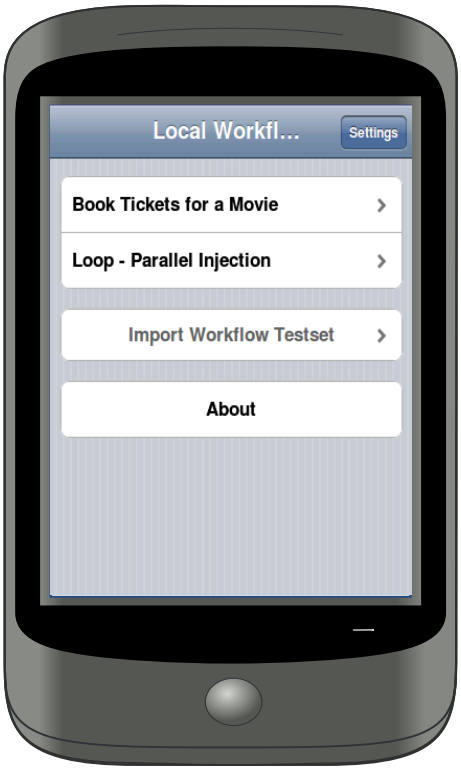

Figure 8.1: Screenshot: Client Application

First: On most mobile devices, installing system services and self-written software is only possible if it is rooted. We do not want to give any details about how to root a mobile device ☺, but afterwards it is possible to execute Ruby applications and therefore also the CPEE, the Repository and the Worklist. With the client stored locally, all needed components are accessible directly on the mobile device resulting in similar security issues and speed as native applications.

Second: To make it distributable over different markets (e.g. Google Apps Marketplace [15], Apple - App Store [13]) and installable like native applications, different frameworks, wrapping these kind of projects into a native application, supporting different platforms, are available e.g. PhoneGap [72].

We start this chapter with an introduction of the overall architecture of the client. After we introduced the general functionality of our client implementation we make a little extension to our Cinemas example to make it more usable in the real world. With the resulting business process we explain the usage of the client using screenshots of our implementation in action.

In order to execute the example we need a worklist to interact with the user which is described later in this chapter. We further explain the algorithm we use to identify activities interacting with the user and activities interacting with the Repository and how we store this information in the client application. The further usage of this information is implemented in the Wallet allowing the user to define personal preferences for each business process. These preferences are divided in user inputs (predefined values) and service endpoints by exploring the Repository.

A discussion about the functionality of the Worklist and its interface is given at the end of this chapter. We further give a detailed explanation about how the Worklist uses templates defined with the activities.

8.1 Architecture and Deployment

In Figure 8.2 we illustrate what components are installed locally on the mobile device by grouping them together. These are namely the already introduced CPEE, the Repository, the Injection Service and the Injection Handler. To derive the needed functionality for the client application we additionally implemented the Worklist component which is described in detail later in this Chapter and the client itself.

Outside this group are resources which may be used by the client during some operations. These are business processes that can be imported over the web into the local client (including templates referenced within them) and services registered in the Repository.

As Figure 8.2 illustrates the interaction of the components is as follows:

1. Import Process Description and Templates from the Web
2. Adapt Injection Requests by Browsing the Repository
3. Initiate an Execution
4. Monitor the Execution Instance
5. Execute the Business Process
6. Register Worklist Task
7. Request Associated Template
8. Request Worklist Task Data

In the following section we will discuss each of this points in detail and show how we implemented the desired functionality.

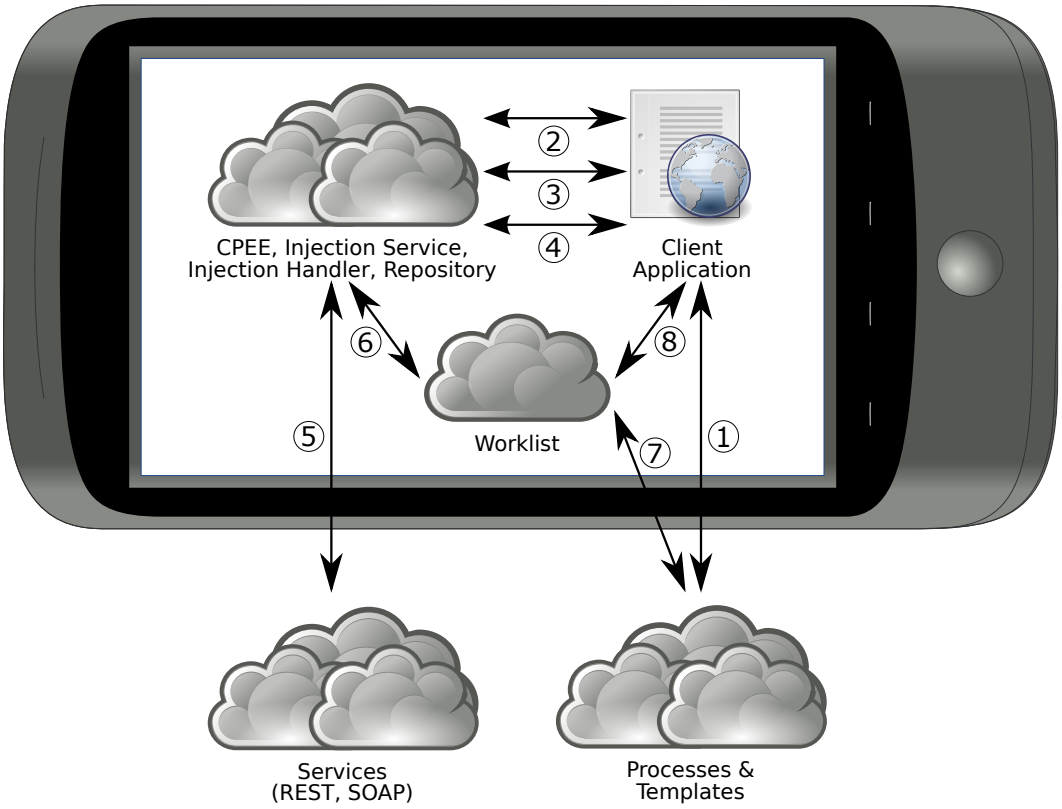

Figure 8.2: Architecture and Deployment of the Client Implementation

8.1.1 Import Process Description and Templates from the Web

To execute business processes locally on the client they need to be defined first. As we did not implement a process designer for mobile devices so far we need to import these descriptions from somewhere else. Therefore we support importing process descriptions in CPEE syntax using HTTP GET requests. The server hosting these definitions may be running locally on the mobile devices or somewhere in the web[29].

Whenever such a process description is imported the client parses it for (a) call statements referring to `templates` (see Listing 5.7 for an example) and (b) call statements referring to `services` (see Listing 5.6 for an example) from any Repository. All explored activities are added into the Wallet of the application. The Wallet is used to store the personal preferences of the user i.e. input data requested by templates and the services used from the Repository. For details about the Wallet see Section 8.3.

8.1.2 Adapt Injection Requests by Browsing the Repository

We describe this functionality later in this chapter (see Section 8.3) and will therefore not discuss it here.

