MODEL DRIVEN DEVELOPMENT OF SPARK TASKS BY MEANS OF ECLIPSE ACCELEO

MODELEM ŘÍZENÝ VÝVOJ SPARK ÚLOH POMOCÍ ECLIPSE ACCELEO

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MAREK ŠALGOVIČ**
AUTOR PRÁCE

SUPERVISOR **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Department of Information Systems (DIFS) Academic year 2021/2022

Master's Thesis Specification

24614

Student:	**Šalgovič Marek, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Information Systems and Databases
Title:	**Model Driven Development of Spark Tasks by Means of Eclipse Acceleo**
Category:	Parallel and Distributed Computing

Assignment:

1. Get familiar with the Apache Spark platform for distributed Big Data processing and with the programming languages it can be used with. Get familiar with Model Driven Development (MDD) and with Eclipse Acceleo. Research the options and existing projects that are used for modeling data processing tasks, especially for the Big Data processing.
2. Design a meta-model for modelling tasks for Big Data processing in Apache Spark and describe its usage with Eclipse Acceleo. Make generation of Spark applications source code from their models possible.
3. After consulting with the supervisor, integrate the designed meta-model and source code generation of Spark applications using their models into Eclipse Acceleo. Verify the functionality by creating models of several Spark tasks for processing Big Data and by generating the corresponding source code.
4. Test the solution, evaluate and discuss the results. Publish the resulting software as open-source.

Recommended literature:

- Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. First edition, 256 pp., O'Reilly Media, 2015. ISBN 978-1-449-35862-4.
- Databricks Spark Reference Applications [online]. 2017 [seen 2021-09-29]. Available at [https://databricks.gitbooks.io/databricks-spark-reference-applications]
- Michele Guerriero, Saeed Tajfar, Damian A. Tamburri, and Elisabetta Di Nitto. Towards a model-driven design tool for big data architectures. In *Proceedings of the 2nd International Workshop on BIG Data Software Engineering (BIGDSE '16)*. Association for Computing Machinery, New York, USA, 2016. ISBN 978-1-4503-4152-3. Available at [http://dx.doi.org/10.1145/2896825.2896835]
- Michele Guerriero, Damian Andrew Tamburri, and Elisabetta Di Nitto. 2021. *StreamGen: Model-driven Development of Distributed Streaming Applications*. ACM Trans. Softw. Eng. Methodol. 30, 1, Article 1 (January 2021), 30 pp. ISSN 1049-331X. Available at [https://doi.org/10.1145/3408895]
- Matúš Bútora. Modelem řízený vývoj Spark úloh. Master's Thesis. Brno University of Technology, Faculty of Information Technology, Brno, 2019. Available at [https://www.fit.vut.cz/study/thesis/21682/]

Requirements for the semestral defence:

- Items 1 and 2 finished and item 3 in progress.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Rychlý Marek, RNDr., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	October 21, 2021

Abstract

This thesis deals with the Model-Driven Development of Big Data tasks in the Apache Spark environment. In the beginning, the reader is introduced to the Apache Spark framework and necessary details. Afterward, a closer look at the Model-Driven Development methodology is provided, and its advantages and disadvantages are described. The second part describes the designed meta-model for modeling Spark tasks. The designed Profile diagram features that extend the Class diagram are described in detail. Afterward, the code generator is implemented. The input of the generator are models that satisfy the designed meta-model. The thesis also contains example models and their evaluation.

Abstrakt

Táto diplomová práca sa zaoberá modelom riadeným vývojom Big Data úloh v prostredí Apache Spark. Na začiatok je čitateľovi predstavený framework Apache Spark a potrebné detaily. Ďalej sa priblíži problematika modelom riadeného vývoja a popíšu sa jeho výhody a nevýhody. V druhej časti je popísaný navrhnutý meta-model pre modelovanie úloh Sparku. Detailne sú popísané vlastnosti navrhnutého profilového diagramu, ktorý rozširuje diagram tried. Následne je implementovaný generátor kódu, ktorého vstup sú modely vyhovujúce navrhnutému meta-modelu. Práca taktiež obsahuje príklady modelov a ich vyhodnotenie.

Keywords

Apache Spark, Model-driven Development, Eclipse Acceleo, UML, Profile Diagram, Code Generation, Big Data, Meta-modeling, Scala, M2T

Klíčová slova

Apache Spark, Modelom riadený vývoj, Eclipse Acceleo, UML, Profilový Diagram, Generovanie kódu, Big Data, Meta-modelovanie, Scala, M2T

Reference

ŠALGOVIČ, Marek. *Model Driven Development of Spark Tasks by Means of Eclipse Acceleo.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšířený abstrakt

Svet sa dňom, čo dňom vyvíja vpred a ľudia sa snažia pochopiť, čo sa deje. Na to aby niekto mohol niečo správne pochopiť, musí analyzovať dáta, ktoré má k dispozícii. V dnešnej dobe je ale týchto dát čoraz viac a viac a ich objem rastie exponenciálne. S postupným vývojom internetu sa začali objavovať termíny ako Big Data. Pojem Big Data popisuje dáta, ktoré môžu byť rôznorodé, a prichádzať rýchlo a vo veľkom objeme. Aby boli počítačové systémy schopné tieto dáta spracovávať, začalo vznikať niekoľko distribuovaných riešení. Jedným z týchto technológií je práve Apache Spark. Často považovaný za následníka Apache Hadoop a jeho výpočetného modelu Map Reduce. Spark používa inovatívny prístup k distribuovanému spracovaniu dát v podobe acyklických orientovaných grafov.

S neustálym rastom množstva dát a výkonom technológii na ich spracovanie je potrebné, aby aj ľudia pracujúci s týmito technológiami dokázali udržiavať tempo. Metodológia modelom riadeného vývoja prináša niekoľko výhod pre analytikov, vývojárov a iných odborníkov. Základom tejto metodológie je grafická reprezentácia systému diagramom a následne vygenerovanie výsledného kódu. Abstrakcia, ktorú modelom riadený vývoj prináša, pomáha zvyšovať produktivitu pri analýze a vývoji systému, ale aj pri komunikácii medzi expertami z rôznych pozícií. Modelom riadený vývoj pokračuje v stopách objektovo-orientovaného prístupu k vývoju, ale snaží sa čo najlepšie abstrahovať technológiu a kód, tým že systémy majú detailne definovaný doménový model, z ktorého je možne vytvoriť plne funkčný systém.

Cieľom tejto diplomovej práce je preskúmať a navrhnúť spôsob, ako spojiť modelom riadený vývoj a spracovanie Big Data pomocou úloh v Sparku. V práci sú najprv detailne priblížené obe tématiky - Spark a Modelom riadený vývoj (MDD). Po zoznámení sa s potrebnými konceptami sú predstavené riešenia, ktoré sa zameriavali na podobný problém.

V druhej časti je popísaný navrhnutý meta-model, ktorý sa používa pri modelovaní úloh v Sparku. Tento meta-model vychádza z navrhnutého doménového modelu. Meta-model bol realizovaný ako profilový diagram rozširujúci diagram tried. Doménový model vychádza z prístupu Sparku k spracovaniu rozsiahlych dát. Je reprezentovaný ako súbor acyklických orientovaných grafov. Profilový diagram a diagram tried bol vybraný z dôvodu vysokej podpory existujúcich nástrojov a celkovej znalosti v sfére objektovo-orientovaného vývoja. Profilový diagram definuje niekoľko stereotypov, ktoré rozširujú slovník triedneho diagramu. Tieto stereotypy obsahujú tzv. *tagged values*, ktoré umožňujú konfigurovať konkrétne entity. Ďalej sú popisané detaily, ako sa s meta-modelom pri tvorení konkrétneho modelu pracuje, aby bol vygenerovaný kód Sparku valídny.

V časti implementácie je popísaný Eclipse Acceleo generátor. Acceleo používa štandard od Object Management Group (OMG) zvaný M2T - Model to Text. Implementovaný generátor zo vstupného modelu generuje kód v jazyku Scala. Vygenerovaný kód je navrhnutý tak, aby bol ako model, tak aj meta-model rožšíriteľný. Tak isto je navrhnutý tak, aby bolo možné potenciálne opačné generovanie modelu z kódu.

V poslednej časti je navrhnutý meta-model a implementovaný generátor vyhodnotený a predstavený na súbore ukážkových Spark úloh, konkrétne; WordCount - považovaný za *Hello, World!* program v kontexte Big Data úloh, PageRank - algoritmus, ktorý Google Search používa na vyhodnotenie relevancie stránky, a spracovanie dát rôznych typov, ktoré poukazuje na ostatné vlastnosti meta-modelu. Tieto úlohy sa snažia ukázať ako výhody, tak aj nevýhody tohto riešenia. Rovnako ukazujú aj vylepšenia oproti predstaveným, už existujúcim riešeniam. Zdrojové súbory generátora a Eclipse Papyrus projekt meta-modelu boli zverejnené ako open-source na serveri GitHub.

Na záver je vyhodnotený celý prístup a výstup diplomovej práce, sú priblížené výhľady na budúcu prácu v tejto problematike, ako napríklad priblíženie abstrakcie, aby modely neboli viazané na Scalu alebo opačné generovanie modelov zo zdrojového kódu.

Model Driven Development of Spark Tasks by Means of Eclipse Acceleo

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. RNDr. Marek Rychlý, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

....
Marek Šalgovič
May 14, 2022

Acknowledgements

I would like to help my supervisor RNDr. Marek Rychlý, Ph.D. for his help and consultations with this thesis. His insight and advice helped both with the design and the implementation.

Contents

Chapter 1

Introduction

With the world constantly evolving day to day, people want to understand what is happening. To properly understand something, one needs to analyze data. In today's age, the volume of the produced data rises exponentially. The term Big Data was first coined in 2005 and was a direct consequence of the development of Web 2.0. Many distributed frameworks were developed with the need to process data that contains greater variety, arriving in increasing volumes and with more velocity. One of these frameworks is Apache Spark. The successor to Apache Hadoop's Map Reduce uses an innovative programming model for distributed data processing. Chapter 2 describes Spark in more detail and mentions some details and workflows necessary for this thesis.

As the volume of the data increases, the need to develop applications in processing frameworks follows the same trend. To reduce the necessary work to develop these applications, people search for an optimization. One possible approach is to formalize the design and then automate the development. A Model-Driven Development approach introduces a way to develop software using graphical modeling. Afterward, the source code of the application is generated. An overview and advantages of Model-Driven Development and Model-Drive Architecture can be found in chapter 3.

This thesis' motivation is to design a method to formally model Big Data processing tasks in Spark and implement the solution. With the rundown of the essential concepts out of the way, the next chapter 4 describes the related and already existing work in this field. Previous projects offer inspiration but also an insight into what can be improved.

Chapter 5 goes over the proposed meta-model design while describing it in detail. Analyzing the previous solutions turned out the already existing ecosystem for modeling is designed with extensibility in mind. The meta-model to describe Spark tasks is an extension of the UML Class diagram using the UML Profile diagram. By using already established tools, the meta-model becomes easier to extend and also learn.

The aim of chapter 6 is to take a closer look at the implementation of the code generator in Eclipse Acceleo. Acceleo implements the „MOFM2T" standard, from the Object Management Group, for performing the model-to-text transformation. The generator consists of OCL templates and Java services and produces any text from a model input.

In the last chapter 7, I evaluate the proposed meta-model and the implementation of the generator using sample test cases. The first test is a Word Count application often considered to be the *„Hello, World!"* program for Big Data processing. The next test case is the Page Rank algorithm used by Google Search. This case focuses on presenting more features and the possibility of modeling a more complex task. Finally, the last example showcases the use of the remaining features and type system using Spark's datasets.

Chapter 2

Apache Spark

Apache Spark is an open-source framework that enables data scientists, data engineers, and machine learning engineers to run large-scale data processing distributed across a cluster, mainly Big Data processing. It ensures necessary features, such as data parallelism and fault tolerance. Spark provides expressive and intuitive API for several popular languages - Python, Scala, Java, SQL, and R. The Spark ecosystem also offers many tools and settings to make Big Data processing more straightforward. One of these tools is a command-line interface in Python and Scala used for quick ad-hoc data analysis and simple applications. It was designed to effectively support various types of workloads - batch processing, stream processing, interactive queries and interactive algorithms, machine learning training, and graph analysis. That means Spark is a good choice for developers since they only need to use one engine for multiple types of data processing.

2.1 Apache Hadoop Map Reduce

Spark is considered a successor to Map Reduce of the Apache Hadoop framework. Both technologies analyze Big Data, but their approach is very different. Apache Spark claims to be 3X - 100X faster than Map Reduce [7].

To understand the differences between these two popular processing engines, let us look at Map Reduce. Hadoop's Map Reduce distributed processing is as simple as three operations or steps.

