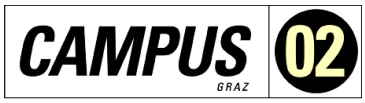
MASTER'S THESIS

Prototype of a Portal for Scientific Data Access Based on Scala

carried out at



Course Programme
Information Technologies and IT Marketing

By: Florian Topf
PID: 1210320012

Graz, on December 16, 2013

.....
Florian Topf

Declaration of Authorship

I hereby declare, that I have written this thesis without any help from others and without the use of documents and aids other than those stated, that I have mentioned all used sources and that I have cited them correctly according to established academic citation rules.

.....
Florian Topf

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....
Florian Topf

Acknowledgements

The RTD work described in this thesis was accomplished within the EU FP7-SPACE project IMPEx (Integrated Medium for Planetary Exploration, project number 262863). The author is thankful to all contributions and suggestions made to the IMPEx portal project by the participating data providers in particular with regard to the establishment of web service interfaces and metadata trees.
The overall process of elaborating this thesis and the IMPEx portal subsequently was supervised by the IMPEx project manager DI Tarek Al-Ubaidi from the IWF (Institut für Weltraumforschung/Space Research Institute) of the Austrian Academy of Sciences in Graz. Additionally Prof. DI Manfred Steyer gave guideance with his didactic role at the Campus 02 and as supervisor of this thesis.
The author would also like to thank Martin Odersky and his team for the excellent documentation of the programming language Scala and related libraries provided at the dedicated website[1] which in particular supported the preliminary development of the IMPEx portal described in this thesis.

Florian Topf

Graz, on December 16, 2013

[1] Scala language website: `http://www.scala-lang.org/`

Abstract

Recent studies by the author have shown that the "Integrated Medium for Planetary Exploration" (IMPEx) FP7-SPACE project is evolving towards becoming a fully operational Virtual Observatory (VO). The service-oriented architecture provides state-of-the-art web services and a robust XML data model which enables the exploitation of observational- and simulated data collated from research performed within planetary plasma- and magnetospheric physics.
The author shows in this thesis primarily that the elaborated infrastructure of IMPEx is extendable with new services, complying with the IMPEx data model and protocol. Secondly, the author focuses on a recently revived software paradigm: *functional programming*.
This paradigm originates from initial attempts to develop a general language for evaluating mathematical expressions: *lambda calculus*. This thesis examines the features provided by this formalism and how they are transformed into a modern programming language. It is shown using the example of the object-functional language Scala how functional aspects are merged into a wider spread paradigm in software development: *object-oriented programming*.
The result of this thesis is a centralised portal for IMPEx which enables the usage of IMPEx web services with frameworks available in the scope of Scala. It is shown how functional aspects support distributed and scalable software with their concise syntax and semantics, as well as handling of XML structured documents. It is concluded that Scala can be considered a new base technology for the development of more integrated simulation environments enabling complex scientific workflows.

Kurzfassung

Vorangegangene Studien des Autors haben gezeigt, dass sich das FP7-SPACE Projekt "Integrated Medium for Planetary Exploration" (IMPEx) in Richtung eines vollständig definierten Virtuellen Observatorium (VO) entwickelt. Die serviceorientierte Architektur stellt state-of-the art Webservices und ein robustes XML Datenmodell zur Verfügung, welche die Nutzung von Observations- und Simulationsdaten aus Forschungsergebnissen im Bereich der Plasma- und Magnetosphärenphysik ermöglicht. Der Autor zeigt in dieser Arbeit primär wie diese erarbeitete Infrastruktur mit neuen Services erweiterbar ist, die ebenfalls mit dem IMPEx Datenmodell und Protokoll konform sind. Sekundär beschäftigt sich der Autor mit einem kürzlich wiederbelebten Softwareparadigma: *Funktionale Programmierung*.
Dieses Paradigma hat ihren Usprung in den ersten Versuchen eine allgemeine Sprache für die Evaluierung von mathematischen Ausdrücken zu entwickeln: das *Lambda-Kalkül*. Diese Arbeit untersucht die Eigenschaften dieses Formalismus und zeigt auf, wie diese in eine moderne Programmiersprache transformiert werden. Es wird am Beispiel der objekt-funktionalen Sprache Scala gezeigt wie funktionale Aspekte mit einem weitaus mehr verbreiteten Paradigma verschmolzen werden können: *Objektorientierte Programmierung*.
Das Ergebnis dieser Arbeit ist ein zentralisiertes IMPEx Portal, welches die Verwendung von IMPEx Webservices mit Hilfe der entsprechenden Frameworks in Scala ermöglicht. Es wird gezeigt, wie funktionale Aspekte mit deren kompakten Syntax und Semantik die Entwicklung von verteilter und skalierbarer Software unterstützen sowie die Handhabung von XML Dokumenten vereinfachen. Es wird zum Schluss gekommen, dass Scala als neue Basistechnologie für die Entwicklung von integrierten Simulationsumgebungen gesehen werden kann, welche in weiterer Folge komplexe wissenschaftliche Workflows ermöglichen.

Contents

1. Introduction **2**
- 1.1. Present Status and Motivation . . . 2
- 1.2. Conceptual Formulation . . . 4
- 1.3. Objectives of this Thesis . . . 4
- 1.4. Approach and Methodology . . . 5
- 1.5. Composition of this Thesis . . . 5
- 1.6. Work done by the Author . . . 6

2. Elements of Functional Programming **7**
- 2.1. Lambda Calculus and the Turing Machine . . . 8
  - 2.1.1. Lambda-Expressions . . . 8
  - 2.1.2. Lambda-Conversion . . . 10
  - 2.1.3. Lambda-Definability . . . 11
- 2.2. Functional Style . . . 13
- 2.3. Expressions and Types . . . 15
  - 2.3.1. Primitive Types . . . 17
  - 2.3.2. Polymorphism . . . 17
- 2.4. Evaluation Strategies . . . 18
  - 2.4.1. Call-By-Name . . . 18
  - 2.4.2. Call-By-Value . . . 19
- 2.5. Data Structures . . . 20
  - 2.5.1. Tuples . . . 20
  - 2.5.2. Lists . . . 21
  - 2.5.3. Trees . . . 22
  - 2.5.4. Arrays . . . 23
  - 2.5.5. Abstract Data Types . . . 24
- 2.6. Advanced Functional Features . . . 25

3. Application to Object-oriented Programming **28**
- 3.1. The Scala Programming Language . . . 29
  - 3.1.1. Functions and Evaluation . . . 30
  - 3.1.2. Abstractions and Classes . . . 32
  - 3.1.3. Types and Polymorphisms . . . 35
  - 3.1.4. Collections and Operations . . . 39
- 3.2. Functional JavaScript . . . 44

4. Evaluation of Object-functional Capabilities **48**
- 4.1. Domain Specific Languages and Parsers . . . 49

4.2. Parallel and Concurrent Programming 54
4.2.1. Actors and the Akka framework 55
4.2.2. Scala Futures and Use Cases 57
4.3. Side-effects and Basic I/O 58
4.4. Scala Web Frameworks 62

5. The IMPEx Portal **68**
5.1. Purpose and Aims 69
5.2. User Requirements 71
5.2.1. Search Capabilities 72
5.2.2. Visualisation Capabilities 75
5.2.3. Communication Capabilities 77
5.2.4. Processing Capabilities 79
5.2.5. User Interface Requirements 81
5.2.6. Backend and Administration features 84
5.3. Architectural Design 86
5.3.1. Overall Architecture 86
5.3.2. The IMPEx Registry 91
5.3.3. The IMPEx Messaging API 96

6. Results and Discussion **102**

7. Conclusions and Outlook **106**

A. IMPEx Configuration **109**

B. IMPEx Portal Map **112**

C. IMPEx Portal Main Classes **120**

Glossary **122**

List of Figures **126**

List of Tables **127**

Listings **128**

Bibliography **129**

1. Introduction

The EU FP7-SPACE project "Integrated Medium for Planetary Exploration" (IMPEx) is currently evolving towards a standardised Virtual Observatory (VO) and is reaching a major milestone in development of distinct services and databases which were specified in Topf (2012a). After a challenging process of defining homogenous web service interfaces and elaborating a robust data model for specific semantics, the project is now entering its first public review phase. This thesis is a product of this phase as it is discovering new possibilities to provide the actual IMPEx services to a wider audience for scientific and educational purposes through one single entry point, the IMPEx portal.

1.1. Present Status and Motivation

Currently the IMPEx project is in its implementation phase after definition of the IMPEx architecture which is comprised of three distinct service layers according to Topf (2012a, pp. 25): the user layer, the tool layer and the resource layer. The basis of the IMPEx infrastructure is formed by the resource layer which provides access to stateless data services through metadata registries for observational and simulation data. The data services are comprised of searchable XML trees compliant with the IMPEx data model described in Topf (2012a, pp. 30). These metadata files are accessible through the database endpoints described in the IMPEx configuration file (see appendix A). On top of the resource layer all database endpoints provide computational services via SOAP interfaces which are needed to exploit the actual data for a selected subset in the metadata tree. Finally the IMPEx tools AMDA, 3DView and ClWeb are glued together with a SOAP interface, the IMPEx workspace, to realize exchange of selected metadata from one tool to another Topf (2012a, pp. 27). At the moment the IMPEx tools are integrating the data and computational services in their environment to be able to process and visualise the data in different ways. To exchange view configurations between tools on the client side the XML-RPC based interface SAMP is used which is actually the only way to be able to synchronously work with two tools in the IMPEx environment. This can be seen as a drawback to usability but the solution with SAMP solves the problem pragmatically and still doesn't conflict with the major goals of IMPEx. Even in the respective recommendation document of IVOA for SAMP by Taylor et al. (2011) the provided solution is described

as having a "modest scope" and hence is not designed to provide an optimal messaging system for all different scenarios. This kind of inter-tool communication does not provide security or overall integrity of the messages being delivered and therefore the tools must take care of the validation and the establishment of the respective secure environments where SAMP calls can be processed.

The main idea behind this thesis is to look forward into the future with regard to tool- and more specifically server integration including also an efficient use of the client machine through modern web-based technologies. The main reasoning behind increased integration of computational aspects on the server-side is that rudimentary inter-tool communcation on the client-side can be avoided but also security aspects can be handled centralised as opposed to the current architecture. The community of the IMPEx project also sees a problem in the different clients as it is difficult to find the needed functionalities through the current project environment. Furthermore the trade-off between interoperability and the splitting of functionalities over a wide range of applications, each with its own proprietary interactive model is seen as a major risk in the future where the system is adding new models and tools. There must be a clear settlement of the IMPEx infrastructure between integration and service orientation.

Within the IMPEx project this particular challenge was identified as a key aspect which needs to be studied further to gain momentum for follow-up projects. The proposed IMPEx portal and its related research and technical development is seen as an optimal testbed for further integration in the user- as well as the resource layer. This increased integration does not interfere with the two key aspects of IMPEx with regard to service orientation: the data model and the web service interfaces. They will still provide the connection to the data- and computational services of IMPEx and will not loose its basic feature to be composable in a variety of configurations. But the IMPEx portal must be designed in a way to on one hand increase the usage of the client's computing power and on the other hand to overcome the problem of extensive data transfer between databases and tools within the service ecosystem of IMPEx. The idea here is to provide a homogenous API to the scattered services of IMPEx at the portal which can be be exploited both locally and from remote machines.

The server-side technology of the portal which connects to the IMPEx services must be able to handle the XML trees and results from the web services in a way which supports the functional orientation of the models and databases which are settled behind those interfaces. With this approach it is easier to decompose the large datastructures and modelling routines of the IMPEx simulation databases to be able to delegate particular standard procedures of the workflows described in the IMPEx science cases in Topf (2012a, chap. 5) to the portal server or even the clients machine. This thesis will use the functional programming paradigm as a basis to develop the architecture of the portal since it will also be relevant in the future to integrate simulation models

directly into web enabled programming languages rather than building up complex interfaces to imperative low level programming languages such as Fortran[1].

To understand the reasoning behind using aspects of functional programming for the IMPEx portal this thesis will provide a detailed study of all necessary key elements by showing the fundamental ideas of functional programming and their application in the real world. This approach will clearly show where the IMPEx infrastructure can be improved among other things by functional aspects with regard to tool and service integration to eliminate the risks identified within the current highly distributed environment.

1.2. Conceptual Formulation

Based on the present status of IMPEx and the respective open task of developing an IMPEx portal, the author has formulated a major research question which will be answered in course of preliminary studies of the chosen base technology as well as evaluation of portal user requirements:

"Which concepts and abilities of functional programming are suitable and needed for implementation of an up-to-date portal for scientific data and model resources access focusing on scalability and expandability?"

1.3. Objectives of this Thesis

In accordance with the major research question four main objectives of this thesis were elaborated:

1. Study of basic and advanced concepts of functional programming and their application to object-oriented programming languages such as Scala and in a wider sense JavaScript.
2. Study of the applicability of object-functional programming languages such as Scala in distributed frameworks and respective tool chains.
3. Design and conception of a prototype portal for the IMPEx project as a first test case for an integration of new tools in the IMPEx infrastructure using Scala. Exploitation of the IMPEx protocol (tree.xml, methods.wsdl) and visualisation of XML based data.

[1] Fortran is a traditional procedural programming language still widely used for numerical calculations in the scientific community. See also: `http://www.fortran.com/`

4. Study of the applicability of the resulting IMPEx portal in the context of advanced workflow systems and enterprise service bus solutions for future consideration.

1.4. Approach and Methodology

The main steps of the technical developments in this thesis are comprised of:

- Definition of the main elements of functional programming and study of their advantages in distributed computational data networks and service-oriented infrastructures.
- Analysis of the application of functional principles into object-oriented programming languages in particular to Scala but also JavaScript.
- Definition of main concepts of object-functional programming dealing with scalability, immutability and concurrency which are needed for the implementation of the IMPEx portal.
- Evaluation of the flexibility of object-functional programming languages such as Scala with regard to further integration into web based applications, common frameworks and tool chains.
- Architectural design and prototyping of the IMPEx service portal based on the results of the preceding study of Scala and JavaScript concepts and respective documentation for future projects.

1.5. Composition of this Thesis

This thesis is composed of the following theoretical chapters:

1. Introduction to elements of functional programming in general.
2. Definition of advanced functional programming concepts and their evolution.
3. Application of functional programming aspects into object-oriented programming.
4. Definition of main features of object-functional programming based on Scala.

The practical sections of this thesis are part of the IMPEx project and are dealing with the design and implementation of an IMPEx portal using functional principles elaborated in the theoretical chapters. These functional principles are contained in

the used base technology, the programming language Scala. The practical sections are comprised as follows:

1. Evaluation of extended features of object-functional programming based on Scala.
2. Listing of purpose & aims of the IMPEx portal.
3. Evaluation of the current IMPEx web services and the used data model.
4. Definition of the user requirements and the architectural design of the IMPEx portal including the presentation of selected functional aspects which are prototyped in course of this thesis.

1.6. Work done by the Author

The design and implementation of the IMPEx portal is content of a working contract provided by the IMPEx project where the author of this thesis is the selected contracting partner. The emphasis in this task is on finding modern and useful concepts for the realisation of a service-driven portal and respective documentation of the complete RTD process.

2. Elements of Functional Programming

Functional programming has a long history and its roots are found in mathematical theories in the field of predicate logic elaborated in the thirties of the 20^{th} century. During the time of the great depression a couple of mathematicians were working on solving a challenge in logic which was introduced by David Hilbert in the twenties in the melting pot of Germany's mathematical research institutions, the Unversity of Göttingen. He named the challenge "Entscheidungsproblem" (German for "decision problem") which was basically comprised of the search for a general algorithm or method which is able to decide in finite time upon a given set of axioms of a formal language, if a constructed logical expression is generally valid or not. In this case "generally valid" means that the proof must imply that every construct satisfying the axioms is again valid in the formal language. The challenge was officially proposed in the major work of David Hilbert and Wilhelm Ackermann "Grundzüge der theoretischen Logik" in 1928 describing a formalism of elementary mathematical logic, which is known today as first-order logic (see: Hilbert & Ackermann, 1928).

The term "decidable" in this regard would play a big role in the field of computer science in the future, since the logic principles defined here form the basis of the well known computability theory dealing with the question if a problem is solvable by a given machine and it's defined syntax and semantics or not. Kurt Gödel was making the first attempt in 1931 by showing that the challenge is not solvable for logical expressions based on arithmetics of natural numbers. He proved in his "Unvollständigkeitstheorem" that there are expressions possible for which can not be decided if they are generally valid or not (see: Gödel, 1931).

Two distinct logicians finally came to a more complete solution to Hilberts challenge indedepently of each other in the year 1936, namely Alan Turing (Turing, 1936) and Alonzo Church (Church, 1936). They both made their conclusion that this problem is generally unsolvable by using newly introduced formalisms which were greatly influenced by the formal language Gödel used in the "Unvollständigkeitstheorem". Both scientists practically worked on the same issue at this time. They studied formal mathematical systems and tried to make theoretical approaches in answering the question on whether a hypothetic machine could generally solve mathematical problems by applying a consistent formal language or not. They worked out different approaches to come to an conclusion - the *Lambda Calculus* and the *Turing-Machine*

which were consequently used as a basis for computational theory in the upcoming decades. In order to understand the principles behind functional programming one must look into these assumptions in computational theory which from the beginning had to deal with decision problems such as described by Hilbert and Ackermann. As a first step this thesis will look on the results of Turing and Church which resulted in the definition of the "Church-Turing-Thesis" and had a tremendous impact on the forthcoming achievements in theoretical informatics in the 20^{th} century.

2.1. Lambda Calculus and the Turing Machine

In course of their investigations Church and Turing implicitly developed a definition of a construct which is called an *algorithm* in informatics today. The idea behind their work was that every mathematical problem can be constructed through predefined rules complying to a formal language. Despite of solving a mathematical problem through their investigations for the well-known "Entscheidungsproblem" their contribution to computational theory was of equal importance. The lambda calculus was originally designed for examining the solvability of mathematical problems and therefore basically provided a way to evaluate functions. That is one reason why lambda calculus as a notation is seen as the historical founding father of functional programming - everything is expressed by mathematical functions. So what does an algorithm contain in the formal language of lambda calculus and how does this approach apply to common programming languages? According to Lippe (2009, p. 23), lambda calculus being a theory of functions is slightly different to the traditional meaning of a function in mathematical theory. In mathematical theory a function is a calculation specification which basically maps a quantity of inputs into a quantity of outputs with a static binding. Each definition has a concrete result which is yielded by the function body. In lambda calculus each of these three components of the calculation specification is represented by a so-called *λ-expression*. This approach, considering functions as a set of rules as stated in Barendregt (1981, p. 3) goes back to a "old fashioned notation" tributed to Dirichlet[1], since functions are commonly considered as graphs consisting of tuples from arguments and values in today's mathematical theory.

2.1.1. Lambda-Expressions

There are various deviations of the applications of *λ-expressions* throughout the literature which all implement the same basic axioms, which were originally proposed

[1]Peter Gustav Lejeune Dirichlet was a famous german mathematican, who worked in the field of number theory in the 19^{th} Century. See: `http://www-history.mcs.st-and.ac.uk/Biographies/Dirichlet.html`

by Alonzo Church in "The Calculi of Lambda-Conversion" (Church, 1941). Church's classical abstract notion is referred to as *pure untyped lambda calculus* in accordance to Hudak (1989, p. 364) and has the following elements:

- Variables or identifiers in the form of lower-case letters e.g. $x, y, z, ..$
- Function abstractions e.g $\lambda x.A$, which defines an anonymous function with x as parameter and A as the function body.
- Function applications e.g (FA), which applies the function F to the expression A

In function abstractions the parameter is usually bound to the function body A over the λ-notation if it is also occuring inside A, but not necessarily since it can also occur "freely" in lambda expressions. An example of a free variable in a function body would be the *λ-expression* $\lambda x.y$, where y is not bound like λx and moreover whose result is always y independently of the argument x. Note that the function body in this case can also also be a further *λ-expression* itself. The basic version of a function abstraction is represented by the *identity function* $\lambda x.x$ which takes x as parameter and simply returns x in its body. It can be seen in this basic example that x is now clearly bound to the function body. A respective application on a *identity function* with an argument y is written in lambda calculus $\lambda y.(\lambda x.x)$ (Michaelson, 1988, p. 16). The definiton of a function, it's parameters and application are united in one single *λ-expression*. The previous examples show that function applications are left-associative and abstractions are right-associative, following the general associativity properties in mathematical theory. It may seem a drawback of lambda calculus that a function abstraction and hence an application can only take one parameter at once. According to Lippe (2009, p. 25) a function which takes two parameters e.g. $f(a, b) = a + b$ is represented in lambda calculus as $h^* = \lambda a(\lambda b(a + b))$. The reasoning behind this approach goes back to the findings of Curry (1930) in the field of combinatory logic where he concluded that a function with n arguments can be attributed to n functions with one argument. This principle is today known as *currying* and its programmatic advantages will be discussed later in this thesis.

It is important to say at this point that the notion of free and bound variables in lambda calculus is crucial for any composed abstraction and application of lambda-expressions. In accordance to Harrison (1997, pp. 13) a *λ-expression* can be used in a primitive recursive function if one inductively proves it's free and bound variables. Lippe (2009, p. 28) shows the induction for $M \in A$, where $FV(M)$ is the quantity of free variables and $BV(M)$ is the quantitiy of bound variables by using combinatory logic. As one can see in this proof the variable x is free in M when $x \in FV(M)$, which means that it is only occuring within the function body and is not changeable from outside. On the other hand it is bound in M when $x \in BV(M)$. This includes any variable denoted with λ if it's also occuring in the function scope. So far, with these examination possibilities of *λ-expressions* one can define functions which apply to other functions. These abilities already sound familiar to a well-trained program-

mer as they are found in popular functional programming languages which will be shown at a later stage of this thesis.

2.1.2. Lambda-Conversion

With the *λ-expressions* defined before and their properties on abstractions and applications there is a need to define basic operations to be able to make transformations of combined expressions and to show equality of two certain expressions. For this purpose Church developed two basic axioms in Church (1941), which define the so-called *convertibility* of *λ-expressions*. They both have implemented a mechanism to substitute and simplify terms in lambda calculus (Barendregt, 1981, pp. 22):

- α**-conversion**, called "renaming": $\lambda x.M \iff \lambda y.[y \setminus x]M, y \notin FV(M)$
- β**-conversion**, called "application": $(\lambda x.M)N \iff [N \setminus x]M$

As explained in Turner (2007, p. 4) the substitution terms are read in α-conversion as "replace all bound variables x in an expression M with y and for β-conversion as "replace all bound variables x in expression M with N. It is important to note that in order to make the "rename" operation work properly, the new variable y must not occur as free variable in expression M. According to Hudak (1989, p. 364) these conversion rules together with the common equivalence relations namely reflexivity, symmetricity and transitivity form a consistent system in mathematical theory.

It is mentioned in Harrison (1997, p. 18) that proving equality of two *λ-expressions* may not be as critical from a computational perspective as transformations of complex *λ-expressions* into simpler terms. This can also be achieved by the above mentioned conversions by removing the equational symmetry in their application so that the transformation only occurs in one direction, left to right. These operations are called α-reduction and β-reduction respectively. If an *λ-expression* can not be reduced further by β-reductions which means that there is no contained term remaining to be simplified by this rule it is called to be in *normal form* (Peyton Jones, 1987, p. 24). Note that there are *λ-expressions* which have no *normal form* because their reduction leads to the original term.

In order to be able to use the reduction from a computational point of view one must define a final "state" where one would only have one distinct *normal form* for a particular *λ-expression*. Therefore Alonzo Church and his scholar John B. Rosser defined the *Church-Rosser Theorem* to proof uniqueness and existance of a *normal form* (Church & Rosser, 1936, p. 479):

1. When $M = N$, then there exists Q with $M \xrightarrow{*} Q$ and $N \xrightarrow{*} Q$.

2. If $M \in A$ with $M \xrightarrow{*} N$ and $M \xrightarrow{*} \overline{N}$, then there exists $Q \in A$ with $N \xrightarrow{*} Q$ and $\overline{N} \xrightarrow{*} Q$.

The first theorem states according to Hudak (1989, p. 365) that if there are two equal λ-*expressions*, there is one unique *normal form* where both can be reduced to with λ-conversion. In this case it does not matter in which order the expressions are reduced. The second theorem only adds, that reduced λ-*expressions* only differ in possible α-reductions (Harrison, 1997, p. 20). It is important to note at this point that there are two different ways of reduction if there are more reducable terms in an expression, namely *normal-order reduction* and *applicative-order reduction*. They differ from the order of evaluation of the contained terms. Turner (2007, p. 17) mentions those approaches in relation to the well known *non-strict evaluation* or *call-by-name* reduction and respectively *strict evaluation* or *call-by-value* reduction of expressions. It will be shown in section 2.4 how these approaches apply to real-world functional programming but also why and respectively when it is necessary to choose a dedicated reduction strategy in programming.

2.1.3. Lambda-Definability

It can clearly be seen from the previously mentioned elements of lambda calculus that there are still definitions missing which justify Church's formalism as a programming language in the end. One particular construct which occurs quite frequently in programming are boolean values. So how can boolean algebra be represented by λ-*expressions*? Following the definitions of Lippe (2009, p. 39) boolean values are described with the following λ-*expressions* which are basically two-digit functions:

$$true \equiv \lambda xy.x$$
$$false \equiv \lambda xy.y$$

Barendregt (1981, p. 133) gives a proof for these definitions, by applying them to an $if - then - else$ control structure which can be represented as a three-digit function $\lambda xyz.(xyz)BMN$. Due to applying β-reduction and inserting the boolean representations above, where B is the boolean value, the result will be the argument M if $B = \lambda xy.x$ and N respectively if $B = \lambda xy.y$. One can then also represent all logical operators known such as *or*, *and*, or *not* (see: Harrison, 1997, p. 28). As one can already see with this basic example there is no difference between data structures and control structures in lambda calculus.

It is interesting in that respect that also natural numbers can be expressed as λ-*expressions* and hence are functions in Church's formal system. Moreover, there is no other way to represent data in lambda calculus. These special λ-*expressions* are called *Church numerals* which display natural numbers as "descendant" of zero.

These are defined as follows according to Turner (2007, p. 5) for:

$$\overline{0} \equiv \lambda f.\lambda x.x$$
$$\overline{1} \equiv \lambda f.\lambda x.fx$$
$$\overline{2} \equiv \lambda f.\lambda f.f(fx)$$
$$\overline{3} \equiv \lambda f.\lambda f.(f(fx))$$
$$\ldots$$

So natural numbers in lambda calculus are nothing more than a function-applying function f which is repeatedly applied to an argument x. Based on this definition one can implement all arithmetics on natural numbers. In this thesis only one example is given on the basis of a function which returns the successor of a specific natural number represented in *λ-expressions* based on Barendregt (1981, p. 135):

$$successor \equiv \lambda n.\lambda f.\lambda x.f(nfx)$$

The following equation shows how one can find out the successor of a specific *Church numeral* by applying β-reduction on the expression, with the original number as argument. Note that the bound variables of the applied number (in this case 1) are changed to avoid confusion:

$$\begin{aligned} successor(1) &\equiv (\lambda n.\lambda f.\lambda x.f(nfx))(\lambda z.\lambda y.zy) \\ &\Rightarrow_\beta \lambda f.\lambda x.f((\lambda z.\lambda y.zy)fx) \\ &\Rightarrow_\beta \lambda f.\lambda x.f((\lambda y.fy)x) \\ &\Rightarrow_\beta \lambda f.\lambda x.f(fx) \equiv 2 \end{aligned}$$

One important issue for the forthcoming considerations in this thesis is still open: The representation of general recursive functions in lambda calculus, since they are not formally implemented in the syntax of *λ-expressions* according to Lippe (2009, p. 42). It is not possible to define a function which applies to itself (e.g. (AA)) because a function as already discovered, is not named and hence one cannot "refer" to it. In this case the function would have to be duplicated to be applied to the same function.

Nevertheless it is possible to write iterative procedures like recursions in lambda calculus by introducing the notion of *λ-definability* which is basically an assumption of previous claims of Curry and Gödel in the field of elementary number theory as presented in Church (1936, p. 346). Church states in his thesis that every function of natural numbers is "effectively calculable" if it is definable by *λ-expressions*. Moreover it can be rewritten completely without recursion through the introduction of a construct which will be later known as the *Y-combinator*. According to Harrison (1997, pp. 31) the *Y-combinator* defines a fixed point Y for a recursive function f as a *λ-expression* which can then be used to rewrite the recursion to some simple lambda-abstractions, so that one satifies the equation $f(Yf) = Yf$. It is important to note that the rewritten lambda-abstractions do not have free variables anymore and can be considered as a closed term. There is a practical application of the *Y-combinator* described in Paulson (1996, p. 18) with a basic implementation of a factorial function.

At a later stage in 1937, Turing proved in his thesis on *Computability and λ-definability* (Turing, 1937, pp. 160) that the approach taken by Church can also be applied to his well-known *Turing Machine*[2] concept. He basically operationalised the lambda calculus as an algorithm on his hypothetical computing machine and showed that his conclusions on the proof of computability of mathematical functions is equivalent to Church's. So what does that finally mean for the implementation of functional programming languages? Michaelson (1988, p. 9) gives a satisfactory answer: "Any of the mentioned computational models, can be translated from one to another, so any fundamental computer language in a broader sense can implement notions of the lambda-calculus as a programming language". However, as stated in the beginning of this chapter, the question if a programm will terminate cannot be answered with the mentioned theoretical approaches. These assumptions are now known as the "Church-Turing thesis" which today act as a reference point for the evaluation of computability in programming languages, even if one can only proof their compatability to Church's and Turing's assumptions. A computational model which satisfies these claims is said to be "Turing complete". Note that it will be of significant importance in further studies of this thesis that both control structures and data strucutures can be reduced to functions in a programming language.

2.2. Functional Style

Although lambda calculus is seen as a major cornerstone of computational theory it was not used in the beginning as a model for programming languages when the first computers were build. The reason behind that fact was that Turing's formalism was more "tangible" since it already described a type of machine which was also able to memorize and maintain states, as opposed to Church's approach. As with the design of the Turing machine, the first computer architecture developed by John von Neumann[3], consisted of a shared memory for data and programs and a central processing unit which was able to execute instructions from a memorised program (Piepmeyer, 2010, p. 4). Naturally, computer programs were constructed on top of this architecture as sequences of instructions which are processed one after another. This approach was very close to Turing's original concept. In the *Von Neumann architecture* both data and instructions were "reduced" down to the very same machine code and transferred to and from the central processing unit in order to be subsequently used in combined calculations. The results were put back to the memory at finalisation of the respective program steps. A computer program running on this architecture consists of a defined *fetch-execute cycle* according to Sebesta (2012, p. 19), and relies on saving interim results during program execution and hence needs intermediate variable assignments (and additional memory). Repetitions in a program

[2]The Turing Machine Concept: `http://mathworld.wolfram.com/TuringMachine.html`

[3]The von Neumann architecture of computer systems. See: `http://www.csupomona.edu/~hnriley/www/VonN.html`

code were realised as iterative loops utilizing subsequent memory cells rather than recursion, since it was more intuitive from the architectural point of view (Piepmeyer, 2010, p. 5).

This clearly contradicts the ideas of Church and the lambda calculus, where no algorithm in his formalism relied on temporary variables and self-applicable functions built up the basement of his computation model. A few scientists recognised this problem during the fast evolution of computer systems in the fifties of the 20^{th} century; the quick elaboration of this imperative programming style tremendously increased the traffic between memory and central processing unit and John Backus[4] named this effect two decades later the "Von Neumann bottleneck" as a general problem of conventional programming languages in Backus (1978, p. 615). The focus of software development and language design went in the direction of expressing how to do a certain task in terms of execution steps and machine states rather than building up foundations to abstract the hardware and provide possibilities to define algorithms more pragmatically with mathematical expressions (Harrison, 1997, p. 3).

In fact there were some practical counter-measures in the evolution of programming languages to overcome this close relation and strict dependency of software development to its underlying hardware architecture, especially with the introduction of the *List Processing Language (LISP)* by John McCarthy in 1958. LISP brought a completely new way of creating programs by introducing and interpreter in order to be able to define programs with semantics similar to the lambda calculus (Lippe, 2009, p. 108). The programming language merely consisted of atoms and lists as so-called *S-expressions* for both data and programs, where atoms defined immutable values and lists represented (data-) abstractions or (function-) applications, as described in Sebesta (2012, p. 677). The most important aspect of LISP was though that it could extend the lambda calculus in order to effectively write recursions without the *Y-combinator*, by assigning a name to the respective function (see: McCarthy, 1960, p. 12). This so-called *applicative style* of programming was the first step in many sucessive attempts to avoid states and local variables for the sake of computing efficiency by taking full advantage of Church's achievements with his calculus.

It is important to be said at this point that LISP was later extended with imperative features such as loops (Harrison, 1997, p. 24). Nevertheless, there were successful implementations of pure functional languages where all imperative constructs were intentionally omitted such as it is the case with Haskell. This group of languages provides a practical implementation of the lambda calculus with its different reduction strategies (Sabry, 1998, p. 13). The main idea of pure applicative or functional programming is to define a stand-alone program as a single function which is combined with other functions and where there is no semantical difference between data and program logic. (Piepmeyer, 2010, p. 6). As a result of this functional encapsulation

[4]John Backus, developer of the Fortran Programming language. See: `http://www.thocp.net/biographies/backus_john.htm`

there is one feature which can only be provided by this type of programming languages according to Hudak (1989, p. 362), the *referential tansparency*. The structure of pure functional programs does not allow different results for the same input depending on when the program is called. In functional languages there are no assignment statements, every variable in the program is immutable (Hughes, 1989, p. 2). This fact automatically removes side-effects which may occur in imperative languages where variables can be changed outside of the scope of a particular function or procedure.

One must note that in both imperative and applicative cases the intermediate programming language is still translated to machine code by its compiler complying with the *Von Neumann architecture* since the hardware didn't change dramatically in its conception over the years. It is mentioned in Backus (1978, p. 639) that due to the popular usage of imperative style in programming, there was also not much freedom in principle hardware design. Nevertheless, the discourage of mutable state and the focus on the evaluation and reduction of expressions (e.g. combinations of functions and immutable data) before execution makes it possible to write more efficiently structured and fail-proof code. The modular design of functional programs favours concurrency, pure functional expressions are thread-safe, and each component of a composed function can be checked for its correctness based on mathematical properties (Backus, 1978, p. 617). This functional approach has gained more momentum in recent years since most of the computers today have more CPUs available simultaneously. Compiler for functional languages can efficiently evaluate program code in order to create parallel running threads. The underlying hardware is capable of distributing them automatically at runtime and the "Von Neumann bottleneck" can be avoided more effectively than in the past attempts of functional programming.

2.3. Expressions and Types

In accordance to the definitions of expressions in lambda calculus, functions in functional programming are always treated as first-class values. This means that they have three distinct features: They can be assigned to (or encapsulated in) data structures, applied as arguments to functions or be returned from enclosing functions (Bird & Wadler, 1988, p. 4). These features form so-called *higher-order functions* which are treated completely the same as values and can also be defined as anonymous function abstractions similar to the syntax in lambda calculus (Wampler, 2011, p. 12). With these approaches every function can be used as an abstraction of one single parameter. In order to provide multi-parameter functions one can implement *currying*, which has also an advantage in providing a certain degree of modularity in program code according to Hudak (1989, p. 382).