[29] If the web server is not locally on the device it must support Cross-Origin Resource Sharing [73]. This technique allows client-side requests of resources from other domains then itself.

8.1.3 Initiate an Execution

When the user has defined all preferences in the Wallet a new execution instance of this process description is created. Our client implementation does this by using the RESTful API provided by the CPEE. A short description how a new instance can be created and initialized is given in Listing 8.1 in pseudo-code syntax.

LISTING 8.1: Create and Initialize a new WEE/CPEE Instance

```
1 $instance_uri = CREATE new instance using HTTP POST at the WEE/CPEE root URI
2 LOAD $process_description FROM localStorage
3
4 # Loading the Process Description with parameter
5 SET transformation property at $instance_uri/properties/values
6   Name: property        Value: transformation
7 using HTTP POST in $process_description
8
9 INITIALIZE $instance_uri/properties/values with parameters
10   Name: endpoints        Value: FIND 'endpoints'        in $process_description
11   Name: handlerwrapper   Value: FIND 'handlerwrapper'   in $process_description
12   Name: data-elements    Value: FIND 'data-elements'    in $process_description
13   Name: description      Value: FIND 'description'      in $process_description
14 using HTTP POST
```

If all these operations are executed without an error (HTTP-status code 200 is returned) the user is allowed to start the execution by sending *running* at `cpee:uri/properties/values/state` using the HTTP-method PUT.

8.1.4 Monitor the Execution Instance

After the execution is started by the user the client application starts to monitor the properties of the execution instance. The CPEE therefore allows other applications to subscribe them self for certain events. Whenever an event occurs the CPEE uses a WebSocket [70] connection, which was established during the subscription, to inform other applications about it. We already described this functionality during the implementation of our Injection Handler and in the excursus about the WEE/CPEE (see Section 3.2.1 for details). Our client application uses this functionality to get informed about ...

1. ... changes of the instance state e.g. ready, running, finished, ...
2. ... changes of the controlflow description. As our business processes include activities resulting in the injection of microflows from the Repository we need to parse the controlflow description every time it is changed to update the locally stored list of activities (requesting user interaction) with their *ID*'s. This is partly the same operation as performed when a process description is imported.
3. ... changes of the actual position of the process execution. We use this position to check if an interaction with the user is requested by the actual activity.

Because not all of today's mobile devices support WebSockets we implemented a fall-back where the client application polls the properties of the instance (state, position, description) regularly. This is not as resource efficient as WebSockets but it is better than a complete fail. We further strongly believe that this problem will be solved in the near future as to our best

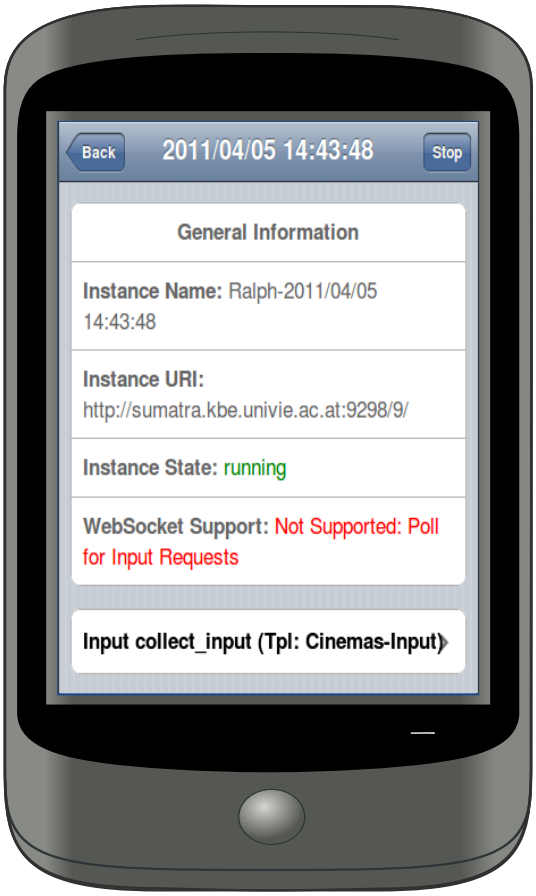

Figure 8.3: Screenshot: Monitoring the Execution Instance

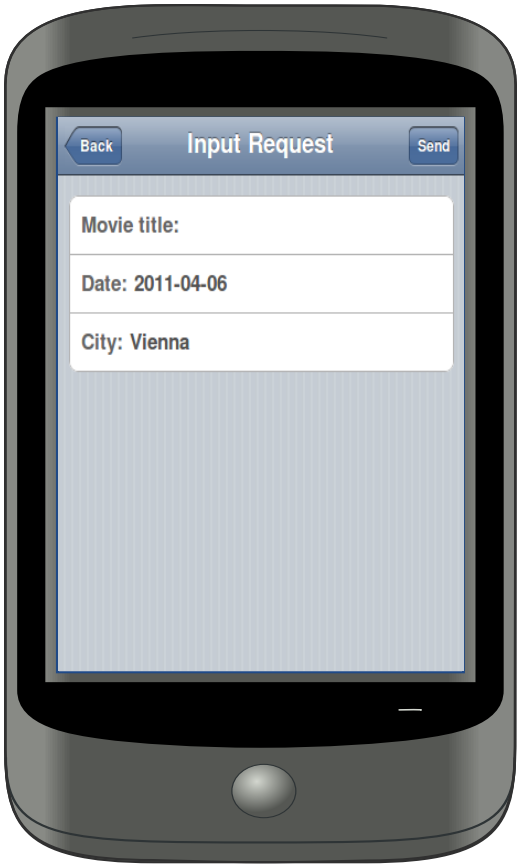

Figure 8.4: Screenshot: Example Worklist Task Data Request

knowledge all upcoming mobile devices will support WebSockets. To avoid wasting resources the user can define the polling interval individually. If, for example, the user knows that a long-running process is executed the polling interval can be raised to high numbers, or even better, the user leaves the monitoring screen of the process. If this screen is accessed again later the client reconnects to this instance and starts monitoring again. This way valuable resources like processing power and therefore battery life can be saved.

We already mentioned that we check if the actual activity requests user interaction every time a position change is recognized. If this is the case, the application offers the user to open the request from the worklist. We use the API provided by the Worklist, which is described in Section 8.4.1, to realize this functionality. Figure 8.3 shows a screenshot of the monitoring screen where the activity `collect_input` requests some user interaction.

8.1.5 Execute Business Process

Because of all injections happened during the process execution so far the CPEE is able to execute actual services. We discussed this already in detail in Section 7 and will therefore not do this here again.

8.1.6 Register Worklist Task

In Section 5.3.4 and 4.2.3 we explained what possibilities are offered to associate templates to external calls. Whenever such an activity is executed the CPEE posts the included information to the Worklist to register this activity at it. Our implementation of the Worklist is able to compute these provided information and create worklist entries out of it. How this is done is described in Section 8.4. The CPEE gets a callback response from the Worklist causing the CPEE to wait for the actual response (using the provided `callback-id` introduced in Section 4.2.3).