- **Map**: each worker node inside a cluster applies the `map` function and produces new data stored in temporary storage.
- **Shuffle**: Worker nodes now redistribute data created by map in the previous step with their output key.
- **Reduce**: workers now apply the `reduce` function on shuffled data.

Because of the simple nature of this approach, multiple aspects might be improved upon. First, the Map Reduce API is too complicated and requires a lot of boilerplate code. Secondly, it lacks the possibility of combining other data processing workloads such as machine learning tasks or stream processing. Finally, complex data processing tasks need to chain multiple Map Reduce operations. That makes Map Reduce tasks very disk dependent. The data are written back to the local disk of each worker node after every Map Reduce stage.

2.2 Spark's design and architecture

To improve these shortcomings, other data processing frameworks were developed. One of these frameworks is Spark which uses a multi-stage approach to distributed processing instead of a two-stage one. Spark uses in-memory storage after each computation. Also, it includes libraries for workloads such as interactive queries (Spark SQL), real-time data stream processing (Structured Streaming), machine learning (MLlib), and graph processing (GraphX). Under the hood, Spark constructs a directed acyclic graph for its computation.

Spark's design philosophy centers around four key characteristics [5].

Modularity

As mentioned above, Spark operations can be applied across many types of workloads and written in any of the supported programming languages mentioned above. In addition, Spark also provides highly documented libraries that include the following modules - Spark SQL, Spark Streaming, MLlib, and GraphX.

These modules can all be used together in a single application without the need to learn different engines and libraries to process the desired task.

Extensibility

Spark design focuses on its fast in-memory processing engine. That makes decoupling of computation engine and storage possible. Moreover, since the storage is wholly decoupled, an extensive range of input data sources can be used - such as Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, Apache Kafka, cloud storages Azure Storage and Amazon S3.

The community even maintains a list of libraries that enriches the Spark ecosystem. That would not be possible if Spark's design was not extensible or easy to use.

Ease of use

Using Spark, one can build Big Data processing applications with a simple programming model in popular languages. The primary abstraction of data is called Resilient Distributed Dataset (RDD). RDDs bring simplicity to the API. Higher data abstractions are constructed upon it - DataFrames and Datasets. Programming Spark application is then simply using actions and transformations on these data structures.

I will describe the Resilient Distributed Dataset and other data abstractions below in more detail 2.2.2.

Speed

As already mentioned, Spark outperforms Hadoop in speed. Futhermore, Spark is able to utilize computer hardware thanks to its fundamental architecture design fully. Spark transforms computations to directed acyclic graphs (DAG). DAGs are then processed and optimized by graph algorithms and redistributed to worker nodes across the cluster for parallel execution.

2.2.1 Spark architecture

As already mentioned, Spark is a distributed data processing system. That means multiple machines inside a cluster collaborate to execute the submitted task. Let us take a look at the architecture of this distributed engine [3] [6]. Spark uses a standard master/worker pattern shown in 2.1.

- *Driver* - The machine responsible for initiating the computation. This machine has multiple roles. First, it is responsible for instantiating and hosting the `SparkContext` inside its JVM process. Second, it requests resources from the cluster manager node, such as memory or CPU. Finally, it transforms Spark code into DAG and distributes it to worker nodes. That means the driver is responsible for coordinating workers and the overall execution of submitted tasks.

- *Cluster manager* - is the master node of a cluster. It manages the resources of workers and makes them run the executor program. Spark is compatible with multiple implementations of cluster managers closely described in 2.2.2.

- *Workers* - also known as slaves - are responsible for executing the calculation. They are the compute nodes where the executor program lives. When the `SparkContext` is initialized each worker node starts its executor (JVM process). This process does not stop after every step and waits for more commands from the driver. Thanks to this Spark worker nodes are able to compute received operations of constructed DAG faster.

  The fact that the executor is a JVM process makes both horizontal(multiple worker nodes) and vertical (multiple executors on a single node) scalability possible.

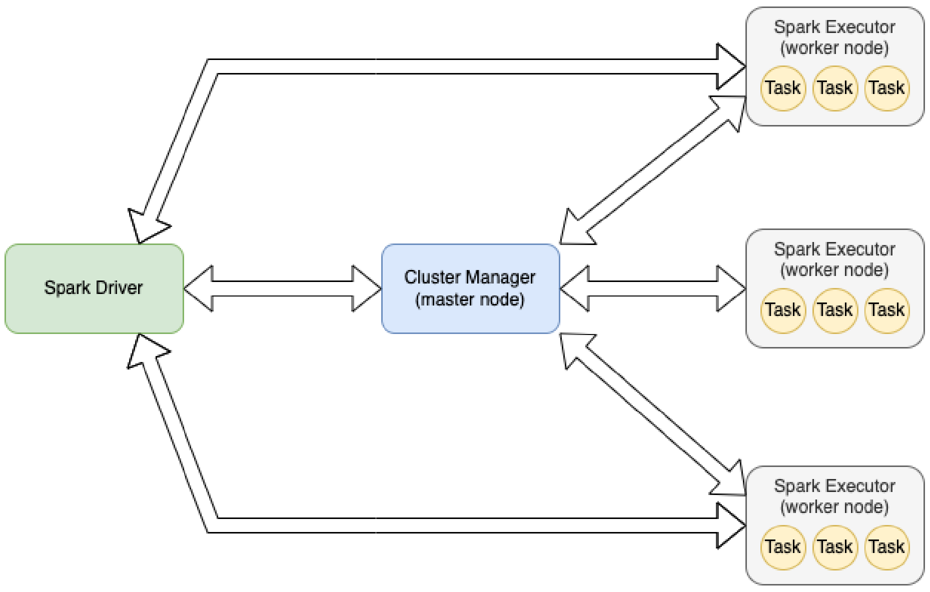

Figure 2.1: Cluster architecture of Apache Spark nodes

2.2.2 Spark ecosystem

Spark creators have won a prestigious award for one of their publications, describing Spark as a „Unified Engine for Big Data Processing“ [19]. As shown in 2.2, Sparks unified stack offers multiple modules, cluster managers, and APIs. Unlike all the other processing frameworks, Spark unifies all components under a single engine. Although Sparks contains its own modules, it is still possible to develop open-source libraries or packages - https://spark-packages.org/. All of these modules are separate from the core computational fault-tolerant engine. That means every Spark application, whether using all or no modules or written either in Python or Scala, is still processed and decomposed into DAG, which the Spark core executes. Using a modular ecosystem brings multiple advantages. The first is the nature of the layered structure. If a component on the lower level is optimized, all the components on any higher level become faster as well. Second, modules were designed to be highly compatible. As a result, when developers need to integrate multiple workloads, they need to set up and maintain only one system. That reduces a lot of cost and time to use.

Now let us talk about the parts in more detail.

Spark Core

Spark Core is the base of Spark functionality. It is responsible for the main features of Spark - task dispatching, memory management, communication with the data sources, and much more. It is also the place that provides the specialized data structure called RDD (Resilient Distributed Datasets) 2.2.2. Spark Core offers API for this structure.

Spark SQL

Spark SQL is used to work with structured data. It offers an abstraction built on RDD called DataFrame. This structure offers more information about the data than the RDD interface. That means Spark SQL is able to optimize the calculations and store them in permanent or temporary tables. Spark also supports SQL and HQL (Hive Query Language) used in Apache Hive. Sparks SQL also follows standard [1] and can be used as a pure SQL engine. Thanks to this module, one can read data stored in various RDBMS or structured file formats such as CSV or JSON.

Spark Streaming

Big Data is often created in real-time, and there is a need for real-time stream processing. Spark Streaming is a module that solves this issue. It provides an API to manipulate and process stream data similar to the Spark Core RDD fault-tolerant usage. The advantage of this module is that developers can use both stream and static data inside a single application. For example, they can treat logs from a web server as they would static data inside a table. Also, streams can be consumed from various data sources such as Kafka, Kinesis, HDFS, or Twitter. This is a perfect example of Spark's ease of use and modularity.

[1] https://en.wikipedia.org/wiki/SQL:2003

MLlib

With the rise of machine learning (ML) and its computational heavy training algorithms, Spark also implemented the support to distribute the computation to multiple machines. MLlib also contains a new data abstraction called Dataset. Dataset APIs offer multiple algorithms, including classification, regression, decision trees, clustering, and many more. As mentioned above, Spark is faster than Hadoops Map Reduce in many ways. MLlib claims to run a logistic regression algorithm 100X faster [1].

GraphX

The last tightly integrated module is GraphX. It can be used to compute graph algorithms in parallel. This module can be easily used with MLlib, SQL, and Streaming which only broadens the possibilities of the Spark framework.

Cluster managers

Cluster managers, also known as task schedulers, are mainly responsible for resource allocation across the worker nodes inside a cluster. A cluster manager decides what node and when should run the task executor. Since the Spark computation cluster might contain hundreds to thousands of nodes, this job is essential. Because of the modular approach of Spark, it is possible to use different kinds of managers, as stated in Spark documentation [16].

- Standalone - the out-of-the-box solution. It is native to Spark, and its main goal is to be easy to use. It only supports FIFO (First in First Out) scheduling which might not be optimal.
- YARN - released with Apache Hadoop version 2.0, and the advantages are the data locality inside HDFS and better scheduling and resource management than the standalone deployment.
- Mesos
- Kubernetes (experimental)
- Amazon EC2

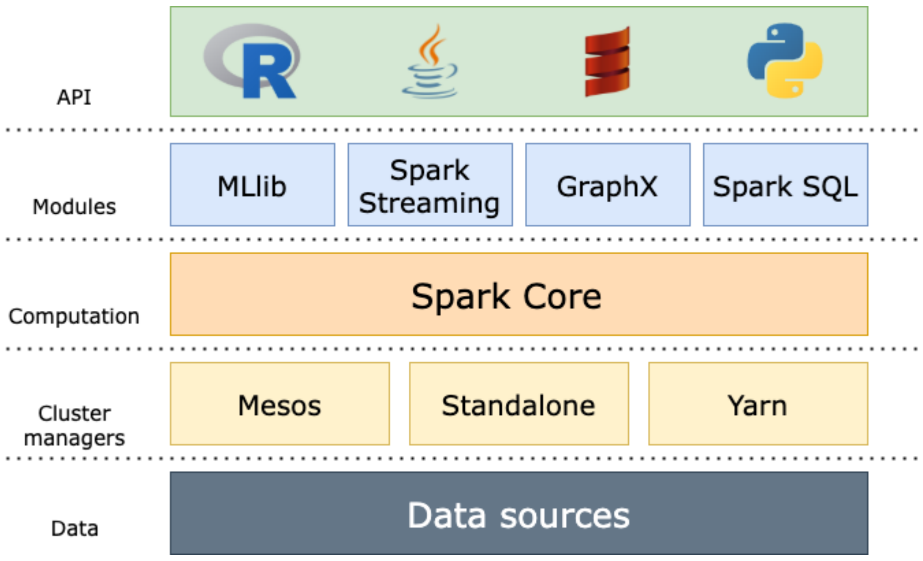

Figure 2.2: Unified stack of Apache Spark

2.3 Resilient distributed datasets

Spark achieves its high speed by using a data abstraction called Resilient Distributed Dataset. They can be described as read-only collections of objects with assured fault-tolerant parallel processing, hence the name RDD. They were first introduced in the paper [21]. Let us go over these features in more detail [6]:

- **Resilient** - fault-tolerant, RDD can be recalculated after a fault of a node or when a partition is missing. That is possible because a lineage graph of all operations is kept.
- **Distributed** - data reside on several nodes of a cluster
- **Dataset** - collections of data with primitive or structured values that represent records of the data you work with.
- **In-Memory** - as previously mentioned, Spark calculates and holds data in memory to make the processing faster. It tries to keep as much data and as long as possible to optimize time and size.
- **Immutable** - RDDs are read-only structures. That means they do not change after they are created. After using a transformation on RDD a new RDD is created.
- **Lazy evaluated** - another feature that can optimize the calculation is lazy evaluation. It means RDD is only calculated after an action prompts the execution. That means that multiple transformations might be chained before the evaluation.
- **Cacheable** - even though Spark uses in-memory computations, it is still possible to hold data in persistent storage.
- **Parallel** - since the data is distributed across the cluster, parallel processing is possible.

- **Typed** - RDD records have types - primitive, e.g., Int, String, or structured, e.g., tuples (Int, String) or objects.
- **Partitioned and Location-Stickiness** - records of RDD are split into logical partitions and distributed across the cluster. One can also define placement preference for RDD.

Users can manipulate RDD with two types of operations. Let us describe them in more detail.

- **Actions** - unlike transformations, action calls do not return another RDD. Actions start the worker nodes' evaluation of the chain of transformations. Actions results are sent back to the driver node. Examples of these operations are count, reduce, foreach, and others. Actions can also save the result to data storage instead of sending it back to the driver node, such as the saveAsTextFile function.
- **Transformations** - operations applied on existing RDD that create new RDDs. DAGs that are later optimized consist of a chain of these operations. Transformations use lazy evaluation, and two types of transformations exist.