An example of an *higher-order function* using *currying* in Haskell is given by Lipovača (2011, p. 60) as follows:

$$multThreeInt :: Int \rightarrow Int \rightarrow Int \rightarrow Int$$
$$multThreeInt\ a\ b\ c = a * b * c$$

The function $multThreeInt$ in principle takes one parameter from the type Int and returns a so-called "partially applied function" in the form $Int \rightarrow Int \rightarrow Int$, therefore enclosing a follow-up function. As one can see this combination mechanism for functions is very powerful and adds more abstraction to the program logic as opposed to imperative languages. Note that *currying* is a feature which can only be provided by functional languages since imperative procedures in general are not able to implement self-applicable functions (Petricek & Skeet, 2010, p. 140). A common built-in higher-order function in modern functional programming languages is map which applies a function to all elements of an appropriate data structure hence ommitting imperative loops. This function in the context of Haskell is described in Lipovača (2011, pp. 66) together with other practical examples using *currying*. The second combination mechanism frequently used in Haskell and similar languages is the *functional composition*. Consider two functions f and g where the output of f goes straight into g. In imperative programming languages one would usually save an intermediate result. In functional programming there is a special operator circumventing this memory allocation and synchronising the two functions. It is written in the form of $f.g$ which translates to $g(f\ input)$ according to Hughes (1989, pp. 8). Another elegant feature of functional programming languages in general is *pattern-matching* which removes the need of $if - then - else$ control structures. To round-up the basic capabilties of expressions *pattern-matching* is shown by a basic example of a recursive function in Haskell, taken from Thompson (2000, p. 60):

$$factFun :: Int \rightarrow Int$$
$$factFun\ 0 = 1$$
$$factFun\ n = n * factFun(n - 1)$$

As one can see here, the function $factFun$ matches a basic case where the input n is 0, otherwise the recursive version is called. This style of recursion can only work because there is a starting or more precisely also a termination condition provided with the first case. This construct represents a *conditional expression* according to Sebesta (2012, p. 708). This function should give a first glimpse on how programs are typically designed in functional programming by omitting most of the well-known imperative control structures. The so-called *equational reasoning* of functions is playing the foremost role in considerations about the design of data structures in modern functional languages according to Cousineau and Mauny (1998, p. 92).

2.3.1. Primitive Types

One distinguishing feature of functional programming, the specific treatment of primitive data types as they were already used in the previous examples with Int or function types with $\rightarrow$, were not directly inherited from lambda-calculus. There were some preliminary attempts in combinatory logic to add type features of set theory to lambda calculus as described in Harrison (1997, p- 38) but all data types mentioned here evolved independently in the beginning. What was in fact demanded in all programming languages, starting from the first implementations, was a mechanism to define collections of "similar typed" values in which the same functions are allowed to operate on. In computational theory there are generally two types of data, the primitive or basic data types and the derived or composite data types (Bird & Wadler, 1988, p. 7). According to Hudak (1989, p. 377) types in functional programming languages are first-class citizens which basicly means that types are treated the same as a value and every expression must have reference to a particular type. In general, functional as well as imperative programming languages frequently use the following primitive data types: $integers$, $booleans$ and $characters$. All these *primitive data types* characterize a set of particular values in every programming language but despite of being traditionally only bound to a value to avoid errors during program execution, a type can be also used and processed as independent element in functional programming languages (Pepper & Hofstedt, 2006, p. 111). Furthermore, also functions and patterns explicitly represent a particular composite type in functional programming languages according to Rabhi and Lapalme (1999, p. 18); the Haskell function $factFun$ for example is representing a function type from Int to Int and the parameters are implicitly matched against a simple pattern involving the integer value 0.

2.3.2. Polymorphism

The before mentioned higher-order function map also has, among similarily designed procedures in functional programming, another property which is justifing this separate handling of types and values: *polymorphism*. In case of map this means that it has implemented a *type variable* in it's defintion which provides the possibility to use this function on any type given as a parameter to the processed data structure at it's execution (Thompson, 2000, p. 87). It is important to mention at this point that in both primitive and composite data types there is only one reference to a distinct set of values with their predefined ranges having a particular minimum and maximum (Lippe, 2009, p. 59). Nevertheless these types can be further composed into *abstract data types* which do not exactly specify their mapping into sets and ranges of specific concrete data types by intention. Separate handling of types and values is of particular advantage here because abstract data types are able to hold different primitive or composite types with *polymorphism* depending on their parametrization which will be shown in section 2.5 in more detail.

It is also important to note that there are different systems existing which define how and when types must be provided for a particular value. The most important distinction lies in the way how a type can be assigned to a value. In static type systems every value is related to a particular type permanently which cannot be changed afterwards. In dynamic type systems the type can change during program execution (Piepmeyer, 2010, p. 21). Type systems can additionally be distinguished in whether they implicitly or explicitly provide type declarations. Haskell for example is statically typed, where one can provide a type for an expression, otherwise the type is detected implicitly at compile time (O'Sullivan, Goerzen, & Stewart, 2009, p. 19). This feature of programming languages is called *type inference* and it implies a clever type-system where every expression can be deduced to one particular type with respective rules comparable with the reduction strategies of lambda calculus (Bird & Wadler, 1988, p. 44). As one can clearly see here, static implicit type systems give a lot of work usually done by the programmer in the hand of the compiler which in the end completely removes type checks at runtime and prevents type errors automatically at compile time.

2.4. Evaluation Strategies

Besides of higher-order functions, functional programming provides another distinct feature which makes the designed software more structured and modular: *non-strict evaluation*. As already briefly mentioned in section 2.1.2 there are two basic evaluation strategies implemented in lambda calculus, the *normal-order reduction* and the *applicative-order reduction* which differ in the way and order how they evaluate expressions.

2.4.1. Call-By-Name

In order to understand these features in real-word programming one most look at the *functional composition* mechanism which was described before in section 2.3. It is a good example to show the motivation of non-strict evaluation; the practical realisation of normal-order reduction in programming languages. The function f which is applied to function g in $g(f\,input)$ is only evaluated here if the the input produced by f is actally used in g. This means that the evaluation is diverted as long as the results from f are not needed by g and are possibly completely dismissed at compile time if there is nothing "needed" from f by g (Hughes, 1989, p. 9). The computational advantage of this procedure is that in such a functional composition enables the compiler to automatically detect if functions are depending on each other and hence is able to decide on its own which parts of the program can be run in parallel. Hudak (1989, p. 383) mentions in this respect that normal-order reduction is inefficient if it is

naively implemented in real-world programming although it's considered optimal in lambda calculus because of its capability to handle recursions with the *Y-combinator*. There is an example given in this paper where one can clearly see the main problem: Repeated expressions are evaluated separately, there is no possibility in the reduction strategy to detect equal expressions so that the computation is made only once. The functions are non-strict in the sense that reduction is made on demand with *call-by-name* evaluation but all expressions are evaluated separately. In order to overcome this inefficient evaluation strategy an extension of *call-by-name* was introducted according to Paulson (1996, p. 11): *lazy evaluation* or *call-by-need*. Haskell, as a pure functional language implements *lazy evaluation* (Thompson, 2000, p. 337).

The difference between these two approaches using *non-strict* functions is that the extended version is building up the dependency plan and reduction order of expressions not as usual in a tree-like structure. As a consequence of the requirement to avoid the evaluation of expressions occuring more than once a graph structure is used instead. When the compiler is reducing expressions with *call-by-need* it only cross-links to equal expressions with pointers and the resulting shared graph nodes are only evaluated once (Pepper & Hofstedt, 2006, p. 34). It is implicitly mentioned in Paulson (1996, p. 41) that this evaluation strategy is applicable but not efficient in *strict functions* when bound variables are occuring inside the function abstractions. A graph reduction strategy would copy the whole function here without evaluating these variables before. Furthermore, the evaluation order of non-strict functions is always left-associative like in the example above which might not be ideal in imperative flavoured functional languages such as LISP. LISP is utilizing strict functions instead according to Knight (1990, p. 105).

2.4.2. Call-By-Value

In contrast to this variations of *call-by-name*, the applicative-order reduction strategy introduced with lambda calculus utilizes strict functions more efficiently because each argument of a function is evaluated right-associative before the actual function body (Petricek & Skeet, 2010, p. 303). This means that expressions are considered undefined as a whole and are not evaluated as long as their arguments are not fully defined and evaluated (Pepper & Hofstedt, 2006, p. 32). As one can imagine this approach is taken by most imperative programming languages because side-effects and mutable data are practically not allowing any other reduction strategy and therefore dependencies cannot be resolved automatically by the compiler. On the other hand, applicative-order reduction can not deal with *conditional expressions* and recursions as they are used in pure functional languages because evaluation would possibly not terminate according to Michaelson (1988, p. 56). One big drawback of the powerful *call-by-need* mechanism in pure functional programming is that one cannot influence programmatically when an evaluation takes places. This means that one has most probably to deal with unevaluated so-called *closures* in more complex programs; func-

tion abstractions which are not reduced and evaluated as long they are not applied to another function (Hudak, 1989, p. 385). A closure in LISP but also in Haskell can inhabit other function definitions which are either returned by or only used within the function scope. A lazy evaluation strategy would not be able to detect these nested functions and their dependencies so they will never be evaluated independently. It can clearly be seen here that the efficiency of *call-by-need* is far worse than *call-by-name* when applied to programming environments where there are side-effects and arguments must explicitly be evaluated before their application. Nevertheless, there are advantages utilizing *lazy evaluation* in pure functional programming languages when it comes to the point of data abstraction and representation

2.5. Data Structures

As already briefly mentioned in section 2.3, besides of primitive types there are also *composite types* in functional languages as well as they are existing in imperative languages. Usually these composite types form a so-called data structure according to (Lippe, 2009, p. 59) if they are defined over a specific combination of primitive data types. That means the main focus is on which data types are used but the semantics for processing the encapsulated data are basically provided by the consisting primitive types themselfes.

2.5.1. Tuples

The basic form of combination of two distinct primitive data types in functional programming is the *tuple* according to Bird and Wadler (1988, p. 32). From the mathematical point of view one can see this construct as a cartesian product of two types and there is only one operation predefined by the data type itself: a selector to access each element (Cousineau & Mauny, 1998, p. 20). All other operations and relations between the actually used primitive types are free in the sense that it is up to the implementation and hence depending on where they are processed, how to interprete them. Note that tuples have a fixed number of elements in their definition as well as they have a fixed type which is written in Haskell syntax as follows: $type\ item = (String, Int)$ (Thompson, 2000, p. 71). This definition also implies that a tuple as a composite type must implement *polymorphism* on it's parameters in Haskell in order to maintain the freedom of choice with all available data types. A typical use case of tuples as a return type of a function is called *tupling* which is illustrated in Rabhi and Lapalme (1999, p. 17) with a polynomial root function returning two integers.

2.5.2. Lists

A practical extension of the tuple is the *list* where one can have a variable length of items in an ordered sequence. This sequence has one particular type and therefore a list is considered as a *linear data structure* (Rabhi & Lapalme, 1999, p. 18). In contrast to the tuple this data structure is providing more operations on itself independently of the types provided by it's final declaration. It is mentioned in Thompson (2000, p. 92) that $type\ String = [Char]$ for example is the definition of string type in Haskell which means that a string is handled like a list of characters. The available operations include the polymorphic map function which was described in the context of function compositions in section 2.3 but also more simpler generic operations such as $reduce$ which i.e. can be used to implement $length$ returning the number of elements in a list. Another representative example of a higher-order function for lists is the $cons$ operator, a constructor function which adds one element to a list (Cousineau & Mauny, 1998, p. 60).

A typical pattern used with lists in functional programming is provided by the head- and tail-argument which is written in the form $x : xs$ in Haskell. This pattern can be found in all implementations of higher-order functions for lists since it enables the use of recursion when iterating over all elements according to O'Sullivan et al. (2009, p. 34). Practical examples of these functions as well as other widely used operations including $filter$ or $fold$ are described more detailed in Rabhi and Lapalme (1999, chap. 2.5). Nevertheless, a common filter possiblity on lists in Haskell must be mentioned explicitely here: *list comprehensions*. Lipovača (2011, p. 15) gives a simple, yet powerful example of this type of expression with $[x * 2 \mid x \text{ <- } [1..10], x * 2 >= 12]$. This statement yields a list as follows: $[12, 14, 16, 20]$. The beginning and ending brackets define that the result will be a list. The first part of the comprehension divided by the pipeline symbol denotes a simple function $x * 2$ and the intervall $[1..10]$ to whose values the function is applied to. The second part additionally provides a condition so only elements yielding a result greater than 12 are saved in the new list.

In contrast to strict evaluation languages such as LISP, lazy evaluation languages like Haskell are able to maintain a list similar to it's mathematical counterpart, the *sequence*. With this property a list can for example be declared as $[1..]$ which basically constructs a list of all natural numbers. The respective entries are then evaluated only on demand when they are actually accessed by the program (Bird & Wadler, 1988, p. 48). Laziness also provides the possibility of composing functions more efficiently on recursive data structures as with strict evaluation (Rabhi & Lapalme, 1999, p. 72). Since every data structure is immutable and persistant in functional languages every iteration made with general recursion is creating an intermediate result. This procedure usually consumes linear stack space with strict evaluation depending on the recursion depth, while loops in imperative programming languages would consume only constant stack space independently of the iteration steps (O'Sullivan et al., 2009, p. 87). When using lazy evaluation in conjunction with the previously defined recursion and the tail-argument, the intermediate result can be removed from

the stack space immediately after it is further processed in the data flow and the program can be run with using only constant stack space too. The exact behaviour of the evaluation in both cases is compared by example in Cousineau and Mauny (1998, pp. 83). It is important to note that in order to work properly in constant stack space, a so-called *tail-recursive* function must embed a helper function whose stack space is re-used in every iteration. An appropriate tail-recursive version of $factFun$ used in section 2.3 would look as follows in Haskell[5]:

$$
\begin{aligned}
&factFunTail :: Int \rightarrow Int \\
&factFunTail\ n = factFunAux\ n\ 1\ where \\
&factFunAux\ 0\ accu = accu \\
&factFunAux\ n\ accu = factFunAux(n-1)(n * acc)
\end{aligned}
$$

In this example $factFunAux$ is representing the helper function which is the last action executed by $factFunTail$. Furthermore $accu$ is a second parameter given to this function which is saving the intermediate results in every iteration. Again pattern matching is used to declare a terminating condition. These provided cases clearly show why it is important to take caution in operations on data structures with functional languages and their distinct capabilities. Okasaki (1999, p. 1) gives an advice in this regard in his introduction to purely functional data structures: "The inappropriate use of complex types in functional languages might influence the asymptotic behaviour of programs and lead to increased memory allocation which dissolve the gained advantages of modularity provided by their purely functional concepts".

2.5.3. Trees

As opposed to lists, *trees* are a hierachical, *non-linear* data structure because an element in the tree can have one or more successors. Generally consisting of nodes and branches connecting nodes with each other, a tree is also a recursive and polymorphic type by definition (Pepper & Hofstedt, 2006, p. 130). It can be build up starting at one distinct root element traversing through its nodes and sub-nodes down to leafs, which are nodes without further branches. A node itself can hold any other known data type in the used programming language and with it's sub-nodes it can be considered as a tree itself. The longest path from the root element to the outermost leaf is called *depth* of a tree according to Cousineau and Mauny (1998, p. 136). Trees have the same capabilities as lists with regard to application of composable higher-order functions since they can be deconstructed in practically unrelated paths. These can be processed in parallel and hence create intermediate results on purpose (Rabhi & Lapalme, 1999, p. 78). It's important to note that these intermediate results can be also avoided similarily as with lists. Again, iteration is done with a recursive helper function but this time this function is called twice in each recursive step creating a

[5]Based on instructions in: `http://scienceblogs.com/goodmath/2006/12/20/tail-recursion-iteration-in-ha-1/`

cascading *tree-recursion* instead of a flat *tail-recursion* as it was shown with the list type (Pepper & Hofstedt, 2006, pp. 41).

The very basic operations needed in *trees* are its constructor functions $node$ and $leaf$ which are composed further into patterns creating specific *tree* manipulation functions (Fokker, 1995, pp. 57). These manipulation functions are divided into single operations, such as add or $remove$ for specific nodes and global operations such as $depth$, $size$ or in more general $traverse$ which iterate through the whole tree (Rabhi & Lapalme, 1999, p. 39). Generally trees perform better in searching and sorting algorithms as lists if they are implemented as a so-called *search-tree*. A search-tree is usually represented by a abstract data structure - the *binary tree* - where each node only consists of two sub-nodes at maximum (Fokker, 1995, pp. 59). With this structure one can maintain a specific order of tree nodes, e.g. the left sub-tree contains smaller elements and the right sub-tree contains greater elements than the respective root node. This opens more possibilities with regard to the optimisation of asymptotic behaviour because algorithms can "focus" on a specific sub-tree, when searching for a specific value recursively. It only needs to "decide" at each node if the searched value is considered greater or smaller than the value related to the actual node (Okasaki, 1999, p. 12). However this type of tree can only be *polymorphic* with types who have an ordinal relation but they can be searched much faster because they are arithmetically balanced according to Bird and Wadler (1988, p. 236).

2.5.4. Arrays

Another frequently used composite data type in both imperative and functional programming is the *array* which is also an ordered sequence and linear like the list according to Rabhi and Lapalme (1999, p. 40). The only difference between these two data structures from the technical point of view is that array elements are explicitly indexed and have a known size at compile time. This has asymptotic advantages in their implemented access algorithms since it needs only constant time to access one element via the index (O'Sullivan et al., 2009, p. 271). Nevertheless in functional languages the array is handled different as it's imperative counterpart. The original design of an array is strictly tied to the *Von Neumann architecture* since the indexed elements are intended to be located in subsequent memory cells so that there is no linking between data cells needed (Pepper & Hofstedt, 2006, p. 287). The updates on the data structure are usually destructive and don't need additional memory cells but this is not possible with immutable data structures in pure functional languages; a copy of the array is created each time an element is updated, the data structure is said to be *persistent* or *multi-threaded* (Rabhi & Lapalme, 1999, p. 82). Although being memory inefficient, arrays can also be composed like lists, they have practically the same higher-order functions on sequences available such as map and they can be also lazy with regard to initialisation of particular elements (Pepper & Hofstedt, 2006, p. 26). Each cell of an array is accessible by it's fixed index independently if the

cell is actually evaluated and initialised with a value or not (O'Sullivan et al., 2009, p. 272). However, in order to achieve single-threaded arrays through functional language constructs there is a need for advanced higher-order functions which enable the creation of data structures where only one version exists at a specific time. It will be shown in section 2.6 that at least some complex data structures can be constructed solely by function applications.

2.5.5. Abstract Data Types

It is not explicitly mentioned throughout the literature but it turns out that the more operations are defined on a data structure and the less it is focused on the particular data types used the more it is considered as an *abstract data type*. This vague argument is nevertheless supported by Lippe (2009, p. 60) with the statemant that the distinction between a data structure and an abstract data type cannot be drawn easily. However with abstract data types one is focusing completely on the operations made within a predefined domain without being strictly tied to groups of primitive or composite types which can be applied to them (Bird & Wadler, 1988, p. 221). Rabhi and Lapalme (1999, p. 86) also states that abstract data types are basically definitions of algebras not visible to the user which can have several different concrete implementations. It is not surprising that abstract types can in fact be practically constructed by using the before mentioned concrete data structures such as *lists*, *trees* and *arrays* and therefore provide the same modularity as those. Furthermore abstract operations can be adapted to work in the native environments of the chosen concrete data structures. In Haskell for example, an abstract data type ist defined by a $module$ code block which defines different operations for the concrete types related to this abstract type according to Lipovača (2011, p. 105). It can be seen as some kind of data encapsulation since the $module$ only provides a single set of interfacing functions to the outside world but the concrete implementation is hidden.

All these complex data types can be efficient in algorithms with regard to their specific asymptotic behaviour if one keeps in mind which operations will be made on them. There are a lot of different applications existing especially with recursive types such as trees as it was already shown to some extend with the *search-tree*. Nevertheless, there is a even more general abstract type defined in functional programming languages named *set* which is having the same properties like it's mathematical counterpart: It is forming an unordered collection of elements occuring only once which all have the same data type. There are also a variety of applications existing with lists which do not allow duplicates according to Thompson (2000, p. 321) It completely depends on which algorithmic conditions one needs when choosing a concrete implementation but all types of *sets* contain higher-order functions like $union$, $intersect$ or $difference$ which are well known from set theory (Cousineau & Mauny, 1998, pp. 66).

With the abstract definition of sets there is the possibility to implement a *map* using the advantages of a concrete *binary-tree* data structure representation for example. In this specific case a key is mapped to one specific value but only the key needs to be unique. With a binary-tree there is the possibility to achieve sorted key-value pairs, if the key is from an ordinal data type. Nevertheless, there are simpler list- and array implementations existing not necessarily having any order which are illustrated in Rabhi and Lapalme (1999, pp. 98). A widely used example in programming is the *associative array* which is a list of key-value *tuples*. It is also stated in this book that it depends on the purpose of a *map* and it's requirements with regard to asymptotic behaviour in search algorithms which conrete data structure should be used. At last Pepper and Hofstedt (2006, p. 324) notes that a *map*, like an *array* can also be constructed as variations of functions whose fundamentals will be explained in the next section. More detailed descriptions of algorithms designed as abstract data types as well as more precise efficiency measures can be found in Cousineau and Mauny (1998, p. 157-230), Rabhi and Lapalme (1999, p. 114-154), Okasaki (1999, p. 171-184) and Pepper and Hofstedt (2006, p. 287-358).

2.6. Advanced Functional Features

So far this thesis has described features of pure functional programming languages which in total rely on two fundamental aspects: *immutability* of data structures and *non-strictness* of functions. There was already one example shown where these aspects are not efficient as their imperative counterparts with the modification of already defined data structures. Nevertheless it was at least possible to keep saving intermediate results at a minimum and as efficient as possible with *tail-recursion* but still there is the need of reallocating the program stack in each iteration. This is not desireable if compared to imperative, destructive updates but it is obviously needed if one keeps in mind the *referential transparency* of functional languages where the flow of data must be declared explicitly and no side-effects are existing (Rabhi & Lapalme, 1999, p. 202). Besides of that declarative style it is sometimes needed to maintain some global state in a program which can be changed by independent parts of the software. Moreover copying big data structures for the sake of providing modular and fail-safe programs might not be ideal in some applications as one can imagine. So how do pure functional languages maintain such imperative aspects as mutable data structures?

So far, one has only seen higher-order functions which are realizing *parameter-passing style*. This means that we have the intermediate result explicitly "carried around" in the recursive iteration (Sabry, 1998, p. 16). However, there are two more abstracted possibilities existing in pure functional programming according to Hudak (1989, p. 393) which build upon those functions as they were discovered in the previous section the so-called *effect-passing style* and the *continuation-passing style*. These

two possibilities were recognised as being equally powerful in their capabilities in the past and they were combined to one fundamental unit of pure functional programming which is called a *monad*. Monads are a mathematical construct coming from category theory which are defined in this domain on the basis of so-called *functors* (O'Sullivan et al., 2009, p. 354). A functor is nothing more than an abstract and polymorphic data type which represents a mapping from a certain *category* - a class of types it can be applied to, to another *category* - a class of types which is returned by this functor (Pepper & Hofstedt, 2006, pp. 218). Type classes in that respect can be seen as a group of similar types to which specific functions can be applied to. A built-in type class in Haskell which was implicitly used in previous sections is Ord which represents all types having an ordered structure according to Thompson (2000, p. 220).

The principle "mapping" capabilities of a *functor* already sound familiar if one looks back to the definition of the higher-order function map. In fact the functor, being a type class itself, inhabits a function $fmap$ according to Pepper and Hofstedt (2006, pp. 219) which generalizes the concept of the map function. The difference here is that $fmap$ does not focus on a specific type but provides the possibility to achieve the mapping with an implicit type constructor as it is shown by example in Lipovača (2011, p.146) where a concrete implementation for a specific type is completely ommited. In addition to that a functor usually provides a second operation in its definition which is enabling the combination of functor results. It is named $join$ in Haskell and can be compared to a "reduce" operation on lists (O'Sullivan et al., 2009, p. 354). Note that monads in Haskell, compared to their originating functors, have a slightly different naming of their contained operations but practically they are doing the same. However in this thesis it is more important to understand, that a vast amount of day-to-day operations in functional programming is implemented through these generic concepts which will be later proved in the practical chapters. Occasionally monads open the door to more capabilities which are not possible in the pure functional domain as described so far: the maintenance of state and the handling of I/O operations. The first paradigm which is realised by monads is the *effect passing style (EPS)*. As the name already indicates it leverages the possibilities of non-strict evaluation to create side-effects. This reduction strategy can be used to implement I/O streams as lists with pure functional languages which are acting as an practically infinite buffer. An extensive example using an abstract data type with already familiar *request/response* features forming an coupled *transaction* is provided in Hudak (1989, p. 395). This data type also incorporates pattern-matching cases to define whether a transaction was successful or not.

The second paradigm, *continuation passing style (CPS)* brings one back to the *functional arrays* which were indicated in the previous section. The idea here is to handle arrays just as an abstract data type in the form of a monad which can continuously call on its applied functions. This procedure omits the creation of multiple threads when modifing elements of the array since there is only one instance of the monad existing at a time (Wadler, 1995, pp. 15). To achieve this, a special monad integrates some kind

of internal state which is passed around among it's defined higher-order functions according to Rabhi and Lapalme (1999, p. 203). The basic continuation which was implemented with *tail-recursion* before, is captured within the monadic structure and only a dedicated beginning condition and the desired transformation rule is supplied by the user without any indication of a specific type[6]. The reason why this allows only single-threaded data structure lies in the way the compiler handles monads according to Pepper and Hofstedt (2006, p. 259). It is indicated here that the compiler recognizes monads and treats them as single entity similar to a single-threaded arrays in imperative languages. Moreover monads also allow *non-determinism* with both paradigms in the form of function applications which might not return a result at all like it was indicated in the case of *transaction*. Finally the monadic array also allows an "exception" condition to occur when e.g. map is used on an empty array. The notion of this monadic structure called $Maybe$ is given in detail in Lipovača (2011, p. 147).

There are a few more applications of monads in pure functional languages such as Haskell. Nevertheless in the frame of this thesis the mentioning of traditionally imperative features which are translated into a more abstract construct is limited to very basic cases on purpose. It is important to see with these examples that pure functional programming can inhabit all advantages of imperative languages making at least as powerful in the end together with their specific expressiveness. However, there is the need to think deeply on which algorithms are implemented with pure functional languages. The elegance of providing modular and fail-proof code can only have computational advantages when applied to the appropriate data structures. It was shown that lambda calculus although having powerful language formalisms, is not efficient when naively transformed to functional programming. Nonetheless there are some features described here originally coming from lambda calculus which will be found again in much more advanced programming languages as it will be shown in the upcoming chapters. Those features are enabling asynchronous messaging features and parallel programming mechanisms as well as purely functional data structures and even more useful higher-order functions. Especially the increasing need of parallelised software for multicore- and cloud computing is making functional aspects more and more popular in non-functional programming languages. In addition to that advanced type systems based on the definitions elaborated during the evolution of functional languages described so far will be able to process external data formats and interprete related *domain specific languages (DSLs)* more efficiently which is an indispensible feature in today's programming.

[6]Note, this is only the case in Haskell, since it is a strictly typed language with implicit naming.

3. Application to Object-oriented Programming

In functional programming languages as it was proved in the previous chapter there is the focus on dividing a problem into smaller parts: *functions* - which are self-contained and have a particular task assigned to. Every problem is seen as an algorithm, a combination of different independent functionalities which can be used in a variety of contexts. In parallel to the elaboration of functional and imperative programming languages there was an evolution of another software paradigm: *object-oriented programming*. It was identified as described in Wotawa and Bloem (2003, p. 116) that in order to overcome the fast increase of complexity in software there was the need of an organisational unit in programming which keeps things abstracted to a certain degree for the purpose of reuseability and modularity. Object-oriented programming languages basically deal with *objects* having particular relationships with each other. A definition of an object is called a *class* which can be instantiated as often as it's needed and therefore is representing a set of objects from the same type. At the beginning, object-based functionality was simply integrated in imperative programming languages as "rules" of handling particular types of data. According to Marick (2012, p. 33) eventually these rules where deeply hidden by the language itself - the compilers took more and more work away from the programmer.

A fundamental difference between functional and object-oriented programming is that in functional languages one focuses on the implementation of a generalised constructor function which makes a distinction of a specific data type applied to it by it's internal implementation. In contrast to that in object-oriented languages the intention is to have each object supplied with an own implementation of operations for a specific type. In the very first elaborations of object-oriented languages such as *Smalltalk* this resulted in syntax and semantics where everything is treated as object which are able to interact among each other with a specific messaging framework (Sebesta, 2012, p. 85). One will see the difference of object-oriented and functional programming at a later stage with the principle of inheritance and subtype polymorphism between related classes as opposed to higher-order combinator functions between related types together with parametric polymorphism in functional programming (Smaragdakis & McNamara, 2000, p. 2). Altogther it seems that object-oriented programmming is nothing more than a generalisation of abstract data types as they were described in previous sections according to Sebesta (2012, p.525). This assumption will be proved in the upcoming sections too.

3.1. The Scala Programming Language

Scala, whose name is derived from "scalable language", is a rather new concept of a programming language which was introduced first in the year 2001 by Martin Odersky and his team at the École polytechnique fédérale de Lausanne. Generally speaking, Scala can be seen as an elaboration of the object-oriented programming language Java since it is running on Java Virtual Machine (JVM) and is able to exploit and interoperate with all existing Java-APIs. It grew out of a prototype of a functional programming language already based on Java called Pizza. This was the first attempt to get the most out of the statically typed Java compiler and it's mature optimisation model for elaboration of functional features such as pattern matching or first-class functions. The achievements in this research project partially evolved into Java generics, the application of parametric polymorphism to classes (Pollak, 2009, p. 5).

The plans with Scala were slightly different. The goal was to develop a hybrid language based on a strong object-oriented background combined with a deep understanding of the advantages of functional programming. The result tackled the most significant parts of both worlds and they were made practically interchangeable as it will be shown in the upcoming sections. The main reason for this fusion is clearly described in (Wright, 2009): Syntactic tricks like the *continuation passing style* or *closures* were often needed when building especially robust and fail-proof object-oriented code but it was very difficult to apply this paradigms to the available languages at the time. Instead complex design patterns for object-oriented languages were elaborated to help the programmers construct the right architecture for their particular use cases.

While functional programming remained more an academic discipline during the decades there was nevertheless one successful application in the telecommunication industry with Erlang in the time, developed by Ericsson[1]. Occasionally one of its core concepts, a concurrent messaging framework based on so-called *actors*, can be found again in the extended features of the Scala programming language (Wampler & Payne, 2009, p. 16). The issues tackled with Erlang are of particular importance for the reasoning about the significance of establishing an object-functional language like Scala in todays programming. It is mentioned by Odersky, Spoon, and Venners (2011, p. 53) that the Java community in particular was searching for new possibilities to get away from threads which share mutable data and where synchronisation in concurrent programs is achieved by complicated locking mechanisms. With regard to scalability, Scala and it's functional and object-oriented foundations made it possible to built up libraries capable of elaborating asynchronous threading models as an alternative approach.

[1] Ericsson is a swedish telecommunication company, see also: `http://www.ericsson.com/`

One of the very basic concepts enabling these capabilities in Scala is that everything is represented as an object which can be treated the same like it's pure object-oriented relative. Even primitive types are objects. Furthermore functions are treated as first-class objects which define them as versatile as their pure functional representation. The strict hierachical type system of Scala is almost identically to Java, yet with some significant differences. Scala has implemented *type inference* which means that every object is referred to one specific type at compile time (Braun, 2010, p. 2). In addition to that advanced abstract data types and immutable data structures add important notions of functional programming in this regard. Although it seems that object-oriented and functional programming is difficult to combine in the domain of Java because of it's tight relation to imperative features, Scala is balancing and supporting both concepts extensively and is widely recognised as a successful attempt by the community according to Wampler and Payne (2009, p. 6). At this point it is assumed that the reader is already familiar with the basic concepts of object-oriented programming. All the traditional object-oriented and pure functional aspects will be only shown as an application within an object-functional programming language in the forthcoming sections.

3.1.1. Functions and Evaluation

According to the findings in chapter 2 evaluation strategies play a fundamental role in programming in general. The way the compiler chooses to evaluate functions can be of importance. The reduction steps which have to be undertaken in order to simplify an expression down to a concrete value are a critical measure in complex operations. Scala therefore implements both substitution models discovered in the course of functional programming: *call-by-name* and *call-by-value*. Scala uses *call-by-value* by default, the compiler immediately reduces an expression i.e. a simple function parameter down to its value as soon it's in encountered in the evaluation procedure. Therefore each similar expression is only evaluated once as already elaborated in section 2.4.

Listing 3.1: A simple function definition in Scala

```
1 def manipulate(val x: Double, val y: => Int): Double = ↩
    {..}
```

Nevertheless, *call-by-name* is considered as equally important in Scala as shown in Pollak (2009, p. 107) since it provides a useful on-demand evaluation. It is important to note though that there are cases where only call-by-name will terminate but call-by-value would fail. An explanatory case in Wampler and Payne (2009, p. 190) gives a proof on this assumption with a custom loop structure implemented as a function-applying function which would run infinitely with call-by-name evaluation. In the example above, imagine `y` is another function which is typically reduced immediately because it's another function parameter with call-by-value, independently if it's

needed or not. However, in this case the parameter declaration is written with an arrow which indicates the use of call-by-name. Whatever stands behind the parameter `y`, it is not evaluated until it's actually applied within the function body.

It can be also seen in this basic example of a function in Scala that parameters provide their specific type as a postfix separated by a colon as well as the return type of the function after the brackets. Altough function applications in Scala are evaluated similar to the β-reduction in lambda calculus there can be more than one parameter applied to the function. Those are reduced from left to right on their own and in course of further evaluation of the function body in the curly brackets then subsequently replaced by its arguments therein. On top of this rather imperative definition of a function there is the possibility of nested function definitions within a block delimited by curly brackets in Scala, enabling closures as well as a special lexical scope for variables and expressions (Loverdos & Syropoulos, 2010, p. 46). In addition to functions, value definitions themselfes can have the same evaluation properties. One can either define them with the `def` prefix as the function above which denotes it as *by-name* or with the `val` similar to the parameters as *by-value*. Naturally `def` evaluates only when the result is actually needed as opposed to `val` which is immediately evaluated at it's definition. In fact, the call-by-name strategy used by the value definition of `y` in the previous example could also be rewritten as `def y:Int`.

Note that value definitions prefixed with `val` are immutable and acting completely the same as member variables in Java declared as `final`. In order to be able to update a value after it's definition it must be prefixed by `var` instead. Braun (2010, p. 28) mentions in that respect that the immutable version of a value is perfectly fitting into the functional paradigm of Scala and is sufficient for most applications therein. Moreover the immutability of variables used in functions makes them practically thread-safe. Nevertheless a mutable value can be from advantage in a variety of common software development situations, in particular when thinking about performance in real-time applications with caching or buffering mechanisms.

It is not surprising that the Scala language exploits the principles of higher-order functions to a great extend. Each function is treated the same as in functional programming. Familiar constructs such as anonymous functions with function types as well as function-applying and function-returning functions are flavouring the syntax and semantics of Scala. An impressive example of *currying*, showing the potential of first-class functions within Scala, is shown in Odersky (2011, p. 21). Currying is used in this case to abstract functionality by introducing a common function which takes another individual function as parameter. Either an anonymous function type or a named function can be applied to it concretizing the combination algorithm. Loop imitating procedures such as the *tail-recursion* can also be defined similar to as it was shown with Haskell having the same properties with regard to stack space allocation. Moreover Scala, according to Piepmeyer (2010, p. 128) provides a possibility to make an annotation to a tail-recursive function with `@tailrec`. This syntactical feature tells the compiler to explicitly check whether the function is really tail-recursive

or not. As one must recall from the pure functional programming languages that functions as they are treated in Scala are essentially named abstractions which can be composed over higher-oder functions to create new abstractions. This feature is very powerful when thinking about reusability.

3.1.2. Abstractions and Classes

Scala provides an interesting new way to deal with data as opposed to pure functional programming because of it's object-orientation. Besides of functions which create and encapsulate primitive and complex data types there are more possibilities with the usage of classes. A class defines a new complex type and implicitly a respective constructor which is able to create objects of this particular type (Wotawa & Bloem, 2003, p. 118). As it is common in object-oriented programming languages one can create new instances of this class, called objects with the prefix `new`. Additionally as it is the case in Java, member variables as well as methods of a class can be accessed from outside by the infix operator `.` except members which are declared with the prefix annotation `private`. It is interesting to mention that the `.` operator is acting similar to the *functional composition* in Haskell described in section 2.3. Nevertheless the class construction in Scala is slightly different to it's underlying Java framework which makes it more powerful with regard to flexibility by providing different ways to actually create objects. Besides of the general auxiliary constructors denoted with `def this(...)={..}`, which are in this form also existing in Java with a slightly different syntax there are other possibilities in Scala to provide parameters at class construction. In the example below there is one distinct example with the class `Response`. A class in Scala can have constructor parameters which are part of the class' primary constructor besides of all statements written in the class' body (Horstmann, 2012, pp. 59).