8.1.7 Request Associated Templates

The Worklist has to request the template definitions (XSLT stylesheets) associated with the registered task to provide its user interfaces. It does this at the time a client application requests this data. Details about the implementation of the Worklist are given in Section 8.4 and will therefore not be discussed here.

8.1.8 Request Worklist Task Data

When the client application gets informed about a `position change` event it checks local data if the actual execution position is registered as an activity requesting user interaction. If this is the case the client application offers the task to the user. In Figure 8.3 we show at the bottom of the screen how the user gets informed about such an activity.

The client application is now using the API of the Worklist (see Section 8.4.1 for details) to request the data (mainly the HTML representation) of the activity. Knowing the `worklist-uri`, the `activity-id`, the `template-id`, the preferred `language` and the `platform` enables the client application to request the right data from the worklist. The HTML representation of such a worklist entry is shown in Figure 8.4.

8.2 Real World Example: Book Tickets for a Movie

To make the example of booking a movie, we used so far, more like a real world case we added two activities requesting user interaction. First is the activity `collect_input` which requests from the user to enter a `Title`, a `Date` and a `City`. With this information we call the operation `search_and_book` from the Cinema domain inside our Repository by the activity `a01`. While the values of Title and Date are provided as parameters to a01 (see section 5.3.1 for details on input parameters) the value of City is used inside a service constraint (see section5.3.2 for details about service constraints) for it.

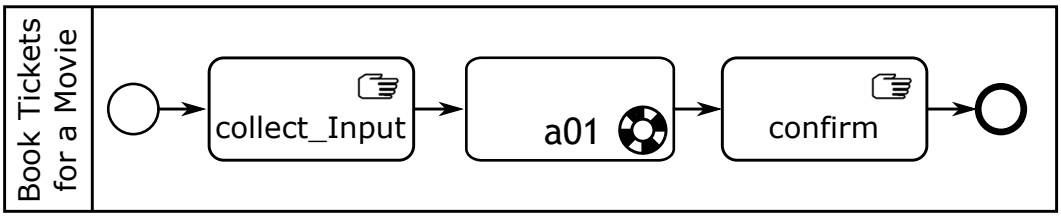

Figure 8.5: BPMN: Book Tickets for a Movie Process

After the operation has finished we added the activity called `confirm`. The purpose of it is to summarize the results of a01 for confirmation of the user. In contrast to `collect_input` this activity does not provide any inputs for the process but requests parameters from it.

8.3 Wallet

To support the user by storing personal preferences according to locally stored business processes we implemented the Wallet. The Wallet allows the user to define (a) default values for different inputs defined within referenced templates and (b) refine the set of services used by activities injecting microflows from the Repository. Storing this preferences locally allows to re-use them each time the business process is executed. This saves the user time and allows to further personalize the business process.

In the following we will explain how we derive the data used by the Wallet from the process description during its import. Next we introduce how the user can define personal preferences for input data and the injected services. At last will we show how this data is used during the execution of such a business process.

8.3.1 Deriving Data from Process Descriptions

Whenever a new business process is imported by the client application its description is parsed for activities ...

1. ... requesting user interaction.
2. ... causing an injection.

Activities requesting user interaction are those `call` elements which define a `template` element within their definition. This kind of activities can be found in the original process description (which are the ones stored within the Wallet and focused here) or inside class level microflows[30] injected from the Repository (which are not stored inside the Wallet and therefore discussed in Section 8.1.4).

As we deal only with process descriptions in the CPEE syntax each of these activities defines an `uri` element and a `name` element inside its `templates` element[31].

The client application uses this information to request the input templates provided at `uri`. It identifies the referenced stylesheet using the attributes *name* and *platform*. The value for the attribute *name* must be the same as provided by the element `name` and the *platform* must match with the locally stored value for the platform of the device (e.g. iPhone).

Now the client application searches each remaining template (which only differ in the *xml:lang* attribute defining its language e.g. EN = English, DE = German) separately. It searches for `input` elements which are not of the *type = 'hidden|button'*. Each of these elements provide an attribute named *id* which is further used to identify the input in the context of the according activity. They further provide a *caption* attribute which represents the caption for the language of the stylesheet used inside the Wallet. The optional *value* attribute defines the default value for this input used inside the Wallet for this language. We think that it is useful to have default values for inputs in a language sensitive context as it allows different default values for different languages, e.g. Vienna (EN), Wien (DE) for the name of the city.

After all the inputs for each provided language are found they are stored locally on the client. Each entry provides information about the activity it is defined in, the template name it is defined in and its caption and value for each supported language.

Activities causing an injection are the second kind of activities the client application searches for when a new business process is imported[32]. In CPEE syntax these `call` elements can be identified by defining a `service` element inside their definition (see Listing 5.6 for an example). The client application stores the value of the `endpoint` element referenced by the

[30] The reason why we decided against supporting class level inputs inside the Wallet is that activities inside a class level microflow change without being noticed by the client application. Out of the same reason it is not possible to define any personal preferences for these activities as we do not store information about class level information any longer then the execution of the process takes.

[31] This is also the case for activities defined within a class level microflow as the language used inside the Repository is transformed into the syntax used by the CPEE during the injection.

[32] We do only parse the original process description because any injection caused by an already injected class level microflow is not allowed to be changed as this may interfere with the original intend of the class level microflow.

attribute *endpoint* from the identified `call` element. This value is used later to guarantee that the user is unable to extend the scope of the endpoint when browsing the Repository.

8.3.2 Setting Personal Preferences

After all necessary information about a business process is educed during its import the user has the possibility to personalize it. As shown in Figure 8.6 the Wallet provides each identified activity to be changed. It divides the collected information in (a) activities related to user interactions and (b) activities related to injections. In our example these are `collect_input`, `confirm` and `a01`. See Figure 8.5 for a BPMN representation of this process.

User Interaction

Each entry leads to an activity identified to request some user interaction. These activities are represented by a set of inputs derived during the import of the according business process (see Section 8.3.1 for details). Each input is presented with its caption and the actual value provided by the defined language of the device (see Section 5.3.5 for details). If the template, the inputs are derived from, defines some default values these default values are stored in the Wallet upon its first use. The user has the possibility to change the value for each input. Doing so allows defining of personal preferences for each process separately. If the user does not want to define a preference for a particular input he/she can remove the entry. Entries not defined in the Wallet must be entered by the user each time the activity is executed. If an entry is removed from the Wallet it disappears in all supported languages. Figure 8.7 shows an example of how the Wallet for a particular activity may look like.

Service Injection

The client application parses each business process during its import for activities causing an injection. Our client application allows the user the narrow the set of services used during the execution by browsing the Repository starting at the URI originally defined in the business process. We argue that narrowing does not interfere with the original intend of the business process because there can be no services included that are not also in the original set. We implemented this by storing the initial value of each service reference and forbid the user to browser beyond this URI by disabling the `Parent Resource` entry in the UI if the original value is reached. In Figure 8.8 we give an example how browsing the Repository looks.

By going through the different resources defined within the Repository the user can choose which one to use for this process. We realize this by parsing the ATOM feed described in Section 4.4.1 to show the user all resources defined at the requested endpoint.