Transformations can be described as having either narrow or wide dependencies. The two types of transformations are illustrated on 2.3

Any transformation that can be evaluated from a single partition as input is narrow. *Narrow transformation* is computed on a single partition of RDD without the need for knowledge of the other partitions - without any exchange of data between nodes. Some examples of these transformations are `map`, `filter`, `union`, or `flatMap`.

The second type is so-called *wide transformation.* Computing this transformation will collect data from multiple nodes, process it, and shuffle it. This transformation can lead to repartitioning of data and even a change in the number of partitions. These operations are time-consuming since a ton of data has to move around. Examples of these operations are `groupBy` or `orderBy`.

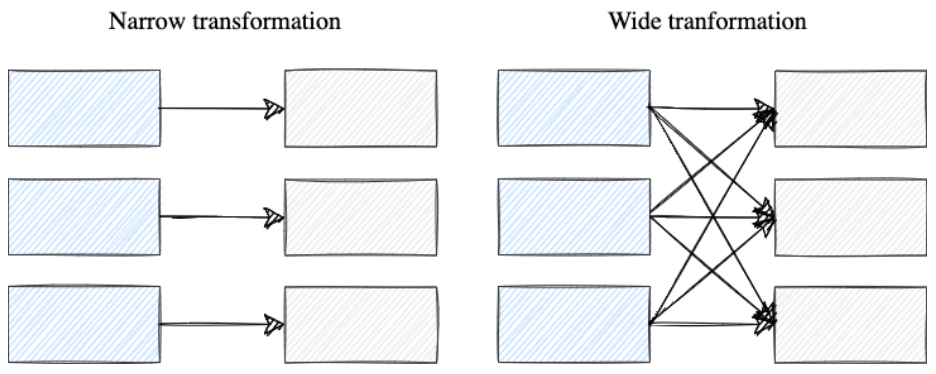

Figure 2.3: Two types of RDD transformations

DataFrame and Dataset

With the development of Spark and its module came a need and a possibility to create new data abstractions. A new data structure abstraction called *DataFrame* has been introduced in the Spark version 1.3 release. A few versions later, specifically 1.6, the next abstraction was added called *Dataset*. In the major version of Spark 2.0, these were merged [18]. DataFrame is only an alias for `Dataset[Row]` in Scala programming language or `Dataset<Row>` in Java. That means it is Dataset with the type `Row`. These data structures are heavily tied to the Spark SQL module.

DataFrame

DataFrame takes features from both RDD and relational database tables. That makes it immutable, distributed, fault-tolerant, and structured in columns and rows. They were designed to make data processing easier with higher-level structured abstraction. In addition, providing SQL-like language to manipulate the data makes it possible for a wider audience of developers.

Dataset

Datasets are the newest data structure in the Spark ecosystem. It provides benefits of both RDDs (strongly typed, lambda functions) and Spark SQL's execution engine. Even though the Dataset API is available only in Scala and Java, Python and R are still able to utilize multiple benefits due to their dynamic nature.

Let us take a look at why merging Dataset and DataFrame APIs was beneficial.

- **Static typing**- Syntax and semantic analysis are already being checked at compilation which prevents possible run-time errors.
- **Highly abstract data structure** - as I already mentioned, DataFrame is an alias for `Dataset[Row]`. That makes the data highly structured with the possibility to map it to a user-defined class in a supported language.
- **Ease of use** - the API offers functions similar to SQL aggregation operations. It may lead to a reduced number of operations from the use of RDDs.
- **Performance** - Spark SQL uses an optimizer called Catalyst. Thanks to this optimization, using DataFrames can lead to better performance than RDDs which do not use any built-in optimizer. The part of Catalyst is the Tungsten memory manager. Tungsten converts data to binary form that takes up less memory than serialized data.

2.4 Example of Spark code in Scala

Finally, let us look at some code snippets of the Spark framework in Scala. These snippets will be helpful when discussing the code generation of the Spark application.

I already mentioned that a new RDD is created after using a transformation on an already existing RDD. There must be a way to create RDD from pure data. Listing 2.1 shows how new RDD can be created in Scala. The usage of transformation and action operations are displayed on listing 2.2. Finally, usage of Dataset and conversion back to RDD is shown in listing 2.3 taken from [17].

```
// sc - SparkContext
// create rdd using parallelize
val rdd1 = sc.parallelize(1 until 10)
// create rdd from file
val rdd2 = sc.textFile("/path/to/data/file.txt")
```

Listing 2.1: How to create new RDD

```
// sc - SparkContext
// transformation of previously created rdd
val odd = rdd.filter(x => x%2 == 1)
// action does not return new rdd
odd.foreach(println)
// prints out: (the order is random because RDD is distributed)
// 1
// 7
// 9
// 3
// 5
```

Listing 2.2: Usage of transformation and action on RDD

```
val linesDS = sc.parallelize(Seq("Spark is fast", "Spark has Dataset", "Spark
    Dataset is typesafe")).toDS()
val words = linesDS.flatMap(_.toLowerCase.split(" ")).filter(_ != "")
val groupedDS = words.groupBy("value")
val countsDS = groupedDS.count()
countsDS.show()
val rddFromDS = countsDS.rdd
// +--------+-----+
// |   value|count|
// +--------+-----+
// |    fast|    1|
// |      is|    2|
// |   spark|    3|
// | dataset|    2|
// |     has|    1|
// |typesafe|    1|
// +--------+-----+
```

Listing 2.3: Usage of Dataset

Chapter 3

Model-driven Development

Model-driven Development is a methodology of building complex software with the use of simplified abstractions of already existing components. The main idea is to move the development to a higher level of abstraction that is later used to generate the source code. These components are visual building blocks that show the application and the business needs instead of looking through complicated lines of code. MDD helps bridge the gap between developers and business domain experts.

Model is the fundamental part of the MDD paradigm. It defines the behavior, structure, and functionality of a system or a part of it. The MDD approach focuses on the construction of a visual representation of a software model. The model specifies how the system works and is used to generate the necessary code.

MDD methodology claims to have multiple advantages:

- It increases developers' productivity because most of the code is generated from the model or its components.
- The fact that MDD increases the system's abstraction level makes the communication between developers and other business experts easier. Business experts often do not understand the complex code but can communicate their ideas to developers using a simple graphical representation.
- Since the code is generated, it becomes more consistent across the codebase because of reusable components.
- With enough support for code generation, modeling a system is independent of the underlying technology.

In any technical decision, there always comes a trade-off. So let us mention some flaws that might hold MDD back.

- The complexity of some artifacts might be too high to model properly with abstract modeling. That may lead to manually editing the generated code, creating inconsistencies across the codebase.
- The usage of models or UML is very different across the developers. Some may sketch the model to get the idea across. Others might use it to design the software, later coded by the developers. These two approaches differ from the MDD methodology of using modeling as a programming language.

- To create proper code generation tools, most of the MDD notations need to be standardized.

There are still a few things a developer could question, such as performance, maintainability, or scalability. Performance could be compared to traditional compilers, where it took a long time and effort to create optimal compilers, while MDD is still a developing concept. Maintainability comes from the usage of MDD. If the developer changes the code without reflecting the change in the model, they can introduce many inconsistencies. This problem could be solved with a reverse code generation to regenerate the graphical model from code. Finally, scaling is the biggest advantage MDD offers. Providing a higher level of abstraction to a system worked on by hundreds of developers can hold the generated codebase and diagrams more consistent across the company [15].

Model-driven architecture

MDA is an approach to software design, development, and implementation introduced by The Object Management Group [1]. MDA offers guidelines for structuring software represented by models. It separates the business and application logic from the underlying platform. That means platform-independent models of applications or systems can be realized on virtually any platform. It also means the separation of the two that creates a possibility for each to develop at their own pace. The business quickly responds to business needs, and the technology improves with the new development [10].

3.1 UML - Unified Modeling Language

The Unified Modeling Language is a formally defined and standardized modeling language to specify, design, and document software systems (or other systems). It is a programming language independent. UML is widely known and used to model systems, especially object-oriented software.

There are three common ways people use UML [8]:

- **as a sketch** - usually by hand without the need of modeling tools
- **as a blueprint** - with the use of modeling software to generate parts of the source code
- **as a programming language** - the model is designed so it can run without any other interaction

In UML, we want to work with the abstraction of reality, and with different language means, we create the abstraction of the system - the model. In this model, we try to capture relationships between different elements of the designed system. As the systems we try to design can be pretty complex, we need to use multiple types of diagrams. Therefore, UML offers several types of diagrams that can be categorized into views. This categorization helps capture only the relevant parts for the modeling and ignore the others.

The set of views that is known as the 4+1 Views of Software Architecture(see figure 3.1) with examples of a diagram is as follows [20]:

[1]https://www.omg.org/

- **logical view** - an abstract description of the structure of units of implementations; modeling of its components and relationships(class diagram, sequence diagram)
- **process view** - the model of system behavior (activity diagram)
- **deployment view** - model the components required for deploying the system (deployment diagram)
- **implementation view** - describes how components are organized in packages and modules (component diagram)
- **use case view** - captures the user requirements of a system (use case diagram)

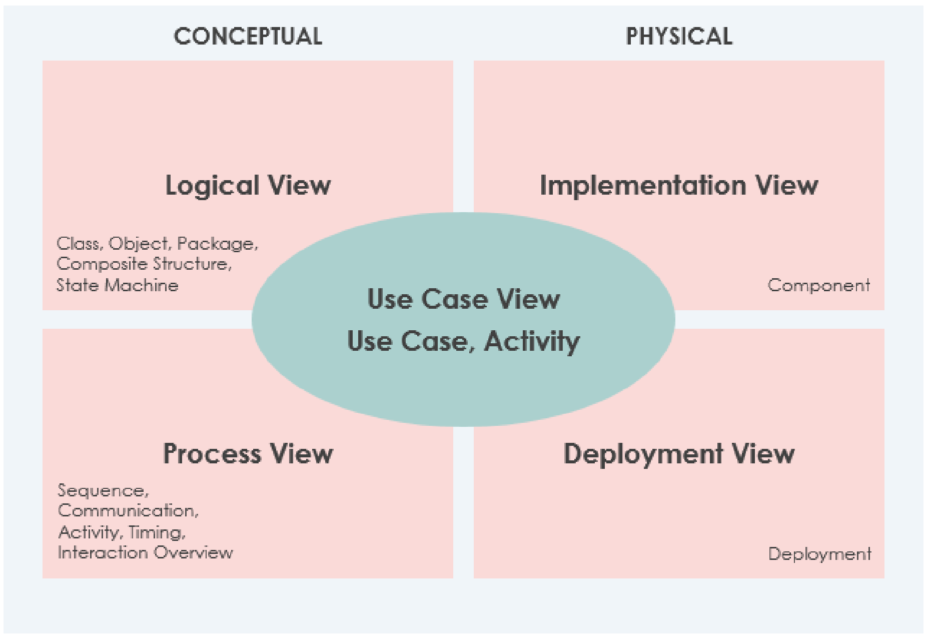

Figure 3.1: UML views

3.2 UML profile

UML, on its own, is a very generic modeling tool. However, when a developer wants to model a domain-specific problem, the standard UML might not be sufficient. The need for an extension of UML that would specify the domain-related problem made the OMG consortium include an extension mechanism called UML profile [12]. Profile package included in UML provides the means to extend UML meta-classes that still follow the UML standard. The usage of profiles can lead to adjusted UML for a specific domain problem. That means the code generators for a specific domain can be developed easier. This technique is not used to create a new type of diagram; instead it extends an already existing type. Most UML editing software (CASE - Computer-Aided Software Engineering) supports the creation and usage of profiles because they only graphically extend elements of already existing and well-known diagrams.

To define UML profile we use three standard elements as defined in [8]:

Stereotypes

Stereotypes signify that an element has a particular use or intent. Stereotypes are most often shown by specifying the name of the stereotype between two guillemots - `«stereotype»`. The number of stereotypes an element can be extended by is not limited. In that case, they are separated by a comma: `«stereotype1», «stereotype2»`. As shown in figure 3.2, while modeling Java classes, we can define stereotypes such as Entity Bean and Session Bean that inherit from the Bean stereotype. Other well-known stereotypes often used in the use-case diagram are `«include»` and `«extend»`.

Tagged values

Tagged values are tied to stereotypes. A stereotype may have multiple tagged values associated with it. A piece of extra information is provided in a key-value manner. Figure 3.3 shows how tagged values are graphically represented. They can also be displayed in a note attached to the stereotyped element. As stated, tagged values are used to extend the properties of UML and are most commonly used for code generation, version control, configuration management, or authorship.