In this case there are two parameters given at the class instantiation. One is initialising a member variable (`token`) the other is just providing an temporary parameter which can be only accessed inside the created object by default. Every member variable can be directly used within the class without the keyword `this` as opposed to Java like it is shown with `def toXML` in the example below.

Listing 3.2: Classes and subtyping in Scala

```
1 abstract class ResponseA {
    val token: String
3 }

5 trait ResponseT {
    val timeStamp: Date = new Date
7   def toXML: NodeSeq
  }
```

```
9
  class Response(val token: String, hash: String)
11   extends ResponseA with ResponseT {
     def this(val token: String) = this(token, md5(token))
13   def toXML = <time>{timeStamp.toString}</time>
  }
```

The three basic elements of the example also show the general class hierarchy as it is designed in Scala. Compared to Java one immediately sees a yet unkown annotation `trait` but misses the well-known annotation `interface`. In fact a trait is similar to an interface, it defines a type just like a class but compared to the interface a trait can have member variables (Raychaudhuri, 2013, p. 70). Unlike the interface members in traits can also be implemented not only declared with parameters and return types as it is shown in the example above.

An abstract class on the other hand can be used just like in Java; one can declare member variables and methods without concrete implementation and they are concretisised through subclasses with the `extends` annotation. In contrast to the Java counterpart abstract members or methods are implicitly defined as abstract by default but there can be a concrete implementation in the Scala version of an abstract class as it can be seen by a simple example in Subramaniam (2009, p. 97). Nevertheless a concrete implementation can be refactored in the subclass like in e.g. `Response` with the `override` prefix annotation. Altogether abstract classes provide yet another possibility to assign a group of similar objects to one *base class* or *supertype* which has similar advantages as *type polymorphism* as it was seen in pure functional programming with Haskell

In fact an abstract class is exactly what was referred to an *abstract type* in pure functional programming which can in Scala contain concrete definitions of functionality as opposed to the pure object-oriented version hence making them "more" functional. After these assumptions with regard to abstract classes and traits it is natural to ask the question where the significant difference lies between those types. In fact the only differentiation is that traits cannot have constructor parameters like classes but abstract classes are able to have them (Pollak, 2009, p. 18). It is also mentioned in Odersky et al. (2011, p. 275) that the keyword `super` used to access members of a super class is *statically bound* in abstract classes and *dynamically bound* in traits which means that the targeted member can change in implementation depending on the order of inheritance of multiple related traits. Scala has also some syntactical features which makes it more functional with regard to its notation. Any method with a parameter defined in a class can be used as an infix operator itself (Wampler & Payne, 2009, p. 168). The resulting syntax is rather similar to a function application in lambda calculus. The hypothetical method `def +(a: Response)` implemented in the class above for example could be used to implement an object-combinator for two objects e.g. `x` and `y` from the type `Response` which can be then exploited with the notation `x + y`, conceptually similar to numerical algebra.

While Java is based on a single inheritance model Scala with its traits is providing more flexibility with multiple inheritance with supporting the so-called *mixin* strategy (Braun, 2010, p. 84). In accordance with the example given in this section, functionality from traits can be "mixed" into classes in addition to `extend` with the `with` annotation. A single inheritance is always made with the `extend` annotation independently if the supertype is an abstract class or a trait but Scala classes can be extended by multiple `with` annotations, each providing concrete or abstract definitions from another trait. Note that traits can also extend other traits and abstract classes so there can be a complex hierarchy of traits in Scala each of them providing certain functionality to their subclass. In Scala every class is a so-called *reference type* and is inherited implicitly from the base type `scala.AnyRef` which in fact is a substitute of it's Java counterpart `java.lang.Object` (Suereth, 2012, p. 28). Similar to the abstract data types in pure functional programming this base type has implemented common functionality which are inherited to all subclasses, for example methods for cloning or proving equality of objects.

In addition to that all primitive types like `scala.Int` are inherited implicitly from the base type `scala.AnyVal`. Moreover in both cases these base types are again inherited from the supertype `scala.Any` according to Odersky et al. (2004, p. 3). As one can imagine this hierarchy has certain advantages when thinking about generic functionalities which are common for a group of types. For example the method `.toString` used before is defined in the base type `scala.Any` and concretised in its subclasses whether it's a primitive or an abstract data type. At the very bottom of the Scala class hierarchy resides the type `scala.Nothing` which is a subtype of every other type above. This special element of the hierarchy provides possibilities in Scala to signal abnormal termination e.g. as a generic return type or as an element type of empty collections as one will see in section 3.1.4 in detail. There is also a base type `scala.Null` in the inheritance model of Scala which denotes all reference types to a special type `Null` but unlike in Java it's not only a keyword but an object as everything else in Scala. One will see reasoning behind this particular class- and type hierarchies more clearly in the next section.

One element frequently used in object-oriented programming is still missing in this section about classes: *static* objects. They are not existing as such in Scala but there is a more "tangible" version implemented here which is in fact manually created quite often in other languages: *singletons*. Instead of defining a class with the `class` annotation one uses the annotation `object`. Those objects are values and have no constructor parameters since they cannot be instantiated more than once just like singletons. It is indicated in Odersky et al. (2011, p. 110) that singletons are far more flexible with regard to type polymorphism and overloading as their static counterpart. In fact a frequently design pattern used in Scala named *companion object* removes the need of static members. An *object* with the same name as a particular *class* can be used for example to create new instances of this class. An example is shown below which creates the previously defined class `Response`. As one can see the *companion object* acts

like a factory for its related class, but there can be done even more with this construct with pattern matching as one will see in the next section.

Listing 3.3: Companion objects in Scala

```
object Response {
  def apply(token: String, hash: String) =
    new Response(token, hash)
  def apply(token: String) =
    new Response(token, md5(hash))
}
// typical call to companion object
Response("12345", "c34df78")
```

Note that the example above also implicitly shows an interesting feature called *dynamic method dispatch* which is used here to distinguish between two methods with the same name but with a different amount of parameters. This procedure can be also done with different types as one will find out later. It will be proven that they are conceptually the same as typed calls to higher-order functions as they were elaborated in the previous chapter where substantial information for the code to be executed is supplied at runtime (Odersky, 2011, p. 36). So far, data abstractions were done with a new type of polymorphism which is solely coming from object-oriented programming: *subtyping* - in the form of a simple yet powerful inheritance model with classes. With this paradigm one can create effective data structures by abstracting the functionality in a hierachical fashion which has a fundamental impact in Scala on how its data structures are organised. This fact will be discussed in detail in section 3.1.4.

3.1.3. Types and Polymorphisms

As one has seen in the previous chapter there are two fundamental data types existing in Scala: *value* (or literal) data types and *reference* (or abstract) data types. Each single data type is part of a hierarchical structure referring to one single base type which inherits functionality of each superclass in its branch. But what if a particular group of types is sharing the same functionality whose relation is independent of the tree hierarchy? It makes perfectly sense to apply the same paradigm which was elaborated in pure functional programming here: *type parameters*.

Listing 3.4: Type parameters in Scala

```
trait ResponseT[T] {
  val timeStamp: Date = new Date
  def toXML: T
}
```

As one can see in the example above the already known trait `ResponseT` got a type parameter enclosed in square brackets named `T`. This identifier is further used in the definition `toXML` which opens the possibility to provide a concrete return type when the trait is actually extended by a class e.g. `class Response extends ResponseT[NodeSeq]`. It is also possible to omit the type parameter and let the compiler automatically check for the correct type via its *type inference* mechanism (Pollak, 2009, p. 100). The advantage of this kind of polymorphism lies in the fact that complex class hierarchies and redundancy across different branches can be ommited. It is important to note that evaluation is not touched by type parameters since they are substituted before reduction takes place. This procedure is called *type erasure* and has it's origins in Haskell but is also used in Java according to Loverdos and Syropoulos (2010, p. 147).

At this point one has to recall that everything in Scala is an object having one distinct type. While Haskell had a basic classification of groups of types according to Thompson (2000, pp. 225), Scala is more powerful in that respect because of its object-orientation. But what still remains open until now is how functions or primitive types are designed with objects. In fact one can use primitive types as they were defined in Java but one can also use the inbuilt primitives with `scala.Int`. The Scala primitives are organised in classes which are conceptually similar to the primitives defined over lambda expressions in Church's calculus. The advantage of an object-oriented approach to primitives is that one can use inheritance to elaborate subsets, in the case of `scala.Int` for example natural numbers which can inherit and extend functionality from its base class (Horstmann, 2012, p. 6). In addtion to that functions are static objects with an `apply` method similar to as it was shown with companion objects before . A function in Scala is designed to take one parameter type at once which is then transformed into a result type through the function body. Scala supports up to 22 parameters in a function since all possible functions definitions are predefined in the Scala API according to Braun (2010, p. 104). Nevertheless one can use *currying* to create functions with more parameters by chaining up function returning functions just like it was shown in section 2.3 with Haskell. This procedure requires a special return type, which is a function itself.

Listing 3.5: An anonymous function application in Scala

```
def power(x: Int, f: (Int, Int) => Int) = f(x,x)

power(5, (x,y) => x*y)  // possible application of power
```

Scala, as a object-functional language also handles anonymous functions besides of named functions enabling the same capabilities as it was shown in pure functional programming (Chiusano & Bjarnason, 2013, p. 24). The respective example above shows on one hand how a basic *function type* can be written in Scala with the variable `f` but also how an anonymous function is applied to another function as a parameter. Note that internally the call `f(x,x)` is expanded to `f.apply(x,x)`. The anonymous function gets an identifier within the definition and is represented as a static object

just like a named function (Odersky, 2013, p. 93). Furthermore methods can also be handled as function types as soon as they get applied as such as parameter to another method in the respective class.

It remains to be discussed why Scala was designed to provide both generic, type parameterised functions from functional programming as well as class subtyping from object-oriented programming. The first advantage is that one can use type parameters to restrict classes and definitions to a fraction of the class hierarchy with so-called *type bounds* according to Wampler and Payne (2009, p. 259). With type bounds one can, like it's shown in the example below, provide an upper bound to which the given type must apply to. In this case the type variable `T` must be a subtype of the type `ResponseT`. One can also restrict the type variable to be a supertype of `ResponseT` with e.g. `[T >: ResponseT]` and even mix those two bounds into an upper- and lower-bound of a type variable.

Listing 3.6: Type bounds in Scala

```
1 def processResponse[T <: ResponseT](response: T) = {..}
```

The second ability enabled by the interaction between parametric- and subtype- polymorphism is called *variance* (Odersky, 2011, p. 56). In order to understand this mechanism there is the need to take a data type which was not yet explained in detail in association with Scala: *lists*. Lists in Scala are typically immutable and may contain a variety of different types. Together with a special type parameter annotation one can project a subtype hierarchy onto this complex data structure at creation. Consider a type variable `A` of the object *list* which is denoted with a plus postfix in its definition e.g. `List[A+]`. This notation defines that the type `A` is *covariant* for this data structure. If one creates now a list with a concrete type like `List[parent]` and another list `List[child]` where `child` is a subtype of `parent`, then these resulting list data structures would have the same subtype relationship as its contained types. The opposite is represented with a minus postfix and is called *contravariant*. A contravariant type parameter states that every supertype of the given type handed over to the respective data structure is passing the same supertype relationship to it. The abilities of variance also open the possibility to generalize methods and functions besides of classes so that they can handle a distinct predefined group of types in parallel too (Raychaudhuri, 2013, p. 96). A type variable without any postfix, as it was already shown in previous examples, is called *invariant* which doesn't hand over any hierarchical information of the type applied to the data structure. According to Suereth (2012, p. 137) all mutable data structures in Scala a generally invariant since the read and write operations must be made type-safe.

So far one has seen impressive ways to decompose data structures through object-oriented mechanisms but in Scala there is yet another way to make decompositions which is originating from pure functional programming: *Pattern matching*. The object-oriented way of decomposition with traits and abstract classes has its drawbacks when it comes to the point of adding functionality in base classes; in most cases there

is the need to update functionality manually in all subclasses as mentioned by Emir, Odersky, and Williams (2007, p. 2). Of course Scala also provides possibilities for *type conversion* and *type-checking* in that respect with `asInstanceOf` and `isInstanceOf` but they are generally discouraged in this language according to Odersky et al. (2011, p. 321). Type conversion, whereever it is needed in Scala can be done e.g. with defining `implicit` methods or values, which are providing unique capabilities in their particular scope of creation. There is generally no need to stick to a class hierarchy when converting to other types according to Subramaniam (2009, p. 100). Consider a function `implicit def IntToString(i:Int) = i.toString`. Whenever an integer is applied to a value from type string in the definitions scope e.g. `val s:String = 25`, the integer is implicitly casted to a string. It will be shown later in section 3.1.4 that inbuilt Scala implicits are simplifying operations on generic collections.

Pattern matching is ubiquitous in Scala and hence is used in a variety of situatons; In the example below with `matchObject` it is providing a new perspective for type-checking of objects.

Listing 3.7: Pattern matching in Scala

```
case class A(value: Int)
case class B(value: Int)

def matchObject(value: Any) {
  value match {
    case a: A => println("class A")
    case b: B => println("class B")
    case _ => println("everything else")
  }
}

matchObject(A(1)) // prints class A
```

Before going into the details of this specific matching mechanism one must introduce a "light" version of classes in Scala annotated with the prefix `case`. *Case classes* are implicitly implementing a companion object with the `apply` method so that the annotation `new` can be ommited at instantiation (Schinz & Haller, 2011, p. 9). In the example above functional decomposition is shown by solving a common problem in programming: A decision statement from where it is to be "chosen" which kind of object is actually handled. The definition `matchObject` takes a parameter of type `Any` so any data type of Scala can be applied. The matching mechanism is similar to a `switch` statement; in this case a specific type is matched in the form of the previously defined case classes. Besides of that, pattern matching can be done on constructors from case classes e.g. `case A(_)` but also on constants, tuples and wildcard patterns denoted with underscore (Odersky, 2011, p. 48). As one can see the wildcard can be

used anywhere in the provided pattern to omit the specification of a particular type or literal.

Along with the research done within this thesis another interesting Scala type was discovered which plays an important role in pattern matching with objects: `Option`. This type can be applied to any other type in the Scala class hierarchy for example `Option[Any]`. The option type has two subtypes: `Some` and `None` according to Wampler and Payne (2009, p. 41) and is used for the same purposes as $Maybe$ in Haskell. This special type can be used to pass optional objects which can be matched against either the case `Some(A(_))` or `None` when applied to the previous example. Altogether the capabilities with regard to types in Scala seem to exploit the best of both functional and object-oriented programming enabling more flexible and fail-proof solutions. It is mentioned in Odersky et al. (2004, p. 14) that object-oriented decomposition should be used when more subclasses are implemented. If there is a flat class hierarchy encouraged where more methods are created pattern matching is the preferred way for decomposition in Scala.

3.1.4. Collections and Operations

In Scala one immediately encounters familiar data structures already introduced with Haskell in section 2.5. Again a very basic sequence originating from functional programming is used throughout application develoment in Scala: *tupels*. A simplified version of a tupel is also existing in the form of a *pair*, which contains two elements and is initialised as follows: `val couple = ("Florian", "Sabrina")`. In this case the value will be automatically recognised as a pair of strings. Of course one can use any type in a pair or tupel; one can even mix different types within this ordered sequence and create a relationship between those types. The most important use case of tupels, *tupling* is widely used in Scala just like in Haskell but they can be also used as a pattern here (Pollak, 2009, p. 66). For example one can define a pattern value `val (boy, girl) = couple`, so that `boy` is assigned to the first value in the `couple` pair and `girl` to the second. Of course there are also standard accessors for elements in tupels. One could also yield the second value of the previously mentioned pair with `couple._2`. Note that this is the only inbuilt operation available for tuples and pairs according to Piepmeyer (2010, p. 145).

As one can imagine the most fundamental data structure of functional programming, the *list* sequence is also vastly used in the design of Scala applications. In principle they are recursive and immutable just like in Haskell as opposed to imperative arrays, which are typically mutable and flat as described in Odersky (2011, p. 63). Arrays as already discovered are not part of a good coding style in functional programming since it is difficult to process them effectively with functional standards such as higher-order functions. Nevertheless, lists are homogenous and ordered too so they can only contain one distinct data type just like an array.

A typical list instantiation in Scala looks like this: `val herbs: List[String] = List("sage", "clover", "ginger")`. The applied type is explicitly noted here which is not necessary since the compiler is able to automatically infer the type parameter needed for this instantiated complex type. Lists are also recursive collections so there must be a base type which can be used to iteratively create lists out of it. This type is called `Nil` and represents an empty list (Chiusano & Bjarnason, 2013, p. 35). Together with the construction operation `::` which is pronounced as *cons* one can already create a list with `x::Nil` where `x` can be from any known Scala data type. This statement returns a list with exactly one element and is similar to the standard syntax for collections as parameterised instantiation e.g. `List(x)` (Braun, 2010, p. 171). If one looks back to the previously defined list of string one could also write `val herbs = "sage"::"clover"::"ginger"::Nil` according to Odersky (2011, p. 64). Note that the creation of this list is right-associative as it is in Haskell, so at first a list `"ginger"::Nil` is created which is then cons'ed with the next element on the left until every element is added to the list.

A common pattern which occurs when working with lists is `x::xs`, where `x` is representing the head and `xs` the tail of a list as it is the case in Haskell. In Scala there are also two attributes called `head` and `tail` which can be used e.g. to recursively take out one element, or to access the first element of the list like e.g. `herbs.head == "sage"` or `herbs.tail.head == "clover"` (Wampler & Payne, 2009, p. 66). The third inbuilt operation for lists in Scala is called `isEmpty` which can be used to test any list if it contains something or not. Note, these operations are from importance when dealing with recursions as it is shown in Odersky et al. (2011, pp. 348) since patterns are used throughout the iteration procedure to identify specific conditions. They are typically denoted with the construction operator; e.g. `x::Nil` can be used as pattern to detect lists with a single element. With these capabilities it is possible to signal termination of a recursion when the iteration is complete.

Besides of the general language elements needed for manually creating recursions Scala also provides inbuilt first-class methods for the list collection, like `length`, `last` or `init`. The latter for example extracts all elements of a list but the last one according to Loverdos and Syropoulos (2010, p. 66). The functional nature of Scala's list implementation is revealed when looking at it's specific access function: `herbs(2)` returns "ginger". Unlike Java's accessor with square brackets Scala's version is written as a function application. Interestingly most of the methods applied to objects can even be written like functional compositions in Haskell. In fact, here in Scala the instantiation of a list provides a companion object with an `apply` method so the access function is equivalent to `herbs.apply(2)` according to Pollak (2009, p.54). While accessing an item in a list is possible in constant time the search for a specific value with the `contains` infix or concatenation of lists with the `++` infix may take up to linear time with recursion depending on the length of the list like it was the case with Haskell. A reference implementation of the `++` operator is shown in the listing below. It represents the actual library function as it is implemented in Scala as a named definition `concat`. According to Odersky et al. (2011, p. 350) a set of basic

library functions are similarily constructed as this example by using the *cons* operator, pattern matching and *tail-recursion* optimisation.

Listing 3.8: Concatenation of Scala lists

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  xs match {
    case Nil => ys
    case z :: zs => z :: concat(zs, ys)
}
```

In the case above the accumulator list is `ys` which is returned by the definition when the iteration terminates. Compared to the Haskell example with the factorial function in section 2.5.2 this tail recursion is more readable and compact with the `case` statement. Furthermore, the list operations can be abstracted away from the contained type with a type parameterised declaration as it is shown above. In most cases it is not necessary to explicitly provide a type here because *type inference* can automatically detect return types. Sometimes Scala also declares type specific operations with *implicit* methods which are applied to the function automatically by the compiler. This opens the possibility to only provide generic sorting functionality for sequences which are concretizised by specific predefined `Ordering` types applied at compile time (Suereth, 2012, p. 205). This can be useful when dealing with *domain specific languages (DSLs)* where specific data formats with recurring semantical patterns are transformed to Scala objects. These features will be shown on the basis of real-world applications of Scala at a later stage.

The basic example before also shows that functions are first-class objects and higher-order functions are rounding up the toolbox available for list types. Practical implementations for modification, extraction and combination of list types are therefore type independent and tail-recursive as well (Raychaudhuri, 2013, p. 151). A typical use case of such a function is `map` or `filter` which can be applied to any collection in Scala. Both definitions usually apply a sequence of extractors and conditions which are forming an anonymous function. A simple mapping on the `herbs` list can be written as follows:

Listing 3.9: Mapping of Scala lists

```
herbs map (x => (herbs.indexOf(x), x))
   == List((0, sage), (1, clover), (2, ginger))
```

Each element is transformed from a single value to a tuple with an additional index extracted from the position of the processed element. This example may be very simple but it is obvious that one can modularize transformation procedures very efficiently ommiting temporary variables and manual iterators. The definition of `map` in the Scala library shows that it is generic with regard to its type and function parameter. Note that even methods can represent a function type as it is stated in Odersky

(2011, p. 70) which can also be applied to objects in general with `map`. It is also revealed in Piepmeyer (2010, pp. 198) that all function-combinating elements of a collection are also generalised over higher-order functions like `reduce` or `fold` which are even more abstract but also tail-recursive in their nature.

To round-up the most important sequences in Scala one must look at the *non-linear sequences* implemented in the standard library. Through the hierarchical nature of the Scala types non-linear sequences inherit functionality from a base class *seq* as defined in Odersky (2013, p. 140). This inheritance relationship is shared with the linear sequences such as lists so all capabilities already mentioned in this regard are available as well because they are provided by the same parent type. According to Odersky et al. (2004, p. 6) this is not the case with strings and arrays since these sequences are directly imported from Java. Nevertheless they are casted implicitly to sequences when applying functions such as `filter` as stated in this paper. A basic version of a non-linear sequence in Scala is the *vector* type. A vector is represented as a so-called *shallow tree*; the depth of the tree structure is kept at a minium (Braun, 2010, p. 173). As opposed to linear sequences this type of collection can access it's elements in constant time proportional to the depth of the tree. The only syntactical difference between lists and vectors is the *cons* operator which can be denoted either with `+:` to add an element at the leading position or with `:+` to add an element at the trailing position of a vector. Otherwise the instantiation of vectors is done the same way. A typical operation on sequences frequently used with vectors is `flatMap` which combines a function mapping on elements followed by a concatenation of the results.

Together with the *range* type vectors are subtypes of so-called *indexed* sequences which all have the advantage of search in constant time (Loverdos & Syropoulos, 2010, p. 50). A range is even a more simpler type as a list since it only defines a start, an end and a stepping number, e.g. `(N to 1 by -1)` or with an exclusive end `1 until N`. This syntactic features and its compact definition makes ranges also very fast as it is proven in Suereth (2012, pp. 192). Ranges can be used in a variety of situations in Scala for example in imperative iterations like `while` and `for`. But `for` can also be expressed very efficiently when dealing with sequences in general. A loop over the `herbs` list might look like this: `for(h <- herbs) yield (herbs.indexOf(h), h)`. This statement would create the same result as the previous example with `map`. The most important part of this loop is the *generator function* which extracts one element after the other from the list in this case but this could also be a range type for example as shown in Horstmann (2012, pp. 21). One could even use curly brackets to be able to add more generators and additional conditions in the `for`-expression. The reason behind having such imperative structures in Scala is that it is more convenient for a developer coming from imperative programming. In any case Scala rewrites for-loops into a respective `map`, `flatMap` or `filter` internally depending on which particular sequence it is applied on (Wampler & Payne, 2009, p. 62). The extendible nature of `for` allows one to create custom iterating mechanisms over more complex data structures. In order to work properly with such arbitary data

structures one must only define the above mentioned substitution-methods in the structure's declaration. Those will be called implicitly by the compiler when a non-standard type is encountered in a `for`-expression according to Odersky (2013, p. 91). As one can imagine such a language design opens a lot of possibilities when working with pure object-oriented data structures but also when handling external data formats resulting from database queries or XML definitions.

Besides of sequences, *sets* and *maps* are completing the collection library of Scala to support all common data structures originating from pure functional programming. As one already knows sets are unorderd but elements are occuring only once in this data structure. Instead of `head` and `tail` operations there is only one basic operation existing for sets: `contains` (Wampler & Payne, 2009, p. 174). All other higher-order functions known from sequences are also available for sets through this basic operation. It is important to note, that Scala provides a set of casting methods for each collection type to transform a particular instantiation into another. For example one could use `toSet` on a list to create a set with all duplicates removed. A more frequently needed type of an unordered set in Scala is the *map* which is constructed similar as in Haskell: A unique key is associated with a value building up a single element of the map data structure in the form of a pair.

An interesting feature of the Scala class hierarchy is that sets as well as sequences inherit from the base class *iterable*. It can be seen from examples in Loverdos and Syropoulos (2010, p. 381) that there is a significant amount of operations shared. For example, maps have an `apply` method for accessing elements via their unique key so they are also built up with companion objects and operated on with function applications. Besides of that maps provide a `get` operation which also accesses an element via it's key. This operation returns an `Option` type instead which is more convenient for further processing and decomposition as it is clearly shown in Horstmann (2012, p. 44). To handle basic exceptional cases when working with maps one can also define a default value directly at instantiation. Other sequences like lists can be transformed directly to maps with the `groupBy` operation for example which returns a map of lists bundled by provided conditions. These conditions can be built up by function applications or through pattern matching for example. The obvious similarity of Scala semantics compared to operations found in DSLs is intentional by it's principle language design; the declarative style of programming in Scala is linked to well-known database languages and data exchange formats to aim at an optimal integration of common frameworks in this versatile language. The degree of flexibility with abstract operations on custom data structures will be shown by the examination of real-world applications of the Scala language in chapter 4.

The mentioned variations of collections and operations in this section also show that one must be very cautious when selecting a data structure for a specific use case. If chosen properly one can drastically reduce the length of code and create modular, abstract operations which can be re-used in other projects or bundled in custom Scala libraries. When compared to the original concepts in Haskell and pure functional

programming in general, the object-oriented fashion of Scala is adding a lot of simplification with regard to types and polymorphism with more elaborated type-checking mechanisms and common vocabulars for processing of data structures. For practical applications within the IMPEx project it will be necessary to evaluate if Scala is able to handle IMPEx internal data formats suchs as the XML based *IMPEx data model*) efficiently. Furthermore it will be necessary to measure how well Scala can access web services and serve data to HTTP clients.

3.2. Functional JavaScript

In the last section of this chapter the author wants to draw attention to a language which grew historical through the evolution of the *World Wide Web* to a powerful language for client-side manipulation of data originating from common HTTP web servers, namely *JavaScript*. The reason for that is on one hand that it will naturally be used in course of development of client-side applications in IMPEx but on the other hand also because it is enriched with functional aspects already known to a careful reader of this thesis. On the basement of the latter assumption resides that typical JavaScript code is designed and executed almost solely through functions. JavaScript functionality is embedded into HTML code with event-triggers which are basically representing functions calling other functions. Modularity of code plays a big role in the development of complex JavaScript applications.

Today, JavaScript is classified as a so-called *prototype-oriented* language which can be seen as a subtype of general object-oriented languages according to Stefanov (2008, p. 149). A prototype as such is a kind of function encapsulating properties and methods with a specific lexical scope which can be accessed through one variable from outside. As such it is representing a closed unit of functionality which can be instantiated and re-used. According to Haverbeke (2011, p 96) every object in fact is a prototype in JavaScript or can at least be re-used and modified as a prototype with it's originally provided properties. Every function in JavaScript automatically has a `prototype` property assigned which can be seen as some kind of constructor pointing back to the respective functional definition. As one can see in the example below properties and methods of a prototype-function are denoted with `this` and private properties of the prototype are written just like local function variables. Access of properties within a prototype is maintained with the `this` annotation the only exception are private properties. As it is shown in the second paragraph of the example, functionality can be added (or modified) through the `prototype` keyword which extends the original constructor of the applied class. The constructor is finally called with the `new` prefix as it is usual in object-oriented programming languages which in this case creates a new instance of `Response`.

With this basic concept it is fairly simple to chain and combine functions in JavaScript and it already provides some sort of inheritance because one can use existing prototypes, modify them from outside and treat them as new, seperated prototypes.

Listing 3.10: A prototype definition in JavaScript

```
function Response(token) {
  var timeStamp = new Date(); // a private property
  this.token: token;
  this.hash: md5(this.token);
  this.getTimeXML: function() {
      document.write("<time>"+ timeStamp + "</time>");
  };
}

Response.prototype.printMsg = function() {
  document.write("Token: ", this.token,
     " Hash: ", this.hash);
};