If the user reaches the instance level of the Repository all properties defined in the properties schema (see Section 4.2.4) are presented. The client uses the caption defined for the according language and the actual values from the service description (see Section 4.3 for details). How such a presentation looks is shown in Figure 8.9.

The URI of the chosen resource is set as the value of the according endpoint in the process description. Because of this they do not have to be considered during an execution any more.

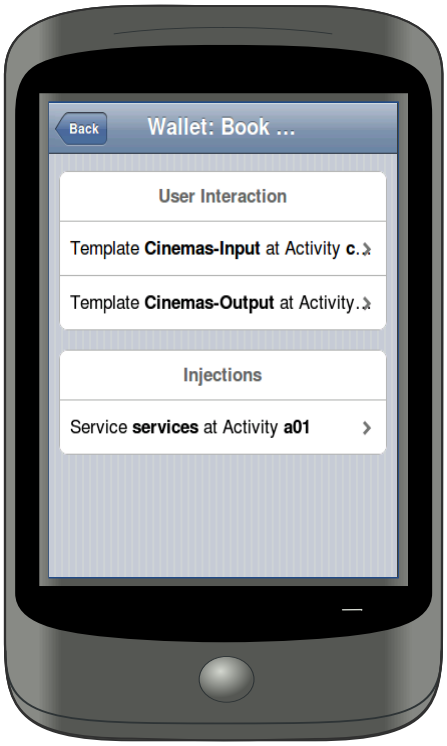

Figure 8.6: Screenshot: Wallet Overview

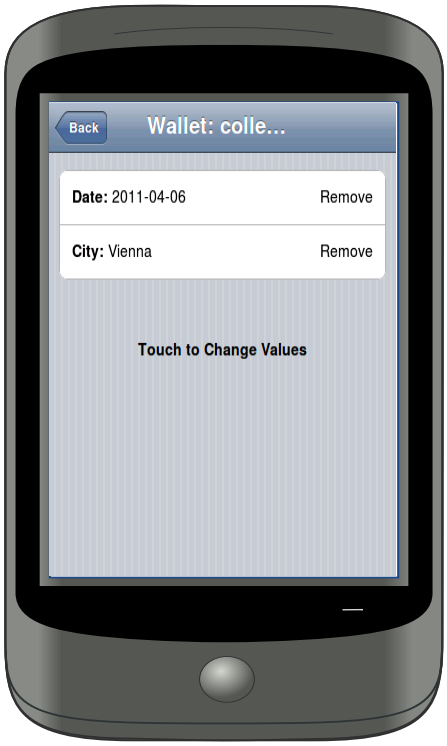

Figure 8.7: Screenshot: Setting Preferences for an User Interaction

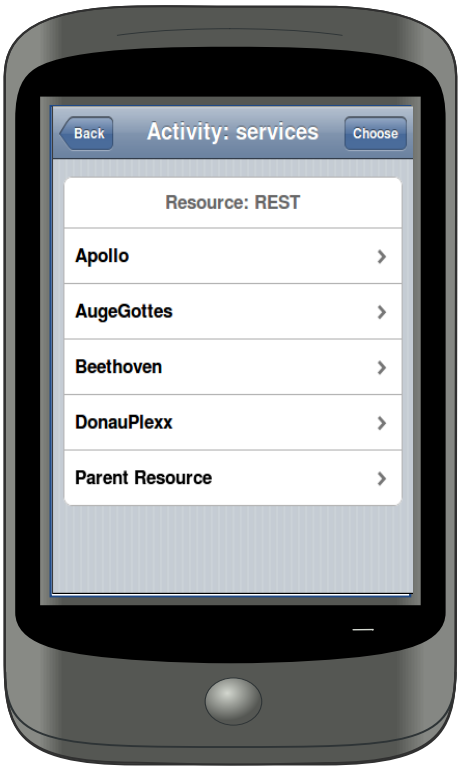

Figure 8.8: Screenshot: Browsing the Repository

Figure 8.9: Screenshot: Properties of a Service within the Repository

As shown in Figure 8.2 we use a Repository which is present on the device. We decided to do so because each user is able to define his/her own set of services in this personal Repository. If the data of this personal Repository is automatically synced with public repositories[33] using e.g. a scheduled syncing job or maintained manually by the user itself depends on the preferences of the user.

8.3.3 Using the Wallet during the Process Execution

During the execution of a business process our client application recognizes activities requesting a user interaction. We already explained that it requests the actual task data from the Worklist, which is a piece of HTML. Whenever the client application loads such a piece of HTML it parses it for all defined `intput` elements and checks if their *id* attribute is defined in the Wallet for this process. If this is the case, it sets the values of the input according to the value defined in the Wallet for the selected language and disables to edit it in the HTML form. If the *id* is not found inside the Wallet the user must enter a value before the execution can be continued.

We implemented it this way to provide as much privacy for the user as possible. When all data is stored locally on the mobile device, and the Wallet can not be parsed directly from activities, the data of the user are safe from unauthorized access.

8.4 Worklist

This section focuses on the Worklist we implemented for our client application. As shown in Figure 8.2 the Worklist is designed to collect pending tasks from different CPEE instances (6) and make the desired XSL transformation (7) if a client requests task data (8). We will introduce therefore the API we implemented for CPEE instances to register worklist tasks and for clients to request these data. We will further introduce the protocol our implementation follows and how the Worklist is kept up-to-date about already completed tasks.

8.4.1 RESTful API

The API consists mainly of two parts: (1) operations provided for the CPEE to register tasks in the worklist and to inform the Worklist about their completion and (2) operations provided for clients to request data about a particular task.

Interaction with WEE/CPEE instances consists of two messages. The first is used to register a task in the worklist. We already described in Section 5.3.4 the *info* attribute and what its purpose is. This attribute is used here to additionally provide the needed information about execution instance with the call. Listing 8.2 gives an example of how a call to the Worklist may look like.

[33] Because the data provided by the Repository is standardized, it is possible to create a Federation of Clouds (see e.g. [19, 74, 75]). Doing so will enable users to collect the data imported into their personal Repository from many different Repositories over the web.

LISTING 8.2: Example of a Call Registering in the Worklist

```
<call id="confirm" endpoint="worklist">
  <parameters>
    <method>post</method>
    <info>true</info>
    <templates>
      <uri>'http://gus.lan/input-forms/cinemas.xsl'</uri>
      <name>'Cinemas-Output'</name>
      <lang>'EN'</lang>
    </templates>
    <parameters>
      <title>data.selected_title</title>
      <date>data.selected_date</date>
      <time>data.starting_time</time>
      <hall>data.hall</hall>
      <res_nr>data.reservation_number</res_nr>
      <cinema>data.selected_cinema</cinema>
    </parameters>
  </parameters>
</call>
```

The content of the `templates` element is moved into the `parameters` section, by the Activity Handler, and then renamed as follows:

- `uri` is moved to `templates-uri`
- `name` is moved to `templates-name`
- `lang` is moved to `template-lang`

This is useful because now we handle information about the associated templates like any other parameter defined inside the `parameters` element (see Section 7.3 for details about the parameter handling).