Constraints

Although stereotypes and tagged values are graphically represented in a diagram, constraints are not. Constraints impose rules and restrictions on model elements. Constraints can be defined in informal language or using OCL - Object Constraint Language. Most diagram editing software supports this language and can check the validity of the diagram based on these constraints.

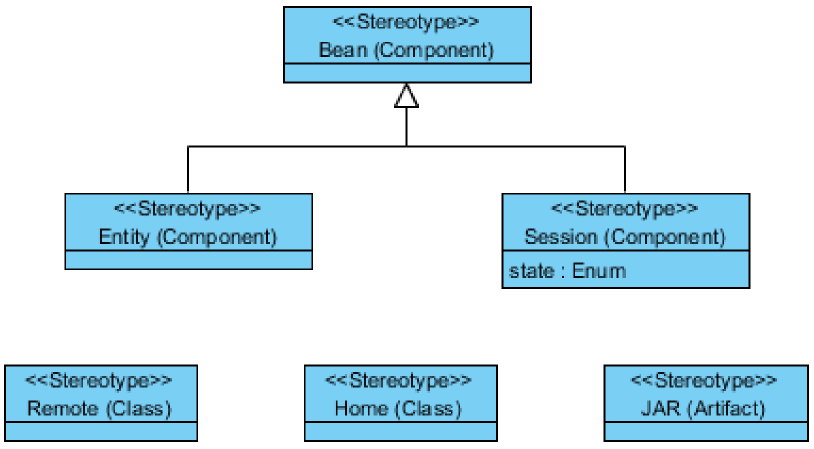

Figure 3.2: Example of stereotype

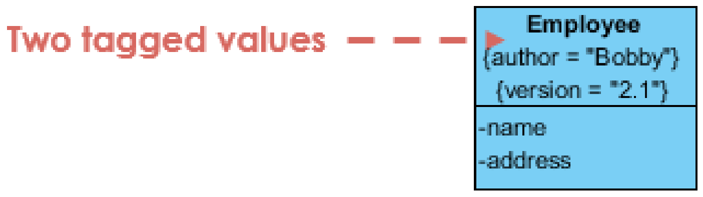

Figure 3.3: Example of tagged values

3.3 Eclipse Papyrus

Eclipse Papyrus is an open-source graphical editing tool for UML 2 as defined by OMG. Eclipse Papyrus targets to implement 100% of the OMG specification. It provides an editor for almost all UML diagrams and supports UML profiles [13]. Papyrus also provides complete support to SysML to enable model-based system engineering. It includes an implementation of the SysML static profile and the specific graphical editors required for SysML. Papyrus is an excellent addition to the Acceleo workflow and Eclipse ecosystem overall.

3.4 Eclipse Acceleo

Eclipse Acceleo is a code generator implementing the Model-to-text specification [11] defined by OMG. Acceleo lives in the Eclipse ecosystem, which is rich in modeling features. Acceleo is not only a code generator but has many features to extend Eclipse IDE with helpful tooling. Acceleo can generate any kind of code with its MTL (Model to Text) language that is compatible with any EMF-based model (Eclipse Modeling Framework) such as UML or SysML. The code generation language uses a template-based approach. A template is a text containing dedicated parts where the resulting code is then calculated from the input model. The dedicated parts are expressions specified on the element of the input model to extract pieces of information defined in it. Acceleo uses OCL (Object Constraint Language) to extract these pieces of information.

Acceleo supports incremental generation. This feature allows people to modify pieces of generated code without losing these modifications when the code is regenerated, allowing the usage of generators to be more flexible. These areas are defined using the `[protected]` tag.

Acceleo is written in Java and is deployed as an Eclipse plugin and integrated into the Eclipse IDE. This plugin also brings multiple tools for the development of an Acceleo generator, such as an editor with syntax highlighting, auto-completion, error detection, a debugger to check the state of generation step by step, and a profiler.

Since only running an Acceleo generator using Eclipse IDE is not optimal, Acceleo can also be used as a stand-alone application. The parsing and the generation engine are generated as a Java class that allows an Acceleo generator to be programmatically integrated into any Java application.

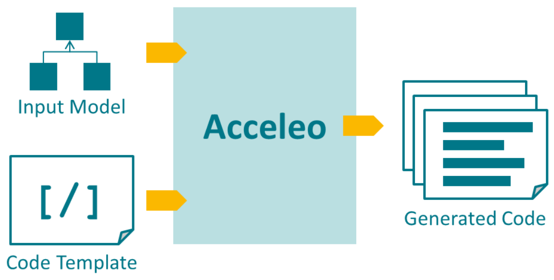

Figure 3.4: Eclipse Acceleo [2]

[2]https://www.eclipse.org/acceleo/overview.html

Java services

Using OCL inside `[query]` construct in Acceleo might not always be enough to extract or modify the relevant information. Therefore, Acceleo offers an option to invoke Java code inside an Acceleo template. These services allow developers generate even the most complex requests. In order to use a Java service, use the Acceleo `invoke` operation in order to tell Acceleo to call your Java method and return the result. Java services are limited to parameters and return values with a type from one of the meta-models used in the generator or a primitive type (String, Integer, Real, Boolean, etc.).

Chapter 4

Related work

With the continuous rise of MDD several other projects with a similar goal in mind have been developed. The goal of this thesis is to design and implement a model-driven tool to create Spark applications. By closely inspecting already existing solutions we can pinpoint the advantages and disadvantages of the given solution.

4.1 Executable UML

First described in a book Executable UML: A Foundation for Model-Driven Architecture [2]. Both a method and language to develop software, Executable UML (xUML) offers highly abstract and platform independent solution to generate source code from defined model. Even though xUML is a subset of Unified Modelling Language - UML, it uses slightly different semantics. The model does not include any code, but parts of the model are mapped to specific code snippets for target platform. Executable UML is a perfect example of Model-driven architecture in practice.

4.2 Map Reduce generator

With the rise of the need for Big Data processing applications the field also tries to develop tools to make the analytic work more straightforward. The first proof of concept for Model-driven Development of Big Data processing applications was published in [14]. Designing a meta-model for Map Reduce applications and subsequently generating a working source code proved that a Model-driven approach could be applied to processing tasks to bridge the gap between developers and complex systems by abstraction. Even though this project uses a Map Reduce approach to data processing it still brings valuable insight into how Spark application can be modeled. The conclusion of this paper showed that MDD approach increased the productivity without any noticable performance overhead.

4.3 The custom Spark generator

This project was developed as a master thesis [4] with a similar goal. The meta-model of this solution does not follow any standard. That makes it harder to learn. Even though the solution works, the modeling approach is quite complex. Using this meta-model you could model very complex applications as it is quite similar to modeling a Scala program by assigning variables, calling and defining a function, and determining the order of execution. The takeaway from this solution is the need for a meta-model to design the application with a much less complex diagram by trying to reduce the modeling space of an application. Also, the new meta-model has to use standard modeling language instead of a custom one, to make it easily integrable to Acceleo.

4.4 StreamGEN

A very robust solution described in paper [9]. They developed a system with a similar approach and technical stack using UML profile and Eclipse Acceleo. StreamGEN is a system to generate Streaming Applications to multiple platforms - Apache Flink and Apache Spark. StreamGEN consists of a modeling language - StreamUML and an Acceleo code generator - StreamCGM. The meta-model is defined using UML profile to extend the Class diagram and the Composite Structure diagram. The authors decided to generate the application source code as Java code because it is widely supported by different platforms. Since this solution is mainly focused on streaming applications it defines multiple domain-specific stereotypes such as a **WindowTransformer**, **WindowedStream**, or **RandomlyPartitionedStream**. Because the meta-model mainly focuses on stream processing some features are missing

Chapter 5

Meta-model design

In this chapter, I will introduce the designed meta-model that can be used to create a model of Big Data processing in Spark. I will describe its usage, advantages and disadvantages, and thought processes behind the design.

As the previous chapter indicates I decided to extend the existing UML diagram with UML profile to add vocabulary to already existing, well-known language. UML and various domain-specific stereotypes are well known and have great support for both modeling and code generation - especially Eclipse Acceleo. To keep the trend of well-known principles I decided to extend the most common UML diagram - the Class diagram.

The advantages of this approach are extensibility, reusability, and ease of use. The class diagram can be easily extended with different profiles and used by different generators to create a more robust solution. The class diagram is also well known and more easily read than code. Finally, as stated in the previous chapter the Model-driven design makes diagrams reusable solutions as they can easily model parts of the system.

The proposed design also brings a few disadvantages. The designer needs to understand the Spark framework and Scala programming language. It does not abstract the internals enough to seamlessly work without a prior knowledge of Spark. Also, the more advanced features of Spark are unable to be modeled (Streaming, MLlib) as their workflow is structured differently than the generic Spark processing pipeline.

This design is mostly inspired by the related work [9], that was mentioned earlier. They took a similar approach by extending both the Class diagram and the Composite Structure diagram. Although this approach can bring better and more robust abstraction, I find using only one diagram more maintainable and reusable. This can bring a few hurdles that I will discuss later.

Now let us take a look at a Domain model of a typical Spark application. As we can see in figure 5.1, a Spark application structure is rather simple. An instance of Spark application has its own configuration. Most Spark applications use the command-line interface with provided arguments for control. After the Spark application is configured the flow of the program can be represented as a directed acyclic graph. This is similar to how Spark actually works on the inside. These processing graphs begin with the source node. Afterward, multiple transformations can chain to modify the input data provided by the source node. The last type of processing node is the action node. Action node represents an action on computed RDD (or Dataset).

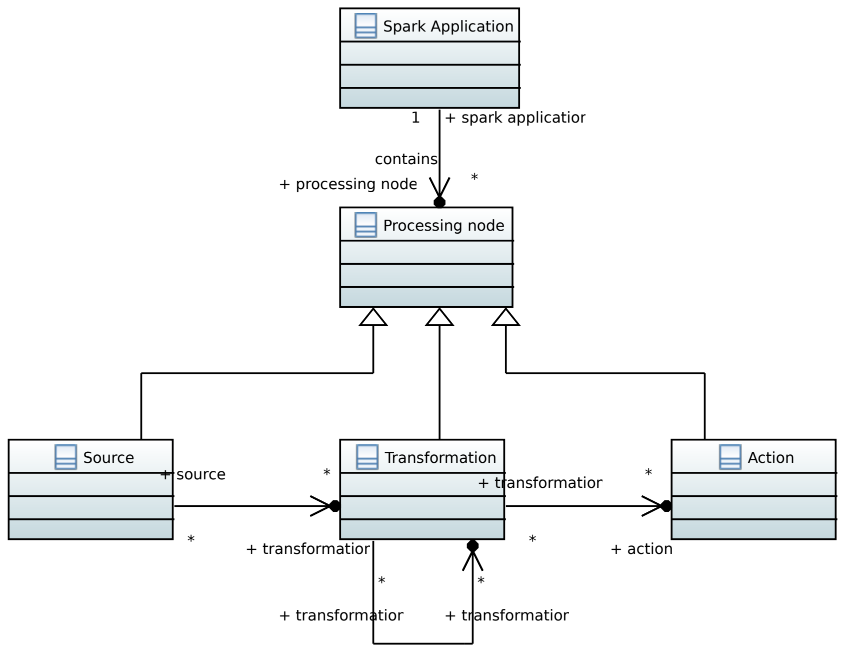

Figure 5.1: Domain model of Spark Application

Up until now, the order of generated code did not matter because the RDDs or transformation nodes were not evaluated. If the respective transformations were chained correctly the flow of the program was apparent. The problem arises when a single Spark application uses multiple actions. Their order must be deterministic because a premature action can modify the not yet evaluated source. That can be avoided by explicitly selecting the priority of a specific action.

With this domain model in mind I will define the meta-model in more detail. Figure 5.2 and table 5.1 show the designed UML profile. As already mentioned UML profile extends the most common diagram - the Class diagram. The most important types of elements of a UML diagram, or more specifically the Class diagram, are relationships (meta-class Association) and classifiers (meta-class Class). The meta-model also extends a less known meta-classes - Model and Datatype, to display the model with a better composition and provide a tuple data structure often used in Spark.

5.1 Description of provided stereotypes

In this section I will describe the designed profile diagram and defined stereotypes in detail. It is necessary to mention the designed meta-model is designed with the Scala programming language in mind. Many tagged values of stereotypes require the user to specify Scala code snippets to define a function or an array of some sort.

First I will describe miscellaneous elements that diagram provides. These elements all contribute to the type system of the meta-model. The `Snippet` primitive is defined to distinguish the fields that expect String primitive or Scala code snippet to be provided. The structered DataTypes `Option` and `Program Argument` are defined as types used in the meta-model to introduce structured tagged values. `Option` is key-value map where key is a string literal and value is a Scala code snippet (either a string literal in quotation marks or a variable name). `Program Argument` defines Spark Application program arguments, their data types and default values. Enumaration File Source Type narrows down the values for `«File Source»` stereotype.