var httpResponse = new Response("1234af56e");
httpResponse.getTimeXML() // typical property access
httpResponse.printMsg() // new method access
```

Besides of the notation above one could also add functionality over the `prototype` property to an already instantiated object like `httpResponse`. In that respect JavaScript is, as well as Scala, a multi-paradigm language. But JavaScript is considered as loose with regard to purity of functional aspects while Scala is rather strict if one looks at it's type system for example. As one can see JavaScript has no types declared for function parameters and local variables; in fact no type is ever provided at declaration since JavaScript is dynamically typed (Stefanov, 2008, p. 98).

Nevertheless according to Resig and Bibeault (2012, p. 32) all functions in JavaScript are first-class objects just like in Scala, so the basic support for pure functional programming as discussed so far is here. JavaScript has also an familiar way to define scopes over functions; a function is basically only able to access data defined within its body. But in addition to that similar to functional programming encapsulated functions can access local variables of their calling objects dynamically with the `this` annotation (Fogus, 2013, p. 55). Independently from where a function is executed it will check it's current scope when working with the `this` property. Another aspect which makes Javascipt more functional is the popular usage of anonymous functions which can be found frequently in so-called *callback functions*. These functions usually take a function as parameter which is called somewhere in the body afterwards. But, as discovered in Scala too, one can use e.g. `function(test){ return "Test:"+ test;}` without assigning it to a variable and directly apply the statement to another

function. This is one feature where JavaScript does mimic pure functional languages with higher-order functions to a certain extend.

Due to the fact that everything is an object in JavaScript, functions can naturally be applied and returned by other functions. Haverbeke (2011, pp. 75) shows some practical examples with the already known `map` and `reduce` functions which can be easily translated to JavaScript code. It is important to note that instead of recursions, `for`-loops are used for iteration here because recursive structures are not optimised by JavaScript internally and could easily create stack overflows in web browsers according to Ullman (2012, p. 262). Moreover, recursions aren't really useful and necessary from the language design's perspective because it would create a lot of intermediate data structures but everything is mutable in JavaScript by default anyway. It seems that the mindset followed by JavaScript with combining functional and object-oriented programming paradigms has its differences when compared to Scala's approach because it is an interpreter language. JavaScript doesn't apply more complex capabilities from object-oriented or functional programming such as subtyping and generic functions because it is dynamically (and implicitly) typed and there is no way to specify types prior to execution. So in fact, there is no possiblity to program type-safe which can and does lead to unpredicatable code. JavaScript has a complex internal type casting system which can not be influenced from outside. Whenever JavaScript is comparing two different types it tries to convert one type into another automatically. The only safety measure here is that one can use the `typeof` expression to check types of a variable during runtime (Stefanov, 2008, p. 27).

Yet another aspect of Scala can be found in almost any program written with JavaScript: *closures*. They especially make use of the functional scope explained before by defining functions within other functions which are then only accessible locally. In addition to that, closures expose functionality to the scope outside by returning functions which are usually modified by their applied arguments. As a result, JavaScript often uses so-called *function factories* which are conceptually quite similar to companion objects in Scala as discovered by the author. Moreover with the prototype property, closures can be used effectively as generator functions providing a set of methods over infix operations which are modified depending on the prototype's arguments. All functions in JavaScript can dynamically add arguments with an inbuilt `apply` method which adds another possiblity for functional composition called *partial application*, typically used together with closures (Haverbeke, 2011, p. 91).

As already indicated, some elements of functional programming are completely left out in the original concept of JavaScript. Besides of tail-recursion, these elements are immutability, monadic data types and pattern matching, to name the most important covered in this thesis as discovered by the author. Fortunately a lot of extension libraries were elaborated during the years where JavaScript gained more and more popularity in web development. For example *underscore.js* provides recursive built-in functions such as `map` which are also tail-optimised to avoid stack overflows. This capability also implicitly requires and makes use of immutable data structures so local

variables can completely be left out as opposed to traditional iteration mechanisms such as the previously mentioned `for`-loops. The implementation details are further described in Fogus (2013, p. 154-158). On the other hand there are libraries which enhance JavaScript's object-oriented nature and glues it more to the functional paradigms found in it's standard concepts. One interesting example is *TypeScript* which is a meta-languague compiling to plain JavaScript. TypeScript adds real classes, interfaces and a complex inheritance model as it was seen in Scala with *mixins*. In addition to that, one can define generic functions with type parameters and a static typesystem can be used which is also enriched with a *type inference* mechanism just like it is given by Scala. Maharry (2013) provides a detailed introduction to this extension of JavaScript and gives a good overview about the motivation with regard to complex web applications.

In this thesis it will be of great importance when working with data formats served by the IMPEx web services, to process requested data sets appropriately on the server and delegate them efficiently to the client. It will be necessary to identify a common exchange format as well as to define data access mechanisms for this specific format on the server as well as on the client. Both mechanisms must be extendable and should be elaborated exploiting the same software paradigms to simplify the architectural design. With the previously mentioned aspects there is already a perfect foundation given to be able to write conceptually similar software on the server and on the client. The extensive capabilities of Scala and it's similarities to JavaScript with it's loose set of functional and object-oriented aspects enriched with extension libaries provide a powerful basement for further, more specific evaluation with regard to technical requirements of the IMPEx portal. The most important fields will cover parsing of external data formats and handling of DSLs where functional programming has distinct abilities for developing user defined libraries. Furthermore it will be needed to look into the implementation of asynchronous communication mechanisms in Scala to enforce scalability through parallel execution of particular tasks. Here, it will be needed to refer back to the originating concepts of functional programming to identify the most suitable data structures and related algorithms for manipulation. Finally it will be necessary to see if Scala is advanced enough to be able to handle all capabilities of the *Web Service Architecture Stack* whose principles were already discussed in Topf (2012b, p. 12-26).

4. Evaluation of Object-functional Capabilities

So far, one has seen a selection of object-functional aspects which seem at the first glance rather general and abstracted in their purpose. In order to be able to evaluate these basic capabilities for their usefulness in development of a highly distributed, domain-specific and service-oriented web-based framework as it is needed for the IMPEx portal one has to look beyond these theoretical definitions and seek respective indications in real-world applications. Most of the basic and fundamental technical requirements needed for successful implementation of the IMPEx portal are already known to the author because of previous studies in Topf (2012b) and Topf (2012a). These studies aimed at the evaluation of developments in scientific web-based applications as they are created in course of the EuroPlaNet and IMPEx projects. A clear path was shown in these papers where the elaboration of an IMPEx portal is mainly set to go. For now only two major aspects will be studied in this chapter since they are needed despite of specific user- and technical requirements proposed by the IMPEx community which will be elaborated afterwards. At first, a parser for XML documents will be required which is able to process the contained information in a way which makes them at most useful for the user and specific tasks but also for other web services using the capabilities of the IMPEx portal. It will be of particular importance here to transform and save the extracted metadata in a Scala native data structure so that the information is kept type-safe and can easily be reused for other functionality made available within the portal architecture. Since all XML descriptions served by the IMPEx infrastructure are also dynamic in their amount of contained resource entries and respectively in their size there must be a well elaborated mechanism at the IMPEx portal to serve the content of the XML documents in a homogenous and efficient way. The validation of the XML structure will be also of significant importance here so there must be also a procedure to interprete *XML Schemas* with a respective *domain-specific language* made available through Scala features as one will see in the upcoming sections.

In addition to that, the server architecture operating the IMPEx portal software must be able to handle requests asynchronously and in parallel because their execution might be time-consuming or relying on web services which are distributed across the IMPEx infrastructure. Since IMPEx is currently also using a predefined set of web services deployed over various WSDL descriptions it will be necessary to provide generic interfaces to these capabilities via customised Scala libraries. Those in-

terfaces will be able to exploit functionality accessible from different providers distributed over the network which may update and extend the provided services independently. The portal architecture must be flexible enough to extend all its accessible capabilities dynamically upon changes coming from the service providers. One will see in the upcoming sections that Scala is able to handle all IMPEx definitions empowering a service-oriented architecture. Moreover it will be verified that Scala is credible as a modern functional language which is fostering the creation of custom "sub-languages" providing common semantics for specific dedicated purposes as it is needed for the IMPEx portal.

In any case a choice must be made on the used server technology which is basically able to provide web services based on Scala and can be easily used as a messaging middleware for all included IMPEx functionalities. The mentioned backend features needed for the IMPEx portal must be provided in an appropriate way so that they can be exploited by any common frontend implementation technique. The chosen middleware to IMPEx services must be flexible enough to serve its capabilities to either remote or local clients by using standard web protocols. Various discussions on *stackoverflow*[1] indicate the growing usage of Scala in web-based enterprise applications for specific purposes where scalability and fail-proofness play a significant role. Note that *stackoverflow* is a moderated Q&A forum for aspects of software development and is not considered as quotable here. Nevertheless, the search for respective discussions have revealed a listing of various enterprises such as Twitter[2] who admittedly use Scala for their infrastructure. It is discussed in Venners (2009) that Twitter uses Scala for message queueing systems between their server daemons and frontends. The decision within IMPEx to use Scala as programming language was primarily influenced on the basis of these and other open discussions on the web whose claims will be proved in course of project-specific evaluation studies in the upcoming sections.

4.1. Domain Specific Languages and Parsers

As already indicated, a significant portion of the IMPEx infrastructure is handling and processing information stored in structured documents which are served in XML format. The primarily used XML format is describing resources accessible through the IMPEx service providers. The provided XML trees are complying with the definitions of a custom *XML Schema* which is built upon the well-known SPASE standard, already studied and evaluated in Topf (2012b, pp. 41). The resulting *IMPEx data model* is naturally designed for a specific problem domain which was identified in course of elaboration of the user requirements for the IMPEx infrastructure. It's specific purpose within this project is to describe the participating simulation models, archived

[1] Stackoverflow, moderated Q&A forum for programmers, see: `http://stackoverflow.com/`
[2] Twitter, see: `https://twitter.com/`

simulation runs and respective datasets so that the databases available through the service providers are easily exploitable through their common metadata. In particular the models cover two fundamentally different plasma and magnetospheric simulation environments, namely the HMM and PMM models which are described further in Topf (2012a, pp. 30).

Of significant interest in this thesis is the fact that XML is used here to create an own markup language for describing how data is organised in scientific databases available via their dedicated IMPEx web services. This markup language provides the necessary information for the access layer to databases so that clients can identify useful datasets through the available structured documents before calling the respective web service access methods. The implementation details of the database structure are hidden by the provided standardised metadata. The metadata is primarily used to construct the SOAP messages needed for accessing the underlying datasets through the before mentioned web services but they will be also used to enrich the user interface of the IMPEx portal with useful information describing the hierarchical content of the XML trees. For that reason every single resource stored within the XML structure is e.g. indexed with and addressable through an XPath variable stored within a special XML element `ResourceID`, as it will be shown later by example. It is only important to note here that this used index represents also a hierarchical structure within the XML structure which will have an impact on the architectural design of search and filter algorithms for the portal.

In any case, the foreseen Scala library making use of this XML data model needs to provide standard procedures to iterate through the given metadata trees. The planned functionalities must be made composable so that they can be combined for specific purposes. Moreover there must be the possibility to elaborate a generic and versatile parsing mechanism which is tied to the *XML Schema* and is easily extendable when the *XML data model* is updated. The main idea here is to identify reccuring query patterns on specific XML elements within the trees which are then generalised and abstracted into respective access methods available through the portal. As one will see later there is a variety of use cases where basic queries can be chained to form one unit in the planned interface to IMPEx services and data trees. The result should in fact form a so-called *internal domain specific language* expressed by means of Scala syntax and semantics which provides a dedicated vocabulary enabling the efficient exploitation of information stored within the XML documents for consequent web service access. According to Meredith (2012, p. 26) especially strictly typed functional languages such as Scala are predestined for providing an "abstract language syntax" for working in a specific domain because it is rather easy to identify groups of related types in domain-specific models in such languages.

Domain-specific languages, according to Ghosh (2011, p. 4) have one significant feature: They usually provide a dedicated language with specific semantics which is able to map from artifacts of a problem domain into artifacts of a solution domain. The most important aspect relevant for this thesis is that such languages usually provide

a domain-specific model which enables a common terminology between domain experts and software developers but also for potential users. In fact the artifacts created through domain-specific languages provide some kind of glue between the problem- and solution domain. It gives the user an abstracted way to communicate and translate problems to be understood by the solution domain without having particular knowledge of their internally used data structures. In the perspective of IMPEx an example could involve a specific `ResourceID` stored in the tree which is tied to a set of operations provided through a solution domain artifact. In fact all elements of the trees can be represented appropriately within Scala's object-oriented nature and have a dedicated set of methods available in their scope. Every element has a specific static type so that every operation made on elements can be considered as type-safe and fail-proof respectively. This limited expressivity as pointed out in Ghosh (2011, p. 13) is making the foreseen access interface to IMPEx services and methods *domain-specific*.

It is already defined partly through the service descriptions of IMPEx data providers which message calls will be part of this *internal domain-specific language* and their concrete definition will be part of the user requirements. It will be of importance in any possible situation that those message calls are self-describing and *request/response* formats are seamlessly fitting into the specific purpose within the IMPEx infrastructure. A certain flexibility must be given to implement different input formats for example. Note that at this point the *internal domain-specific language* is not directly made available for subsequent exploitation of functionalities to the portal client so the Scala solution must be further embedded within a common web-based interface. It is technically independent of the actual implementation of access mechanisms elaborated in the context of Scala. This external interface will be the only point where client requests are accepted, handled and processed. It will wrap the provided domain-specific library for acessing IMPEx services and resources in an additional protocol layer which will be defined later in the architectural design. This layer will be build upon the HTTP protocol and the capabilities available through this standard should be sufficient for providing a homogenous access layer to the user.

What is fundamentally needed for the specific *internal domain-specific language* planned here are efficient parsers for XML documents and appropriate abstractions of their content within Scala. The latter requirement makes Scala a particularly good candidate since the language is providing optimal support for creating custom data types as it was shown in section 3.1.3. Furthermore, implicit conversions are helpful when designing domain-specific libraries where custom data types are inherent and also frequently used. The implemented data structues in Scala must naturally be interpreted and translated automatically in various situations to comply with the used internal *IMPEx data model* at any time. The abstraction provided through the data model must therefore be directly transformed into appropriate Scala objects. The contained metadata can be then exploited just as if they were stored within XML structured documents and extracted with respective XML specific query languages.

With regard to general parsing capabilities Scala as described in Odersky et al. (2011, p. 759) provides so-called *parser combinators* which can be composed into custom parsing mechanisms for external data formats coming from any source outside of Scala's native language environment. There is a dedicated data type `Parser` implemented where one can define e.g. regular expressions for processing needed custom data formats meaningfully. A few basic examples of standard combinators provided by the respective Scala library are shown in Braun (2010, p. 180) together with illustrative definitions of self-defined parsers. Note that it is also emphasised in Wampler and Payne (2009, p. 230) that parser combinators in Scala provide a good alternative to well-known parser generators, which create classes and objects out of parser definitions. Those code generators are usually bulky and not easy to debug according to Chiusano and Bjarnason (2013, p. 155) and it's mentioned here that in most of the common use cases parser combinators are sufficient for handling external data formats. It is yet to be evaluated, whether it is necessary to implement custom parser libraries for external formats used in IMPEx or not.

For now, preliminary studies of the capabilities within Scala have shown that XML strings are first-class objects in Scala. Literals are automatically recognised as XML data by Scala's type inference mechanism independently if they are read from the filesystem or constructed programmatically. This makes a lot of sense since XML is vastly used as cross-language communication mechanism and nearly every higher programming language is able to process this format natively. The syntax of Scala is also helpful here because the usage of infix operators and function applications without brackets are providing good possiblities to write libraries with vocabularies similar to data query languages as they are known from e.g. database implementations and XML standards. According to Odersky et al. (2011, p. 655) Scala has essential functionality made available through it's standard libraries for processing strucutured data which is e.g. serialised and sent over the network. It is shown clearly here that Scala has particular advantages as a host language when a standardised communication framework based on XML is playing a fundamental role within a service-oriented architecture. XML strings in Scala are represented through one of the following classes or data types, which are naturally sub-types of each other: `NodeSeq`, `Node`, `Elem` or `Text`. For developers already familiar with XML definitions, these type annotations are extremely helpful.

As one can see in the example below XML node sequences can easily be constructed from scratch and be parsed by using an syntax similar to XPath. The double backslash is signalling a relative path search, whereas a single backslash would represent an absolute path search. XML attributes, when used, can be extracted with the `@` prefix followed by the attributes name. Text nodes, as shown in the example are extracted with an respective class method following a node selector statement.

Listing 4.1: Example of XML processing in Scala

```
1 var ResourceName: String = "Hybrid_LATMOS"
```

```
3 var xml: NodeSeq =
    <SimulationModel>
5     <ResourceID>impex://LATMOS/Hybrid</ResourceID>
      <ResourceHeader>
7       <ResourceName>{ResourceName}</ResourceName>
      </ResourceHeader>
9     <SimulationType>Hybrid</SimulationType>
      <CodeLanguage>Fortran2003</CodeLanguage>
11  </SimulationModel>