For its RESTful API, the Worklist implements the following messages at the root URI, e.g. `http://worklist:uri`.

'Register Task' - Message: Is used to register an activity at the Worklist to be executed by a user.

HTTP-Method: POST

Parameters: There are at least the parameters caused by the `info` parameter which are namely `call-instance-uri, call-activity, call-endpoint, call-lay, call-oid` and as well parameters resulting from the definition of a template which are namely `templates-uri, templates-name, template-lang`. It is possible that there is any number of additional parameters if they are defined inside the `parameters` element of the call statement.

Response Parameter: none

HTTP-Status: 201 (Created)

'Syncing After' Notification: This notification is sent by the CPEE every time an activity is finished. Its purpose is to keep the list of pending tasks up-to-date inside the Worklist. Whenever this notification is sent the Worklist checks if the related activity is registered and removes the according entry.

HTTP-Method: POST

Parameters: Three query parameters. First is the

1. `vote:` representing the event which is in our case *syncing_after*
2. `topic:` representing the topic of the event. For *syncing_after* it is *running*
3. `notification:` provides detailed information about the event like e.g. the ID of the activity causing the notification or the URI of the according CPEE instance

Response Parameter: Query parameter name `continue` indicating if the execution should continue (value is *true*) or if it should stop the execution (value is *false*).

HTTP-Status: 200 (OK), 501 (Internal Error)

For the client there is only one message defined which allows to request the specified task data from the Worklist.

'Request Worklist Task Data' - Message: This message is used by client applications to request specific task data for an activity depending on the language and the platform.

HTTP-Method: GET

Parameters:

1. `instance:` represents the URI of the instance the requested task belongs too
2. `activity:` refers to the *id* of the activity causing this worklist task
3. `name:` defines the name of the template that should be used together with this worklist task
4. `lang` and `platform:` further specify the template that will be used for the transformation of the task data.

Response Parameter:

1. `html:` represents the resulted HTML code of the transformation

HTTP-Status: 200 (OK), 501 (Internal Error)

8.4.2 Worklist Interaction Protocol

In this Section we discuss how we realized the protocol of our worklist implementation. In Figure 8.10 we give an overview about the interaction between the client application, the Worklist and the CPEE instance if an activity requests user interaction.

1. Register Worklist Task: When an activity inside the business process refers to the Worklist it uses the *Register Worklist Task* - Message. Using all the provided parameters the Worklist is able to add the activity to the list of pending worklist tasks. It further stores all provided parameter to make them available later in the requested XSLT stylesheet. All tasks in this list are offered to client applications at `http://worklist:uri` using the *Request Worklist Data* - Message. If all this is done, the Worklist subscribes it self for `Syncing After`

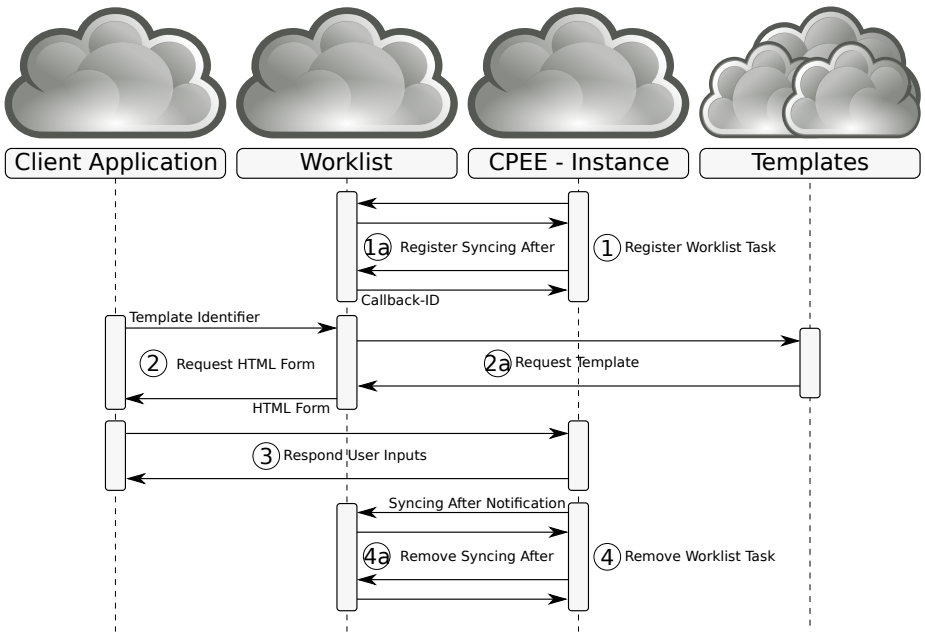

Figure 8.10: Interaction Between Client and Worklist

events of the instance. This way the Worklist gets later informed (see Step 4) when an entry can be removed from the list.

2. Request HTML Form: If the client application finds a pending activity in the Worklist because of the execution position it requests the data from the Worklist using the *Request Worklist Data* - Message. When such a request is sent to the Worklist it checks if the activity is registered in the list of pending tasks. If this is the case, it requests the associated XSLT stylsheets of the activity. After it identified the particular stylesheet by checking the *name*, *xml:lang* and *platform* attributes it start the transformation. All provided parameters, expect the one named `data`, are copied into XSLT stylssheet by defining them as `xsl:variable` and making them accessible during the transformation. In our example the `confirm` activity makes use (beyond the parameters related to the *info* attribute) of this possibility (see Listing 8.2 for an example). If a parameter named `data` was provided with the registration the transformation is performed on this data. For example the activity called `perform_select` inside the microflow of `search_and_book` (see Listing 8.3) provides an XML data structure using the parameter data. When a client requests its task data the provided XML data ist used for the transformation.

LISTING 8.3: Example of a Call Defining the Data Parameter

```
1<call id="perform_select" endpoint="selector_service" endpoint-type="outside"
      http-method="post" info="true" default-tpl-name="mobile" default-tpl-lang
     ="en">
2   <templates> ... </templates>
3   <input name="data" variable="list_merge"/>
4   <output name="show_id" variable="show_id"/>
5   <output name="target" variable="endpoint"/>
6   <output message-parameter="status" type="status"/>
7   <output name="movie_title" message-parameter="movie_title"> ... </output>
8 </call>
```

After the transformation is done the Worklist responds the result of the transformation to the client application. In our implementation the templates are designed to generate HTML output which can be directly displayed by our application.

3. Respond User Inputs: Using the XSLT variable `$instance-uri` and `$callback-id` makes the template to respond directly to the CPEE. This allows the client application to be independent of the Worklist. It is on behalf of the template designer that the message sent by them fits the interface of the according activity in the process description. Listing 8.4 gives an example how such a callback in HTML may look like using JavaScript.

LISTING 8.4: Example of a JavaScript CallBack

```
1<script type="text/javascript">
2  function send() {
3    var callback = '<xsl:value-of select="$instance-uri"/>/callbacks/<xsl:
         value-of select="$callback-id"/>';
4    $.ajax({
5      url: callback,
6      type: 'put',
7      dataType: 'text',
8      data:{'title': $('#title').val(), 'date': $('#date').val(), 'city': $
         ('#city').val()},
9      success: function(res){
10       jQT.goBack();
11     },
12     error: function() {alert('Unable to send data to CPEE instance at ' +
         callback);}
13   });
14 }
15</script>
```

In line 3 the XSLT variables `$instance-uri` and `$callback-id` are referenced.