- `«Spark Application»` - this model stereotype defines Spark Application as a whole. Spark applications usually need a configuration. The most common attribute - `master` is defined as an explicit tagged value. Next tagged value is `conf`. `Conf` uses the defined `Option` type to configure the Spark object using key-value pair configuration. Similiarly `arguments` define input arguments for Spark Application and their default values. `Imports` is a list of packages to be imported into the Spark Application. Last, `initialCodeBlock` is used to inject specific code at the start of the generated application.

- `«Processing Node»` - the processing of input with transformations and finally evaluating it with action is the core of Spark applications. The processing node stereotype is abstract and defines the processing graph node. Multiple stereotypes extend it.

- `«Source»` - the entry point of the data processing. Source node represents the initial creation of RDD (or Dataset). Tagged value `priority` is used to determine the order of execution to prevent any undefined behaviour occuring. Source stereotype is extended by specific ways of RDD creation.

- `«RDD Parallelize»` - the most basic creation of RDD. The tagged value `array` represents a Scala code snippet of an array of data, for example - `array="Seq(1,2,3)"`.

- `«File source»` - Spark supports multiple file types to load the data from. The type of file can be selected using the tagged value `format` and the `File source types` enumeration. The enumeration is used to limit the input scope. The path to the input file is configured using the `filePath` tagged value. Similar to `conf` in Spark Application stereotype, file source can be configured with `options` tagged value. The fact that Spark application can load both RDD and Dataset from files means it might be necessary to provide a data type for Dataset.

- `«SQL»` - represents the SQL query from the Spark SQL storage. Using the tagged value `query` to define an input query. Also uses `datatype` tagged value.

- `«JDBC»` - the last supported data source is by loading data from a database. The `URL` tagged value is self-explanatory. The `options` and `datatype` work the same as in the file source. The necessary options are also explicitly used as tagged values - `user`, `password`, `tableName`.

- `«Transformation»` - transformation nodes are the central part of the processing graph. The designed meta-model supports both RDD and Dataset transformations that make this stereotype abstract. The tagged value that both transformations inherit is the `func` value. It is used to define the transformation provided by the Scala Spark API[1]. For example, if we want to use the map function to double the value of records in RDD we would set the tagged value as follows: `func="map(x => x*2)"`.

- `«RDD transformation»` - stereotype representing the RDD transformation. Doesn't contain any tagged values because it inherits `func` from abstract transformation.

[1]https://spark.apache.org/docs/latest/api/scala/org/apache/spark/index.html

- `«Dataset Transformation»` - works similar to RDD transformation but for Dataset API. The purpose of this stereotype is to be able to tell the generator to convert the RDD to Dataset.

- `«Action»` - the next defined stereotype extending the Class meta-class represents the last node of the graph - the Action. Actions' `func` values work the same way as in terms of transformations. Most of the action functions just return a value. `«RDD Action»` and `«Dataset Action»` stereotypes are also defined and work similarly to transformation.

- `«Code Block»` - even though Spark Application mostly consists of the previously mentioned processing graphs, there is still need of some utility code either to display or format outputs or assign values to variables. The tagged value `code` contains a Scala code snippet. The stereotype is also part of the processing graph and the provided code is generated while traversing the graph. The output of the previous node is injected into the code using `$out` variable.

- `«Variable»` - in some specific cases we need to introduce a variable into our computation. We might need to iterate in a loop while joining constantly changing RDD with another RDD. Or we just want our mapper to use program arguments. The need of this stereotype might be more apparent when I discuss it's usage in a later chapter. It is possible to store a specific output of a processing node by connecting this stereotype with Information Flow relationship to it. Tagged value `dataType` defines the dataType variable will hold and `isRDD` is a bool flag to tell the generator variable will contain RDD of the provided data type.

- `«Flow»` - this stereotype extends Information Flow relationship and is used to increase the ability to define the order of execution and reuse already defined transformations. `Priority` is used when a transformation output is sent to multiple nodes. The generated code will traverse the graph in depth-first search manner. `Tag` helps to name a specific flow in the graph. It is only useful if a specific transformation has input from two different nodes. The output of this transformation for each of these nodes is accessed via the same tag. If the Information Flow relationship does not use this stereotype the priority is always considered to be traversed last. Also the graph is generated for each tag from the input flows.

- `«Argument»` - extends the association relationship. Some of the transformations provided in Spark API use another RDD/Dataset to process the input e.g. `zip`, `join` or `intersect`. This relationship helps to inject a `Variable` stereotype as a function argument to a transformation node.

- `«Tuple»` - while working with RDDs in Spark one can use transformations, such as `groupByKey` or `reduceByKey`, that need tuples as their data type. This stereotype extends DataType meta-class making it possible to use it as a data type in model when necessary.

- `«Loop»` - similarly to `«Spark Application»` stereotype this stereotype also extends the Model meta-class to display a proper composition of the application. `«Iteration Loop»` and `«Conditional Loop»` stereotypes define the loop condition since this stereotype is abstract. Both `iterations` and `condition` tagged values are Scala code snippets and represent the behaviour of the loop.

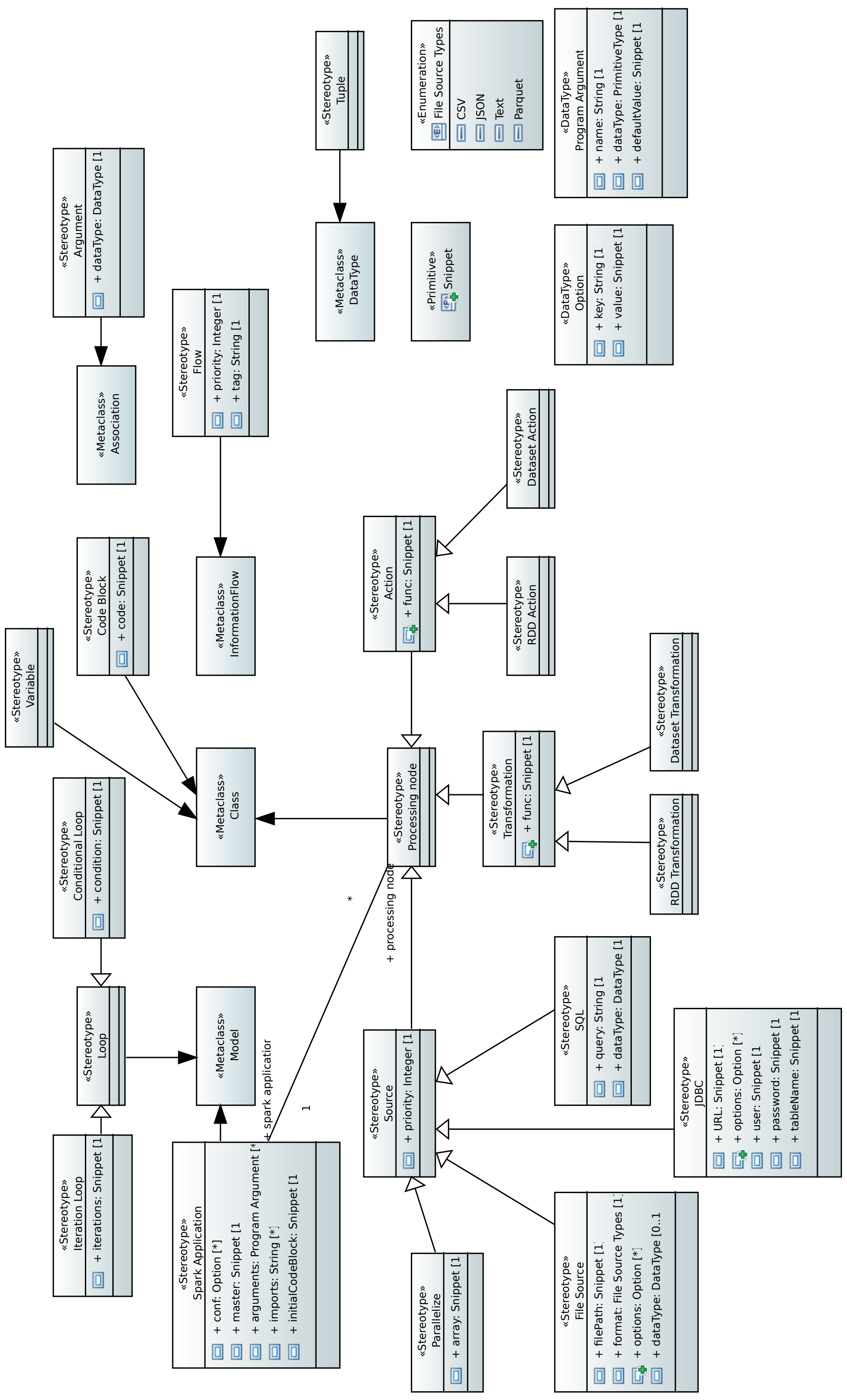

Figure 5.2: UML Profile diagram describing Spark Application

5.2 The usage of the profile

There are a few non-standard design choices that need to be described to understand the usage of the proposed meta-model. In this section I will go over the details that are necessary to understand to use the designed meta-model properly.

Order of execution

The source node and the action node can have multiple outputs and inputs respectively. The transformation node can have both multiple inputs and outputs. This creates a situation where one diagram can define multiple processing pipelines. To determine which code is generated first, the source nodes and Information Flow relationships have priority values. The order of generation is as follows:

1. find the highest priority source node
2. generate the code representing graphs in depth-first search manner where higher number has higher priority
3. select next source node and repeat

In many Spark Application the processing nodes might only contain one output. To make the modeling process more straight-forward, even the Information Flow relationships without the `«Flow»` can be used to model the computation. These relationships are then considered to have the lowest priority when possible branching occurs.

Data Types

The graph, created using Information Flow relationships as edges and processing nodes as nodes, instantiates the processing pipeline of the application. Each edge must contain data type of the RDD that is being processed. This data type is selected using *conveyed* property of the Information Flow relationship.

To define data types used by different elements across the diagram, such as Information Flow relationships, variables, and sources, the package element of a Class diagram is used. If the package is contained inside the `«Spark Application»` stereotype model element, the appropriate data structures are also created by the code generator. Class diagram offers primitive data types - Integer, Real, Boolean and String. It is also possible to modify the multiplicity of the attribute to create a list. Meta-model supports the definition of a structure type with primitive data types and their multiplicity. The generated case classes and data types are later described in chapter 6. On the other hand, some data structures, or in this case Scala case classes, can include properties that can not be modeled in the UML Class diagram. The meta-model can still support these data types as an imported anonymous data type. To import the data types, use the `imports` tagged value of the `«Spark Application»` stereotype. Afterward, create a package element with a DataType element inside. This element does not need to have any properties defined, but requires the name to match the imported name. The difference between imported and generated data type can be seen in figure 5.3.

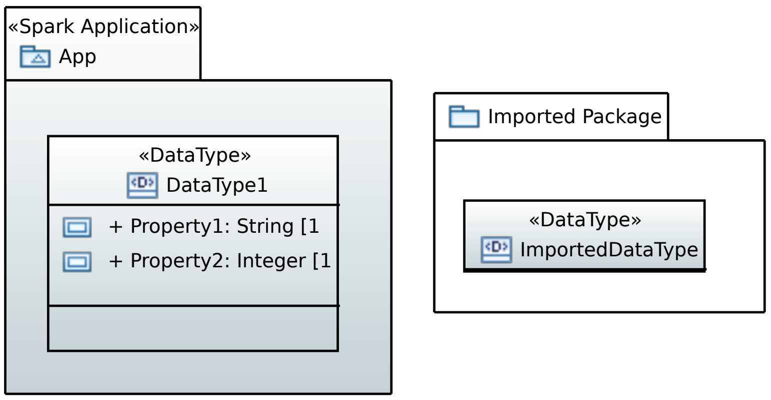

Figure 5.3: Difference between generated and imported data type

Loops

The analyzed related projects did not include any form of loops in their design. Therefore, the addition of loops to the meta-model brought the possibility to model more complex applications. To properly include this feature, both the readability and functionality of the diagram must be considered. Similarly to the Application element, loops are also represented as the model element of the Class diagram. The model element can display what nodes of the computation are inside. Now it is crucial to determine when the loop scope starts and where it ends. As the generator traverses the computational graph, when the first node with a new loop scope is visited, the loop header will be generated. The closure of the loop happens when the generator visits the first node outside of the model element. The meta-model also supports nested loops and thanks to the model element the diagram remains readable.