13 var resourceID: String = (x \\ "ResourceID").text
```

With these basic capabilities one can already construct libraries for serialisation and deserialisation of XML structures smoothly, e.g. by introducing respective parsing and writer methods for classes or by defining implicits for translation of custom data types in a dedicated scope. Pollak (2009, p. 78) also gives some examples how pattern matching can be used here to support more abstraction in deserialisation mechanisms for XML documents implemented with Scala. Pattern matching can be used in this context to define conditional search algorithms for XML data structures quite efficiently. Since the *IMPEx data model* is producing large and complex XML documents describing the contents of simulation databases complying to it's extensive *XML Schema*, it is of great importance to look at possibilities beyond these simple parsing capabilities. As discussed in Topf (2012b, pp. 13) *XML Schemas* provide useful information about specific XML structures such as validation rules, element relationships and type constraints for example.

Of course it would make perfect sense to make use of this information when transforming XML data into internal Scala data structures. It is indicated in preliminary studies in Meredith (2012, p. 78-80) that a so-called *data binding* mechanism - automatic conversion from XML data models to Scala objects - can be achieved by external Scala libraries, such as *scalaxb*[3]. On the other hand there are additional libraries available for Java which provide data bindings as it is described with the *jaxb* library in Ullenboom (2012, p. 1153-1158). Both of these possibilities must be considered within the architectural design of the IMPEx portal, where it finally needs to be decided whether to write the *internal domain-specific language* for exploiting the IMPEx data trees based on parser combinators and inbuilt Scala functionalities or by using parser generators with external libraries. Yet it is already indicated in Meredith (2012, p. 79) that especially in situations where a communication and data exchange protocol is based on domain-specific XML data models external libraries are more flexible. The reason for that is that external libraries such as *scalaxb* implement generic *monads* for parsing which are independent of specific domain models and therefore easily maintainable and extendable through automatic and implicit conversion in this case.

[3]Scalaxb, XSD and WSDL data binding for Scala, see: `http://scalaxb.org/`

4.2. Parallel and Concurrent Programming

As already indicated at the beginning of this chapter it will be needed for the IMPEx portal that requests coming from local or remote clients can be handled *asynchronously* and in *parallel*. For now there will be a close look on what Scala can provide in order to be able to define libraries which enable *concurrency* of their operations independently of the later used network transport protocol. The most important issue which needs to be taken care of is that blocking operations and shared data over different threads on the server should be avoided in any possible situation since it is considered as not optimal with regard to composability of concurrent tasks as stated in Haller and Sommers (2011, p. 26). Moreover error handling is significantly harder to implement as well as error recovery according to the practical examples described in Wyatt (2013, p. 36). It was shown in the theoretical chapters of this thesis that functional programming and its fusion with object-functional programming in Scala are fostering the usage of non-blocking operations and shared data can be avoided by using immutable data structures. Moreover, the whole functional programming domain with more fault-tolerant procedures, better modularity and more concise code are predestined for two branches of computing which are considered important for data analysis tools like they are implemented and used in IMPEx: *multicore- (or parallel-)* and *cloud- (or distributed-) computing*.

According to Neward (2008, pp. 3) the main reason for industries to use Scala and object-functional programming are these particular fields of computer science so the attraction in using Scala in this thesis is originating from these assumptions. Both capabilities enable concurrent programming to different extends and based on specific aspects and requirements which are outlined in Wyatt (2013, pp. 32). In the case of IMPEx, parallel computing it is important for the operation of simulation models and distributed computing is predetermined through the elaborated service-oriented architecture within the project. As one can imagine, it will be a standard situation in the latter case for example that *asynchronous events* are triggered which must be *non-blocking* to other tasks in the distributed framework provided by IMPEx. In any scenario of concurrent actions within IMPEx mutable state will be an critical element which needs to be treated in an efficient and fail-proof way. As it was already mentioned in the theoretical chapters of this thesis one must move away from modification of mutable state whereever possible and look more on the transformation of immutable values respectively. It was also shown that composability within functional programming helps in abstraction and modularisation of functionality so that parts of a program can run independently and concurrently in the long run. One will see here that the usage of functional combinators and the concentration on transformation rather than modification will help in shrinking the complexity of certain parts of the software.

4.2.1. Actors and the Akka framework

The previously mentioned motivation goes also in-line with the current shift of paradigms in the design of web-based systems from *structured* to *event-driven* reactive systems where there is a particular focus on the scalability of systems rather than solely on performance (Gupta, 2012, p. 14). With regard to IMPEx it is yet to be decided whether performance or scalability is the more important issue to be considered. Currently the main focus lies on the elaboration of a *concurrent* and *non-blocking* I/O so asynchronous events will most probably dominate the message passing throughout the IMPEx infrastructure. There will be a focus on the description of specific dataflows in the service-oriented system similar to workflow descriptions as they are already known within web-based environments. Following the assumptions in Allen (2013, p. 10) this is also part of an optimal domain-driven design and gives excellent opportunities in exploitation of the data model and respective web services available within IMPEx. The foreseen architecture of the portal will keep each service *loosely coupled* as it was put to practice in the original design of the IMPEx infrastructure. The first practical implementation of an asynchronous messaging system within Scala are the so-called *actors*. The principle behind actors is inherited from the functional programming language Erlang and basically consists of an abstracted way to work with standalone threads and additionally provides a communication mechanism between them (Haller & Sommers, 2011, p. 25).

As it is also stated in Haller and Sommers (2011, p. 28) the manual work with Java threads is considered as a last resort when there is no other way to synchronize specific funtionality and shared data between threads is explicitly needed within a functionality implemented in Scala. Note that in this case the developer would need to take care of the synchronisation with dedicated locks and related error handling manually. *Actors* help in this regard because they do not enforce locking and the developer doesn't need to take care of synchronisation functionality by him- or herself. They are also embedded in the "Akka middleware framework" which is included out-of-the-box in Scala. In principle, actors in both cases are able to send and receive messages to and from other actors as well as they are able to react specificly on certain messages (Piepmeyer, 2010, p. 263). Each actor is comprised of a set of objects and each object within an Akka enabled program code can and must be related to only one specific actor. Each modification of an object from an external entity must be handled by sending messages to its related actor. In addition to the provided general messaging middleware, actors can also have queues so incoming messages are stacked for processing. The control flow for the queue implementation in actors is shown by example in Raychaudhuri (2013, p. 265) where it is also noted that receiving and handling of messages is done by two different threads. These basic features clearly show why scalability and modularity is fostered by this concept and reliability of communicating services can be increased.

The main advantage of actors in the scope of this thesis is that they primarily share nothing and their underlying processes react on predefined events. Therefore one ex-

plicitly defines the flow of data when developing with actors. Through the provision of message queues the processes are never blocked by default. It also does not depend on whether the actor is accessible locally or remotely if appropriate network protocols are made available to the Akka library (Gupta, 2012, pp. 218). Note that actors are designed to encapsulate state and there cannot be any concurrent action within the actor implementation itself. Nevertheless it is possible to maintain a global state in actor systems with more complex functionality of Akka. In fact Akka does not always give away a thread per actor because sometimes this would create too much overhead. Instead the library provides a few thread pools which are shared automatically by the implemented actors in the Scala application (Pollak, 2009, p. 150). This approach allows a kind of load balancing and relatively fast messaging systems because threads are reused rather than created at each message call. In fact actors are completely fitting into the functional paradigm because they tend to send immutable data to one another according to Raychaudhuri (2013, p. 261). One interesting aspect for this thesis is the possibility to group actors by a specific domain where they are supervised by a single actor. With that capability one can build up tree-like dependencies between actors and grouped actors. Each node in the tree can be seen as a kind of supervisor which takes care of handling its child actors. Of course all supervised actors at a specific node can also be run in parallel and they can persist and maintain their own internal state within the dependency tree (Suereth, 2012, p. 221). Criticial data is processed by the supervisor, portions of data are delegated to the underlying worker-actors on demand and the supervisor also recognizes and handles errors coming from its actors.

What is also interesting when working with actors is the way how they can be parallelised. Since actors are nothing more than objects one can spawn the same actor as often as its needed and they will be having their own independent resources and workload respectively. It is noted in Allen (2013, p. 25) that actors should only get one specific task or responsibility in a chain and there must be a clear definition of this task in order to be able to leverage all functionalities provided by actors. With regard to IMPEx it is even recommended at the user requirements in Topf (2012a, p. 21-23) that each service and database should only serve functionality and data for a particular set of tasks. Any possible scenario where tasks are chained which rely on each other would be a favourable choice for actors. If these chains are multiply called through an appropriate web protocol they are also exectuted in parallel. Yet there is one thing which is needed when working within a concurrent and distributed environment: *synchronisation*. Besides of the standard capabilities of actors one can explicitly use a special `sychronised` actor where one can define then blocking operations. Those can also be limited in their amount of processes which can make use of the synchronised actor according to Odersky (2011, pp. 131).

Again it is important here to have a clear view of the foreseen architecture of the IMPEx portal when looking at these capabilities. The separation of tasks must be clearly defined within the architectural design as well as the distribution of supervisors, if needed across the planned system. Correlating tasks must be isolated so that

they only use their own dedicated resources in order to achieve physically separated, standalone processes within the system. Wyatt (2013, p. 480) mentions in this regard that blocking tasks must only be used if there is no possibility to solve the problem with asynchronous calls. What is particularily important for the development of the IMPEx portal is that such messaging frameworks provided with actors are closely related to the elaboration of domain-specific languages. In fact Akka can be seen as internal domain-specific language for concurrency tasks which should be brought in-line with the development of tasks within the domain of IMPEx resources.

4.2.2. Scala Futures and Use Cases

Besides of Scala actors and the more advanced Akka library there is another mechanism which enables concurrent computations quite easily. The so-called `Futures` can be spawned at any location in the source code where one is then able to work with asynchronously working processes where usually no result is immediately available. In fact one can work with a `Future` object independently whether there is actually a result stored in it or not. This capability makes it possible that the program can continue working on other tasks and passes around an "unfinished" variable which only blocks when it is finally accessed for computations. In fact the `Future` object has two possible callbacks which are changed asynchronously and can be accessed within the program: `onSuccess` and `onFailure`. As one can imagine a `Future` is exactly the same what was introduced with the `Option` monad in section 3.1.3. It can also be used just like any other monadic structure for functional composition so they can leverage any combined functional capability described so far as well as they can be freely shared across the application. An illustrative example of the usage of `Future` objects in Scala is provided in Alexander (2013a, p. 436-444).

Altogether the global trend, according to Wyatt (2013, pp. 75) is in the direction of *functional reactive programming* which leverages the concepts of `Futures` but also the handling of large data streams with lazy evaluation as one will see in the next section. The main advantage with that approach is that one can establish *non-deterministic* procedures with unpredictable outcome without violating the functional paradigm especially the "referential transperancy" discussed in section 2.2. Haller and Sommers (2011, p. 48) mention in that respect that data streams in reactive programming mark specific events where re-evaluation takes place so even large portions of non-deterministic data can be handled quite efficiently and moreover fail-proof here. It is clearly stated in Raychaudhuri (2013, p. 281) that any application of concurrency mechanisms in Scala including `Futures` must be well elaborated: At first one has to look at the pieces of architecture where one can work straight-forward with functional aspects which enable concurrency out-of-the-box e.g. immutable data structures. Afterwards non-deterministic elements with actors are selectively added where necessary and as last resort shared, mutable data is used. Before using locks and explicit threads one should consider the introduction of a *transaction-styled* communication

with a chain of tasks which all need to be successful in order to complete the whole procedure. It is again very important to isolate independent tasks in order to create lightweight processes and to keep the need of complex error handling at a minimum. When working with actors it will most probably be neccessary to introduce supervisors so that most of the control structures can be moved out of the worker actors. In the end business logic and error handling should be completely decoupled as it is the case in service-oriented architectures. Nevertheless, when working in distributed environments like in IMPEx one must be extremely cautious in what technology for concurrent programming is used. Especially RPC-styled communication which emulates synchronous method dispatch is completely ignoring the scalability of distributed, concurrent computing and moreover preventing the efficient use of the above mentioned capabilities.

Nonetheless the *event-driven model* is certainly a mechanism which needs to be taken into consideration when elaborating the *IMPEx portal* architecture. With all the service endpoints which need to be exploited the distributed environment can not be considered as a reliable source of information. There is definitely the need to introduce mechanisms for handling asynchronously acting, remote procedures. In the service-oriented architecture there is no easy way to guarantee message delivery so there must be also a capability for automatic retries and special states so the system knows whether it needs to block other tasks or not. It is important to note that asynchronous messaging and non-blocking tasks should keep any possible delay between sending and receiving at a minimum by keeping the actor tasks concise, in particular within service-oriented systems according to Wyatt (2013, P. 84). At the current stage there is no concrete descision possible for the IMPEx portal since it relies on how the messaging itself is designed in the architecture. Nevertheless it is highly probable that web service endpoints need to be exploited with non-blocking communication mechanisms and error handling is supervised from a third entity like it is concepted with actors. Sometimes it might be also needed to introduce blocking communication when service calls are relying on critical data needed for successful continuation. Maybe a mechanism for creating *transactions* is needed in that respect. It is up to the implementation of the access layer to web services from the *IMPEx portal* perspective, whether a event-driven application model needs to be elaborated manually or is already provided through respective web frameworks.

4.3. Side-effects and Basic I/O

As already mentioned in the theoretical chapters in section 2.6 of this thesis one fundamental language design pattern available in pure functional programming sticks out as it allows operations coming from imperative programming: *monads*. It was also indicated before that especially two aspects mainly are covered by this abstract data type: functions allowing *side-effects* and purely functional I/O operations allowing *non-determinism* without loosing *referential transparency*. Moreover, the last two

sections have revealed more use cases of monads for IMPEx with regard to parsers and concurrent programming. The reason why side-effects and I/O with monads is studied in detail in course of evaluation of the base capabilities needed for the IMPEx portal is obvious: On one hand there are situations arising in programming where side-effects are needed and cannot be circumvented e.g. when maintaining a global state for a group of related procedures. On the other hand I/O operations will be needed when dealing with both web-based and local resources. It must be noted though that both aspects can be covered by simply exploiting imperative structures within Scala by using Java libraries. Nevertheless, for the IMPEx portal the aim should be to stay as pure as possible because it doesn't seem to be worth to use Scala as "yet another object-oriented programming language". All the advantages of the new programming paradigm seen so far already give a good indication for reasoning about structured programs flavoured with pure functional concepts.

Although monads are also abstract data types in Scala as they are in Haskell, they are slightly more versatile. In Scala monads are represented through classes which enable generic type constructors easily through type parametrisation and variance (Raychaudhuri, 2013, p. 161). Furthermore there is no need to explicity provide a transformation mechanism from the applied type to the equivalent monadic type since class instantiation automatically generates a monadic type out of this given type. The only operation which needs to be implemented explicitly in Scala is the `flatMap` method. This method does the same as the equivalent procedure $fmap$ in Haskell and is a so-called *functor* as already discovered. It provides a generic operation which enables the application of functions on a given type for subsequent transformation to another type just like the `map` function. In addition to that `flatMap` does not only map the given function on each element of the applied data type to yield a new type - the resulting type will also be encapsulated in the respective monadic type again. With this feature one has the ability with monads to wrap a given type into a generic type to be able to work with it in an abstracted way.

As the monad always returns an expected type it is *referential transparent* to the scope outside independently on how the applied type is treated inside of the monadic data structure. This enables to work in a pure functional environment with imperative aspects abstracted into a monadic data type as it is stated in Suereth (2012, p.264). Naturally it depends on the use case whether to aim at implementing monadic data types into Scala programs or to choose the equivalent imperative way of achieving e.g. side-effects. Sometimes it may not be desireable to use abstractions of types in such highly manner but the idea to extend a monad with generic *combinators* - methods which provide specific behaviour within this data structure - seem to go inline with the assumptions made so far. If there is a way in Scala to implement generic methods for a set of types having a certain relationship in the class hierarchy it must be clear that such capabilities will be needed also for enabling complex operations within the planned *domain-specific language* for exploiting IMPEx trees and services.

Chiusano and Bjarnason (2013, pp. 201) gives another motivation for using monadic data types within Scala by introducing it's inbuilt "internal domain-specific language for I/O operations". With the capabilities described here it is possible to keep I/O purely functional by using either inbuilt or self-defined monads. The main design difference compared to equivalent imperative functionality is that functions describing data are clearly separated from functions interpreting the data. So the only thing which needs to be taken into consideration when working with monads is that pure operations must be abstracted away from operations having side-effects. The examples elaborated in Chiusano and Bjarnason (2013, pp. 240) also show how this is achieved with object-functional elements in Scala: every method having side-effects, for example printing to the console, is encapsulated in a predefined data type in the form of a class. This means that when the "effectful" method is called through a function, not the method itself is returned but an instantiation of the data type containing this method. As it is also illustrated there one can easily add new functionality to this class and it's instantiations respectively.

Since it is planned in course of IMPEx to implement a *domain-specific language* for data access it is a natural conclusion to describe the data model by means of Scala objects and to provide an interpreting mechanism for the model-specific metadata. With regard to parsers this will be done with functional compositions as already discovered and depending on whether the focus is on input or output operations there will be the need to add more compositions for specific tasks. The most important issue will be here too to keep the "effectful" actions away from the pure functional operations. It is noted in section 2.5 that one has always to look which data structure is needed in a particular software solution and how the previously described constraints can be implemented within the given architecture. It must be clear that within monadic data structures one has the same disadvantages of e.g. non-determinism and unpredictable results as they are existing in imperative programming but monads themselfes are still composable within the purely functional programming paradigm and therefore they are perfectly embedded in their host language Scala.

The only thing which is additionally needed when working with I/O in IMPEx especially for capturing inputs, is a mechanism which enables *continuation passing style (CPS)*. The reason for that lies in the fact that when dealing with input there is sometimes no chance to predict the length of a given input. Therefore there is the need for *non-strict* evaluation and more importantly *non-blocking* operations. Firstly there needs to be a way to deal with growing data structures without running permanently into stack overflows and secondly long running operations accessing external data should by no means block the whole program since they might not return a result at all in the end. Of course, stack overflows can be prevented in functional programming by using recursive data structures and tail-recursive iterations - this must be kept in mind when working with monads. Non-blocking operations can easily be achieved by using the described `Futures` in the previous section which are monads by themselfes since they also encapsulate any applied type in a generic type. This mechanism is absolutely needed when working with e.g. networked I/O. According

to Wampler and Payne (2009, p. 286) it seems that for inputs with arbitrary length one has to look also at the possibilities with *call-by-need* evaluation within Scala. Furthermore there is the need for a special data structure which can grow practically indefinitely on demand. Scala provides this with the `Stream` data type. In principle streams can be used to realize functional buffers which read inputs divided into packets of fixed length rather than reading the whole input at once.

As one can imagine there is yet another problem: How can buffers or streams be leveraged in Scala without the use of imperative loop control structures which cannot be composed into pure functional programs? One possibility would be to search for equivalent functional data structures such as lists which are composable and also recursive. In fact the only difference between lists and streams in Scala is that streams do not evaluate their tails until they are actually accessed (Alexander, 2013a, p. 331). Only their head element is evaluated at any time. Every other capability for accessing and manipulating elements in a stream is working with the same functional aspects as with lists. A typical monad for handling arbitrary inputs from an outside scope would encapsulate this *lazy stream* into it's own data structure from where the streams is then growing depending on the length of the input data. The significant difference between traditional imperative buffers and streams in Scala is that buffers and their respective file handles must be closed explicitly when the end of an input is reached (Chiusano & Bjarnason, 2013, pp. 280). In functional programming like in Scala there is the possibility for automatic closing of external resources with *call-by-need* evaluation as described in Suereth (2012, p. 272). Any other action similar to imperative file handles would not be pure because it would require accessing the internal state of the monad from the scope outside.

Sometimes it may be needed to create effects in order to maintain an internal state of the program besides of I/O related effects which rely on external data. This means one needs a mechanism which provides the same capabilities with pure functions for internal effects. In fact the approach is practically the same as with external effects - a monad is used to wrap different data types into one particular monadic type as explained in Raychaudhuri (2013, p. 158). This is of certain interest within IMPEx since involvement of different remote procedures does not only need asynchronous messaging mechanisms as already discovered but also stateful operations which delegate state related information from one entity to another. Note, that monads are already capable of doing this as it was seen with actors which can persist state internally by creating a supervised group of actors who are able to share information about their common global state. They simply mutate different data structures in their internal scope and return the result in form of e.g. a *state monad* type to other monads which may continue changing the state encoded in the monadic data structure. The only issue which needs to be taken into account again is that operations mutating internal state must be clearly separated to those operations creating external effects according to Chiusano and Bjarnason (2013, pp. 265). It will be seen in the next section that it is not needed for the most use cases in IMPEx to elaborate custom monadic structures since they are already provided either by inbuilt Scala libraries as it was seen with

parsers and concurrency mechanisms or are additionally enabled through external libraries. Nevertheless their internal capabilities must be kept in mind during the elaboration of technical use cases at the IMPEx portal.

4.4. Scala Web Frameworks

At the current stage of this thesis one has seen a variety of potential capabilities of Scala that will be needed in course of the development of the *IMPEx portal*. Yet, there is one specific glue missing which sticks together the base technologies and makes them available to the user: A framework which enables communication and utilisation of all portal-specific functionalities through web-based protocols. It is considered here that the reader has basic knowledge about the HTTP protocol whose stateless *request/response* messaging model was introduced in conjunction with web services in Topf (2012b, p 15). Furthermore it is assumed that the reader knows about commonly known Java web architecture APIs. In course of research made for the *IMPEx portal* regarding web development libraries which are built on top of the HTTP protocol, two most common frameworks were discovered which use Scala as a host language: Lift[4] and Play[5].

The first framework - Lift was not considered further although having excellent support for actor-based systems and XML processing mechanisms according to the introduction in Perrett (2011a, p. 6-10). The reason lies in the fact that Lift requires a Java application server or at least any Java-based software which is able to expose Java servlets like Apache Tomcat[6]. Lift only deploys applications as a servlet container so it is not a standalone solution. Since there is no existing ecosystem for Java based web applications existing in the IMPEx project the descision was made upon the second framework Play which in fact is a standalone framework at a first glance with its included non-blocking I/O web server called Netty[7]. The only alternative would have been to use Java libraries which enable the HTTP protocol such as JAX-RS[8] but with the limited resources given it was not considered feasible to implement the full *Web Service Architecture Stack* described in Topf (2012b, p. 14-17) more or less from scratch.

One of the most interesting facts discovered in the first steps of evaluation of the Play framework was that the structure of applications here is inspired by the famous *Ruby on Rails*[9] web framework which is commonly known as a fast implementation and deployment system for web-based Ruby applications. The main focus of such

[4]Lift web application framework: `http://liftweb.net/`
[5]Play web application framework: `http://www.playframework.com/`
[6]Apache Tomcat web container and -server: `http://tomcat.apache.org/`
[7]Netty client-server web framework: `http://netty.io/`
[8]Java API for RESTful Services: `https://jax-rs-spec.java.net/`
[9]Ruby on Rails web framework: `http://rubyonrails.org/`

frameworks is on a common coding standard for all parts of the web-based software and fast result delivery rather than the provision of a complex API architecture which covers a wide range of different use cases going beyond the scope of this project. To undermine these basic facts one must look at the structure of Play based applications and how they are built up during a usual implementation phase. As it is stated in Ellis (2010, p. 10) the deployment time efficiency of Play based applications is much higher than it is within common Java web frameworks. The reason for that lies in the typical structure of those application frameworks which tend to have a complex set of layers for the different parts of a typical web application. These layers expose separated functionality for implementing e.g. request handlers or view templates. Furthermore there is an additional layer needed for encapsulation of the web-based application as such in order to be finally enabled by another separated entity - a suitable Java based HTTP server for so-called Java web archives (Hilton, Bakker, & Candeo, 2013, pp. 6).

In comparison to that, the Play framework provides every needed capability for modern web-based applications through one layer and also includes a Netty HTTP server as mentioned before in it's standard installation. Furthermore it ships an own *Scala Build Tools (SBT)* based console for compiling, testing and deployment without the need of additional software except of a Scala compiler of course. Especially in the IMPEx project there is the need to be flexible for quick changes and there must be an efficient way for testing, staging and operation of applications. This is a fundamental cornerstone for the successful realisation of the project goals. The high degree of RTD requires capabilities to serve the community with up-to-date software for efficient evaluation and integration of instantly needed modifications. Both Play and *Ruby on Rails* are equipped to overcome the most common challenges of modern web applications like big data handling, concurrent processes, efficient ways of persisting data, full support for client-side HTML5 technologies and zero-downtime deployment strategies (Alexander, 2013b, p. 71). The latter two challenges are the most interesting from the authors point of view since efficient client-side technologies have revolutionised the market and a shift of paradigms with regard to task distribution in a client-server architecture. Furthermore automatic deployment strategies of applications running on top of the JVM have not only become more efficient in the evolution of Java, they also do not require that production software is taken offline for the time of updating. This is seen as crucial for modern web-based applications which are frequently extended and updated according to the users need.

Another advantage of Play is that Java and Scala can be intermixed throughout the whole software whereever it is needed. This opens a lot of possibilities since all existing Java libraries can also be used in the context of web-based applications. Together with the fact that a "full-stack framework" is provided with database connectors, the integrated web server and an extended I/O API including Akka features can be practially used for any enterprise application (Typesafe, 2013). Moreover it can also be integrated easily in existing Java applications, since Play also provides a possibility to export an application into a servlet container if needed. Interestingly the Play application architecture focuses specifically on a clean REST-API and does foster

bottom-up development by beginning with the HTTP interface of an web-based application. This basically means a developer starts with a Play application by defining the routes and respective HTTP methods (`GET`, `PUT`, `POST`, `DELETE`) of the planned software.

This is in-line with the planned implementation roadmap of the *IMPEx portal* as one will see later in the user requirements definition. Moreover, according to Hilton et al. (2013, pp. 68) "URL-centric design" is considered a good style of coding because it helps to perfectly adopt the HTTP architecture without breaking its stateless nature. It is also mentioned here that traditional web-based Java applications lack of this because they produce REST-based HTTP interfaces with a lot of payload e.g. in `GET` requests. This may lead to URL structures which are not stateless at all and depending on specific content of `GET` parameters. In fact Play helps the developer to implement a more concrete *resource-oriented architecture* where each defined HTTP route is acting and existing as a resource on it's own decoupled from other provided functionality. This makes the whole application much more composable and pure functional programs can easily be translated to the web when following the paradigm of REST (Reelsen, 2011, p. 9).

On top of the functionality provided for the transport layer, Play embraces the *Model-View-Controller (MVC)* pattern which is a common design paradigm for web-based applications. A developer designs the controllers and models of the application and finally provides user-relevant information in views. The controller handles the requests coming from the client and calls a respective model which is then transformed to display the requested information in the view (Petrella, 2013, pp. 30). In Play the defined HTTP routes are always directing to a static controller object and a respective method which provides a specific `Action` type to the client. `Action` types are implemented with actors of the Akka library which are working asynchronously on incoming requests as it is naturally required by the HTTP protocol. The `Action` type is therefore also monad which can be composed with other actions or be encapsulated by generic actions needed in the controller such as e.g. filters, validators or manipulators of incoming requests before they are handled by the wrapped action.

A controller in Play does nothing more than dispatching the request and delegating it to a respective model e.g. a Scala class which is then extracted in the views of the web application. These views can be either provided by an inbuilt template engine based on Scala or the bare REST interface can be used by JavaScript MVC frameworks for example. Routes can be also called everywhere in the controller code since after registration of a route at compile time a respective access method is implicitly included in Play. Note that in contrast to typical Java based web applications both routes and views of an application are also type-safe because the content is checked explicitly at compile time. As one can imagine a view may include conditions, iterations or method calls in Scala syntax embedded into HTML markup and those are not checked e.g. when using traditional view templates in other Java web frameworks (Petrella, 2013, p. 59). Moreover, the application configuration and the plugin

settings of a Play application are type-safe because their syntax is checked at compile time too which leads to the assumption that the Play framework is particularly focused on fail-proofness throughout the whole application like it's host language Scala.

Of course it must be also noted here that the fast evolution of client-side technologies including e.g. client-side AJAX calls and different HTML5 capabilities have brought a change in the design of MVC frameworks since a server must be able to handle different streams of data the client is requesting and consuming in parallel (Hilton et al., 2013, p. 351). Limitations were for example the vast consumption of resources since every request in pure Java-based web applications created an additional thread and memory consumption increased drastically with the amount of clients connecting to the server. One has already seen that asynchonous code in Scala tends to re-use threads from a thread-pool and they are never blocking each other. It is stated in Brikman (2013) that especially when requests are taking longer time to respond a "thread-per-user" policy is not scaling up properly and costing a lot of performance under heavy network load. The non-blocking I/O of Play has also some advantages when it comes to *reactive programming* where the client subsequently receives results coming from the server which are only preceded by a single request before. The framework can provide stream processing rather similar to Scalas basic I/O features described in section 4.3 where the server is continously pushing data and automatically closes the resource when the end of data is reached. So there is no need for the client to ask continuously whether a result is already available to the request or not.

In fact the HTTP body parser of the controller actions mentioned before is an integral part of this stream processing. It does not matter if the request is fully parsed by the action, it is implicitly "existing" all the time and its payload is automatically recognised and typed to e.g. XML, JSON or plain text by the Play framework. It must be noted here that Play provides a similar versatile syntax for processing JSON documents like it was discovered with XML in Scala. Play also eases the efforts needed to transform JSON objects to Scala objects and vice-versa. Responses from a Play framework server are encoded into well-known HTTP status codes before being dispatched, such as `OK` (200), `NOT FOUND` (404) end even `NOT IMPLEMENTED` (501) which is a dedicated response action itself. With this rather simple mechanisms to work with the HTTP protocol the developer is explicitly motivated to build applications which embrace asynchronous messaging and complete statelessness. Moreover this basic approach goes in-line with the cornerstones of functional programming and Scala respectively.

For the *IMPEx portal* it will also be needed to embed web service calls in the controller actions which delegate respective user requests to a remote server and forward the results either unchanged or transformed back to the user. In that respect it will be also needed to encapsulate asynchronous calls within the controller object's methods. Interestingly, Play also uses Akka features as a basis here by applying the `Future` type introduced in section 4.2.2. This `Future` type encapsulates the response from the

web service call and can be mapped onto any subsequently used variable. Usually a call involving a `Future` is wrapped into an `async {..}` block in Scala or a special `Action.async{..}` type in Play which identifies an special area where results of e.g. web services are to be used. This area is not blocking in both cases, so Scala does in fact create an own thread here for handling the remote data without blocking the original thread created by the originating request (Petrella, 2013, p. 196). This is an efficient way to work with remote web services because Play is able to suspend threads which are currently not used when making additional remote requests during execution of a local action. This particularly helps with the scalability of the application. Brikman (2013) also mentions that altogether it turns out that Play is *event-driven* and not *thread-driven* which in the end distinguishes Play from *Ruby on Rails* but also from Java web applications using servlets.

The advantages were already clearly described in this chapter especially with regard to concurrent programming nevertheless there is one actual topic which needs to be explained in more detail here, because it might be needed for the IMPEx portal: *WebSockets*. *WebSockets* are a currently developed HTML5 standard which allows to keep HTTP requests open without immediately closing it after an initial response was delivered by the server (Ellis, 2010, p. 171). The advantages with that feature is that the server can use this open connection for subsequent communication without further requests by the client. The server is able to push information to the client which might be helpful when working e.g. with workflows where clients usually wait longer for a responses or subsequently receive a chain of responses. Here the client does not need to poll at the server frequently to recognize when the services are finished, the server just publishes the information as soon as it is available. Altogether one sees that not only Play goes in the direction of *event-driven* applications, the well elaborated web standards also do, which is a rather promising aspect for the sustainability of this framework and its usage within the IMPEx project.

To conclude all the functionalities provided by the the HTTP protocol one must look at the so-called *HTTP-session* which enables the developer to transport information over serveral requests e.g. through the *HTTP-header* with cookies stored on the client-side. So in fact, it is possible to encode state into the client-server communication. This feature can be used to "group" subsequent requests into a user session to be able to recall information which was already sent on the server-side for example. Play does not explicitly force such a functionality because it would not go in-line with the promise to provide a completely stateless interface. If there is a need to persist state it must be completely separated from stateless procedures in Play according to Reelsen (2011, p. 35). This is also required by the underlying Akka features since their monadic data structures would loose their capability to be composed when there is state encoded outside of the actions directly triggered by the client. In fact the same paradigm as one already knows from monads comes into practice here which is similar to *continuation passing style (CPS)*. A cookie is used as a key-value store in the Play framework to persist small amounts of data within an user session and every request is transporting this data from the client to the server through the *HTTP-header*. This

means that the client's session does not share any state with the server as opposed to common Java based frameworks according to Ellis (2010, p. 32). Here a cookie stored on the client-side is only used as an identifier to access session data stored on the server-side. The reason for the different approach at the Play framework lies in the fact that it is more clean with regard to the stateless REST interface enforced here to use a so-called "share nothing" policy between server and client (Reelsen, 2011, p. 250). Note that this cookie is additionally signed so that it cannot be compromised on the client-side.

In addition to that Play provides another cache called *flash scope*, which is able to cache data only for one subsequent request. This might come in handy when implementing REST interfaces where scalability plays a big role because nothing "volatile" is stored at the server. The client can basically communicate with different servers in a cluster and still transport information from one request to an other (Hilton et al., 2013, p. 42). Nevertheless, sometimes it is needed to persist data on the server not only for a given timeframe and a specific user but for some more general purposes. As one has already discovered Scala comes in handy when working with domain-specific models so the general functionality provided by Play to save objects to databases is also dominated by aspects evaluated in section 4.1. Again it is needed to separate descriptive domain-specific data and functionality processing and using this data. Typically this is done with *data access objects (DAOs)* which interprete the models encoded into simple objects of the host language. In the case of Scala such models are often represented by the compact Scala case classes and Play provides DAOs which can directly map those classes into respective database implementations including a variety SQL and NoSQL solutions. It is still not known up to now whether it is needed in the future to persist data at the IMPEx portal but the capabilities need to be kept in mind especially when elaborating the data model representation within Scala.

As conclusion to the first evaluation of Scala capabilities which make the language a versatile companion for web development it is to say that everything needed for the *IMPEx portal* from the basic perspective is not only available but also fully conforming to the pure functional paradigm. One has seen that monads are used throughout many extended features of Scala and they are able to maintain the purity of functional aspects such as *immutability* and *referential transparency*. Some of the aspects described here with regard to domain-specific languages, concurrent programming and I/O seem quite complicated as they require background knowledge reaching back to fundamental functional programming paradigms. Nevertheless the final evaluation of Scala web frameworks in this chapter has shown, that these aspects are smoothly integrated in an efficient way with the Play framework. This framework takes over functional concepts of Scala described here which are needed in web-based applications and efficiently makes them available to the developer without the need custom implementations.

5. The IMPEx Portal

In accordance with the basic motivation of this thesis described in section 1.1 and recent discussions within the community of IMPEx a clear direction was established on what the IMPEx portal must and should provide to the broad scientific and educational community outside of IMPEx. Although there are some limitations on the degree of client- and server-side integration of functionalities, since the actual inter-tool communication via the SAMP protocol was not widely recognised as drawback. Nevertheless a significant portion of the foreseen portal architecture will be able to use the advantages provided by object-functional languages like Scala and JavaScript. One particular aspect was seen as most important, namely the efficient handling of *domain specific languages (DSLs)* in correlation with parsing and processing of XML documents within a service-oriented architecture as it was evaluated in chapter 4.

The following sections will provide a complete path through the software engineering process, where at first the concrete purpose and the aims will be formulated which were previously agreed upon by the project management commitee (PMC)[1]. As a second step the author will look at the existing ecosystem of IMPEx including it's data model and web service interfaces which were already described in Topf (2012a) and remark possible deviations from the originally projected design. The subsequent step will be the definition of user- and technical requirements as well as the portal architecture which are created in an iterative process and in close cooperation with the IMPEx community. During the design phase and its projected engineering process of the IMPEx portal the author will select fractions of the foreseen software where the previously described functional aspects of Scala and JavaScript will be evaluated by real-world measures in the IMPEx project. Furthermore, they will be cross-linked to the original ideas and advantages of pure functional and object-functional programming described so far.

[1] The elaborated user requirements in this thesis were presented and accepted at the IMPEx technical meeting in Toulouse, September 20, 2013

5.1. Purpose and Aims

The IMPEx portal is an official part and deliverable of the work package 5 *Outreach and Dissemination (ODis)*. The idea behind this work package at the time of project creation was to on one hand disseminate the achievements and results of IMPEx through the planetary science community in a proper way and on the other hand to educate newcomers (e.g. students) in the scientific fields of magnetospheric- and plasma physics, where the simulation models as well as the observational data provided by IMPEx are of significant value. All the functionalities of IMPEx should be promoted at a central access point suitable for the different target groups. Although IMPEx already has a website[2] with an integrated tool page the *User Advisory Board (UAB)* of IMPEx which is comprised of external experts in the respective science- and technology fields has critised the current way IMPEx delivers it's services and tools to the community.

The reasoning behind their criticism was that the current way the tools and services are provided does not emphazise the particular role each specific service has in the IMPEx environment nor does it show a coherent project environment. It was identified as an important step that after most of the system is clearly defined now there must be reference implementations of the data model and web services propagated by the simulation databases provided to the community. A potential user of the system must be able to identify those new capabilities independently if the services are integrated in one system or distributed among the participating institutions of IMPEx. This is not the case at the moment so in the group of the UAB there were two particular proposals for changes and improvements which were taken as main direction to go for the implementation of the IMPEx portal:

- Development of a modern and dynamic map of all tools and services available with short and precise background information, including statistics of the provided content (datasets or web service methods).
- Provision of a graphical representation of the path the data takes through the system, emphasizing science cases and including tutorials on integration of IMPEx web services and the IMPEx data model.

After evaluation of the feedback and recent discussions within the project management commitee (PMC), the IMPEx consortium came to the conclusion that there needs to be more focus on the continous monitoring of usability throughout the IMPEx infrastructure. As for now the IMPEx website does not provide a clear view on the capabilities primarily delivered by the project. The user gets the feeling that IMPEx is about extending existing tools such as the participating 3DView Multimission tool, the "Automated Multi-Dataset Analysis tool" (AMDA) and the "Cluster Web tool" (ClWeb) which are building up the core of the tool- and resource layer presented in Topf (2012a, p. 25). Although a significant portion of the project's resources

[2]IMPEx project website: `http://impex-fp7.oeaw.ac.at/`

is actually invested in the update of these tools to work with IMPEx capabilities, the background web services and moreover the definition of the *IMPEx data model* is the key goal of the IMPEx project. The currently existing service interfaces and data model instances are in fact reference implementations for a *Virtual Observatory (VO)* in the field of plasma physics which should be promoted as such. It is concluded that this was particularly critizised by the UAB because the IMPEx website promises the establishment of an "interactive computational framework" (see: The IMPEx Consortium, 2013) where the user might indeed expect a concrete single entry point to the system with all new functionalities available at first glance.

This naturally leads to the question which was asked by the community whether it would be possible to make IMPEx web service and simulation database access available at one distinct place promoted via the IMPEx website. From here the user could decide on and select the needed datasets and functionalities for his/her specific research purpose. The user should be able to either delegate selected datasets via the SAMP protocol to any tool capable of communicating via well elaborated IVOA standards or aquire specific datasets directly for offline processing. An appropriate guidance and documentation through the whole system must be available for the user. Furthermore the IMPEx project must take care of the different operating systems and more importantly the different browsers and devices which will be used by the community since it is currently not practised adequately. 3DView for example can only be used within a Java Virtual Machine (JVM) and AMDA is only working properly on desktop computers as well. Of course the limitations can not be solved completely by the provision of a portal but at least basic functionality should be available to a wide range of devices for example.

In addition to that the dedication to educational and public outreach work in the field of plasma physics despite of being welcomed by the IMPEx community should be more emphasised in the project. In fact there was little progress in those fields since the availability of *IMPEx* services in the current stage of development at the level of tools is not optimal for the public presentation of the particularily implemented capabilities in this project. Especially the start of active usage within student seminars with the current tool-set is prevented due to the relatively steep learning curve in efficient operation of their specific functionalities[3]. Altogether the path for the IMPEx portal was clearly set by the PMC trough various discussions and processing of feedback from the UAB as well as the initial definition and allocation of resources in the *Description of Work (DoW)* of the IMPEx proposal. One important issue was raised by the PMC after definition of the principle goals of the IMPEx portal: The developed system should not duplicate any of the data analysis and visualisation capabilities already provided by the tools of IMPEx. The focus must be on provision of auxiliary services which are either not directly available through the tools or more importantly not existing so far. To name one important aspect here, a common registry of IMPEx services is still not in operation which would add a more integrated interface to all ex-

[3]There was recent progress in usability shown with the newest versions of AMDA and 3DView, which were presented at the IMPEx technical meeting in Toulouse, September 20, 2013.

isting IMPEx services following the *Virtual Observatory (VO)* paradigm as described in Topf (2012a, pp. 12). The complete user requirements definition, respecting the goals and general purpose of the portal as described, but also the mentioned constraints in functionality and technical requirements must be therefore established on top of the existing infrastructure.

5.2. User Requirements

As already mentioned, there were several internal discussions during this year what the portal could finally provide in its user interface. Still, there are no concrete definitions except what was written preliminary in the *Architectural Design Definition*. In addition to that, the UAB Meeting and the related UAB Questionnaire gave an indication on what is needed from the community perspective. In course of preparations for the IMPEx technical meeting in Toulouse the user requirements described here were captured from all available documentation so far.

Three main cornerstones were elaborated in course of these investigations which give a rough overview of these user requirements and the projected implementation work:

1. **Development of a main entry point to the IMPEx project resources:**
   - Focus on visual, intuitive and explanatory features
   - Provision of a map through "the forest of applications"
   - Information on available models and scientific assumptions made etc.
   - Different entry levels and centralised access to simulation databases
2. **Basic workflow integration:**
   - Built on top of the available web service interfaces (IMPEx protocol)
   - Possibility for automatisation of recurring processes
3. **Project Documentation:**
   - Access to important documentation wia website
   - Documentation for non-expert users
   - Science Cases visualisation

This thesis is naturally focused on the underlying architecture for provision of the first cornerstone and further integration into more complex operations of the second cornerstone. It was already clearly shown in chapter 4 which basic capabilities need to be available to achieve these projected goals of the IMPEx portal which were already known from the beginning of the project: XML parsing, processing and consequent distributed I/O with web services using the obtained metadata from the XML trees for data access. However, at the current stage a final roadmap for development must

be determined by a concise classification and priorisation of the user- and technical requirements. Furthermore there must be restrictions on the functionality in order to fullfil the previously described aims in the given time without duplicating work.

The most important constraint on the capabilities from the users perspective is based on a remark coming from the IMPEx community which was covered in the *Architectural Design Definition*: "The purpose of the IMPEx portal is to provide users not familiar with IMPEx tools an additional way to choose simulated or observed data of interest and send them to an IMPEx tool through a SAMP hub". This basically means that the user should get the information needed where to go for working on a specific IMPEx science case including basic capabilities to select, view and download portions of simulation or observational data. Therefore the IMPEx portal must only act as an enabler for more complex operations which can be conducted by 3DView, AMDA or ClWeb. In the opinion of the author this also reflects in which direction the IMPEx project should go for promoting the underlying SPASE-based XML data model and the web service interfaces. The IMPEx portal should also make its domain-specific resource model and remote service connectors transparent to the scientific community for further integration into other tools and extension of distributed database access through the IMPEx infrastructure. Both requirements tightly fit into the assumption of the author that a single gateway to a complex service-oriented architecture increases the usability and extendability of the system beyond the projects lifetime.

The following sections will summarize all aspects required to fullfil all these concretised goals and constraints from the IMPEx community. The requirements will be basically prioritised in the categories *Must (M)*, *Should (S)* and *Could (C)* depending on the decision made by the IMPEx-PMC. It will be also traced in the user requirements from where the requirements come in order to be able to clarify open issues during the architectural design with the respective stakeholder.

5.2.1. Search Capabilities

The *search capabilities* are representing the core of the portal since they are needed to make the metadata from the IMPEx data trees available in a suitable form which is most useful for the projected tasks of the user. This fact also implies that the *search capabilities* must be able to identify the web service methods available for the searched and selected dataset of a particular tree. It is important to note that the following requirements are building up the basement for all other functionalities explained in this chapter.

Number	UR-1.1
Title	Consolidated IMPEx data tree
Priority	M
Origins(s)	IMPEx team Graz
Type	Interface requirement
Description	In order to be able to visualise the dynamic view of the IMPEx data trees in the way it is described in section 5.2.2 they must be consolidated into one single IMPEx data tree. The data structure used at the portal for this combined tree must be capable of the rearrangement functionalities depicted here. It is commonly known that the underlying SPASE data model and its flat hierarchy is basically capable to be stored in one single valid XML document. The IMPEx data tree must include the metadata from the SMDBs, AMDA and CLWeb. The default arrangement of the tree must display the category *Data Provider* at the top level including information about the contained *Data Type*. The user interface should display the tree as an *IMPEx map* where further tool and service usage is indicated with an appropriate visualisation.
Comments	It must be noted here that the observational data trees from AMDA and ClWeb do not comply with the used SPASE data model for the SMDBs described in Topf (2012a, pp. 30). It is considered by the IMPEx-team that the observational datasets will be transformed to comply with the SPASE standard in the future as it was done with the VEX-MAG datasets in Topf (2012b, pp. 32). At the current stage the IMPEx portal must include a way to be able to understand the current proprietary metadata format of those tools. The consolidated IMPEx data tree should also be available through a general purpose interface for other applications in XML format.
Related UR(s)	UR-1.2, UR-2.1 and UR-6.2

Table 5.1.: UR-1.1 Consolidated IMPEx data tree

Number	UR-1.2
Title	Search for IMPEx trees and services
Priority	M
Origins(s)	IMPEx-PMC
Type	User interaction
Description	The dynamic view of the IMPEx trees and services must be updated automatically depending on predefined filters and user queries. This means, the interface must display the trees and services in a form which is capable of re-arrangement based on the following query parameters: • *Target Name*: Mars, Venus, Jovian satellites, etc... • *Resource Type*: Observational or simulated data • *Data Provider*: LATMOS, FMI, CDPP, etc... • *Mission Name*: Venus Express, MAVEN, etc... • *Physical Quantity*: – Magnetic field – Plasma ion moments – Plasma electron moments – Plasma ion spectrograms – etc... In addition to that a to be determined set of elements in the provided data trees should be available for full-text search to the user. These need to be identified in the architectural design based upon usability but also feasability within the scope of portal developments.
Comments	The available IMPEx web services are tied to a specific *Data Provider* and will be therefore not directly involved in the filter and search mechanism described here. The related web services will be displayed in every search result depending on the given parameters depicted above associated with the respective *Data Provider*.
Related UR(s)	UR-1.1 and UR-2.1

Table 5.2.: UR-1.2 Search functionality for IMPEx trees and services

5.2.2. Visualisation Capabilities

The most important aspect of the *visualisation capabilities* will be the display of all resources available through the consolidated data tree and its search capabilities described in 5.2.1. Furthermore it will be needed to enhance the usability of available tools, services and the data trees by providing appropriate documentation on how to use these IMPEx capabilities. Together with the requirement fo a statistical overview of the available services the documentation are considered as secondary features of the IMPEx portal.

Number	UR-2.1
Title	Dynamic provision of IMPEx trees and services
Priority	M
Origins(s)	IMPEx-UAB
Type	Autonomous system activity
Description	The IMPEx portal requires a service which is able to display every attached observational and simulation database dynamically depending on their availability. Furthermore all available IMPEx web services must be displayed accordingly so that the user immediately recognises which functionality is available for a specific selected dataset in the given database. All relevant metadata for e.g. particular simulation runs or instruments must be displayed in correlation to all available physical quantities of the datasets. The interface must also define all input and output parameters as well as the required formats needed for communication with the web services. The IMPEx tools AMDA, ClWeb and 3DView must also be embedded accordingly in this dynamical *IMPEx map* to clearly show their particular purpose.
Comments	There is already a service existing called the *WS scoreboard* which is maintained by the team of ClWeb*. It displays all URLs of the XML data trees and WSDL descriptions of the IMPEx infrastructure and indicates their availability and development status. The service at the IMPEx portal must have a more suitable format of representation for the community outside of the IMPEx team by only providing the most relevant information for potential users.
Related UR(s)	UR-1.1, UR-1.2 and UR-3.2
	*WS scoreboard overview website: `http://clweb.cesr.fr/webservice.html`

Table 5.3.: UR-2.1 Dynamic provision of IMPEx trees and services

Number	UR-2.2
Title	Documentation of IMPEx services
Priority	S
Origins(s)	IMPEx-UAB
Type	User interaction
Description	An appropriate documentation of the available resources and services should be accessible in the *IMPEx map* at any time. It would be of advantage to place links to documentation views in close vicinity to the respective resource. The documentation should include on one hand tutorials and manuals for all IMPEx portal features and on the other hand technical documentation about the underlying data model and web services. All explained and described capabilities should be intuitive and additionally documented with step-by-step dialogs.
Comments	The portal should include an *IMPEx cookbook* in the future which translates the *IMPEx science cases* into hands-on tutorials for introduction to the combined usage of the IMPEx portal, AMDA, ClWeb, 3DView and related external tools. It should be also considered for the future to provide a certain degree of automatisation in the steps needed to accomplish science cases through workflow descriptions as they are used in the *Taverna workflow engine**.
Related UR(s)	UR-5.2
	*Reference implementations for workflows with IMPEx web service communication based on *Taverna* are provided by FMI: `http://www.myexperiment.org/groups/1150.html`

Table 5.4.: UR-2.2 Documentation of IMPEx services

Number	UR-2.3
Title	Statistics of IMPEx services
Priority	S
Origins(s)	IMPEx-UAB
Type	Autonomous system activity
Description	The *IMPEx map* should inhabit a place where statistics on the available resources are displayed in any possible configuration of the dynamic view. These statistics could for example include the number of simulation runs available for a specific target, time ranges of observational data for a specific mission and similar measures.
Comments	This requirement is strictly tied to the final representation of the user interface because there must be a space preliminary reserved for the statistics until the exact parameters are known.
Related UR(s)	UR-2.1 and UR-5.2

Table 5.5.: UR-2.3 Statistics of IMPEx services

5.2.3. Communication Capabilities

The *communication capabilities* basically cover all connections to remote data sources and web service endpoints available in the IMPEx infrastructure. Together with the functionality of inter-tool communication on the client-side, the requirements depicted here match with the originally elaborated requirements for the whole IMPEx infrastructure in (Topf, 2012a, p. 25). Nevertheless the functionality of the IMPEx portal must represent a simplified and reduced set of the originating infrastructure capabilities elaborated in course of the project.

Number	UR-3.1
Title	Delegation of portal data selections
Priority	M
Origins(s)	IMPEx-PMC
Type	Interface requirement
Description	In order to be able to use data selections from the IMPEx portal acquired through data tree search and web service exploitation there must be a possibility to delegate these results to either AMDA or 3DView for further processing and visualisation. This must be accomplished by a SAMP hub which can be started from the IMPEx portal through the JSAMP* toolkit and where other tools can register to if they have a SAMP hub discovery mechanism implemented in their capabilities. It will allow the portal to publish data in the VOTable format over the hub to all connected tools. It should be pointed out at the *IMPEx map* which tool can continue using a specific data selection made by the user via the SAMP protocol.
Comments	The SAMP protocol and architecture are further described in Taylor et al. (2011). It is noted there that the hub mechanism is not designed for excessive throughput of data which must be taken into account at every possible use case of SAMP at the IMPEx portal. Besides of providing an own SAMP hub, the IMPEx portal should be able to connect to Java applications where a hub mechanism is already integrated such as TOPCAT.**
Related UR(s)	UR-2.1 and UR-4.2
	*JSAMP, a Java-based SAMP hub implementation: `http://software.astrogrid.org/doc/p/jsamp/1.3-3/` **TOPCAT, Tool for OPerations on Catalogues and Tables, an IVOA based viewer and editor for table-based data: `http://www.star.bris.ac.uk/~mbt/topcat/`

Table 5.6.: UR-3.1 Delegation of portal data selections

Number	UR-3.2
Title	Exploitation of IMPEx services
Priority	M
Origins(s)	IMPEx-PMC
Type	Interface requirement
Description	The user interface of the IMPEx portal must be able to communicate with all web service providers like AMDA and ClWeb for accessing observational data as well as LATMOS, FMI and SINP for enabling the access methods to simulated data. All defined and fully operational methods must be displayed depending on the actually selected resource since every method needs a `ResourceID` from the SPASE-based XML data trees as input. The following operational method must be included in the portal which is available from each SMDB: • `getDataPointValue` The following operational methods must be made available from FMI and LATMOS only: • `getDataPointValue_Spacecraft` and `getFieldLine` • `getMostRelevantRun` • `getFileURL` (only LATMOS) The following operational methods must be made available from SINP only: • `calculateDataPointValue` • `calculateDataPointValue_Spacecraft` • `calculateFieldLine` Ultimately the following methods must be made available from AMDA and ClWeb: • `getObsDataTree` • `getParameterList` (only AMDA) and `getParameter` • `getTimeTableList` and `getTimeTable`
Comments	It must be noted that the SMDB methods also require an URL to a respective VOTable file as input, providing e.g 3D coordinates for respective calculation of physical parameters. There is a reference implementation of a suitable auxiliary service generating VOTables maintained by FMI* which should be included at the portal too.
Related UR(s)	UR-2.1 and UR-4.2
	*IMPEx-SMDB PHP web service: `https://github.com/IMPEx/IMPEx-SMDB_php_webservice`

Table 5.7.: UR-3.2 Exploitation of IMPEx services

5.2.4. Processing Capabilities

The *processig capabilities* of the IMPEx portal itself are preliminary limited to two major functionalities since the overall goal as described in the introduction of this chapter is not to duplicate functionality of AMDA, ClWeb or 3DView. These tools are able to analyse and visualise data in 2D and 3D environments with capabilities suited for fully accomplishing the *IMPEx science cases* such as the example for "Venus magnetosphere studies" depicted in Topf (2012a, p. 17-20). The IMPEx portal therefore only provides download of selected datasets and the management of time- and coordinate-tables needed for web service exploitation, besides of delegation described in the previous section.

Number	UR-4.1
Title	Download of portal data selections
Priority	M
Origins(s)	IMPEx-PMC
Type	User interaction
Description	In addition to delegation of selected datasets the IMPEx portal must include a possibility to download the data obtained via web service communication in the VOTable and netCDF format depending on what was chosen by the user beforehand. Since a request through remote web service communication could take longer to respond the IMPEx portal should provide a *data box* in the *IMPEx map* of resources and services where an incoming response will be announced to the user and made available for download (and/or delegation).
Comments	It is not clear at the stage of the requirements engineering phase whether the netCDF format structure needs to be natively understood by the IMPEx portal or it just needs to provide URLs pointing to the resulting files of a web service call for download. The IMPEx portal should in any case favour the VOTable format because this format will also be used for the SAMP communication and one common format will save time in the implementation phase due to decreased complexity.
Related UR(s)	UR-2.1 and UR-4.2

Table 5.8.: UR-4.1 Download of portal data selections

Number	UR-4.2
Title	Time- and coordinate-table management
Priority	M
Origins(s) IMPEx-PMC	
Type	Interface requirement
Description	Time- or coordinate-tables in the VOTable format must be provided by URL-links in order to enable web service exploitation, so the portal must also offer an interface for uploading VOTables from the users desktop. These files should be placed in a server-side *data box* displayed in the *IMPEx map* for further usage in web service calls or in SAMP hub delegations to the IMPEx integrated tools. The IMPEx portal must also provide a possibility to link generated VOTables from one service on the *IMPEx map* together with a `ResourceID` from the consolidated data tree to another service available on the *IMPEx map*. Furthermore there should be a possibility given to create timetables from scratch by either entering parameters and data tables manually or by uploading e.g CSVs in ASCII format for automatic transformation. These manually created VOTables should be also made available through URLs in the *data box* from where they can be further used.
Comments	Only the requirements for uploading VOTables and mapping of generated VOTables to URLs must be taken into account in the first iteration of the implementation phase because they are mandatory for other requirements denoted with *Must* in this chapter. The upload functionality in particular was mentioned in internal discussions because it will enable a variety of use cases where the user might want to quickly obtain interpolated magnetic field values of a specific target by using locally stored coordinate-tables for example.
Related UR(s)	UR-3.1, UR-3.2 and UR-5.2

Table 5.9.: UR-4.2 Time- and coordinate-table management

5.2.5. User Interface Requirements

The user interface requirements are primarily settled around the exact definition of the previously mentioned *IMPEx map* and how the portal functionality will finally be displayed to the user. Furthermore the requirements described so far demand an integrated functionality for saving intermediate results on the server during a users working session. As already mentioned in the introduction of this chapter, the display must be as intuitive as possible since the portal should be designed for a broad range of differently skilled and trained users. Therefore there should be a way to adapt the views within the *IMPEx map* in order to make it at most useful for specific users entering the IMPEx portal as it was also demanded by the IMPEx-UAB.

Number	UR-5.1
Title	User sessions
Priority	M
Origins(s)	IMPEx team Graz
Type	Autonomous system activity
Description	At first it must be possible to have a *user session* enabled once the IMPEx portal is opened in the browser. This capability will enable the user to save selected resources or results from web service calls in a temporary cache until the browser is closed again or after a specific predefined timeout. The cached information must also include any manually created file in the VOTable format during a *user session* and be saved in the *data box* on the *IMPEx map* as previously described in section 5.2.4.
Comments	In the first iteration of the portal developments the temporary cache should be implemented as simple as possible to save resources in the engineering phase. This means that an unique user should only be identified as such based on a negotiated session identification between client and server without further credentials needed. For the future it should be considered to integrate a fully operational user management system where user accounts and related data selections are persisted on server-side indefinitely and independently of a particular *user session*. The user will then be able to return to a working session at any time in the future with a login-based authentication scheme.
Related UR(s)	UR-5.2

Table 5.10.: UR-5.1 User sessions

Number	UR-5.2
Title	User interface arrangement
Priority	M
Origins(s)	IMPEx team Graz and IMPEx-UAB
Type	Interface requirement
Description	The portal must include general information about the project as well as thematically grouped resources which are made available through the IMPEx project as mentioned in section 5.2.1. The visualisation of IMPEx services and the consolidated data tree contents must be configurable over predefined filters and represented by a virtual workbench or a thematic map in any configuration depicted in section 5.2.2. The visualisation of datasets should not be a primary goal, there must be a focus on a general purpose access to selected resources. Nevertheless it is desired to have statistics displayed for the available datasets in a topical structure as well as suggestions on how to visualise the selected datasets with IMPEx tools suchs as AMDA, 3DView and ClWeb. Furthermore, there should be a place at the user interface in the future where external tools are advertised too which can communicate with the IMPEx portal through the SAMP hub such as the IVOA-tool TOPCAT*. The virtual map must be able to show paths and indications where selected datasets can be used in the IMPEx infrastructure. There should be also suggestions on how to use the underlying IMPEx data trees and web services for third-party software.
Comments	A typical use case of the *IMPEx map* is shown with step-by-step mock-ups in appendix B. All related capabilities are depicted here as they should be presented to the user when conducting a basic workflow at the IMPEx portal. Note that it must be determined by the team before which tools are capable of further processing of particular selections like in this use case since there is no known way where the system could identify the suitable tools automatically.
Related UR(s)	UR-2.1, UR-2.2, UR-2.3, UR-3.1, UR-3.2 and UR-5.3
	*Tool for OPerations on Catalogues and Tables, an IVOA based viewer and editor for table-based data in the VOTable format: `http://www.star.bris.ac.uk/~mbt/topcat/`

Table 5.11.: UR-5.2 User interface arrangement

Number	UR-5.3
Title	User levels and custom views
Priority	S
Origins(s)	IMPEx-UAB
Type	Interface requirement
Description	The *IMPEx map* should be designed for a variety of different user groups as already indicated in the introduction of this chapter. This means there should be a way to switch between different views of the *IMPEx map*. The *educational view* should emphasize and describe the basics behind the conducted research with the available datasets as well as the capabilities of tools and services displayed in the *IMPEx map*. The focus should be on an explanatory descriptions instead of providing technical details to the available resources. This is closely tied to the tutorials and manuals provided by the documentation services of the IMPEx portal depicted in section 5.2.2. A simplified "help" function to every element in the user interface should be in place to ease the usage of functionalities. There could also be restrictions on the functionality of web services in the *educational view*. The *professional view* should inhabit the aforementioned general user interface functionalities to be able to search data trees down to parameter level and delegate data selections to other tools including the full spectrum of web service access available from all service providers in the IMPEx infrastructure. Furthermore it must provide access to technical documentation needed for external exploitation of the IMPEx data trees and web service endpoints.
Comments	This user requirement should only be considered in the first iteration of developments with respect to the provision of a flexible enough mechanism for configuration of the generated views. One standard view will be served by the IMPEx portal at first based on the default configuration illustrated in the mock-ups in appendix B.
Related UR(s)	UR-5.2

Table 5.12.: UR-5.3 User levels and custom views

5.2.6. Backend and Administration features

The *backend and administration features* are basically concluded from all previously described requirements and also have a direct influence on the priorities assumed in this chapter. The reason for that lies in the fact that an *administrative user interface* must be placed on top of the portal implementation to be able to manage the different remote resources which are dynamically loaded into the system. In addition to that the system must be capable of injecting updates on the *IMPEx data model* and its related instances in the data trees as well as the web service descriptions.

Number	UR-6.1
Title	Administrative user interface
Priority	M
Origins(s)	IMPEx team Graz
Type	User interaction
Description	The administrative user interface must be browser-based and only accessible via well-known HTTP authentication methods to the administrators of the IMPEx portal. In this interface there must be the possibility to manage the used XML-based *IMPEx configuration* file which enables the IMPEx portal to recognise all data providers dynamically, including discovery of the URLs to XML data trees, web service descriptions and online documentation. The loading of updated configuration files must also include the validation of the newly fetched XML data trees from the SMDBs as well as *AMDA* and ClWeb. Furthermore there should be a basic set of operational tests provided by the administrative user interface which can be run at any time to check the availability of service capabilities provided through the IMPEx portal. The configuration data itself must be made available to the whole IMPEx portal immediately after update so that it can be accessed at any time during the operation of the IMPEx portal by every functionality depending on it. Any included resource should be cached and persisted on the server if possible and services capabilities should be configurable with regard to their accessibility within the user interface.
Comments	The actual *IMPEx configuration* file is documented in appendix A. The test cases for checking the service capabilities e.g. the request of specific datasets should be synchronised with the test cases made in course of developments within other IMPEx tools such as 3DView.
Related UR(s)	UR-2.1 and UR-6.2

Table 5.13.: UR-6.1 Administrative user interface

Number	UR-6.2
Title	Handling of data model and tree updates
Priority	M
Origins(s)	IMPEx team Graz
Type	Interface requirement
Description	The IMPEx portal must be able to handle updates of the data model as well as contents of the associated trees in the consolidated representation. Therefore the data structure used for storing the consolidated IMPEx trees must be able to provide a flexible *IMPEx registry* for the portal where selections can be updated depending on changes done remotely at the SMDBs, AMDA or ClWeb. The *IMPEx registry* must be primarily capable of all search- and visualisation capabilities depicted in section 5.2.1 and 5.2.2. Furthermore the foreseen generalised access interface to tree contents and web service methods in the *IMPEx registry* must be designed to enable a dynamic and configurable *IMPEx messaging API* for the portal. The *IMPEx messaging API* must be primarily capable of communication- and processing-capabilities depicted in section 5.2.3 and 5.2.4.
Comments	In course of investigation during the requirements engineering phase a similar registry service provided by the SPASE group* was considered as a suitable basement for the design of the *IMPEx registry*.
Related UR(s)	UR-1.1, UR-3.2 and UR-6.1
	*SPASE Registry Server: `http://www.spase-group.org/tools/registry/`

Table 5.14.: UR-6.2 Handling of data model and tree updates

As a conclusion of the user requirements engineering phase it is important to say that the approach taken in this thesis proved to be optimal. The preliminary studies and evaluation of suitable base technologies revealed that all needed capabilities to accomplish the requirements depicted in this chapter can be provided by Scala in combination with the Play web framework. One has seen from both the technological and the users perspective which capabilities are mandatory for the successful implementation of the IMPEx portal. The technological advantages of Scala tightly fit into the user requirements when referring back to chapter 4 in particular with respect to DSLs, asynchronous programming and web frameworks. In this chapter the required key concepts needed from the users perspective were roughly summarised by `UR-6.2` in table 5.14: an *IMPEx registry* of resources and an *IMPEx messaging API* for subsequent data access which are provided by the underlying architecture of the IMPEx portal. In addition to that a certain focus must be laid on *caching* and *persistence* technologies needed for almost all mandatory requirements described here which involve access to the content of the data trees. It can be seen that a significant degree of operations is repeated in each particular use case, especially with the illustration of the *IMPEx*

map and the respective workflow during a users session. The required infrastructure at the *IMPEx portal* does therefore favour pure functional features since the simple repetitive operations of data selection and web service access must also have the capability to be chained like it is done frequently with object-functional programming features described in this thesis.

5.3. Architectural Design

Before going into details of the architectural design one must look back at the motivational part of this thesis and compare the preliminary ideas with the assumptions and conclusions which were drawn from the user requirements in the previous section. The overall IMPEx infrastructure briefly summarised at the beginning of this thesis already dictates how the most important features of the portal are to be implemented and how the principal structure of the portal architecture must be constructed. In fact the most important issue to tackle in the architectural design will be the communication with the data providers. Those are making the data trees available through XML documents and the data access methods through WSDL descriptors via the respective URLs stored in the *IMPEx configuration file*. As one can imagine this configuration marks the starting point from where the available resources and services of the *IMPEx portal* will be dynamically built up and made available. From this point, the main task of the portal will be to efficiently handle the domain-specific *IMPEx data model* based on SPASE and its instantiations - the data trees from the SMDBs. This means that the portal must integrate a custom DSL for accessing the contained metadata. In addition to that a simple adapter to the observational data model used by AMDA and ClWeb must be provided in order to enable the implementation of a unified search mechanism on both types of trees. Based on the search for particular resources in the trees a respective interface for communication using the SOAP protocol must be integrated in the portal. Together these capabilities will enable the implementation of the full spectrum of requirements depicted in the previous section through a unified messaging middleware established at the portal. It was already shown in this thesis with the evaluation of base technologies provided by Scala and its libraries that all goals are supported from the object-functional perspective so a final architecture will be built up using their distinct functionalities.

5.3.1. Overall Architecture

In order to show the adapation of the overall IMPEx infrastructure within the design and implementation of the IMPEx portal, the original figure from Topf (2012a, p. 25) was taken and modified to show the portal architecture planned in this thesis. As it can clearly be seen in figure 5.1 the common interfaces as they were planned

and implemented in IMPEx are also used here. The *Data Services* and *Compute Services* were slightly modified from the original design as they now provide both access to observational and simulation data. On the *Resource Layer* this enables access to the metadata trees from each data provider in the form of separate XML documents and in a consolidated form as explained in `UR-1.1` (table 5.1 on page 73). On the *Tool Layer* each data provider exposes a WSDL document which describes the access methods available for the resources stored in their respective XML tree. These interface descriptions are the same as they are used by the tools AMDA, ClWeb and 3DView at the current stage of development in the IMPEx project. It must be noted here that at the time of writing of this thesis only the depicted methods in `UR-3.2` (table 5.7 on page 78) can be made available through the *Data Access Service* of the portal. These methods do not cover the full spectrum of web services available through the SMDBs because their concrete implementation is still pending and will be finished until mid of 2014. Nevertheless, the integration of other methods available from the WSDL documents at the portal is planned at a later stage. They will then be subsequently made available at their finalisation via the *IMPEx Messaging API* either for direct access through the portal or with a respective reference to tools which have fully integrated the desired method.

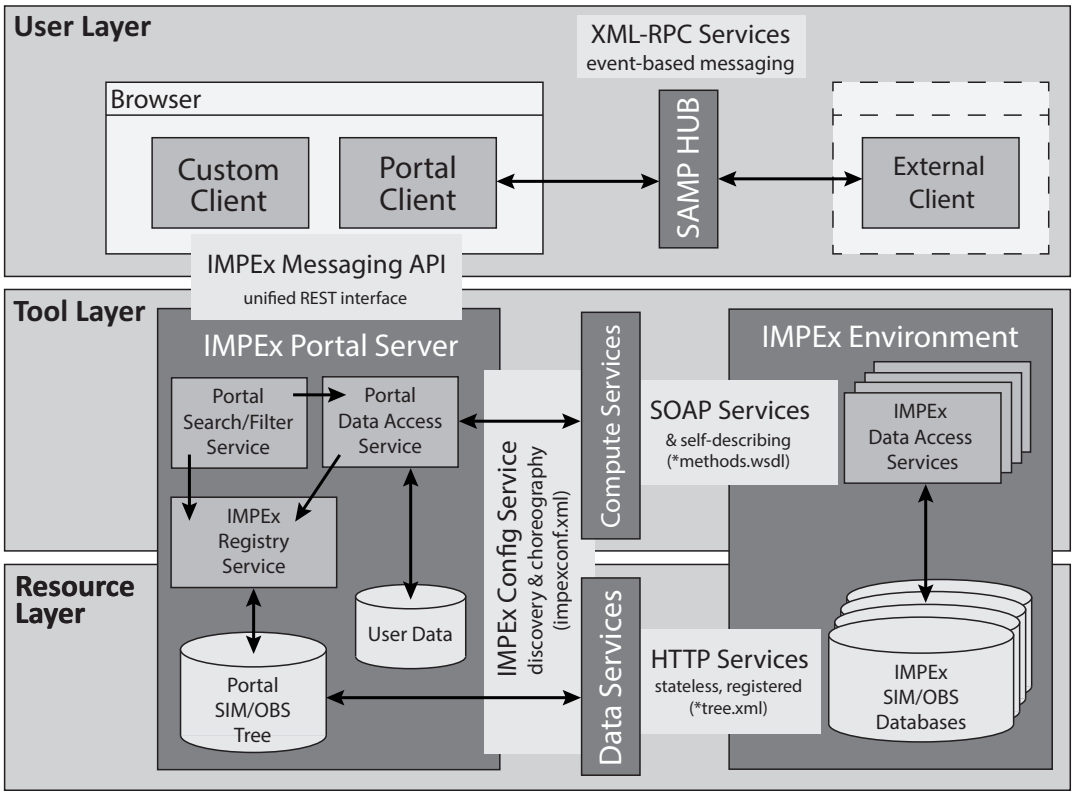


Figure 5.1.: Overall architecture of the IMPEx portal, following Topf (2012a, p. 25)

As one can see also in figure 5.1, a new service is provided by the portal server: the *IMPEx Config Service* which enables the access to the information of the configuration file both locally but also for remote connections. This service is started automatically at server initialisation and reads the content of a locally stored *IMPEx configuration*

file. In the Play framework, there is the possibility to overwrite the method `onStart` from an object `Global` in the default package where one can place any code, which needs to be executed at startup of the integrated web server. For subsequent access to the configuration information during operation of the portal it was decided to use actors from the Akka library. On one hand the Play framework already starts an Akka system automatically at startup of the web server and on the other hand it is a good alternative to classic static objects. In fact actors are classes so they can be spawned in an Akka system as often as needed and they can be also parametrised in contrast to static objects. Nevertheless each actor stays alive and is persisted in the main memory like a static object during operation of the web server instance.

As it is depicted in section 4.2 actors can receive and send messages when they are instantiated in an actor system. As soon as the actor is "activated" it is addressible through a name and one can obtain an `ActorSelection` type referring to it with e.g. `Akka.system.actorSelection("user/config")` as it is also shown in the listing below. Note that an Akka system itself is an actor and in Play every spawned actor is part of the `user` system by default hence a child of an actor named `user`. In the previous case, the name was assembled by a concrete path of the actor hierarchy `user/config`. It is mentioned in Suereth (2012, p. 213) that for hierarchical search engines a respective actor hierarchy can be from advantage which will be seen in the next section in a more complex scenario with the *IMPEx registry*.

Listing 5.1: Synchronous communication with the Config Service