4. Remove Worklist Task: If the Worklist receives a 'Syncing After' Notification it checks if the activity was registered as a worklist task. If this was the case, it removes it from the list, because it has already finished its execution successfully. After this it removes its subscription for `Syncing After` events related to this activity. In either cases the Worklist responds the parameter `continue` with the value *true* to such requests allowing the CPEE to continue with the execution of the process.

Chapter 9

Conclusion

This chapter points out four major results we educed in this thesis and during the implementation of the prototype system. The scope of our contribution to the field is discussed at the beginning. We continue with a discussion of aspects we did not cover in this thesis although they may influence future work. We finish with things we have learned during the creation of this thesis about *The Life, The Universe and Everything.*

9.1 Contributions

1. A scalable and flexible service marketplace.
2. An algorithm to use the marketplace together with adaptive workflow execution engines.
3. A language to describe service interaction in the context of the marketplace.
4. A prototype implementation of the overall system including a client for mobile devices.

9.1.1 A Scalable and Flexible Service Marketplace

This section is also presented in Vigne et al.[27].

In this thesis we presented the structure of a scalable and flexible service marketplace/repository. The marketplace is realized by providing high level use-cases represented by microflows (class level operations), as well as low level microflows (instance level operations) that describe the interaction between the high level use-cases and actual services. Additionally the marketplace provides a set of properties for the selection / filtering of services (constraints).

As the presented marketplace contains no active functionality, but instead concentrates on the multi-level description of services it is a business enabler, but not a gateway or bottleneck. The feasibility of the marketplace (and its implementation) was shown by including real world services, in our case a diverse set of cinema services exposing searching and booking functionality.

9.1.2 An Algorithm to use the Marketplace

The system introduced in this thesis includes a component responsible for injecting the microflows provided by the market, namely the Injection Service. Because injections include significant changes to the structure of the process, the algorithm was developed in respect to workflow correctness criteria defined by Rinderle et al. [67]. These criteria ensure that the process structure and the execution positions stay valid after changes have been made.

We took the concept of loop-unrolling [69], which origins in compiler building, to ensure that (a) each iteration of the loop is executed according to the original controlflow and therefore unaffected by any former injections and (b) the correctness criteria are not violated.

9.1.3 A Language to Describe Service Interaction

In this thesis we introduced a language to describe microflows, provided by the market, for service interaction.

During the design of the language we focused on providing functionality (a) need for adaptive workflow system by following the Open-point approach and (b) to describe services in a simple and comprehensive way to be used by customers. The introduced language to describe the microflow covers all statements and context elements introduced by Stuermer [45] and is extended with statements and concepts needed to design 'injectable' microflows (process snippets) e.g. message-parameter, constraints, ...

We further provide RNG schemas to validate service descriptions against (general controlflow and application domain specific) to ensure data quality and support service provider and client developer.

Our language covers statements to describe class level interfaces, e.g. properties schema, class level operations and also instance level interfaces like e.g. properties instantiation, transitions (see Eder et al. [49]).

9.1.4 A Prototype Implementation

To show that our introduced design and concepts work, we implemented a prototype system. The design sticks strictly to the SOA and REST paradigms and is therefore very flexible. Further does the extensive use of standards like XML or ATOM-feeds enable a wide support for different programming languages.

The marketplace and all other components required by our system benefit from the SOA design (loosely coupled services), making the overall system very scalable. The interfaces are designed to be RESTful, which makes them very easy to use. Further does the HTTP-protocol allow to use them from a number of different clients (e.g. browsers) and allows for easy application development.

The introduced client application is designed for mobile devices and uses state-of-the-art web-technologies e.g. WebSockets, LocalStorage, ... To allow the client to interact with the user we implemented a worklist component for it.

9.2 Further research

With the incarnation of the marketplace as a basis we will concentrate on elaborating more real world use-cases and provide for flexible integration of secure payment (see Mangler et al. [76]), SLA information and SLA negotiation mechanisms (using WS-Agreements [77]) into the marketplace.

In respect to actual research about Federations of Clouds (e.g. [19, 74, 75]) we will elaborate techniques, allowing to use services provided by different marketplaces in a transparent way. Using our standardized language and RESTful API as a base, matching algorithms for resources (application domains, subgroups, ...) and schemas provided by it are needed.

Different strategies when selecting a service can be achieved and implemented very well with our system. We envision a generic preferences function implemented as a service which is able to decide domain-independent which service best fits the current needs of user.

In the end, we envision a generic Service Store, similar to Apple's App Store [13] or Google's Apps Marketplace [15] based on the introduced concepts. Therefore, additional to the aspects mentioned above, we will further research about utilizing the marketplace(s) within client applications for different mobile devices.

9.3 Lessons Learned

9.3.1 Technical Skills

- The idea behind REST: How/Why to use it the right way.
- XSLT stylesheets: How and when they work and when not.
- The principles of Cloud computing/infrastructures: strengths and weaknesses.
- Programming in Ruby and web technologies (HTML5, JavaScript, SVG).
- Writing papers/thesis using LaTex.
- Designing message based architectures.

9.3.2 Publications Related to this Thesis

A Structured Marketplace for Arbitrary Services
Ralph Vigne and Juergen Manlger and Erich Schickuta and Stefanie Rinderle-Ma
to apprear in: Future Generation Computing Systems (2011)

A RESTful Repository To Store Services For Simple Workflows
Vigne, Ralph and Mangler, Jürgen and Schikuta, Erich and Witzany, Christoph
In: Austrian Grid Symposium, 2009-09-28, Linz (2009)

9.3.3 Personal Experiences

- Two and a half years is a very long time for one master thesis (way to long) - but I really enjoyed it!.
- Things can always be done better on second try.
- My English has improved a lot - imagine how bad it must have been before.
- Working in a great team makes for great fun!

"I may not have gone where I intended to go, but I think I have ended up where I needed to be."

Douglas Adams

Bibliography

[1] M. F. Greaver, *Strategic outsourcing: a structured approach to outsourcing decisions and initiatives*. AMACOM Div American Mgmt Assn, 1999.

[2] C. Stricker, S. Riboni, M. Kradolfer, and J. Taylor, "Market-based workflow management for supply chains of services," in *hicss*, p. 6022, 2000.

[3] L. Schubert, K. Jeffery, and B. Neidecker-Lutz, "The future of cloud computing," tech. rep., European Commision: Information Society and Media, 2010.

[4] D. Hollingsworth *et al.*, "Workflow management coalition: The workflow reference model," *Workflow Management Coalition*, 1993.

[5] D. Wodtke, J. Wei\ssenfels, G. Weikum, and A. K. Dittrich, "The mentor project: Steps toward enterprise-wide workflow management," in *Proceedings of the International Conference on Data Engineering*, p. 556–565, 1996.