Tags

The meta-model is designed to be a Class diagram. Classes are not meant to represent objects already instantiated in the application. The overall thought behind the meta-model design is that the nodes represent the structure and the template of computations, while Information Flow relationships instantiate them inside the application's main method. Therefore, all the nodes inside the diagram must be unique. Tagged value `tag` in the `«Flow»` stereotype aims to prevent a definition of multiple classes with the same method, but different name. When the computation requires to process multiple inputs with the same function, it is possible to connect these inputs to a processing node. These input edges must then be distinguished by tagging them with a specific value. To differentiate between the output flows, the same tag must be used in the out-coming relationship. If no tag is used in the output Information Flow relationship, all the inputs are processed and the generator continues to generate the pipeline for each of them. It is important to mention that if only single Information Flow relationship is used it can only hold a single conveyed data type, whereas the inputs types to the processing node may differ. The use of tags is shown in the Dataset processing test case described in the section 7.3.

Datasets

When using Spark's Dataset API, it is often necessary to use the name of the input Dataset variable. An example can be seen in listing 5.1. However, the name of the generated variable holding the value is unknown. Therefore it is necessary to provide this functionality to the model designer somehow. Test case 7.3 shows the code snippet of the solution. Because outputs from all the nodes are cast to RDD, the transformation and action methods need to use API with the $ notation. The input Dataset is then renamed to `this` using an alias method.

```
people.filter("age > 30")
 .join(department, people("deptId") === department("id"))
 .groupBy(department("name"), "gender")
 .agg(avg(people("salary")), max(people("age")))
```

Listing 5.1: Dataset API using the name of Dataset variable

Variables

Another fairly non-standard feature is the use of variables. This element defines a variable in the main method of the application. The name of the element matches the variable inside any Scala code snippets. The injected variable also uses the same identifier inside the transformation and action function.

Stereotype	Inherits from	Type
«Spark Application»	Model	Concrete
«Computation Node»	Class	Abstract
«Source»	«Computation Node»	Abstract
«File Source»	«Source»	Concrete
«SQL»	«Source»	Concrete
«Parallelize»	«Source»	Concrete
«JDBC»	«Source»	Concrete
«Transformation»	«Computation Node»	Abstract
«RDD transformation»	«Transformation»	Concrete
«Dataset transformation»	«Transformation»	Concrete
«Action»	«Computation Node»	Abstract
«RDD Action»	«Action»	Concrete
«Dataset Action»	«Action»	Concrete
«Flow»	«Information Flow»	Concrete
«Code Block»	Class	Concrete
«Variable»	Class	Concrete
«Tuple»	DataType	Concrete
«Loop»	Model	Abstract
«Iteration Loop»	«Loop»	Concrete
«Conditional Loop»	«Loop»	Concrete
«Argument»	Association	Concrete

Table 5.1: The list of defined stereotypes

Chapter 6

Implementation details

For this chapter, I will go over the implementation details of the Spark Application code generator. Both the meta-model definition and code generator are integrated into the Eclipse modeling ecosystem. Futhermore, both Papyrus and Acceleo follow OMG standards. Therefore the designed solution is technologically independent in theory.

The code generator is implemented in Acceleo and can be divided into two parts. First, the generation of necessary Scala classes, and second, the generation of the computational graph. The implementation also contains two kinds of source files. Acceleo `.mtl` template files and Java services, both found in `src` folder. Java services are mostly used to generate the complex computational graph while templates are used to define the folder structure and class implementations.

The final generated **main/scala/{app_name}/** folder has the following structure:

- **actions/** - folder containing action classes
- **dataTypes/** - folder containing dataType case classes
- **sources/** - folder containing source classes
- **transformations/** - folder containing transformation classes
- **{app_name}App.scala** - file with the application object and the main method

The generated folder structure represents the application source code. The best approach to folder structure is to include the generated folder in the `src/` folder of the project. The generator does not generate the whole project but only the needed source code.

6.1 Scala classes

The generator creates multiple classes. To make the model more extensible, each node of the computational graph is represented as a Scala class. Also, the data types defined in the model are generated as Scala case classes. Scala's case classes represent immutable data defined as a data structure with methods. A sample generated data type is shown in listing 6.1. If the case class needs to be enriched with potential methods to modify data, the already mentioned `[protected]` tag is used and represented as a user code comment. This section can be modified and is not included in the model. This code block is not lost after a subsequent generation of the application if a change occurs.

```
case class WordCount(
  var Word: String,
  var Count: Long,
){
//Start of user code WordCount
//End of user code
}
```

Listing 6.1: The generated Counter class

As per Scala's best practices, each class is generated into its own file. This approach to class-based code generation is different than how most Spark Application code is written. I believe that this approach is more extensible and robust, even though it creates more boilerplate code. An example of a generated class can be seen on listing 6.2.

Multiple important details might be pointed out. The class consists of a single method and multiple properties. These properties do not influence the generated application in any way. They serve to demonstrate the potential possibility of reverse model generation. In the future, a tool might be developed that would complement Acceleo's M2T generation by reversing it to Text to Model transformartion. By introducing these properties it is possible to define the stereotype and tagged values of a class.

```
class Counter() {
  val S_rDDTransformation = true
  val TV_func = """reduceByKey(_+_)"""

  def transform[T: TypeTag](rdd: RDD[T]) = {
     typeOf[T] match {
        case type1 if type1 =:= typeOf[(String, Long)] =>
           rdd.asInstanceOf[RDD[(String, Long)]]
              .reduceByKey(_+_)
     }
  }
  //Start of user code Counter
  //End of user code
}
```

Listing 6.2: The generated Counter class

Every single computational node; source, transformation and action contain a single method - `source`, `transform` and `action` respectively. This method implements the logic provided by tagged value `func`. The method uses generic types and reflection because a single transformation or action can have multiple inputs with different types. The use of reflection is closely described in section 6.6. The use of multiple data type inputs to a single processing node can be seen later in section 7.3. Both transformation nodes and action nodes extend RDD and Dataset operations. To abstract away the communication between them, every output is defined as `RDD[T]` of the specific type. This leads to the need of re-typing RDD to Dataset using `toDS()` method, when Dataset action or transformation is used.

6.2 Computational graph

The more complex part of the code generation that creates the flow of the application is the generation of the computational graph. Because of the complex nature most of the generation is located in Java services. The implementation of the computation is located in the main method of the App object. If the previous generated code was represented by classes of the model, this part is represented by Information Flow relationships that instantiate the classes and produce outputs. A few other necessary elements are also included in this part of the generation, namely loops, code blocks and variables.

First, all necessary classes are instantiated. All nodes are stored in variables with a specific name - prefix `S_`, `T_` and `A_` and properly formatted name of the class in the diagram. After that, the graph is traversed depth-first while creating necessary statements. The output of each node is then stored in a variable with a similar name to the node it was produced by. Starting with prefix `s_`, `t_` and `a_` and the name of the node, and ending with suffix `_{tag_name}`. The suffix is omitted if the particular flow does not contain any tag.

6.3 Loops

Some Big Data processing tasks might include a specific loop to process data, e.g., tasks such as k-means clustering, linear regression, or page rank calculation (described in chapter 7 in more detail). Including loops in the meta-model brought many hurdles implementation-wise. Since loops extend model, the graph node must have been searched for recursively in the application model.

Looking at how the loop is structured in code, we can see it wraps statements inside it. To correctly generate the loop while traversing the graph, the header is generated before the node statement when the computation enters a new loop scope. The recursive traversal of the graph keeps track of the loops in the model and currently active loops - meaning loops where the end of the scope is yet to be generated. When processing a new node, the generator checks the scopes it is inside of. If any active loops are missing, the proper loop scope is closed.

As previously mentioned, new variables are defined for each output of a node. Therefore, some of these variables can be defined inside the loop's scope. During the first prototyping of loop implementation, a problem arose when these variables needed to be accessed outside the loop's scope. This situation happens when the graph leaves the loop and uses the output of the last node inside the loop. The implemented code generator handles the problem by defining the variables located inside the loop at the beginning of the graph. Only necessary variables are defined to lower the amount of generated boilerplate code. The solution can be seen in a sample code snippet shown in listing 7.3.

6.4 Data Types

Scala as a programming language is statically-typed. That means all necessary data types must be known during the compilation of a program. The fact that the generator creates much boilerplate code to make the model more extensible also removes the possibility of utilizing Scala's type inference feature. Also, the type system in UML modeling is fairly limited. As previously mentioned, it is only possible for a few primitive types, their multiplicity, or the user-defined structure of these primitives. The import and data type definition outside the model element feature attempts to include more complicated data types. These data types are then generated as {datatype_name}. See the code snippet in section 7.3.

It is necessary to mention the use of primitive types of the UML, especially *Integer* and *Real*. The internal conversion between Dataset and RDD using toDS() and rdd methods, introduces a possible typing problem. To prevent this *Integer* is represented as Long data type in Scala and *Real* is Double. This fact is also necessary to understand when modeling a Spark task using the meta-model. As seen in figure 7.1, the **CountAdder** class uses 1L to define the number as Long.

6.5 Arguments, code blocks and variables

Spark Applications might need to use other code than RDD actions and transformations. This issue is solved by including code blocks and variables. These parts of code are also considered to be processing nodes in the diagram. The generator first defines all variables in diagram with their assigned data type and null value. If the variable is part of the computational graph connected by Information Flow relationship, then a re-assignment statement is generated as the variable is already defined.

Code blocks nodes and initialCodeBlock tagged value are generated at appropriate places as Scala code snippets are used to introduce custom functionality or possibility to model specific work-arounds.

Finally, program arguments are generated as shown in listing 6.3. They represent the application input and can take value of any primitive data type. Futhermore, if a node needs to use it as an argument, simply define «Variable» stereotype with the same name. When the generator creates the variable definitions it omits the variables defined in program arguments, because the generated code would throw variable already defined error.

```
var argMaster = if (args.length > 0) args(0).toString else "local[*]"
var argInt = if (args.length > 1) args(1).toLong else 1
var argReal = if (args.length > 2) args(2).toDouble else 1.1
var argBool = if (args.length > 3) args(3).toBoolean else true
```

Listing 6.3: Generated program arguments

6.6 Reflection

To fully utilize the advantages of model-driven development, we want to reuse already modeled processing nodes as much as possible. With the introduction of tags in the meta-model, it is possible to distinguish the incoming flows to a transformation or action. These flows might convey different data types while still being compatible with the node's function. To support this approach, the generated code uses type reflection and type assertion to limit the generated boilerplate code.

Even though Scala programming language supports method overloading, it can only be used on types without type parameters. As we already know, the processing graph always produces RDD[T]. Therefore, type overloading would not work. The next Scala feature that enables this approach is compound types[1]. Thanks to compound type a Scala

[1] https://docs.scala-lang.org/tour/compound-types.html

method returns values that act as multiple types at once. To distinguish the proper type, the generator uses type casting with the method `asInstanceOf[T]` in the computational graph code. The section 7.3 takes a closer look at the advantages and code snippets of this approach.

Chapter 7

Sample test cases

This chapter will demonstrate the usage, discuss and evaluate the solution. I prepared multiple relevant test cases that display the general usage and some of the features and quality of life design choices in the meta-model. By showing these test cases, I intend to demonstrate the advantages of the meta-model while also pointing out where the model-driven solution to Spark tasks might come short. The diagrams, model meta-files and the generated sources, have also been submitted with this thesis for further inspection.

7.1 Word Count

Word Count is considered to be the *„Hello, World!“* program for Big Data processing. This application counts the amount of each word in an input text. The sample model is shown in figure 7.1.

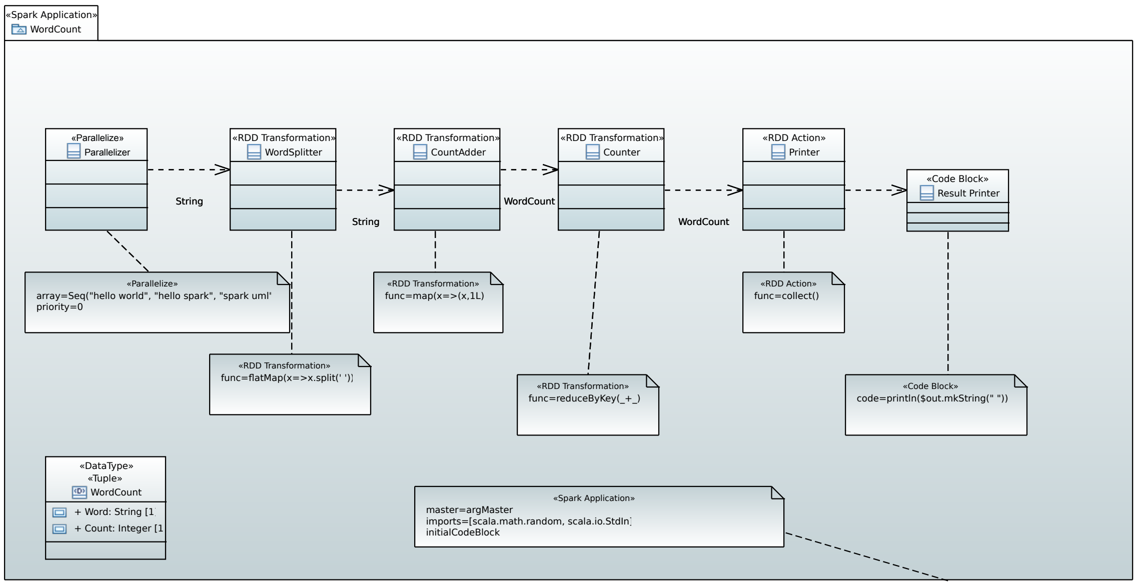

Figure 7.1: Word Count diagram

As seen in the figure, the diagram is a simple directed graph inside a model element. The quick overview of the algorithm is quite simple. First, the input RDD is created. Although it is most often created from a text file, for demonstration purposes, the `«Parallelize»` stereotype is used. Then, the input is split into RDD of strings of specific words using `flatMap`. After that, the Counter transformation processes the input to output defined key-value tuples. These tuples are then reduced by key to calculate the occurrence of each word in the source text. Finally, the action node is used to collect the desired result.