```
// actor creation
Akka.system.actorOf(
  Props(new ConfigService), name = "config")

// actor selection
val actor = Akka.system.actorSelection("user/config")

// blocking message call to actor
val dbs: List[Database] = Await.result(
   (actor ? Some("database")), Duration.2.seconds)
```

An usual way to communicate with actors is to define messages with case classes, which are deconstructed by pattern matching within an actor's `receive` method for example. These classes are used to transport the payload of the messages. As soon as one has the reference of an actor with an `ActorRef` type, one can use message patterns on it which are provided by the Akka library.
In the case of the `ConfigService` actor a frequently used pattern is shown in the example above with `actor ? Some("database")` which simply returns a list of all stored data providers in the configuration. Each provider is represented by a case class `Database` and is referenced from within the actor. What is important to note here is that any returned data from an actor is wrapped in a `Future` monad. In the example above a blocking operation `Await` is used to transform the `Future` type

into a `Result` type containing the final response. `Await` has here a timeout of two seconds applied to.

Another option would be to use `async{...}` blocks to spawn an asynchronous operation on the result anywhere in the code which is not blocking. This feature is also available for the `Action` types in controller classes of Play as shown in the listing below. It maybe needed at some stage of development of the IMPEx portal to carefully think on the optimal way to handle `Futures`, especially with respect to chains of (asynchronous) operations. `Futures` can be used with basic monadic operations like `map` or `flatMap` as well as complex compositions with `for`-comprehensions. This will be of importance at the `RegistryService`.

Listing 5.2: Asynchronous communication with the Config Service

```
Action.async { // special async Action type
  // calls with the same message as before
  val future = ConfigService.request(GetDatabases)

  // future response renders view
  future.map(databases =>
    Ok(views.html.config(databases)))
}
```

It was discovered in the user requirements that the *ConfigService* must also have a capability to update the configuration. The whole XML file is therefore converted into a Scala object at startup and made available to the rest of the system through the actor. Internally the configuration is deconstructed for convenient access to frequently needed information. It will be possible to write an updated configuration to XML as required by `UR-6.1` (table 5.13 on page 84) which automatically updates the available resources at the portal. For the purpose of databinding the parser generator *scalaxb* is used because it can smoothly create case classes out of XML elements through automatically created implicit parsers similar to as they were described in section 4.1.

As one will see in the following sections, this option was taken instead of elaborating own parsers since every needed *XML Schema* was successfully translated by *scalaxb* to respective class representations and associated custom data types. As long as the structure of the *XML Schema* does not change this option is very efficient because it saves resources by ommitting manual creation of parser combinators. Nevertheless it must be noted here that updates of any *XML Schema* used in conjunction with *scalaxb* require the automatic creation of new combinators and case classes, so this part of the system cannot change dynamically during operation. This drawback was already clearly illustrated in section 4.1. In any case, this should not occur frequently because similar *XML Schemas* in the scientific domain are mostly well elaborated at the time of their release and the final integration of the IMPEx extension in the SPASE data model will provide additional stability in this case.

After enabling the access to the configuration file, all data providers are extracted and asserted to either the *observation* or the *simulation* domain together with their respective name stored in the configuration. In order to maintain the different domains of data as well as their associated data providers during operation, each tree will be fetched at startup and assigned to an actor which becomes addressible via the data provider names available from the `ConfigService`. This approach allows the complete realisation of `UR-2.1` (table 5.3 on page 75) with regard to dynamic provision of IMPEx resources at the portal. It will also provide the necessary capabilities for visualisation of the *IMPEx map* depicted in `UR-5.2` (table 5.11 on page 82). The result of a `GET` request is similarly processed as in the communication with an actor which can be seen in the listing below. This result is also a `Future` which can be handled both asynchronously and blocking. All message operations and remote service communications work with monadic operations in Play and Akka respectively. Every response is represented by the same data type which is optimal when actor messaging and remote service communication must be chained for example with `for`-comprehensions.

Listing 5.3: Example of simple GET request in Play

```
val promise: Future = WS.url(treeURL).get()
val treeXML: NodeSeq =
  Await.result(promise, Duration.10.minutes).xml

val tree: Spase = scalaxb.fromXML[Spase](tree)
```

The listing above also shows how the XML document is transformed into an object with *scalaxb* which will be described further in the next section since this is an integral part of the `RegistryService`. This service registers and supervises all previously mentioned `DataProvider` actors. It makes their metadata generally available within the system in particular for the *Search/Filter Service* and the *Data Access Service* respectively. Both services are relying on the structure of the registry which is to be defined before. They will be able to obtain information from the registry as soon as they become active in the system. The *Search/Filter Service* will be able to extract information according to the defined filter mechanisms in `UR-1.2` (table 5.2 on page 74). The *Data Access Service* will enable the communication with the web services described in the WSDLs. Each set of web services will also be asserted to the respective data provider name from the *IMPEx configuration file*. In fact an additional entity is needed for enabling the web service access because they are also maintained through generated access classes with *scalaxb*. This access interface is static at the moment and should therefore be decoupled from the dynamically assembled *RegistryService*. Since the operations provided by the generated classes are blocking, this interface might change in the future. In any case, only the fully operational methods suitable for the portal are finally made available via the interface to the *User Layer* of the portal. One feature will be operational *Data Access Service* from the beginning which is the interface to the *User Data* store. In this case the service provides tools for creation of documents in the VOTable format as well as update and deletion of

results obtained from the web services which respond in the same format on request. Since this capability enables URL access to generated files it is considered as auxiliary service of the *IMPEx messaging API*.

All the three distinct services provided by the portal server can be composed and each of the services is a self-contained component on its own. Loosely coupled, component-based architectures are also a typical use case for Akka systems as mentioned in Gupta (2012, p. 21). The *IMPEx Messaging API* finally provides an unified REST interface to all the capabilities provided at the portal hiding the underlying actor system and network communication which will be elaborated in section 5.3.3. This enables the custom composition of services on the client side and also an optimal abstraction between *Tool-* and *User Layer* because there is a low-level access possible as well as advanced data exploitation through the portal. All the respective client-side requirements from `UR-5.2` (table 5.11 on page 82) must be kept in mind here with regard to granularity of data access. The *IMPEx Messaging API* makes it possible to compose the capabilities of each portal service on the client-side. Note that the `DataProvider` actors are building up the tree database at the portal since each actor is holding it's tree as shown in figure 5.1. The XML documents are updated automatically during operation of the portal. The actor system of Akka provides a scheduling mechanism for sending specific messages to an referred actor. In the case of the portal an `update` message is again fetching the tree from the remote server and updates the actually saved tree in the `DataProvider` actor. This functionality is seen as sufficient for automatic caching of the remote trees in the initial version of the IMPEx portal in order to meet the requirements in `UR-6.2` (table 5.14 on page 85). In addition to that the actor will backup the trees on harddisk once after each update which is also enabled through the scheduler at startup.

5.3.2. The IMPEx Registry

The *IMPEx Registry Service* is a mechanism which needs to be available at startup of the portal server after the resources are downloaded and made available to the system. As already mentioned this service is able to collect the information from the `DataProvider` actors once they are registered there. In fact the `RegistryService` will automatically dispatch selections of the trees after the called `DataProvider` actor has transformed the XML data into an object representation with *scalaxb*. It is also an actor on its own which enables a domain-specific access interface to metadata supporting the two different data models used for observational and simulation data. As already mentioned, the `DataProvider` is identified by the registry through it's unique name stored in the *IMPEx configuration file*. Furthermore, the databases are distinguished by their type (simulation or observation) in the `RegistryService` to be able to handle the two different data models and their object representations. The object representations are returned from the registry either in the original XML representation with *scalaxb*'s `toXML[T]` method or by implicit writers and readers in

the JSON format. The latter functionality is in particular useful for the IMPEx portal client. The features of the registry are provided through a unified access interface respecting the hierarchical structure of both data models which means they are enabling a basic domain-specific language for browsing the content of the XML tree instances.

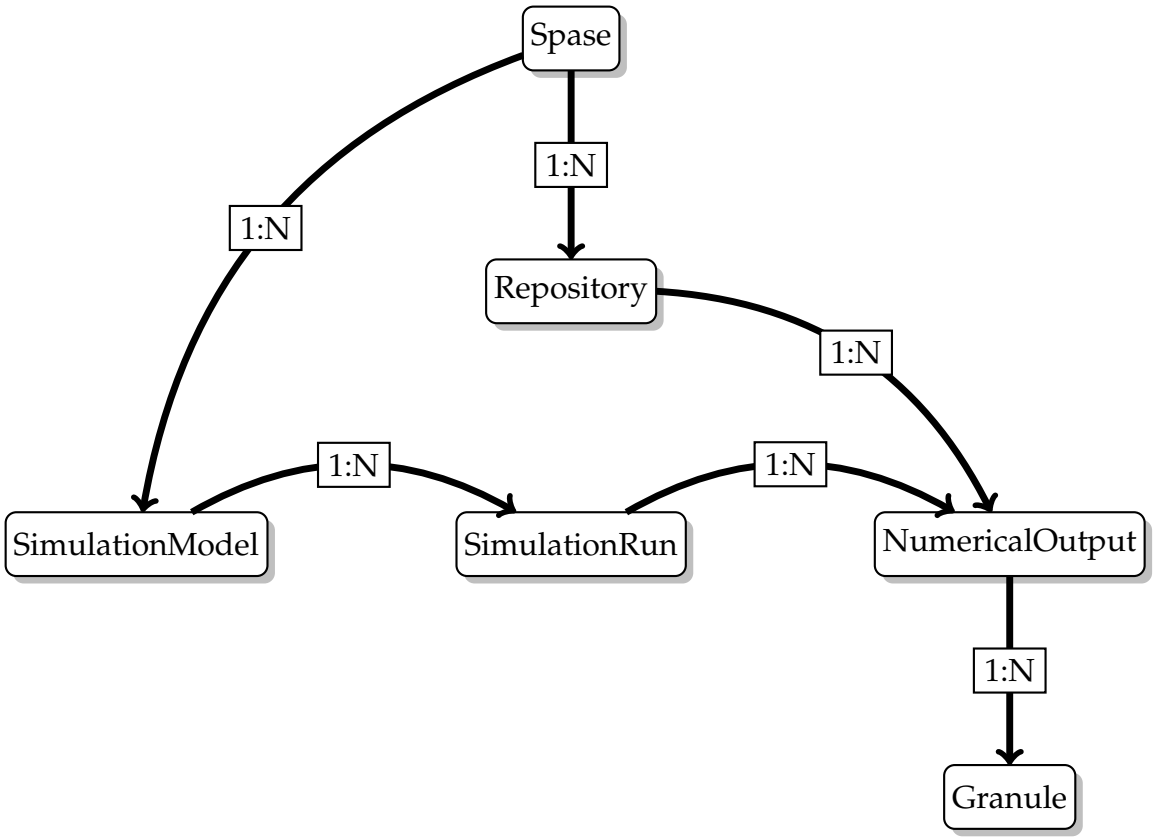


Figure 5.2.: Hierarchy of SPASE based simulation model

Figure 5.2 and 5.3 illustrate the hierarchy and relationships of the key elements of each data model. The elements for the SPASE-based simulation model represent the set of resources which can be obtained from the SMDBs. In this case the hierarchy is not described with a classical XML-tree structure but the elements are identified by a `ResourceID`. Every element depicted in figure 5.2 is a direct child of the `Spase` XML-element. Through the structure of the `ResourceID` element which is an URI there can be informal hierarchical relationships between resources and every URI is built up from *scheme://authority/path* (The SPASE Consortium, 2011, p. 7). In IMPEx all SMDBs are represented with the infrastructure resource `Repository` as *authority*. The `SimulationModel` is only informally tied to the repository through its `ResourceID` for example *impex://LATMOS/Hybrid*. All other elements are formally tied to at least one other element. This means that `NumericalOutput` for example has a reference element pointing to a `SimulationRun` and to `Repository`. The reason for using such a reference model within SPASE is that components can be composed and distributed easier with a flat XML hierarchy (The SPASE Consortium, 2011, p. 8). Through the structure of the `ResourceID` all elements are informally

related to each other so they can be automatically grouped. This also helps when the tree must be constructed dynamically beginning from the root node. Furthermore it is easy to provide a consolidated SPASE tree as it will be needed for `UR-1.1` (table 5.1 on page 73).

The *Registry Service* at the portal will use this path-based hierarchical structure of SPASE to resolve the resources as follows:

Listing 5.4: URL path for the SPASE data model

```
1 GET /{ProviderName}/{SimulationModel}/{SimulationRun}/
    {NumericalOutput{/{Granule = None}
```

A simple `"simulation"` message to the actor will return all available `Repository` elements. The `ResourceID` of a specific SPASE repository (or any infrastructure resource) is tied to the name stored in the *IMPEx configuration file* and can be subsequently used to obtain the respective `SimulationModel` elements and so on. Note that `Granule` is an optional element which is not provided by every SMDB tree. In principle, all available elements of the SPASE data model can be translated by the registry and *scalaxb* respectively, but for now only the mentioned originating- and data-resources will be integrated here. Nevertheless IMPEx-specific elements depicted in the path have nested elements which include domain-specific metadata needed for the *Data Access Service* as well as the *Search/Filter Service* [4]. The latter service will automatically request the respective parent elements at the *Registry Service* based on its implemented search- and filter operations and return a recursive version of the tree selection.

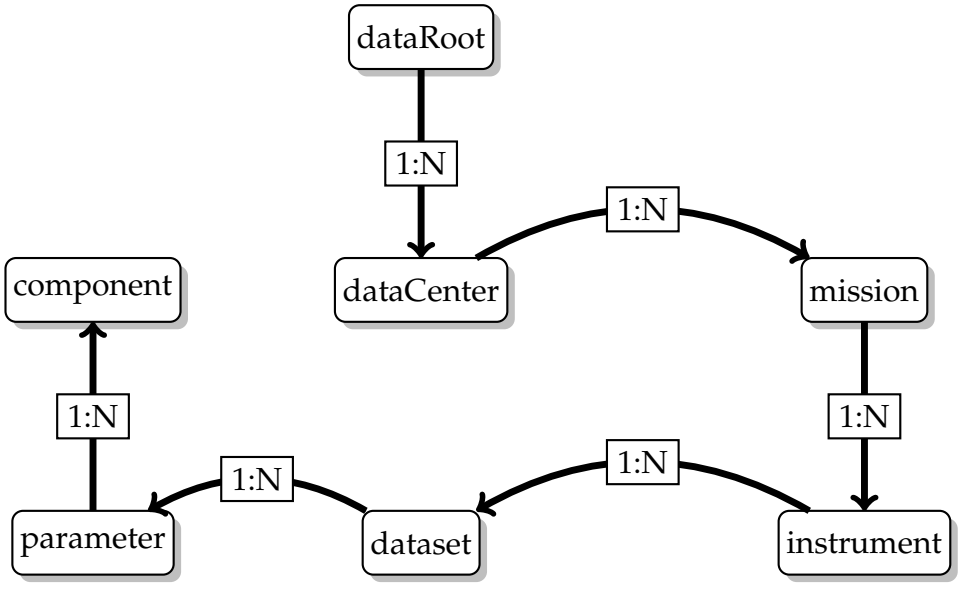


Figure 5.3.: Hierarchy of proprietary observational model

[4] Full documentation of the SPASE data model including all elements of the IMPEx extension is available at: `http://impex.latmos.ipsl.fr/tools/DataModel.htm`

The observational data model is used for accessing data from ClWeb and AMDA. As shown in figure 5.3 the structure is fundamentally different. Each element in the hierarchy is nested within its respective parent element. The XML-tree structure dictates the path and each element has attributes as direct identifiers. In this case the resolving URL path can be constructed as follows:

Listing 5.5: URL path for the observation data model

```
GET /{ProviderName}/{Mission}/{Instrument}/{Parameter}/
  {Component = None}
```

A simple message "`observation`" to the actor will return all available `dataCenter` elements which are again translated to the provider name stored in the *IMPEx configuration file*. Here the attribute `xml:id` identifies e.g. a specific `dataset` or `instrument` element. A `DataProvider` actor tied to an observational data tree deconstructs the path shown above into respective selections, so that only portions of the hierarchy can be displayed similar to as it is done with the SPASE data model. As already mentioned, both data models are dynamically translated into objects with *scalaxb* in the `DataProvider` actor. The `RegistryService` actor is in particular responsible for distinguishing between *observation* and *simulation* data providers because it needs to know which objects are to be send by the actors. It must be noted here that all actor classes defined for the IMPEx portal are accompanied by a companion object which implements the message patterns described in section 5.3.1. All the communication to the related actor is usually done through this object. In the case of the `DataProvider` actor all defined operations take an `ActorRef` value as input which needs to be selected by the `RegistryService` actor before.

As soon as the registry gets a request on a specific `ProviderName` for example it selects the respective actor and sends a message to it. With multiple actors as it is the case in the IMPEx portal `for`-comprehensions provide a lot of possibilities when working with `Futures` as can be seen in the listing below. In order to obtain all `Repository` elements from a specific domain (*simulation* or *observation*) for example, the configuration service is requested for the respective list of databases and then each actor is asynchronously called by the registry. The partial results `databases` and `providers` as well as the combined result of the `for`-comprehensions are yielded into a `Future` monad. In this case the result must be also flattened in the end since `provider` returns a sequence of `Futures` nested in a single `Future`. As already discovered any `Future` type can easily be processed further in respective controller actions of the Play framework.

Listing 5.6: A typical call to a DataProvider actor

```
def getRepositoryType(dType: Databasetype):
    Future[Seq[(Databasetype, Any)]] = {

  for {
```

```
      // requests databases from specific type
6     databases <- ConfigService.request(
        GetDatabaseType(dType)).mapTo[Seq[Database]]
8
      // requests all repository elements one after
10    // another with Future.sequence
      providers <-
12      Future.sequence(
          getChilds(databases) map {
14        provider =>
            DataProvider.getRepository(provider) })
16
     // result is flatten due to nested Futures
18  } yield providers.flatten
  }
```

In order to prevent reloading of larges amounts of metadata from the trees, the respective `Action` method calling the registry actor will be cached as a whole by the Play framework cache API as it is illustrated in Hilton et al. (2013, p. 269). It is noted here that REST-based URL routes defined in Play must be static and not contain e.g. `GET` query parameters in order to be able to use the cache API for entire actions. Especially with the requirement to provide download of large consolidated XML trees for the *observation* and *simulation* domain of IMPEx this cache can be used system-wide for all users. Dynamic routes and their respective `Action` methods will need to combine the cache with the HTTP-based *user session* which can be initiated by calling any of the implemented actions in this Play application. As soon as the user accesses one of the cache-enabled actions, their response will be made accessible through a key-value store for each subsequent request during an *user session* as it was requested in `UR-5.1` (table 5.10 on page 81).

With the capabilities of the *IMPEx messaging API*, the *Registry Service* is finally provided to the user through a REST-based interface mirroring the hierarchy paths of the two different data models. For convenience the two distinct URL-paths will get a prefix which identifies their respective domain: `/simulations/...` and `/observations/...`. Altogether the actor-based approach in the backend of the IMPEx portal is fulfilling the requirement to provide a unified access interface to both types of metadata. As soon as observational metadata integrated in IMPEx becomes available in SPASE-compliant format the `DataProvider` actor can be refactored and the proprietary format can be removed from the data access interface. The hierarchy-based access scheme depicted here is also reflecting the key characteristics of a DSL and this is also in-line with what is proposed from the SPASE community as indicated in the user requirements. Furthermore actor-based systems are loosely-coupled and they can be distributed across the network with the same messaging capabilities as on a local system. This fact is considered as a main driver for usage of the Akka concurrency library in this project.

The preliminary class diagrams of the mentioned actors and companion objects in this section is illustrated in appendix C. Note that these classes are not fully elaborated at the time of writing this thesis and some basic access methods for the registry are still missing. Nevertheless it is clear that both the *Search/Filter Service* and the *Data Access Service* can easily be embedded in this actor-based scheme and act as separate services depending on previous selections from the *Registry Service*. The communication with these three distinct services will therefore be also unified through the *IMPEx messaging API* and made available to the portal client but also for remote access with a REST interface.

5.3.3. The IMPEx Messaging API

As a final step of the fundamental architectural design, the connection to the service-oriented architecture of IMPEx must be completed by introducing the before mentioned unified access interface called the *IMPEx messaging API*. It is clearly demanded by the community as seen in the user requirements to provide such an access interface to both the metadata of the trees and the services exploiting datasets of simulations and observations within IMPEx. In fact all functionalities implemented by the IMPEx portal client are solely relying on the REST proxy provided by this messaging API. As also discovered in the theoretical chapters of this thesis most of the client-side JavaScript frameworks are managing their access to resources through REST-APIs. The reason for that lies in the fact that the stateless nature of the HTTP protocol allows to easily combine requests on the client, so the granularity of access to the capabilities of the portal server can also be controlled on the client-side. Every component in the REST-API is independent and scalable with regard to their possible interactions between components.

This fact allows the creation of a client as it is required in the sense that one can chain REST requests needed to enable workflows as shown with the *IMPEx portal map* in appendix B. It must be noted here that the default view templates of Play are only used during the development of the portal in order to be able to test the respective parts of the interface e.g. the registry or the data access mechanism. This will be indicated in course of the definitions in this section. The portal client will finally be completely abstracted from the portal server in that respect that a suitable MVC framework e.g. *angular.js*[5] will deliver the required dynamic representation in HTML based upon communication with the REST-API. This approach will also show the usefulness of the unified interface for external applications. By using a REST-architecture there is also the possibility to easily combine different elements of the API and realise workflows with respective engines like *Taverna* as noted in the user requirements. So there is no restriction here with regard to using *Taverna workflows* with one REST-interface instead of scattered SOAP-based services.

[5] Angular.js, an open-source JavaScript framework for single-page web applications, see: `http://angularjs.org/`

All the distinct capabilities provided through the API are self-contained which means they do not explicitly rely on any other capability provided at the level of this interface. Nonetheless the basic capabilities of the *IMPEx Registry Service* are implicitly serving information which is mandatory for using more complex services as indicated in the previous section. This means a typical workflow on the client will at first ask the *IMPEx Registry Service* and then choose a suitable service from the *Data Access Service* for example. Since the underlying Play framework of the portal is focusing on a clear definition of a REST-API for clients connecting to it, it is a also logical conclusion in that respect to use only a basic REST-architecture with its predefined HTTP request methods for the *IMPEx messaging API*. In fact this thesis also follows the paradigm of "URL-centric" design which is favoured by the Play framework. The whole architecture is built bottom-up with a clear definition of the HTTP routes needed to browse, select and access datasets from the trees through the respective web services. This means also, as already mentioned in section 4.4 that the focus is on a resource-oriented architecture rather that a service-oriented architecture. It must be noted that the structure of this foreseen protocol layer is perfectly fitting into the design of the SPASE data model hierarchy with its `ResourceID` elements as mentioned in the previous section. Every element delivered by the data models can be directly mapped on a resource which is made accessible through an URL corresponding to its path in the respective XML hierarchy.

Nevertheless the SPASE data model is slightly adapted internally and by default all simulation models are automatically related to a repository stored in the same XML tree of a specific data provider. The repository, as already mentioned is also automatically tied to a `Database` element stored in the IMPEx configuration. All the following elements in the SPASE hierarchy are connected to their formally related parent element. Every call to the registry through the messaging API returns the respective parent element for example `SimulationModel` and all `SimulationRun` child elements. The messaging API exposes this information in a way which shows the child-parent relationships to the client based on the URL paths in the IMPEx portal. This means that child elements are identified by their relative path inside their parents. As soon as one has the relative `ResourceID` of a simulation model for example `HYB` one can obtain the child elements which are in this case simulation runs as shown in the listing below.

Listing 5.7: Example of API calls to the Registry Service

```
1 // returns all Repositories
  GET /simulations
3
  // returns all dataCenters
5 GET /observations

7 // returns all SimulationModels
  GET /simulations/FMI
9
```

```
  // returns all SimulationRuns of LATMOS
11 GET /simulations/LATMOS/HYB
```

The API in this case returns a simple JSON array which contains a JSON object with metadata of one simulation model and an abritrary amount of JSON objects representing the related simulation runs by default. The key for the model metadata is given by its URL path at the portal based on its relative `ResourceID` e.g. `LATMOS/HYB`. The keys for the simulation runs metadata are given by their relative path within the model e.g. `LATMOS/HYB/Mars_14_01_13_SimRun`. With that capability the clients connecting to the portal can easily browse through the hierarchy without loosing the parent-child context within the observation or simulation domain. In the development version of the portal a respective view template will be displayed in the paths to enable browsing through the results in a basic manner with URL lists. Note that some of the relative `ResourceID` identifiers must be translated internally since they contain slashes which would result in incorrect URL paths at the portal. They are replaced with their respective HTML entity code when the related API controller action is returning the information obtained from the registry service. The actual metadata is stored as a whole in the respective value of the JSON object so the original SPASE `ResourceID` of each element is retained in the results. Over the same routes defined in the listing above it is possible to obtain the tree in its original SPASE format with the option to return also the parent elements in a recursive manner. This is a mandatory feature because the original XML format might be requested by other tools connecting to the *IMPEx Registry Service*. For the observational data the `xml:id` will be used instead for identifying the path. All other capabilities of this type of data will be subsequently handled in the same way as simulations through the API. Both the `ResourceID` and the `xml:id` of specific elements will be needed for further exploitation of the *Data Access Service*.

Since the IMPEx infrastructure is providing SOAP-based web services for data access after selection from the trees through the *IMPEx Registry Service* there must be a possibility to transform this services into resource-based entities within the messaging API. This means according to the definitions of Daigneau (2011) that the portal in fact respresents a mixture of a pure resource API with parts of a message API in the form of SOAP services. The portal must be able to act as a SOAP client internally within the *Data Access Service* which dispatches requests to the according data provider. The REST-based interface finally provided by the portal does not trouble the user with manual handling of specific SOAP messages. This means there is no particular technical knowledge needed about the SOAP protocol as well experience in working with the IMPEx data model. The interface smoothly translates the SOAP services into a dynamic an hierarchical map of capabilities which is structured similiar to the *IMPEx Registry Service*. In fact the responsive part of the messaging API can be seen as a kind of REST controller enabling access to a set of similar resources from a particular domain which is always the case with the methods provided by the WSDLs available in IMPEx. Nevertheless this controller is only manually tied to one specific type of element stored in the trees. This information is not available from the WSDL files

so the API for accessing the *Data Access Service* must be able to obtain the respective parts of the XML tree before calling the web services to be usable without explicitly requesting the *IMPEx Registry Service* before. For example, all methods depicted in `UR-3.2` (table 5.7 on page 78) which are available from SINP can only be called with a `ResourceID` from one of the simulation models stored in the SINP tree. The used element type in the SOAP message defintion is only reflecting its nature as a general `ResourceID` from SPASE data model.

The `DataAccessService` is a static object which is organised according to the content of the *IMPEx configuration file*. Every data provider listed in the configuration can be addressed through this object and their related set of methods is made available through a dedicated controller of the Play framework. These methods are provided by classes which are generated from the current WSDLs through *scalaxb*. As already mentioned in the previous section this is the reason for a separate entity in the architecture which makes the SOAP services accessible. Although the code generated by *scalaxb* is flexible with a variety of possibilities how to use the services with its used *cake design pattern*[6] as described in Perrett (2011b) it is static and cannot be changed during operation as opposed to the metadata trees provided by the `RegistryService`. Furthermore it is also needed to think about ways to encapsulate the blocking operations into asynchronous blocks of code when calling web services.

In this regard it makes sense when the respective paths to the controller actions request the needed set of elements from the `RegistryService` in parallel and combine basic operations by default on the server to simplify the workflow on the client-side. In the listing below there are three examples of the *IMPEx messaging API* which define URL paths to methods without any query parameters. The fourth example indicates how the web service is finally called in principle.

Listing 5.8: Example of API calls to the Data Access Service