[6] P. Grefen, K. Aberer, H. Ludwig, and Y. Hoffner, "CrossFlow: cross-organizational workflow management for service outsourcing in dynamic virtual enterprises.," *IEEE Data Engineering Bulletin*, vol. 24, p. 52–57, 2001.

[7] J. Yu and R. Buyya, "A taxonomy of workflow management systems for grid computing," *Journal of Grid Computing*, vol. 3, no. 3, p. 171–200, 2005.

[8] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, 2008.

[9] F. Rosenberg, F. Curbera, M. J. Duftler, and R. Khalaf, "Composing restful services and collaborative workflows: A lightweight approach," *IEEE Internet Computing*, p. 24–31, 2008.

[10] R. Buyya, D. Abramson, and J. Giddy, "A case for economy grid architecture for service oriented grid computing," in *10th Heterogeneous Computing Workshop*, 2001.

[11] S. Jha, A. Merzky, and G. Fox, "Using clouds to provide grids with higher levels of abstraction and explicit support for usage modes," *Concurrency and Computation: Practice and Experience*, 2009.

[12] M. E. Porter, "Competitive strategy," *Measuring Business Excellence*, vol. 1, no. 2, p. 12–17, 1993.

[13] "Apple - iPhone - learn about apps available on the app store." http://www.apple.com/iphone/apps-for-iphone/. [Last access: 26.08.2010].

[14] "Apple - app store - buy, download, and install apps made for mac.." http://www.apple.com/mac/app-store/. [Last access: 02.03.2011].

[15] "Google apps marketplace." https://www.google.com/enterprise/marketplace/?pli=1. [Last access: 02.03.2011].

[16] "Home - android market." https://market.android.com/. [Last access: 02.03.2011].

[17] M. Lewis and N. Slack, *Operations management*. Routledge, 2003.

[18] "Definition of heterogeneous from oxford dictionaries online." http://oxforddictionaries.com/definition/heterogeneous#m_en_gb0375710.007. [Last access: 13.06.2011].

[19] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, 2009.

[20] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of High Performance Computing Applications*, vol. 15, no. 3, p. 200, 2001.

[21] G. C. Fox and D. Gannon, *Workflow in grid systems*. Citeseer, 2006.

[22] K. Nichols, D. L. Black, S. Blake, and F. Baker, "Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers." http://tools.ietf.org/html/rfc2474, Dec. 1998. [Last access: 03.08.2010].

[23] P. Lambros, M. T. Schmidt, and C. Zentner, "Combine business process management technology and business services to implement complex web services," *IBM Corporation*, 2001.

[24] G. Joeris, "Defining flexible workflow execution behaviors," *Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications*, vol. 24, p. 49–55.

[25] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[26] K. Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. Fernandes, and G. Mehta, "Adaptive workflow processing and execution in pegasus," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 16, p. 1965–1981, 2009.

[27] R. Vigne, J. Manlger, E. Schickuta, and S. Rinderle-Ma, "A structured marketplace for arbitrary services," *to apprear in: Future generation Computing Systems*, 2011.

[28] O. Corcho, P. Alper, P. Missier, S. Bechhofer, and C. Goble, "Grid metadata management: requirements and architecture," in *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, p. 97–104, 2007.

[29] P. Missier, P. Alper, O. Corcho, I. Dunlop, and C. Goble, "Requirements and services for metadata management," *IEEE Internet Computing*, vol. 11, no. 5, p. 17–25, 2007.

[30] P. A. Bernstein and U. Dayal, "An overview of repository technology," in *Proceedings of the International Conference on Very Large Data Bases*, p. 705–705, INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1994.

[31] L. Clement, A. Hately, C. von Riegen, T. Rogers, *et al.*, "UDDI version 3.0. 2." http://www.uddi.org/pubs/uddi_v3.htm, 2004. [Last access: 02.08.2010].

[32] "CF SIC code list." http://www.sec.gov/info/edgar/siccodes.htm. [Last access: 11.05.2011].

[33] "NAICS main page." http://www.census.gov/eos/www/naics/. [Last access: 11.05.2011].

[34] "UNSPSC homepage." http://www.unspsc.org/. [Last access: 11.05.2011].

[35] "SOAP specifications." http://www.w3.org/TR/soap/. [Last access: 05.08.2010].

[36] "Web storage." http://dev.w3.org/html5/webstorage/. [Last access: 05.04.2011].

[37] M. B. Blake, A. L. Sliva, M. zur Muehlen, and J. V. Nickerson, "Binding now or binding later: The performance of uddi registries," in *40th Annual Hawaii International Conference on System Sciences, 2007. HICSS 2007*, p. 171c–171c, 2007.

[38] R. Vigne, J. Mangler, E. Schikuta, and C. Witzany, "A RESTful repository to store services for simple workflows," in *Austrian Grid, National Symposium*, (Linz, Austria), p. 1, OCG, Sept. 2009.

[39] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D. L. Mcguinness, E. Sirin, and N. Srinivasan, "Bringing semantics to web services with owl-s," *World Wide Web*, vol. 10, no. 3, p. 243–277, 2007.

[40] N. Srinivasan, M. Paolucci, and K. Sycara, "An efficient algorithm for OWL-S based semantic search in UDDI," *Semantic Web Services and Web Process Composition*, p. 96–110, 2005.

[41] Y. Han, A. Sheth, and C. Bussler, "A taxonomy of adaptive workflow management," in *Workshop of the 1998 ACM Conference on Computer Supported Cooperative Work*, 1998.

[42] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, *et al.*, "Giggle: A framework for constructing scalable replica location services," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.

[43] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: a computation management agent for multi-institutional grids," *Cluster Computing*, vol. 5, no. 3, p. 237–246, 2002.

[44] G. Stuermer, J. Mangler, E. Schikuta, and C. Witzany, "Network based execution of dynamic workflows in grid and cloud based environments," in *Austrian Grid, National Symposium*, (Linz, Austria), OCG, Sept. 2009.

[45] G. Stürmer, *An architectur of a workflow execution engine to enable network based execution of dynamic workflows.* Masterthesis, University of Vienna, 2010.

[46] "HTTP/1.1: status code definitions." http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. [Last access: 09.10.2010].

[47] R. Sayre, "Atom: The standard in syndication," *IEEE Internet Computing*, vol. 9, no. 4, pp. 71–78, 2005.

[48] J. Eder and W. Liebhart, "The workflow activity model WAMO," in *3rd International Conference on Cooperative Information Systems*, 1995.

[49] J. Eder, J. Mangler, E. Mussi, and B. Pernici, "Using stateful activities to facilitate monitoring and repair in workflow choreographies," in *Proceedings of the 2009 Congress on Services - I*, pp. 219–226, IEEE Computer Society, 2009.

[50] E. Sirin, B. Parsia, and J. Hendler, "Filtering and selecting semantic web services with interactive composition techniques," *IEEE Intelligent Systems*, p. 42–49, 2004.

[51] R. Khalaf, N. Mukhi, and S. Weerawarana, "Service-oriented composition in BPEL4WS," *WWW (Alternate Paper Tracks)*, 2003.