As we can see, the diagram uses the `«Tuple»` stereotype to introduce key-value pair data structure. Every Information Flow relationship conveys a specific data type, as previously discussed, these types represent the type of the RDD sent between two nodes of the graph (for example `RDD[(String, Long)]` for the tuple **WordCount**). The only Information Flow relationship without any data type is used when the graph finishes with the `«Code Block»` stereotype node. The output value from the previous node is accessed using `$out` variable. The diagram also shows the need to use `Long` instead of `Integer` in the **Count Adder** class.

As already stated, the generator creates a bunch of boilerplate code to make it more extensible or potentially introduce a possibility to support reverse model generation. Listing 7.1 shows only the part of the main method, where the computation resides.

```
val s_parallelizer =
  S_parallelizer.source().asInstanceOf[RDD[String]];
val t_wordSplitter =
  T_wordSplitter.transform(s_parallelizer).asInstanceOf[RDD[String]];
val t_countAdder =
  T_countAdder.transform(t_wordSplitter).asInstanceOf[RDD[(String, Long)]];
val t_counter =
  T_counter.transform(t_countAdder).asInstanceOf[RDD[(String, Long)]];
val a_printer = A_printer.action(t_counter);
println(a_printer.mkString(" "))
```

Listing 7.1: The generated Word Count computation

One may notice, that the generated computation seems very uniform and easy to read, that is because the code generator executes in the order of the computational graph. This uniform code also simplifies the potential reverse generation.

An argument can be had that this algorithm is only a simple sequence of steps without any complexity. However, the next test case introduces a more complex problem.

7.2 Page Rank

The next chosen test case is the Page Rank algorithm. Google Search uses this algorithm to measure the importance of a website page. Not only is this algorithm used with Big Data, but it also uses a loop. This fact makes it an excellent example to demonstrate the usage of the designed meta-model. The integration into the Eclipse ecosystem that already has modeling features turns out to be a good choice for any modeling task. Papyrus offers the possibility only to display the selected elements. As shown in figure 7.2, some of the elements are hidden, so the diagram is more readable while still holding the necessary information to generate the final application. Now let us look at some of the features used in this diagram.

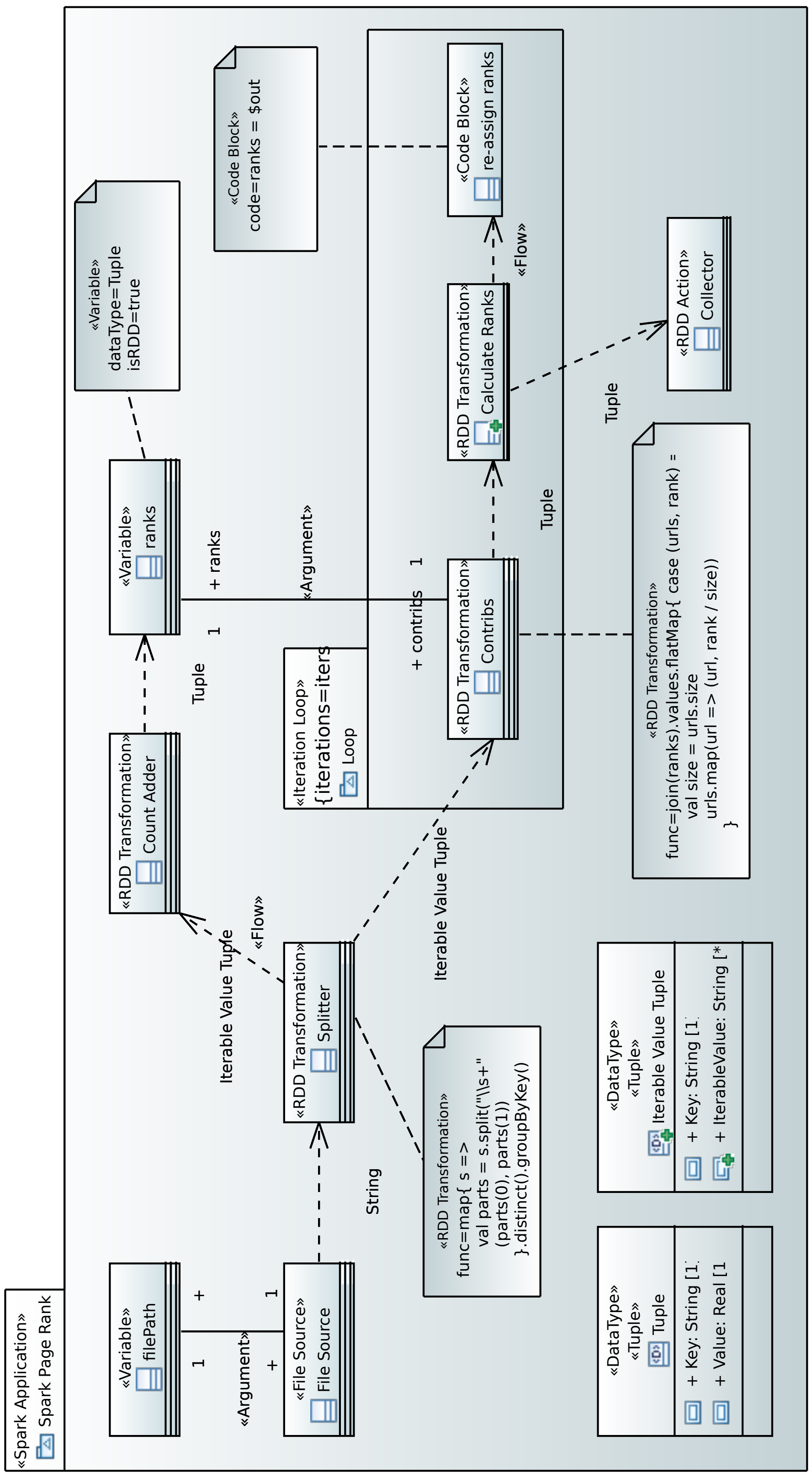

Figure 7.2: Page Rank diagram

First, we can see two tuples defined. The first defines a typical key-value pair already seen in the Word Count example. The second one differs in the multiplicity of the value element. This tuple is produced by the `groupByKey` operation. This tuple data type then represents `RDD[String, Iterable[String]]`.

Next, we can see that two nodes have multiple outputs. To determine the exact order of the execution, the `«Flow»` stereotype has been used for the higher priority path. As I already mentioned, Information Flows without the stereotype are evaluated last in an undefined order. In this diagram, the generator first generates the path to the **ranks** node with the `«Variable»` stereotype. This variable holds the output of the previous node. As can be seen in the diagram, the variable **ranks** data type is `RDD[(String, Double)]`.

The computation node **Contribs** uses the previously defined variable in its transformation. We can see the `«Argument»` stereotype being used to inject the variable into the transformation. In a typical Spark application program, the transformation's anonymous function could reach out of its scope and use all variables defined in the app's main method. Since all nodes are generated as separate classes, all necessary variables must be provided to the method scope. This approach introduces more modeling work but increases the transparency of the used transformation and might also prevent bugs by using variables out of scope. An example of Linear Regression implementation with the transformation using variable out of the anonymous function scope can be seen in an official Spark example on GitHub[1].

Loop

Now let us discuss the loop. Listing 7.3 shows the full generated computation of the Page Rank diagram. The first thing to point out are the variable definitions for the nodes inside the loop. If the variables were not defined before hand, the last statement in the listing would fail, because the `t_calculateRanks` would only be defined in the scope of the loop. The diagram shows that the path enters do loop when the **Splitter** node sends data to the **Contribs** node. Because of this transition, the appropriate loop header was generated. When the generator enters the **Collector** node, it closes the loop because the node is not inside the model element.

The main reason to use a loop inside a Spark application is to modify inputs for a transformation. In this example, we need to modify the **ranks** variable in each iteration. If we tried to model the diagram differently, we might come across a problem. If the **Calculate Ranks** tried to overwrite the data inside the variable node, it would exit the model element, and the variable re-assignment would close the loop. Furthermore, if the meta-model allowed to define multiple classes with the same name, it would disobey the UML standard. That is why the variable re-assignment is modeled as the `«Code Block»` stereotype. This situation shows that the meta-model might be improved, but also that it is flexible enough to allow a specific program requirements to be modeled.

This test case aimed to present multiple features of the meta-model. The loops, the function arguments, and the execution priority were proven to work correctly and offer a relatively intuitive modeling experience. This example also highlights the advantages of this solution compared to already existing ones. Improving the ability to model more complex tasks while also keeping the graphical model readable is one of the essential parts

[1] https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/SparkLR.scala

of the model-driven approach. The last example test case will go over the remaining features supported by the integrated meta-model.

```
var t_contribs: RDD[(String, Double)] = null
var t_calculateRanks: RDD[(String, Double)] = null


val s_fileSource = S_fileSource.source(filePath).asInstanceOf[RDD[String]];
val t_splitter = T_splitter.transform(s_fileSource).asInstanceOf[RDD[(String,
    Iterable[String])]];
val t_countAdder =
    T_countAdder.transform(t_splitter).asInstanceOf[RDD[(String, Double)]];
ranks = t_countAdder
for(loop <- 1 to iters.toInt) {
t_contribs = T_contribs.transform(t_splitter,
    ranks).asInstanceOf[RDD[(String, Double)]];
t_calculateRanks =
    T_calculateRanks.transform(t_contribs).asInstanceOf[RDD[(String,
    Double)]];
ranks = t_calculateRanks
}
val a_collector = A_collector.action(t_calculateRanks);
```

Listing 7.2: The Page Rank computation

7.3 Dataset processing

The final test case highlights the remaining features and provides a deeper look at how multiple data flows can be modeled in a single diagram. The modeled Spark task is relatively simple. We load data from three different CSV files. The first file contains a list of an abbreviation and the full name of American states. The two remaining files, which data we want to modify, are lists of American cities and Sports teams, both containing a state name in their records. In this task, we want to change the state's name to its abbreviation while highlighting the polymorphism of a single processing node. Afterward, to show the possibility of working with specific flows, we filter out all but NBA teams from the team flow. The model also uses a single node to save the modified data. This is accomplished using code block nodes that modify a variable.

The modeled diagram is shown in figure 7.3. First, let us mention the *imported* package outside the Spark Application element. Defining data types outside the model element allows us to utilize it in the diagram, while the generator does not generate the corresponding Scala case class. However, the generator still has access to the name of the data type to generate typecasting properly. The imported class can be found in the submitted source files of this test case in the **imported/** folder.

The computation starts with the highest priority source node - the **AbbrSource** node, loading data and storing the resulting RDD in the **abbrs** variable. The tagged values of the variable define its type as `RDD[Abbr]`. This variable is then injected into the **ReplaceStateWithAbbr** transformation. As previously stated, all processing node outputs are converted back to RDDs. Therefore, the function definition of the Dataset transformation must explicitly cast the **abbr** variable to Dataset using the `toDS()` method. We can also see the usage of the alias *this* inside the function that refers to the input RDD(or Dataset after implicit cast).

```
  def transform[T: TypeTag](rdd: RDD[T], abbrs: RDD[Abbr]) = {
    import spark.implicits._
    typeOf[T] match {
       case type1 if type1 =:= typeOf[City] =>
          rdd.asInstanceOf[RDD[City]]
             .toDS().as("this").joinWith(abbrs.toDS().as("abbr"), $"this.State"
                 === $"abbr.State", "inner")
                       .map(x => x._1.copy(State =
                           x._2.Abbreviation)).as[City].rdd
       case type2 if type2 =:= typeOf[Team] =>
          rdd.asInstanceOf[RDD[Team]]
             .toDS().as("this").joinWith(abbrs.toDS().as("abbr"), $"this.State"
                 === $"abbr.State", "inner")
                       .map(x => x._1.copy(State =
                           x._2.Abbreviation)).as[Team].rdd
    }
 }
```

Listing 7.3: Using reflection to determine the RDD type parameter

Next, the **CitySource** node produces the input RDD and creates a flow tagged with *city* tag. The transformation that replaces the state name with its abbreviation has three output Information Flow relationships. Since the path is using the *city* tag, one of them is ignored. Traversing the graph using the higher priority flow, the next visited node is the code block node. This node is used to change the contents of **outputPath** variable, which is used in the next generated node to execute the defined action to store the data inside a file.