```
1 // returns a list of methods available from SINP
  GET /methods/SINP
3
  // returns the list of applicable simulation models
5 GET /methods/SINP/calculateDataPointValue/inputs

7 // returns the list of applicable numerical outputs
  // returns the list of available VOTables in the session
9 GET /methods/LATMOS/getDataPointValue/inputs

11 // sends a web service request
  GET /methods/LATMOS/getDataPointValue?<params>
```

[6]The cake design pattern is enabling dependency injection by using Scala's mixin strategy, see also Odersky and Zenger (2005)

The first example just returns a JSON encoded list of the available methods of the respective access class provided by *scalaxb* for the WSDL of SINP. As it is noted in the example, a list of elements is returned which are applicable to the web service through the sub-path `inputs/`. These can also be elements of different types for example with the method for `getFileURL` which can take references to simulation runs or numerical outputs as input. The base path to a method will for the development version of the portal provide a form from where the interface to the method can be tested. Some metadata will be needed as descriptive information in the client's user interface in the end but the `ResourceID` of a single element is mandatory for the web service call in any case. The final call on the respective web service is handled via `GET` query parameters which are sent to the base path. Those parameters are in the first iteration only defined by a written *REST API documentation* which will be finalised during the implementation phase. Internally *scalaxb* is used to dispatch the request through the respective acccess class as it is shown by example below with a call to the `getParameter` method from the AMDA database.

Listing 5.9: Example of web service call with scalaxb

```
// create service binding
val provider = new Methods_AMDASoapBindings with
    scalaxb.SoapClients with
    scalaxb.DispatchHttpClients {}

// call method of binding
provider.service.getParameter(
     startTime = "2011-08-10T00:00",
     stopTime = "2011-08-11T00:00",
     parameterID = wnd_swe_he, sampling = None,
     userID = Some("impex"), password = None,
     outputFormat = Some("VOTable"),
     timeFormat = Some("ISO8601"), gzip = None)
```

The portal client will provide a respective user interface for using the methods which is built up manually by using the above mentioned routes and the API documentation. It is considered to improve the responses of the URL paths listed above in the future to include JSON encoded information of all mandatory and optional parameters related to the method through the `inputs/` sub-path based on the definitions stored in WSDL file . Then it will also be possible to use the defined custom types in the WSDL file directly on the client because their restrictions and possible values will be encoded in the response of the `inputs/` sub-path as well. For that feature it will be needed to decide on a shared XML Schema for the IMPEx WSDL files which include common types (e.g. spacecraft abbreviations needed for `getDataPointValue_Spacecraft`), but also message signatures which are the same at every SMDB (e.g. the method `getDataPointValue`).

Results obtained from the web services are always returned in the VOTable format although there are different types available at the portal. In this case the messaging API must also be unified to simplify other operations implemented within the `DataAccessService` object. The VOTable files are saved on the portal server in the *User Data* store as soon as they are returned from the web service. They are addressible through the *SessionID* of the users working session subsequently. As soon as the user calls the `userdata/` path, the portal server returns a list of URLs pointing to all previously acquired VOTables which can then be downloaded as depicted in `UR-4.1` (table 5.8 on page 79). Furthermore there will be the possibility that client implementations may `POST` files in the VOTable format to the path `userdata/create` which returns a link to the storage location on the server.

In conjunction with the requirements for the portal client in `UR-4.2` (table 5.9 on page 80) this part of the API will finally provide a path for updating VOTable files. In the development version `userdata/update` will enable creation of VOTables from scratch. Internally the VOTable will be created and transformed for updates also with *scalaxb* and it's respective XML Schema from IVOA[7]. In some cases the mandatory input parameters include URLs to VOTable files e.g with `getDataPointValue`. In this case the available files from the users working session available in `userdata/` will be suggested by the `inputs/` route shown in the previous listing. This feature can later be extended to support more functionalities for creation of VOTable files e.g. from CSV files. The reason for using a unified response format from the web services lies in the fact that the results must be further used by the SAMP hub delegation mechanism which is using this format by default. Such a delegation mechanism is also a novelty at a central access point within the IMPEx infrastructure since it is only available at tool level up to now with very limited capabilities as shown in `UR-3.1` (table 5.6 on page 77). Note that the usage of URLs in GET queries is generally discouraged because this means the request is relying on other resources whose availability cannot be guaranteed. This drawback must be considered when working with this part of the API on the client.

Another use case on how the functionality of the *Data Access Service* can be realised in a workflow is as follows: The user is browsing the registry with the respective part of the messaging API until a specific `NumericalOutput` element for example is reached which can be used as parameter for a set REST methods. As soon as the controller of the *Registry Service* is asked to display the metadata of this element, it sends a request to the *Data Access Service* before to obtain information on which methods are available for this particular type of element at the respective data provider. This feature will only make sense as soon as this part of the messaging API providing access to the methods is returning the signatures and custom types from the WSDLs dynamically to the client. The improvements needed in order to make the WSDLs more self-descriptive and unified is subject of further studies within the project lifetime of the IMPEx project.

[7] VOTable Schema v1.2: `http://www.ivoa.net/xml/VOTable/v1.2`

6. Results and Discussion

At the beginning of this thesis a concise overview of the fundations of functional programming was shown with the introduction of lambda calculus. The related studies were seen as necessary for further evaluation and elaboration of the research question which asked for suitable elements of functional programming required for the development of the IMPEx portal. This approach in particular was needed to obtain a deeper understanding of the different mindset followed by the functional paradigm and its focus on functions as a versatile component of software design. At first the attention was drawn on the different reduction strategies provided by lambda calculus which provide a tool for simplifying *λ-expressions* in this formal language. This study enabled a clear understanding of the evaluation strategies used within modern programming languages which are based on these reduction strategies. It was also discovered during the theoretical chapters of this thesis that these fundamental cornerstones are necessary to develop memory efficient code so that there is no drawback in using functional style over imperative style when building up a software architecture. The reasoning behind that was clearly shown based on the historical growth of functional programming, through the increase of computing power and the evolution of parallel and concurrent software design as a consequence. Furthermore a few use cases were illustrated where imperative features like I/O and creation of side-effects can be translated to functional programming with lazy evaluation.

The most important issue when working with *λ-expressions* is discussed in particular at the beginning: Everything in lambda calculus can be built up with *λ-expressions*. This means that even control structures and primitive types are represented with this basic unit in lambda calculus. This is an interesting fact which was also re-discovered in the principle assembly of the object-functional language Scala. In this case everything is built up with objects. In both cases a clear overview of their advantages was shown so that one can understand their usefulnes in real world applications like the IMPEx portal in this thesis. It must be noted here that it needs time to achieve a satisfying transition in the software design by using the provided object-functional capabilities instead of their imperative counterparts. The code is structured significantly different in most of the situations depicted here. It was also elaborated in course of this thesis why the pure functional paradigm has not been considered widely for commercial applications until recently. The reason for that was presented clearly: The way functional programs are designed need more attention from the programmer because with functional programming there is a certain degree of overhead created which is slightly less efficient compared to imperative programming. Nevertheless

this argument is more and more vanishing because today programmers do not need to think about every single allocation of memory cells anymore. In fact there are much more resources available and systems are additionally more scalable then they used to be in the past. In any case the seamless integration of Java libraries help the developers to smoothly change their way of programming within Scala.

All the principle elements of functional programming such as first-class values, higher-order functions, recursions and functional composition let the developers design their software in fact in a more efficient way as in imperative programming. Together with immutable values for example it is possible to write self-contained pieces of software which do not depend on other parts of the software. These pieces are also more fail-proof as in imperative programming because they generally avoid side-effects. Those side-effects include local, mutable variables in functions and classes which are generally discouraged in pure functional languages as well as the Scala language. In course of theoretical studies in this thesis a gradual transition was undertaken with the movement from imperative control structures to pattern matching for example. Furthermore a set of possibilities were presented how functions can be composed to achieve more complex, self-contained units. What was from equal importance in the studies of this thesis was the elaboration of different typesystems coming from functional and imperative programming. At first the focus was laid on generic functions with type parameters originating from Haskell which can be found again in Scala. Furthermore a clear distinction was drawn between static and dynamic typesystems and their different approaches were shown on the basis of findings in functional programming languages. In the case of Scala it was proven to be optimal to use a static typesystem which helps in the realisation of *referential transparency* and *type inference* subsequently. It is clearly shown in this thesis that these functional principles are fundamentally useful for the realisation of DSLs in service-oriented systems because they are ensuring type-safe operations.

In the case of the IMPEx portal the usage of polymorphic data structures as depicted in the theoretical chapters with their recursive access- and manipulation procedures is also seen as optimal with regard to the foundations of the *IMPEx data model*. Especially with the vast usage of XML structures a respective tree-like representation is used and their implemented higher-order functions enable an easy deconstruction of the stored metadata. Together with the object-oriented capabilities of Scala this allows the implementation of abstract data types which can interpret the data model and manipulate the content of the XML trees subsequently. In addition to that these data types provide a set of access methods to datasets available within the service-oriented architecture of IMPEx. Through type parametrised class hierarchies enabled by Scala it is also possible to abstract different operations which can be defined type-independent. The extendibility of the portal's functionality is additionally given by the object-oriented subtyping relationships between the implemented and instantiated actor classes for example. Subtyping is also from particular importance with regard to the IMPEx data model because every element in the XML hierarchy is a distinct Scala object but shares a common set of access methods with the other XML en-

tities in the end. The IMPEx portal architecture especially uses implicit and dynamic method dispatching together with pattern matching and case classes to deconstruct this XML hierarchy. This functionalities are solely provided by the parser generators from the *scalaxb* library.

Altogether the domain-specific model of IMPEx and its distributed architecture is hidden beneath an unified access layer which defines an embedded Scala-based DSL. Its semantics represent a common query language which can be used in a variety of use cases ranging from metadata extraction to remote dataset exploitation. The internals of the DSL are built up with modules which capsulate these separate functionalities. These modules are almost throughoutly captured within a monadic data structure as it was shown by the design of an actor-based architecture at the IMPEx portal server. It is important to note that monads allow single-threaded operations and their results are always represented by their particular return type such as `Future` or `Option`. The general outcome of the architectural design is a component-based architecture which is able to interact internally and is providing a set of operations to the outside world over a REST-based interface. The internal interaction is dominated by the message based communication between the actors through their companion objects. This communication can be composed based on monadic operations as it was shown with examples taken from `RegistryService` actor. Since the messages are almost always represented by case classes, subtying also helps here when dispatching the requests over companion objects with pattern matching. This approach clearly shows the advantages of the combination of these particular functional- and object-oriented capabilities. With this fusion it is possible to program the flow of data explicitly by sending around case class instances in the actor system which contain the required information of the request. One can avoid almost every imperative control structure and local variables with this fundamental approach and increase the fail-proofness of the overall architecture in the end. In that respect it was seen as particularily important to evaluate every needed object-functional capability of Scala before, based on the basic aims and goals of the IMPEx portal.

Since it was not the primary goal of the IMPEx project to provide a unified access interface from a centralised service in the beginning there was also the necessity to create an use case in this thesis which emphasises the advantages of actor-based systems. They are an important aspect for reasoning about Scala and it's usefulness within web-based scientific tools. It was clearly shown in the architectural design that Scala does not make any difference between calls to remote web services and actor communication with regard to their return types. In addition to that actors can communicate locally and remotely which is certainly an aspect to think about in future projects because it would enable type-safe communication with an own protocol instead of the currently used SOAP based interaction. Because of their asynchronous nature which is based on an event-driven model, exception handling within actor systems is designed more efficiently as in imperative programming. No operation is blocking by default and there are monadic data types returned in any situation. Furthermore this approach is more flexible when thinking about bus-based messag-

ing systems or transaction-based communication which are certainly considered for future extensions. In that respect it is needed from the beginning to create unified error handling mechanisms which can be easily understood by the clients connecting to the IMPEx portal. The server-side part must additionally be capable of removing non- responding services from the access interface dynamically to avoid web service calls to data providers which are temporarily offline. This capability is also provided by the actor-based system because there is a monitoring functionality embedded in the IMPEx portal architecture which checks and updates information stored within the actors frequently.

The Play framework finally unifies all the capabilities of the IMPEx portal over a REST interface as it already was clearly shown in the practical chapters of this thesis. Since every distinct capability is built up with non-blocking actions here out-of-the-box it can be easily connected to the previously defined actor-based system of the portal. This means that the controllers of the Play-based application are also using `Future` types to return information over the provided HTTP interface to the client. Therefore they can use the same compositional capabilities as they are incorporated in the internal communication of the actor-based system. Since the XML trees are only persisted in the main memory at the moment the response times of the system when loading selections of metadata can be seen as most optimal. Every element of the tree can be requested dynamically in the respective action context and made available to the calling client asynchronously. There is also a fallback mechanism included which makes regular hard-disk dumps of the XML documents in the background so that the metadata trees can be loaded into memory again at any time independent of their availability over the network. It must be noted here that the client is also constructed based upon an event-driven model with service-based components so the exploitation of the REST interface can be designed conceptually similar to the internal architecture of the portal server providing it. Together with JSON encoded data communicated through the REST interface this allows an agile elaboration of the client based functionalities after the successful implementation of the backend capabilities. The reason for that lies in the fact that the data model specific operations can be translated easily to the JavaScript context with this approach. It was concluded that the REST interface must also provide a native XML representation of each response from the system. The used JavaScript libraries for the portal client can finally construct the respective HTML based representaton dynamically by combining operations from the REST interface. In order to justify this design decisions a preselection was made with the *angular.js* MVC framework in combination with the *TypeScript* system in course of elaboration of the portal architecture based upon previous experiences by the author. It was also discovered during the elaboration of the portal architecture that sometimes it will be needed to encode state in subsequent communication with the stateless REST interface. Fortunately there is everything made available from the Play framework out-of-the-box too with its key-value based caching system and its distinct session handling capabilities which together are seen as sufficient in the first iteration within the development phase of the portal.

7. Conclusions and Outlook

It can be clearly seen from the results of the practical chapters that the aims and goals of this thesis were fully accomplished. In the end a prototype using a significant amount of object-functional features was presented here. This prototype is fully inline with the user requirements depicted in this thesis and furthermore it does also comply to the general properties of a *Virtual Observatory (VO)*. In fact the original service-oriented architecture is covered by an additional unified layer here which allows access to metadata over a registry similar to respective implementation of the SPASE group. Nevertheless some issues were encountered which still need to be solved in course of the ongoing software development process. Despite of the vast possibilities of Scala with its complex typesystem including generics and subtyping as well as its advantages with implicit parser combinators there are still some issues with the data binding of XML documents. The *scalaxb* library is still in an immature stage and especially complex XML types are not transformed in an optimal way to Scala objects according to the experiences made at the beginning of the development phase. Furthermore the inclusion of the web services over the included WSDLs in IMPEx is not optimal at the portal because the parser generator of *scalaxb* is creating the contained access methods in a static way. Every call to a remote SOAP server is also blocking which cannot be avoided at the moment. This is an issue which is again related to the immature stage of *scalaxb*. From the IMPEx perspective the WSDLs also have no common parameter types with a few exceptions which are imported from the XML Schema of the *IMPEx data model*. This is certainly an aspect which needs to be discussed in the community in order to unify important parts of the definition in a dedicated XML Schema for the types in the WSDLs of the IMPEx infrastructure.

It was also seen as difficult to move away from the SOAP protocol and provide everything only through the HTTP methods used in a resource-oriented architecture. One has seen that a fully compliant REST interface would avoid `GET` queries which can only temporarily exist. This is the case at the IMPEx portal and will probably not be changed because it is simply not foreseen to hand over e.g. VOTable files to remote services in other ways than based on URLs. These files are typically generated during a users session at the portal and are therefore deleted after a certain time. In addition to that the `ResourceID` URIs used within the SPASE-based data model are sometimes not elaborated well so they must be transformed in the meantime so that they can be translated to URL paths properly. Due to the flat hierarchy of the SPASE data model it might be needed to move away from tree-like respresentations to key-value based maps. Altogether more emphasis must be placed on the definitions of the *IM-*

PEx configuration file and on the unification of the simulation and observation data model until the end of this project and beyond. It was shown by example here that on one hand the configuration is used as a general reference point for the interacting actors to identify resources. On the other hand it is clearly shown here that handling of multiple data models is making the system significantly more complicated with regard to unified search capabilities.

What is still only preliminary defined in conjunction with the general design of the IMPEx portal server are the capabilities provided by the IMPEx client. It was mentioned that a concrete design and and implementation is not considered as useful before the REST interface isn't fully elaborated and operating. In the first stage of developments the interface will provide view templates of the Play framework to be able to test the individual capabilities of the REST interface beforehand. Nevertheless the structure of a Play based application dictates how modern web-based applications should be constructed. Two of the most important issues on the client-side are similar to those identified on the server-side: scalability of the respective components and loose coupling between them. This can certainly be achieved by the mentioned client-side libraries but appropriate libaries for the user interface elements needed for the *IMPEx map* are still in discussion. In any case all needed functionality for working with the domain-specific models and VOTable structures on the client will be available through a respective representation within the JavaScript context based on the transferred JSON objects from the server. Nevertheless the original XML representation of VOTable files will be needed when delegating selections from the client over the SAMP hub which was already taken into account in the architectural design.

Finally it is to say that the here evaluated extended capabilities of object-functional programming such as the actor-based Akka library and the asynchronous nature of the Play framework are extremly useful for the elaboration of fully scalable and resilient web-based applications. One important conclusion was drawn in course of the respective studies: Actor-based systems are also capable of being distributed over the network. This means that the here depicted portal architecture can basically also operate on multiple servers for example. In the future this might be needed when long-running tasks are included in the system for example. A more efficient messaging protocol could be established between this distributed actors in the form of an *Enterprise Service Bus (ESB)*. This means that also data provider could create actor-based systems locally which can be interconnected over the network. It was also recently discovered by the author that `Future` types can also be encapsulated within `Promise` types which enable a more abstract way to work with multiple `Future` instances in parallel. These types also enable the capturing of specific types of futures and the addition of implicit methods for their exploitation and combination. These circumstances let the programming language Scala in overall appear as a good candidate for future projects in the domain of distributed data analysis frameworks in space physics which will remain the main research topic of the author beyond the finalisation of the here described IMPEx portal. With the here described advantages,

the full spectrum of the research work done within this thesis can be justified easily in the scope of the IMPEx project.

Nevertheless in particular with regard to user authentication and session-handling in the frontend as well as caching mechanisms and persisting services in the backend there is still a significant amount of functionality to be studied in the frame of the Scala language. In addition to that the IMPEx community must be made more aware of all advantages of asynchronous and concurrent programming in distributed Scala-based systems whose primary features are already used within the architecture of the IMPEx portal.

A. IMPEx Configuration

The following listing represents the final version of the *IMPEx configuration file* which is fetched regularly on the server to refresh the location and capabilities of all services and resources available to the IMPEx portal. The configuration is also exposed to all participating tools for remote access via the IMPEx portal.

Listing A.1: The IMPEx configuration XML file

```
<?xml version="1.0"?>
<impexconfiguration xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=
  "http://www.impex.org/2012/configuration.xsd
  configuration.xsd"
 xmlns="http://www.impex.org/2012/configuration.xsd">
  <database type="simulation">
    <name>FMI</name>
    <description>
      FMI Hybrid and MHD web archive
        </description>
    <dns>impex-fp7.fmi.fi</dns>
    <methods>/ws/Methods_FMI.wsdl</methods>
    <tree>/ws/Tree_FMI_HYB.xml</tree>
    <tree>/ws/Tree_FMI_GUMICS.xml</tree>
    <protocol>soap</protocol>
    <protocol>http</protocol>
    <info>http://hwa.fmi.fi/beta/login.php</info>
  </database>
  <database type="simulation">
    <name>LATMOS</name>
    <description>
      LATMOS Hybrid simulations
    </description>
    <dns>impex.latmos.ipsl.fr</dns>
    <methods>/Methods_LATMOS.wsdl</methods>
    <tree>/tree.xml</tree>
    <protocol>soap</protocol>
```

```
    <protocol>http</protocol>
    <info>http://impex.latmos.ipsl.fr/LatHyS.htm</info>
  </database>
  <database type="simulation">
    <name>SINP</name>
    <description>
      SINP Paraboloid Model simulations
    </description>
    <dns>dec1.sinp.msu.ru</dns>
    <methods>
      /~lucymu/paraboloid/SINP_methods.wsdl
    </methods>
    <tree>/~lucymu/paraboloid/SINP_tree.xml</tree>
    <protocol>soap</protocol>
    <protocol>http</protocol>
    <info>
      http://smdc.sinp.msu.ru/index.py?nav=model-para
    </info>
  </database>
  <database type="observation">
    <name>AMDA</name>
    <description>AMDA observational database</description>
    <dns>cdpp1.cesr.fr</dns>
    <methods>
      /AMDA-NG/public/wsdl/Methods_AMDA.wsdl
    </methods>
    <tree>
      /AMDA-NG/data/WSRESULT/
                getObsDataTree_LocalParams.xml
    </tree>
    <protocol>soap</protocol>
    <protocol>http</protocol>
    <info>http://clweb.cesr.fr/webservice.html</info>
  </database>
  <database type="observation">
    <name>ClWeb</name>
    <description>
      ClWeb observational databases
    </description>
    <dns>clweb.cesr.fr</dns>
    <methods>/Methods_CLWEB.wsdl</methods>
    <tree>/clweb_tree.xml</tree>
    <protocol>soap</protocol>
    <protocol>http</protocol>
    <info>http://clweb.cesr.fr/webservice.html</info>
```