[52] "Web service definition language (WSDL)." http://www.w3.org/TR/wsdl. [Last access: 05.08.2010].

[53] "XML schema part 2: Datatypes second edition." http://www.w3.org/TR/xmlschema-2/. [Last access: 08.09.2010].

[54] M. Alam, M. Nauman, X. Zhang, T. Ali, and P. C. Hung, "Behavioral attestation for business processes," in *Web Services, IEEE International Conference on*, (Los Alamitos, CA, USA), pp. 343–350, IEEE Computer Society, 2009.

[55] "XML path language (XPath)." http://www.w3.org/TR/xpath/. [Last access: 09.12.2010].

[56] "XSL transformations (XSLT)." http://www.w3.org/TR/xslt.

[57] Y. M. Teo and R. Ayani, "Comparison of load balancing strategies on cluster-based web servers," *Simulation*, vol. 77, no. 5-6, p. 185, 2001.

[58] "Apache HTTP server - load balancer." http://httpd.apache.org/docs/2.1/mod/mod_proxy_balancer.html. [Last access: 31.08.2010].

[59] S. A. White, "Introduction to BPMN," *IBM Cooperation*, p. 2008–029, 2004.

[60] J. Conallen, "Modeling web application architectures with UML," *Communications of the ACM*, vol. 42, no. 10, p. 63–70, 1999.

[61] T. Bray, D. Hollander, and A. Layman, "Namespaces in XML," 1999.

[62] "Relax NG home page." http://www.relaxng.org/. [Last access: 05.08.2010].

[63] "HTTP/1.1: protocol parameters." http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html. [Last access: 09.03.2011].

[64] "JSON." http://www.json.org/. [Last access: 29.10.2010].

[65] "Ruby programming language." http://www.ruby-lang.org/en/. [Last access: 11.04.2011].

[66] E. Schikuta, H. Wanek, and I. U. Haq, "Grid workflow optimization regarding dynamically changing resources and conditions," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 15, p. 1837–1849, 2008.

[67] S. Rinderle, M. Reichert, and P. Dadam, "Correctness criteria for dynamic changes in workflow systems – a survey.," *Data and Knowledge Engineering*, vol. 50, no. 1, p. 9–34, 2004.

[68] S. Rinderle, M. Reichert, and P. Dadam, "Flexible support of team processes by adaptive workflow systems.," *Distributed and Parallel Databases*, vol. 16, no. 1, p. 91–116, 2004.

[69] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, p. 345–420, 1994.

[70] "The WebSocket API." http://dev.w3.org/html5/websockets/. [Last access: 05.04.2011].

[71] "jQTouch — jQuery plugin for mobile web development." http://jqtouch.com/. [Last access: 05.04.2011].

[72] "PhoneGap." http://www.phonegap.com/. [Last access: 15.06.2011].

[73] "Cross-Origin resource sharing." http://www.w3.org/TR/cors/. [Last access: 05.04.2011].

[74] R. Ranjan and R. Buyya, "Decentralized overlay for federation of enterprise clouds," *Arxiv preprint arXiv:0811.2563*, 2008.

[75] R. Buyya, R. Ranjan, and R. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," *Algorithms and Architectures for Parallel Processing*, p. 13–31, 2010.

[76] J. Mangler, C. Witzany, O. Jorns, E. Schikuta, H. Wanek, and I. U. Haq, "Mobile gSET - secure business workflows for Mobile-Grid clients," *Concurrency and Computation: Practice and Experience*, vol. 9999, no. 9999, p. n/a, 2009.

[77] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (WS-Agreement)," in *Global Grid Forum*, 2004.

List of Figures

Listings

Appendix A

Abstract

A.1 English

Today's companies more and more embrace the utilization of inter-organizational services as part of their internal business processes. The benefit from this outsourcing is manifold, ranging from lower maintenance burden to predictable cost. One problem is, that for companies in order to not bind themselves to a single service provider, they have to ensure that the services they consume are interchangeable. Therefore a marketplace for services, based in a common set of rules, that allows companies to stay flexible when selecting business partners is needed.

In this thesis we introduce a hybrid process and service repository acting as a base for such a marketplace. Organizing services into different application domains with a common interface allows easy usage of the provided services for the customers, while the support of interface transformation within service description keeps the vendors flexible. We further introduce a prototype system demonstrating how to use this information on in combination with adaptive workflow execution engine and Cloud infrastructures as a base. To proof the feasibility of the introduced concepts, a mobile client executing a real-world example is introduced.

A.2 Deutsch

Heutige Unternehmen verwenden mehr und mehr inter-organisationale/externe Services innerhalb ihrer internen Geschäftsprozesse. Die Vorteile dieses Service-Outsourcings sind vielfältig, und reichen von geringeren Wartungsaufwand bis zu Kostenvorhersage. Jedoch müssen Unternehmen, um sich nicht an einen einzelnen Anbieter zu binden, dafür sorgen, dass die verwendeten Services austauschbar sind. Um diese gewünschte Flexibilität bei der Serviceauswahl zu unterstützen ist ein Markplatz für Services, welcher auf einer gemeinsamen Menge von Regel aufbaut, nötig.

In dieser Arbeit stellen wir ein hybrides Prozess und Service Verzeichnis vor, welches als Basis für solch einen Marktplatz verwendet werden kann. Die Einteilung in unterschiedliche Anwendungsdomänen mit gemeinsamen Schnittstellen unterstützt Kunden bei der Nutzung der angebotenen Services. Die Möglichkeit innerhalb der Servicebeschreibungen die Schnittstellen/Parameter zu transformieren erlaubt es Anbietern flexibel mit ihren Services umzugehen. Weiter stellen wir in dieser Arbeit einen Prototyp vor, welcher demonstriert, wie diese Information in Kombination mit adaptiven Workflowsystemen innerhalb von Cloud - Infrastrukturen verwendet werden kann. Um die Umsetzbarkeit der vorgestellten Konzepte zu zeigen wird eine Anwendung für mobile Geräte vorgestellt, welche ein reales Beispiel unter Einbeziehung des Marktplatzes ausführt.

Appendix B

Curriculum Vitae

Ralph Vigne

Geboren am:	16. Mai 1980
Geboren in:	Krems a. d. Donau

Ausbildung

Bakkalaureatsstudium Wirtschaftsinformatik	2003 - 2007
Bachelorstudium Politikwissenschaft	2005 - 2010
Masterstudium Wirtschaftsinformatik	2007 - 2011

Publikationen

A Structured Marketplace for Arbitrary Services
Ralph Vigne and Juergen Manlger and Erich Schickuta and Stefanie Rinderle-Ma
to apprear in: Future generation Computing Systems (2011)

A RESTful Repository To Store Services For Simple Workflows
Vigne, Ralph and Mangler, Jürgen and Schikuta, Erich and Witzany, Christoph
In: Austrian Grid Symposium, 2009-09-28, Linz (2009)

A Heuristic Query Optimization Approach for Heterogeneous Environments
Beran, Peter and Mach , Werner and Vigne, Ralph and Mangler, Jürgen and Schikuta, Erich
In: The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2010-05-27, Melbourne, Victoria, Australia (2010)