Finally, the team flow is generated similarly to the city flow. After the abbreviation replacement the traversal continues with the only edge being tagged with the *team* tag. We filter teams only to include the NBA teams to show that the flows are separated again. Now the code block modifies the **outputPath** variable so the same action node can be used to store the result.

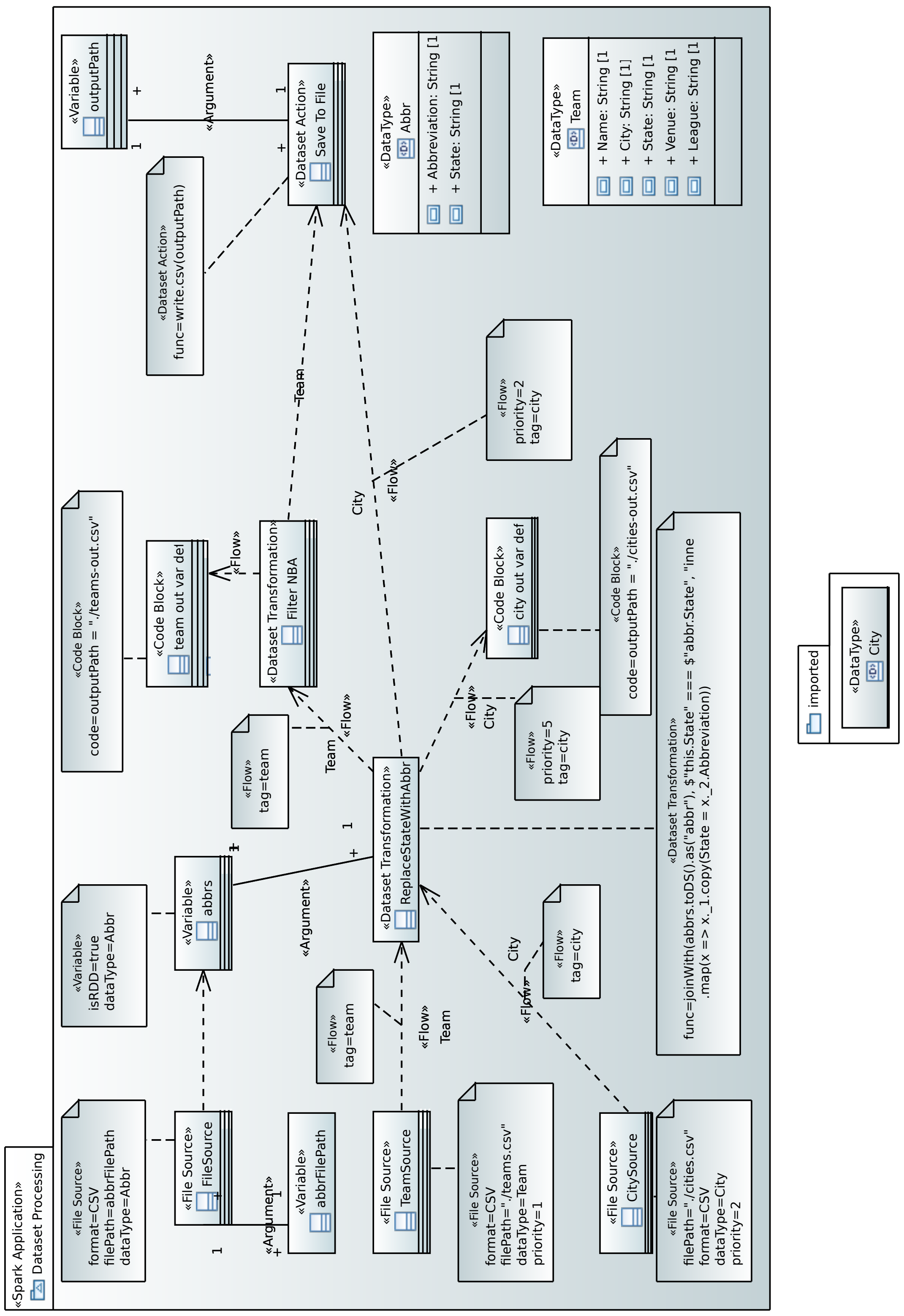

Figure 7.3: Dataset processing model diagram

Chapter 8

Conclusion

This thesis aimed to design a meta-model for Big Data processing tasks in the Apache Spark framework and develop a tool to generate the target source code of the modeled task. First, I had to get familiar with Spark to accomplish this goal. The essential details to understand the framework are described in chapter 2.

Afterward, the concept I had to get familiar with was Model-Driven Development (MDD). The advantages, disadvantages, and details are discussed in chapter 3.

After I researched the necessary concepts to understand the problem, I took a look at already existing tools that try to solve a similar problem. An overview of model-driven tools to solve Big Data processing is in chapter 4.

The integral part of this thesis is chapter 5 that goes over the designed meta-model and the reasoning behind specific choices. In addition, it describes the UML Profile diagram and its stereotypes in detail. Furthermore, the choices behind the generated code structure are also discussed.

The details and highlights of the generator implementation are in chapter 6. This chapter brings a further insight into how the implemented Eclipse Acceleo code generator works and how the specific snippets of source code are generated.

In the last chapter 7, a few test cases are presented to make the reader more familiar with the meta-model and to show the specific features in use. The test cases provide a mix between a simple processing pipeline, complex algorithm, and the processing of data with different structures.

Finally, I published the source code, meta-model and model files as open source. The project can be found on GitHub[1]. In this next section, I will evaluate the solution and propose a possible future work to improve the modeling ecosystem.

8.1 Evaluation and future work

I would like to evaluate the designed meta-model, code generator, and the overall approach to Model-Driven Development of Big Data processing tasks. The designed meta-model improved upon the disadvantages of several previous projects. The possibility of modeling a more complex, more readable diagram in a standardized (UML) environment definitely improves the overall MDD approach. The meta-model introduces features that abstract the code behind a relatively comprehensive graphical diagram.

[1] https://github.com/MarekSalgovic/spark-m2t-generator

The first issue with the modeling approach of Spark tasks is that using this meta-model might actually take more time than provisioning the solution with code. In simple processing applications, the generated code does not have enough volume to justify using a diagram instead of code. Furthermore, code completion and general IntelliSense features of code editors improve productivity more than the diagram's abstraction. The value of using a diagram comes from introducing a uniform or a standard way to design applications while generating reusable code components. Furthermore, graphical diagrams might also convey more information about the application logic.

The second issue comes from the robust typing system of the meta-model. While writing Spark code in Scala, the developer can rely on Scala's type inference feature. In addition, code editors would still highlight possible semantic errors. The implemented generator must explicitly generate the type notation to produce reusable, object-oriented code. The fact that the designer must always select appropriate data types in specific model parts also reduces productivity.

Several solutions to these issues come to mind while preserving the diagram's abstract, reusable and straightforward way of expression. As mentioned multiple times in this thesis, a reverse text to model generation tool could heavily improve the productivity of the workflow. The generated code was designed with this specific idea in mind. Also, OCL constraints could be added to the meta-model to introduce semantic analysis similar to code editors. The data type issue could be solved if the meta-model was designed with a specific target programming language in mind. Spark framework also supports dynamically-typed Python. However, altering the meta-model to exclude the type system would tie it to a specific technology. This approach would not work if the meta-model was to be used with a Java code generator. Java is statically-typed and does not support similar type inference as Scala does.

In conclusion, I consider this project a step in the right direction for a model-driven approach to data processing applications. However, there is still room for several refinements and supporting tools since Model-Driven Development is still a developing methodology.

Bibliography

[1] *Spark MLlib* [online]. [cit. 2021-12-12]. Available at: https://spark.apache.org/mllib/.

[2] Balcer, M. and Jacobson, I. Executable UML: A Foundation for Model-Driven Architectures. january 2002.

[3] Béder, M. *Zpracování síťové komunikace v prostředí Apache Spark*. Brno, CZ, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.vut.cz/studenti/zav-prace/detail/114695.

[4] Bútora, M. *Modelem řízený vývoj Spark úloh*. Brno, CZ, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://dspace.vutbr.cz/xmlui/handle/11012/180380?locale-attribute=en.

[5] Damji, J. S., Wenig, B., Das, T. and Lee, D. *Learning Spark, 2nd Edition*. 2nd ed. O'Reilly Media, Inc, 2020. ISBN 9781492050049.

[6] Laskowski, J. *The Internals of Apache Spark 3.2.0* [online]. 2021 [cit. 2021-12-12]. Available at: https://books.japila.pl/apache-spark-internals/.

[7] Xin, R. *Apache Spark Officially Sets a New Record in Large-Scale Sorting* [online]. 2014 [cit. 2021-12-09]. Available at: https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html.

[8] Hamilton, K. and Miles, R. *Learning UML 2.0*. O'Reilly Media, Inc, 2006. ISBN 978-0-59-600982-3.

[9] Mansour, M., Wolf, M. and Schwan, K. StreamGen: a workload generation tool for distributed information flow applications. In: *International Conference on Parallel Processing, 2004. ICPP 2004*. 2004, p. 55–62 vol.1. DOI: 10.1109/ICPP.2004.1327904.

[10] *MDA® - THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD* [online]. [cit. 2021-12-16]. Available at: https://www.omg.org/mda/.

[11] *MOF Model to Text Transformation Language, v1.0* [online]. 2008 [cit. 2021-12-16]. Available at: https://www.omg.org/spec/MOFM2T/1.0/PDF.

[12] Pagáč, J. *UML profil pro modelování komponentových systémů*. Brno, CZ, 2011. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://dspace.vutbr.cz/handle/11012/54141.

[13] Gérard, S. *PAPYRUS USER GUIDE SERIES: About UML profiling, version 1.0.0* [online]. 2011 [cit. 2022-05-02]. Available at: https://www.eclipse.org/papyrus/resources/PapyrusUserGuideSeries_AboutUMLProfile_v1.0.0_d20120606.pdf.

[14] Rajbhoj, A., Kulkarni, V. and Bellarykar, N. Early Experience with Model-Driven Development of MapReduce Based Big Data Application. In: *2014 21st Asia-Pacific Software Engineering Conference*. 2014, vol. 1, p. 94–97. DOI: 10.1109/APSEC.2014.23.

[15] Selic, B. The pragmatics of model-driven development. *IEEE Software*. 2003, vol. 20, no. 5, p. 19–25. DOI: 10.1109/MS.2003.1231146.

[16] *Cluster Mode Overview* [online]. [cit. 2021-12-16]. Available at: https://spark.apache.org/docs/latest/cluster-overview.html.

[17] Pedamkar, P. *Spark Dataset* [online]. [cit. 2021-12-12]. Available at: https://www.educba.com/spark-dataset/.

[18] Damji, J. *A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets* [online]. 2016 [cit. 2021-12-12]. Available at: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html.

[19] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M. et al. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. oct 2016, vol. 59, no. 11, p. 56–65. DOI: 10.1145/2934664. ISSN 0001-0782. Available at: https://doi.org/10.1145/2934664.

[20] *UML Practical Guide - All you need to know about UML modeling* [online]. [cit. 2021-12-12]. Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-practical-guide/.

[21] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J. et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, April 2012, p. 15–28. ISBN 978-931971-92-8. Available at: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

Appendix A

Memory media contents

The submitted memory media contains source files of the thesis's text, `.pdf` file, Eclipse Papyrus projects and diagrams of the examples task and the meta-model, source code of the generator, compiled Java classes of the generator, `.jar` files of the used libraries, and finally the generated Scala code for the example tasks.

The structure is as follows:

- **bin/** - the compiled generator
- **doc/** - thesis source files
- **libs/** - `.jar` files of used libraries
- **out/** - generated Scala code, project setup and input files for example tasks and **Makefile** to run the generated tasks
- **src/** - source code of the generator
- **spark-metamodel/** - Eclipse Papyrus project of the meta-model Profile diagram
- **sparkWordCount** - Eclipse Papyrus project of the first example task
- **sparkPageRank/** - Eclipse Papyrus project of the second example task
- **dataset-processing/** - Eclipse Papyrus project of the third example task
- **Makefile** - contains the commands to run the generator and generate output

Appendix B

Software versions

The list of used software and their version to compile and run the project:

- **Apache Spark** - 3.2.1
- **sbt** - 1.6.2
- **Eclipse Acceleo** - 3.7.11.202102190929
- **Scala** - 2.12
- **Java** - openjdk version „17.0.2“ 2022-01-18 LTS