```
75    </database>
      <tool>
77      <name>AMDA</name>
        <description>
79        Multi-mission data analysis tool for
          space plasma physics
81      </description>
        <dns>http://amda.cdpp.eu/</dns>
83      <info>
          http://cdpp-amda.cesr.fr/DDHTML/HELP/about.html
85      </info>
      </tool>
87    <tool>
        <name>ClWeb</name>
89      <description>
          Multi-mission space plasma data plotting tool
91      </description>
        <dns>clweb.cesr.fr</dns>
93      <info>http://clweb.cesr.fr/clweb_poster.pdf</info>
      </tool>
95    <tool>
        <name>3DView</name>
97      <description>
          3D multi-mission visualisation tool
99      </description>
        <dns>http://3dview.cdpp.eu/</dns>
101     <info>
          http://3dview.cesr.fr/other/cdpp3dview_tutorial.pdf
103     </info>
      </tool>
105 </impexconfiguration>
```

B. IMPEx Portal Map

The following step-by-step description explains a typical workflow within the *IMPEx portal map*. It represents a common use case of the portal which can be applied to a variety of scientific applications and summarises the major user requirements evaluated in section 5.2. This chapter will describe each of the eight steps with the help of mock-ups depicting the preliminary design as it was presented and agreed upon at the IMPEx technical meeting in Toulouse, September 20, 2013. Note that the mock-ups are only illustrating six steps since the remaining two are recurring steps which will be indicated as such in the enumeration on the next page.

1. Figure B.1 on the following page shows the initial view of the IMPEx map with it's standard tool-based configuration. The *Databases* are representing each included resource providing a data tree; the *Services* are displaying each included resource enabling web service access; The *Tools* branch is showing all IMPEx-associated tools capable of displaying selections made within the IMPEx portal. Ultimately *My Data* represents a custom data store where intermediate selections and results of a workflow are persisted during a users session.
2. Figure B.2 on page 115 shows an example of a search conducted by the user either by choosing respective predefined filters or by making a full-text search by using the search-field. The entered keywords *Mars* (Target) and *LATMOS* (Data provider) automatically highlight all resources related to either one of them. The following sequence of steps needed in a typical workflow is indicated with the grouping of resources in the IMPEx map: *Databases*, *Services* and *Tools*.
3. In this case the user selects the highlighted *LATMOS Simulations* database where a dialog is showing the available simulation runs and respective parameters as depicted in figure B.3 on page 116. The user can then choose one or more parameters.
4. Figure B.4 on page 117 shows how the IMPEx map will display the consequent steps an user can carry out with the current selection. In this case the user can save the selection in *My Data* or delegate it to a web service call from the actually selected data provider in the group of *Services*.
5. In this use case the user opens the dialog of *LATMOS Data Access* where a respective method can be choosen depending on the before selected resource. Note that it is considered here that the actually selected method requires additional acquisition of observational data.
6. Figure B.5 on page 118 again indicates a path to the respective database and service which needs to be used to obtain the required data for *LATMOS Data Access*.
7. It is considered here that the user will open the respective dialogs for *AMDA Observations* and *AMDA Data Access* in order to select the required resource and conduct a web service call on AMDA.
8. The final view in figure B.6 on page 119 indicates that all necessary selections have been successfully accomplished in order to finally visualise the acquired data with the *Tools* group through the SAMP protocol or to save the dataset selection in *My Data*. This is again indicated with respective paths on the *IMPEx map*.

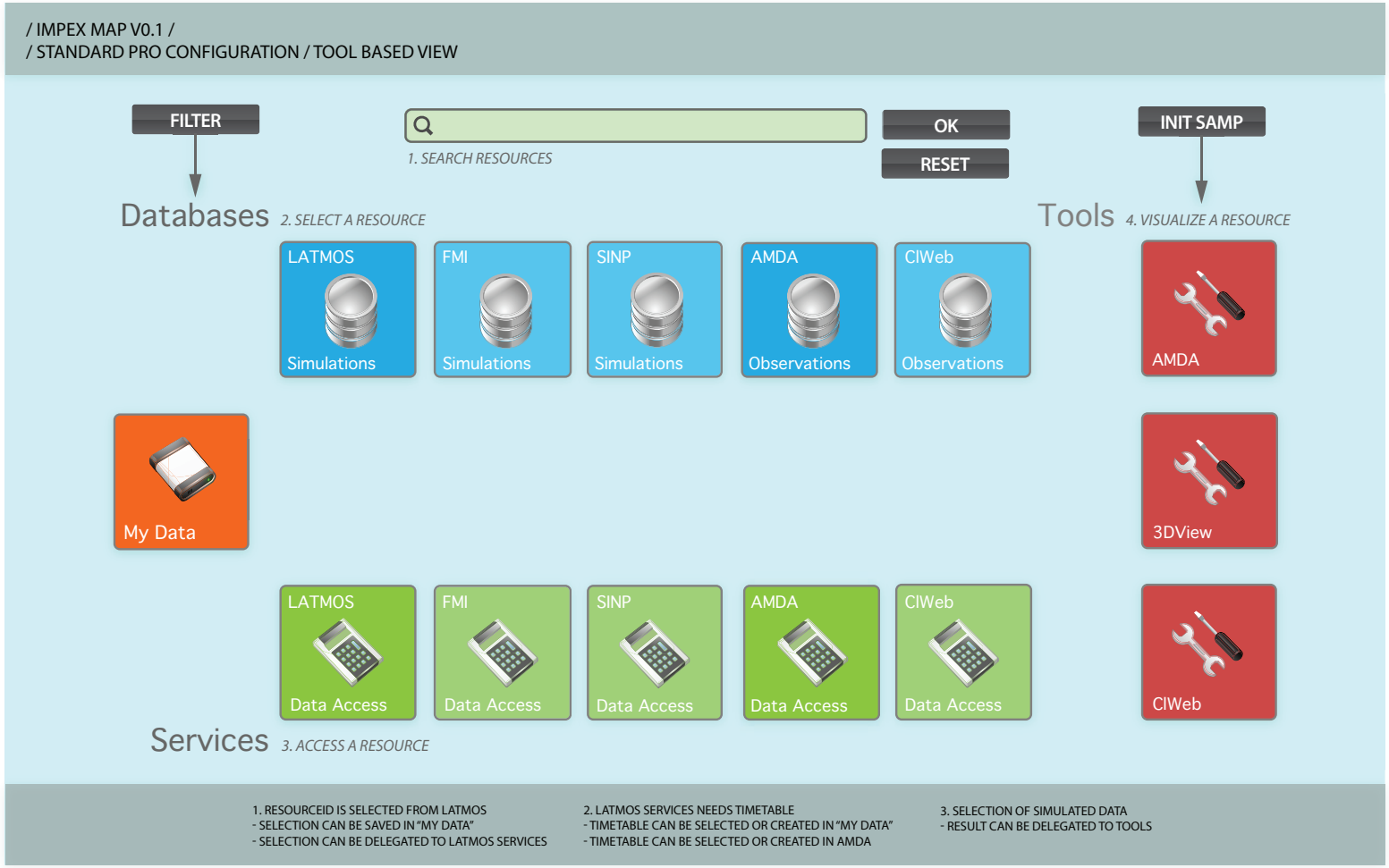


Figure B.1.: IMPEx portal map - Initial index view

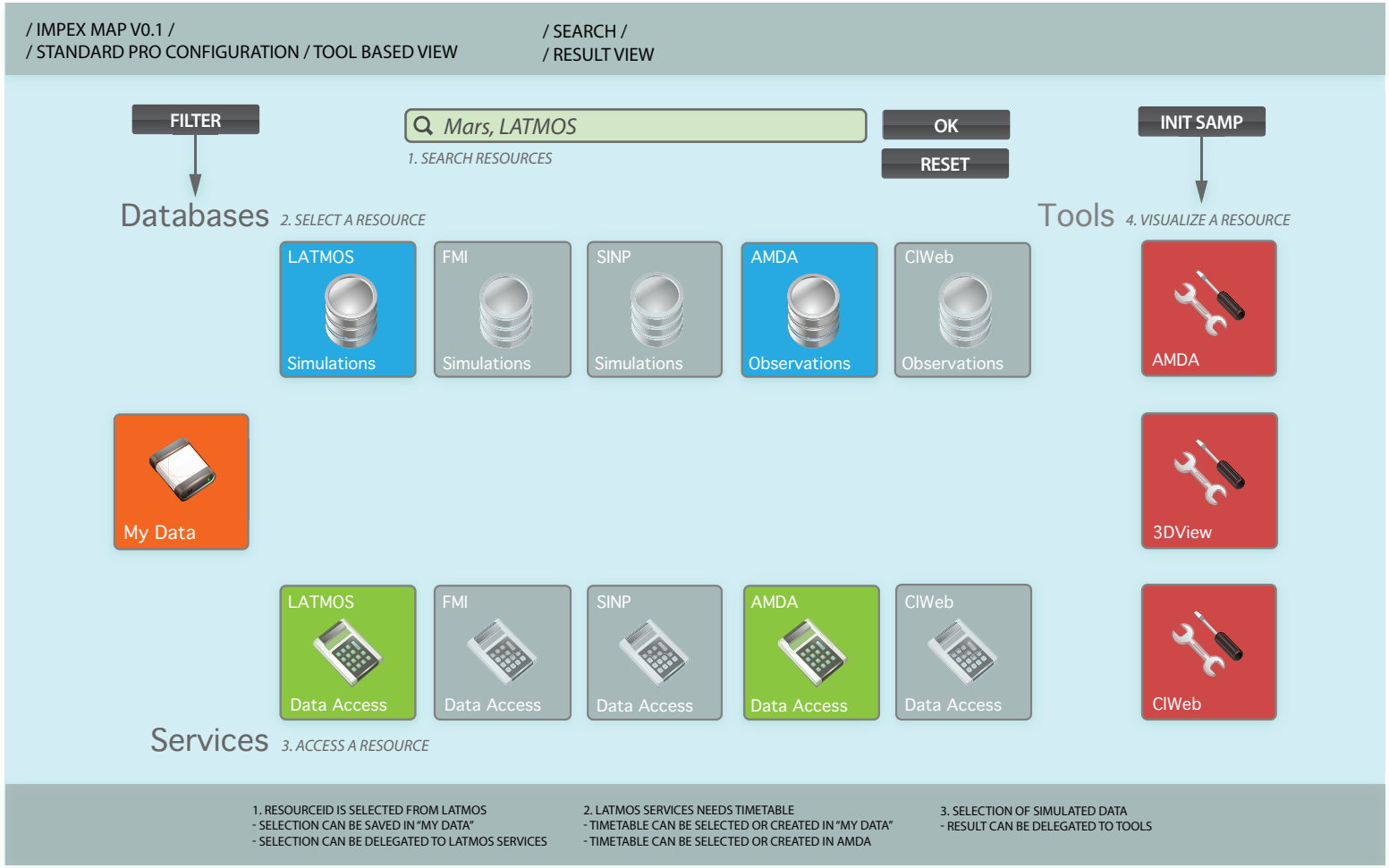


Figure B.2.: IMPEx portal map - Search view

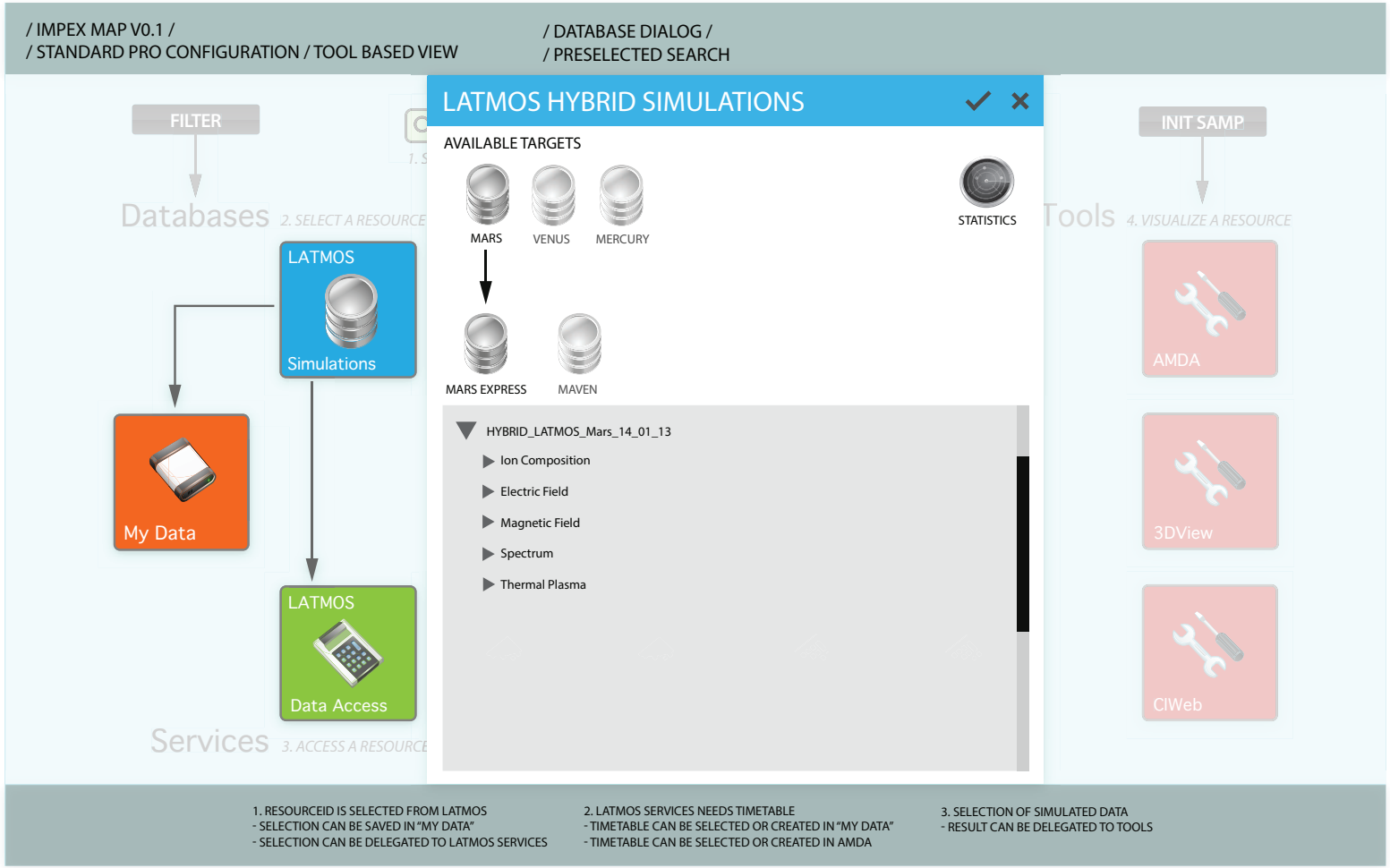


Figure B.3.: IMPEx portal map - Database dialog

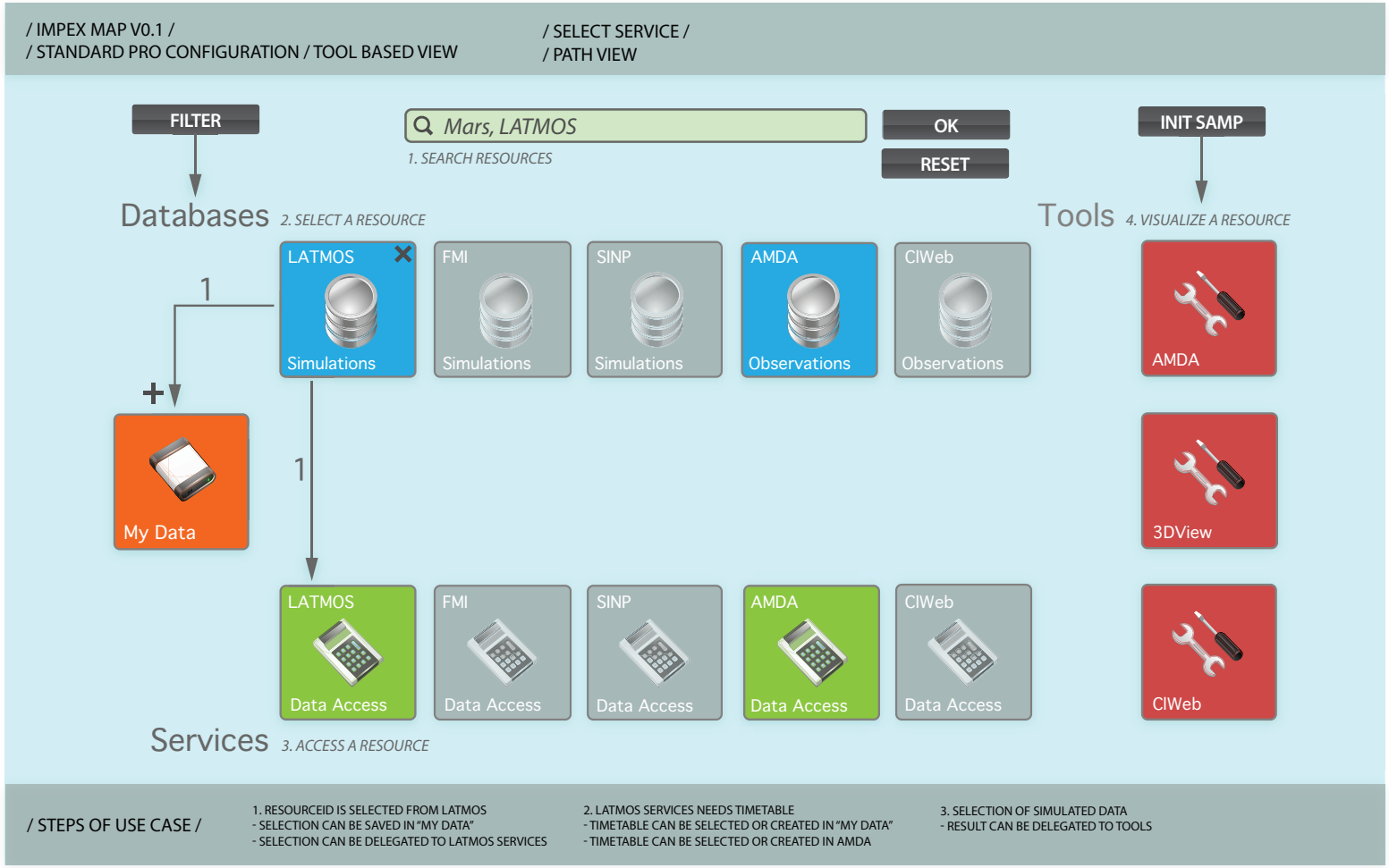


Figure B.4.: IMPEx portal map - Data path view

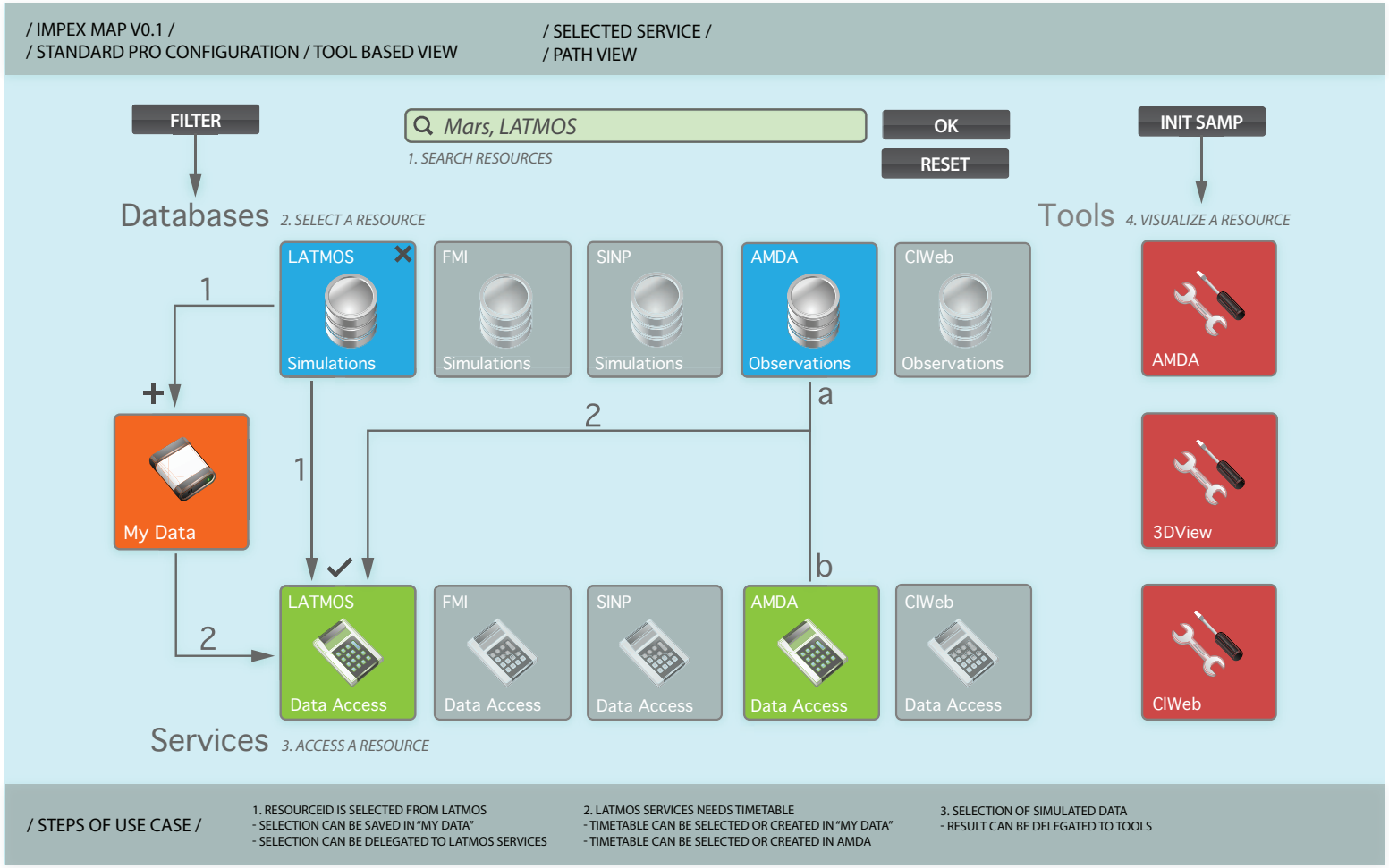


Figure B.5.: IMPEx portal map - Data path view

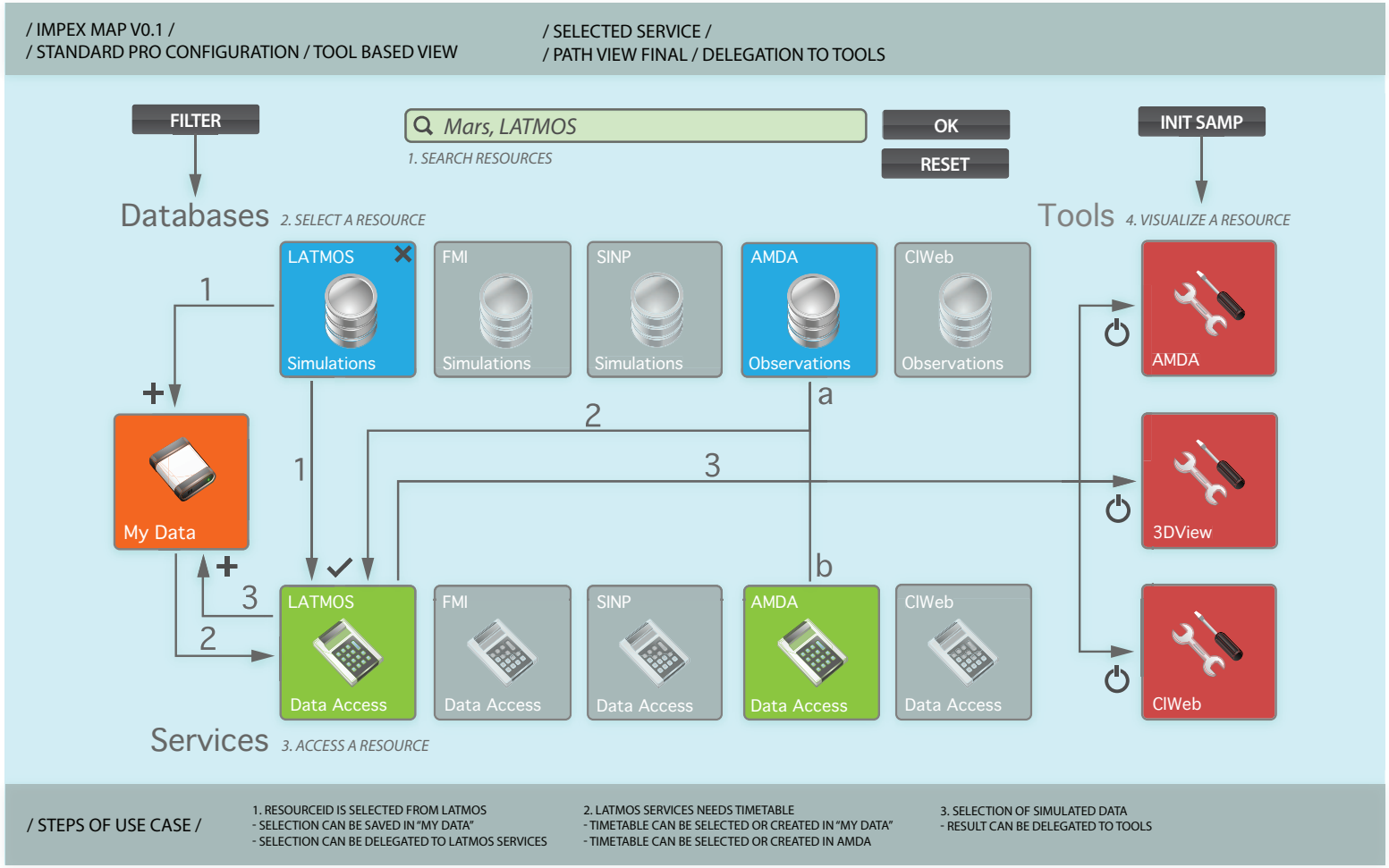


Figure B.6.: IMPEx portal map - Final view

C. IMPEx Portal Main Classes

The following class diagram shows the three basic entities which build up the capabilities needed for the *IMPEx Registry Service* and the *IMPEx Data Access Service* respectively. As one can see the entities are represented by a class derived from the `Actor` trait which identifies them as an actor for subsequent usage with the inbuilt actor system of the Play framework. The requests to an actor are always handled through a companion object which dispatches the respective messages to the actor.

The basic element of the *Resource Layer* at the IMPEx portal is built up with with an `DataProvider` actor. It holds a respective tree from one database stored in the IMPEx configuration which is initially loaded at startup of the integrated web server in the Play framework. The `DataProvider` actors are immediately registered at the `RegistryService` as childs. As one can see in the diagram the `RegistryService` handles all requests sent to it's childs. It simply maps the shared methods from the actors to one set of access methods which is used subsequently by the portal system. It is also basically possible to register new `DataProvider` actors dynamically in this preliminary version.

All entities in the showed class diagramm are directly or indirectly related to the `ConfigService` actor. The `DataProvider` actors are created based on the information stored within the `ConfigService` on startup. The `RegistryService` is using the `ConfigService` as reference point to check the integrity of incoming requests and the pool of registered childs.

The respective source code of the IMPEx portal prototype can be obtained on request via the autor's GitHub[1] profile, see: `https://github.com/FlorianTopf/impex-portal`.

[1]GitHub a project hosting platform for Git repositories: `https://github.com/`

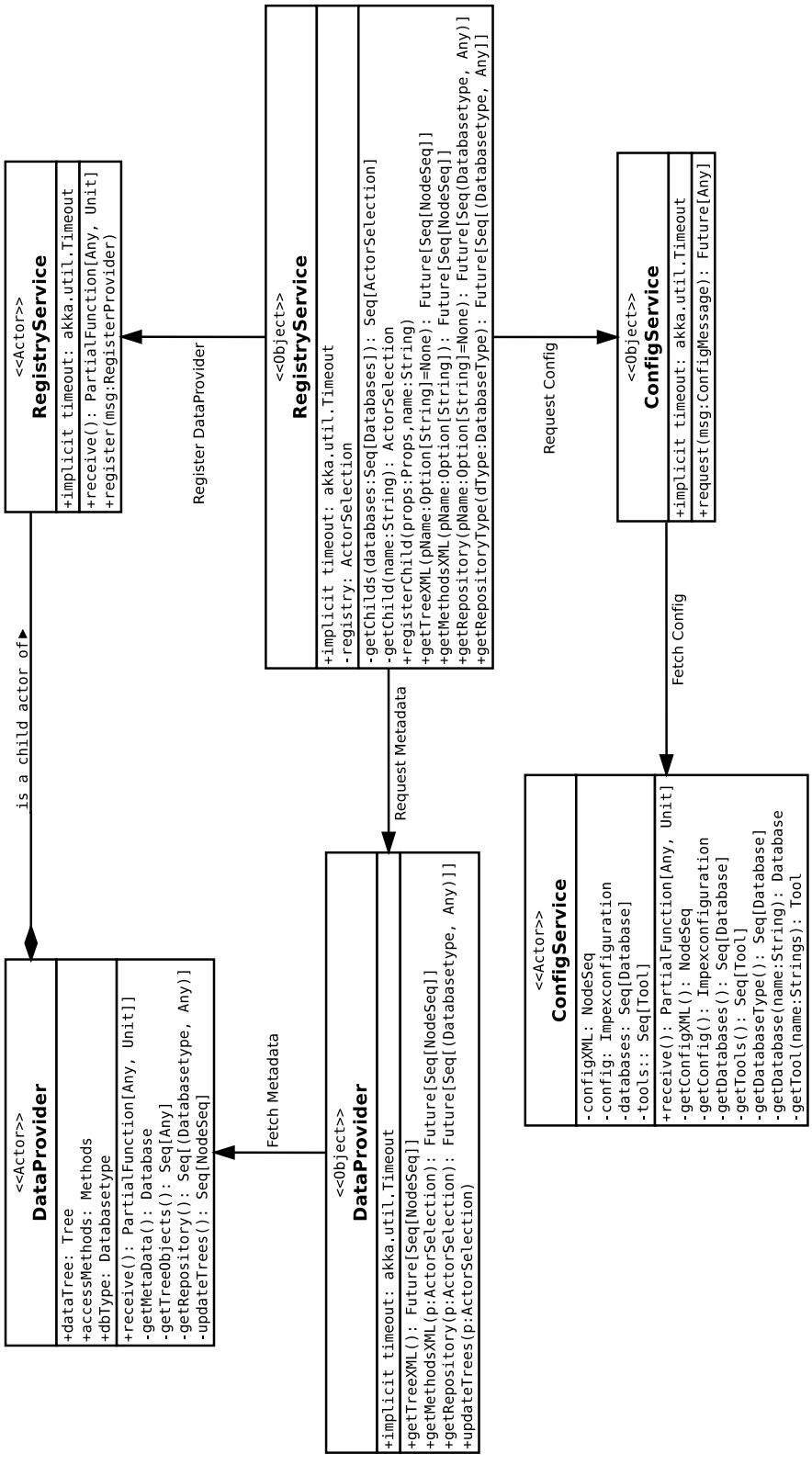


Figure C.1: IMPEx portal main classes

Glossary

3DView 3DView Multimission, a 3D visualization environment for spacecraft trajectories and planetary ephemerides. see: `http://3dview.cdpp.eu/`, p. 2.

AJAX Asynchronous JavaScript and XML, p. 65.

Akka A library for concurrent and distributed programming in Scala and Java, p. 55.

API Application Programming Interface, a specification intended to be used as an interface by software components to communicate with each other, p. 3.

ASCII American Standard Code for Information Interchange, a 7-bit character encoding scheme.

CDPP Centre de Données de Physique des Plasmas, a french data center for space plasma physics located in Toulouse.

CIWeb A web-based scientific analysis tool developed by CESR, Toulouse. see: `http://clweb.cesr.fr/`, p. 2.

CPU Central Processing Unit, p. 15.

CSV Comma Separated Values, a generic way to display table data in ASCII files.

DAO Data Access Object, p. 67.

DSL Domain Specific Language, p. 43.

Erlang A dynamically typed functional language enabling concurrent and distributed programming, p. 29.

FMI Finnish Meteorological Institute, a multi-disciplinary research agency located in Helsinki.

Haskell A pure functional programming language with a strict type system and non-strict semantics, p. 14.

HMM Hybrid and MHD Models, a work package of the EU FP7-SPACE project IMPEx, p. 50.

HTML Hypertext Markup Language, a markup language designed for the creation of static web content, p. 44.

HTML5 The successor of HTML4 markup language for web-based documents adding new features e.g. for media and caching, p. 63.

HTTP Hypertext Transfer Protocol, the cornerstone for data communication in the World Wide Web, p. 44.

I/O Input/Output, p. 26.

IWF Institut für Weltraumforschung or Space Research Institute of the Austrian Academy of Sciences located in Graz, p. iii.

Java An object-oriented programming language, originally developed by Sun Microsystems, p. 29.

JavaScript A browser-driven scripting language, mainly used for creating dynamic HTML website content., p. 4.

JSON JavaScript Object Notation, a notation for structured documents used to prepare data for clients in a web-based application, p. 65.

JVM Java Virtual Machine, p. 63.

LATMOS Laboratoire Atmosphéres, Milieux, Observations Spatiales, a french research laboratory located in Paris.

LISP List Processing Language, the first functional programming language developed by John McCarthy in 1958, p. 14.

MAG Magnetometer.

MVC Model View Controller, a design pattern for web-based applications, p. 64.

netCDF Network Common Data Format, a set of encoding rules for making array-based scientific data machine-readable.

FP7 7th Framework Program for Research and Technological Development, a funding programme created by the European Union, p. iii.

IMPEx Integrated Medium for Planetary Exploration, an EU FP7-SPACE collaborative project, p. iii.

SOAP Simple Object Access Protocol, an XML based messaging protocol for web services, p. 2.

AMDA Automated Multi-Dataset Analysis Tool, a web-based scientific analysis tool developed by CDPP, Toulouse. see: `http://amda.cdpp.eu/`, p. 2.

XML-RPC A Remote Procedure Call protocol based on XML, p. 2.

SAMP Simple Application Messaging Protocol, an IVOA standard based on the XML-RPC protocol, p. 2.

IVOA International Virtual Observatory Alliance, a community dedicated to the development of a common VO, p. 2.

EuroPlaNet European Planetary Network, a EU FP7 research infrastructure project, p. 48.

WSDL Web Service Description Language, an XML based description vocabulary for web services, p. 48.

Play A development framework for web-based applications based on JAVA and Scala, p. 62.

PMC Project Management Commitee, p. 70.

PMM Paraboloid Magnetospheric Models, a work package of the EU FP7-SPACE project IMPEx, p. 50.

REST Representational State Transfer, a programming paradigm for web-based application based on the HTTP protocol, p. 63.

RPC Remote Procedure Call, p. 58.

RTD Research & Technical Development, p. iii.

Ruby A dynamically typed, object-oriented interpreter language, p. 62.

Scala A object-functional programming language running on the Java Virtual Machine, p. iv.

SINP Skobeltsyn Institute of Nuclear Physics, a research institution located in Moscow.

Smalltalk A message-oriented programming language, where data and programs are represented as objects., p. 28.

SMDB Simulation Database.

SPASE Space Physics Archive Search and Extract, a data model for describing and accessing space physics data, p. 49.

UAB User Advisory Board, p. 69.

URI Uniform Resource Identifier, a unique identifier for an abstract or physical resource, p. 92.

URL Uniform Resource Locator, a special version of an URI used in the World Wide Web to address web resources, p. 64.

VEX Venus Express, an ESA planetary mission to Venus.

VO Virtual Observatory, often referred to an distributed online data analysis environment for space sciences, p. iv.

VOTable An XML standard developed by the IVOA for the interchange of data represented as a set of tables.

XML eXtensible Markup Language, a collection of encoding rules to make structured documents machine-readable, p. iv.

XPath A simple subset of the XQuery XML language, p. 50.

List of Figures

5.1. Overall architecture of the IMPEx portal 87
5.2. Hierarchy of SPASE based simulation model 92
5.3. Hierarchy of proprietary observational model 93

B.1. IMPEx portal map - Initial index view 114
B.2. IMPEx portal map - Search and results view 115
B.3. IMPEx portal map - Database dialog 116
B.4. IMPEx portal map - Data path view 117
B.5. IMPEx portal map - Data path view 118
B.6. IMPEx portal map - Final view 119

C.1. IMPEx portal main classes 121

List of Tables

5.1. UR-1.1 Consolidated IMPEx data tree 73
5.2. UR-1.2 Search functionality for IMPEx trees and services 74
5.3. UR-2.1 Dynamic provision of IMPEx trees and services 75
5.4. UR-2.2 Documentation of IMPEx services 76
5.5. UR-2.3 Statistics of IMPEx services 76
5.6. UR-3.1 Delegation of portal data selections 77
5.7. UR-3.2 Exploitation of IMPEx services 78
5.8. UR-4.1 Download of portal data selections 79
5.9. UR-4.2 Time- and coordinate-table management 80
5.10. UR-5.1 User sessions 81
5.11. UR-5.2 User interface arrangement 82
5.12. UR-5.3 User levels and custom views 83
5.13. UR-6.1 Administrative user interface 84
5.14. UR-6.2 Handling of data model and tree updates 85

Listings

3.1. A simple function definition in Scala 30
3.2. Classes and subtyping in Scala 32
3.3. Companion objects in Scala 35
3.4. Type parameters in Scala 35
3.5. An anonymous function application in Scala 36
3.6. Type bounds in Scala 37
3.7. Pattern matching in Scala 38
3.8. Concatenation of Scala lists 41
3.9. Mapping of Scala lists 41
3.10. A prototype definition in JavaScript 45

4.1. Example of XML processing in Scala 52

5.1. Synchronous communication with the Config Service 88
5.2. Asynchronous communication with the Config Service 89
5.3. Example of simple GET request in Play 90
5.4. URL path for the SPASE data model 93
5.5. URL path for the observation data model 94
5.6. A typical call to a DataProvider actor 94
5.7. Example of API calls to the Registry Service 97
5.8. Example of API calls to the Data Access Service 99
5.9. Example of web service call with scalaxb 100

A.1. The IMPEx configuration XML file 109

Bibliography

Alexander, A. (2013a). *Scala Cookbook* (First ed.). O'Reilly Media.

Alexander, A. (2013b). *Scala Cookbook: Bonus Chapters* (First ed.). O'Reilly Media.

Allen, J. (2013). *Effective Akka* (First ed.). O'Reilly Media.

Backus, J. (1978, August). Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, *21*(8), 613–641. doi: 10.1145/359576.359579

Barendregt, H. P. (1981). *The Lambda Calculus: Its Syntax and Semantics* (First ed.). Elsevier Science B.V.

Bird, R., & Wadler, P. (1988). *Introduction to Functional Programming* (First ed.). Prentice Hall College Div.

Braun, O. (2010). *Scala: Objektfunktionale Programmierung* (First ed.). Hanser Verlag.

Brikman, Y. (2013, March). *Play Framework: async I/O withwith the thread pool and callback hell.* Retrieved October 22, 2013, from `http://engineering.linkedin.com/play/play-framework-async-io-without-thread-pool-and-callback-hell`

Chiusano, P., & Bjarnason, R. (2013). *Functional Programming in Scala* (Manning Early Access Program ed.). Manning Publications.

Church, A. (1936, April). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, *58*(2), 345–363. doi: 10.2307/2371045

Church, A. (1941). *The Calculi of Lambda-Conversion* (Vol. 6). Princeton University Press.

Church, A., & Rosser, J. B. (1936). Some Properties of Conversion. *Transactions of the American Mathematical Society*, *39*, 472–482. doi: 10.2307/2268573

Cousineau, G., & Mauny, M. (1998). *The Functional Approach to Programming* (Second ed.). Cambridge University Press.

Curry, H. B. (1930, October). Grundlagen der kombinatorischen Logik. *American Journal of Mathematics*, *52*(4), 789–834.

Daigneau, R. (2011). *Service Design Patterns - Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services* (First ed.). Addison-Wesley.

Ellis, W. (2010). *Introducing the Play! framework* (First ed.). Wayne Ellis. Retrieved from `http://www.the-play-book.co.uk/`

Emir, B., Odersky, M., & Williams, J. (2007). Matching Objects with Patterns. In *ECOOP 2007 – Object-Oriented Programming, volume 4609 of LNCS* (pp. 273–298). Springer-Verlag.

Fogus, M. (2013). *Functional JavaScript: Introducing Functional Programming with Underscore.js* (First ed.). O'Reilly Media.

Fokker, J. (1995). *Functional Programming.* Lecture Notes. Retrieved from http://www.staff.science.uu.nl/~fokke101/courses/fp-eng.pdf

Ghosh, D. (2011). *DSLs in Action* (First ed.). Manning Publications.

Gupta, M. K. (2012). *Akka Essentials* (First ed.). Packt Publishing.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, *38*(1), 173-198. doi: 10.1007/BF01700692

Haller, P., & Sommers, F. (2011). *Actors in Scala* (First ed.). Artima.

Harrison, J. (1997). *Introduction to Functional Programming.* Lecture Notes. Retrieved from http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/

Haverbeke, M. (2011). *Eloquent JavaScript: A Modern Introduction to Programming* (First ed.). No Starch Press.

Hilbert, D., & Ackermann, W. (1928). *Grundzüge der theoretischen Logik* (First ed.). Springer-Verlag.

Hilton, P., Bakker, E., & Candeo, F. (2013). *Play for Scala* (MEAP ed.). Manning Publications.

Horstmann, C. (2012). *Scala for the Impatient* (First ed.). Addison-Wesley.

Hudak, P. (1989, September). Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, *21*(3), 359–411. doi: 10.1145/72551.72554

Hughes, J. (1989, April). Why functional programming matters. *Comput. J.*, *32*(2), 98–107. doi: 10.1093/comjnl/32.2.98

Knight, T. (1990). An architecture for mostly functional languages. In P. H. Winston & S. A. Shellard (Eds.), *Artificial intelligence at MIT expanding frontiers* (pp. 500–519). MIT Press.

Lipovača, M. (2011). *Learn You a Haskell for Great Good!* (First ed.). No Starch Press.

Lippe, W.-M. (2009). *Funktionale und Applikative Programmierung* (First ed.). Springer-Verlag Berlin Heidelberg.

Loverdos, C. K., & Syropoulos, A. (2010). *Steps in Scala - Object-Functional Programming* (First ed.). Cambridge University Press.

Maharry, D. (2013). *TypeScript revealed* (2013rd ed.). Apress.

Marick, B. (2012). *Functional Programming for the Object-Oriented Programmer* (First ed.). Leanpub.

McCarthy, J. (1960, April). Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, *3*(4), 184–195. doi: 10.1145/367177.367199

Meredith, L. G. (2012). *Monadic Design Patterns for the Web* (First ed.). Artima.

Michaelson, G. (1988). *An Introduction to Functional Programming through Lambda Calculus* (First ed.). Addison-Wesley.

Neward, T. (2008, January). *The busy Java developer's guide to Scala: Functional programmng for the object oriented* (IBM developerWorks Series No. 1). IBM. Retrieved from http://www.ibm.com/developerworks/library/j-scala01228/

Odersky, M. (2011, May). *Scala by Example.* Retrieved April 28, 2013, from http://www.scala-lang.org/docu/files/ScalaByExample.pdf

Odersky, M. (2013, March). *The Scala Language Specification - Version 2.8* (Draft). EPFL. Retrieved from http://www.scala-lang.org/files/archive/nightly/pdfs/ScalaReference.pdf

Odersky, M., et al. (2004). *An Overview of the Scala Programming Language* (Tech. Rep. No. IC/2004/64). EPFL Lausanne, Switzerland. Retrieved from http://www.scala-lang.org/docu/files/ScalaOverview.pdf

Odersky, M., Spoon, L., & Venners, B. (2011). *Programming in Scala* (First ed.). Artima.

Odersky, M., & Zenger, M. (2005). Scalable component abstractions. In *Proceedings of the 20th annual acm sigplan conference on object-oriented programming, systems, languages, and applications* (pp. 41–57). ACM. doi: 10.1145/1094811.1094815

Okasaki, C. (1999). *Purely Functional Data Structures* (First ed.). Cambridge University Press.

O'Sullivan, B., Goerzen, J., & Stewart, D. (2009). *Real World Haskell* (Second ed.). O'Reilly Media.

Paulson, L. C. (1996). *Foundations of Functional Programming.* Lecture Notes. Retrieved from http://www.cl.cam.ac.uk/~lp15/papers/Notes/Founds-FP.pdf

Pepper, P., & Hofstedt, P. (2006). *Funktionale Programmierung: Sprachdesign und Programmiertechnik* (First ed.). Springer-Verlag Berlin Heidelberg.

Perrett, T. (2011a). *Lift in Action* (First ed.). Manning Publications.

Perrett, T. (2011b, July). *Using SOAP with Scala.* Retrieved November 22, 2013, from http://timperrett.com/2011/07/26/using-soap-with-scala/

Petrella, A. (2013). *Learning Play! Framework 2* (First ed.). Packt Publishing.

Petricek, T., & Skeet, J. (2010). *Real-World Functional Programming* (First ed.). Manning Publications.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages* (First ed.). Prentice-Hall.

Piepmeyer, L. (2010). *Grundkurs funktionale Programmierung mit Scala* (First ed.). Carl Hanser Verlag.

Pollak, D. (2009). *Beginning Scala* (First ed.). Apress.

Rabhi, F., & Lapalme, G. (1999). *Algorithms: A Functional Programming Approach*. Addison-Wesley.

Raychaudhuri, N. (2013). *Scala in Action* (First ed.). Manning Publications.

Reelsen, A. (2011). *Play Framework Cookbook* (First ed.). Packt Publishing.

Resig, J., & Bibeault, B. (2012). *Secrets of the JavaScript Ninja* (First ed.). Manning Publications.

Sabry, A. (1998). What is a Purely Functional Language? *Journal of Functional Programming*, *8*, 1–22.

Schinz, M., & Haller, P. (2011, May). *A Scala Tutorial for Java Programmers.* Retrieved April 28, 2013, from http://www.scala-lang.org/docu/files/ScalaTutorial.pdf

Sebesta, R. W. (2012). *Concepts of Programming Languages* (Tenth ed.). Pearson Education.

Smaragdakis, Y., & McNamara, B. (2000). *Bridging Functional and Object-Oriented Programming* (CC Technical Report). Georgia Institute of Technology. Retrieved from http://hdl.handle.net/1853/6600

Stefanov, S. (2008). *Object-oriented JavaScript* (First ed.). Packt Publishing.

Subramaniam, V. (2009). *Pragmatic Programming Scala* (First ed.). The Pragmatic Bookshelf.

Suereth, J. D. (2012). *Scala in Depth* (First ed.). Manning Publications.

Taylor, M., et al. (2011, September). *SAMP - Simple Application Messaging Protocol Version 1.3* (Proposed Recommendation). IVOA. Retrieved from http://www.ivoa.net/Documents/SAMP/

The IMPEx Consortium. (2013, May). *The Integrated Medium for Planetary Exploration - Web portal.* Retrieved May 26, 2013, from http://impex-fp7.oeaw.ac.at/

The SPASE Consortium. (2011, October). *A Space and Solar Physics Data Model* (Specification). SPASE. Retrieved from http://www.spase-group.org/docs/dictionary/spase-2_2_2.pdf

Thompson, S. (2000). *Haskell: The Craft of Functional Programming* (Second ed.). Addison-Wesley.

Topf, F. (2012a). *IMPEx - A Service-oriented Space Plasma Physics Virtual Observatory*. Unpublished Bachelor Thesis, FH Campus 02, Graz, Austria.

Topf, F. (2012b). *Service Oriented Architectures in Planetary Sciences*. Unpublished Bachelor Thesis, FH Campus 02, Graz, Austria.

Turing, A. M. (1936). On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, *42*, 230–265.

Turing, A. M. (1937). Computability and λ-Definability. *J. Symb. Log.*, *2*(4), 153-163.

Turner, D. (2007, February). Church's Thesis and Functional Programming. In A. Olszewski (Ed.), *Church's thesis after 70 years* (pp. 518–544). Ontos Verlag.

Typesafe. (2013, October). *Play - The High Velocity Web Framework for Java and Scala.* Retrieved October 22, 2013, from http://www.playframework.com/

Ullenboom, C. (2012). *Java ist auch eine Insel* (10th ed.). Galileo Press.

Ullman, L. (2012). *Modern JavaScript* (First ed.). Peachpit Press.

Venners, B. (2009, April). *Twitter on Scala.* Retrieved September 30, 2013, from http://www.artima.com/scalazine/articles/twitter_on_scala.html

Wadler, P. (1995). Monads for Functional Programming. In *Advanced functional programming, first international spring school on advanced functional programming techniques-tutorial text* (pp. 24–52). Springer-Verlag London.

Wampler, D. (2011). *Functional Programming for Java Developers* (First ed.). O'Reilly Media.

Wampler, D., & Payne, A. (2009). *Programming Scala* (First ed.). O'Reilly Media.

Wotawa, F., & Bloem, R. (2003). *Einführung in die Informatik.* Lecture Notes.

Wright, K. (2009, December). *A (Brief) History of Object-Functional Programming.* Retrieved July 28, 2013, from http://www.artima.com/weblogs/viewpost.jsp?thread=275983

Wyatt, D. (2013). *Akka Concurrency* (Pre-Print ed.). Artima.

If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

(John von Neumann)