Designing DAWN: a Data Analytics Workflow Notation

Master's Thesis

R.M. de Lange

For Albert

Designing DAWN: a Data Analytics Workflow Notation

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE
TRACK INFORMATION ARCHITECTURE

by

R.M. de Lange
born in Amsterdam, the Netherlands

Web Information Systems
Department of Software Technology
Faculty EEMCS
Delft University of Technology
Mekelweg 4
Delft, the Netherlands
wis.ewi.tudelft.nl

ORTEC Consulting Group
Houtsingel 5
Zoetermeer, the Netherlands
www.ortec-consulting.com

Designing DAWN: a Data Analytics Workflow Notation

Author: R.M. de Lange
Student id: 1534068
Email: r.m.delange@student.tudelft.nl, mick@mickdelange.nl

Abstract

Big Data is a popular research and business topic. Due to the potential value that lies in Big Data, much effort is put in attempts to improve Big Data analysis methods. ORTEC is a company that provides data analysis, optimisation, and forecasting solutions. In an attempt to make Big Data analysis easier to use for its customers, ORTEC is developing the Big Data Portal. The intention is to create an easy-to-use cloud service that offers an all-in Big Data analysis solution.

The Big Data Portal allows users to design workflows for their data analysis work. These workflows need to be designed in the portal through means of a notation that is both easy to use and can be extended with the latest in Big Data analysis tools. Furthermore, the portal should process the analysis work in an efficient manner, to minimise costs.

In this design research DAWN, a Data Analytics Workflow Notation, was developed. This design considers the goals set for the Big Data Portal, as well as theoretical base for the workflow notation. The formal definition is derived from nested relational calculus as a theoretical base.

In this work, DAWN is shown to be easily visualisable for graphically editing the workflow. Furthermore, DAWN is flexible as it enables the addition of new user defined functions. The compiler for DAWN, presented in this work, shows portability to other workflow execution systems. The nested relational calculus base is shown to enable optimisations in the workflows for cost reduction in the execution.

Thesis Committee:

Chair:	Prof. dr. ir. G.J.P.M. Houben, Faculty EEMCS, TUDelft
University supervisor:	Dr. ir. A.J.H. Hidders, Faculty EEMCS, TUDelft
Company supervisor:	Dr. S.F. van Dijk, ORTEC Consulting Group
Committee Member:	Prof. dr. E. Visser, Faculty EEMCS, TUDelft

Preface

This thesis represents the final work of my master Computer Science and therefore the final step in my student life. During this period I was able to rely on input and support by many different people, whom I wish to thank in this preface.

First of all I would like to thank my colleagues at ORTEC, and fellow graduate interns. Specifically, I would like to thank Steven van Dijk for his dedicated guidance and detailed feedback on both my written work as well as my code. Furthermore I want to thank Arjan Peters, Chuck Ng, Dragos Tihauan, Vincent Berkien and Sander Knape for their input on my work, the role they played in the Big Data Portal and the discussions we had on the difficulties of graduation work.

From the TU Delft, I would like to extend my thanks to Jan Hidders, who has been a great supervisor. Both in helping me figure out complex issues and telling me to take time of when needed, Jan has been great support. I would also like to thank my fellow alpha group members for the useful discussions and the feedback provided on my presentation. Furthermore, I would like to thank my other committee members, Geert-Jan Houben and Eelco Visser for taking time to read my thesis and provide feedback.

Last but not least I would like to thank my family and friends, especially Tjitske, for their dedication, support and feedback on early versions of my work.

It has been an interesting project, which has brought me new insights and experiences, both from a scientific as well as a personal perspective. The resulting work is something I feel pride for and which I hope you, the reader, will enjoy as much reading as I did writing it.

R.M. de Lange
Delft, the Netherlands
November 22, 2015

Contents

List of Figures

Chapter 1

Introduction

Big Data has become an increasingly popular topic in both business and science. McKinsey & Company has said that Big Data is *"The next frontier for innovation, competition, and productivity."* [15]. This claim is based on the fact that more and more data is being collected by a large variety of applications. This data is steadily proving its value through analysis efforts. There is both business and scientific value in analysing Big Data, as this data often provides insight in human behaviour through sensory data. A large number of fields face the challenges of effectively analysing large amounts of data.

ORTEC, a company that provides data analysis, optimisation, and forecasting solutions, aims to provide businesses with an all-in solution to Big Data analysis challenges. By harnessing the computational power of cloud services into a user friendly portal, ORTEC wants to help business analysts gain value from Big Data analysis. This *Big Data Portal* (BDP) enables customers to gain insights in their data. Business analysts at ORTEC will help the customers design their own data analysis processes. Users will be provided with easy to use tools that enable them to employ the latest in Big Data analysis and cloud technology.

This thesis presents the design of DAWN, a Data Analytics Workflow Notation that allows users to design their data analysis processes in a simple manner. The design builds on the research on *Nested Relational Calculus* (NRC) and nested relational query languages to support the workflow language with a formal basis. Given this formal basis, the design takes the user's perspective into account and aims to provide a user friendly portal for data analysts.

Both a formal definition of the workflow notation and a notation in *JavaScript Object Notation* [1] (JSON) are presented in this work. Furthermore, a compiler is implemented that allows the workflow notation to be executed in *Windows Workflow Foundation* [2] (WF). This is the execution engine chosen by ORTEC to run the workflows. Evaluation of this design shows that the design fits the goals set for the BDP and the application of the workflow notation in this portal.

[1] `http://json.org/`

[2] `https://msdn.microsoft.com/en-us/library/dd489441.aspx`

1.1 Motivation

ORTEC wants to present an easy to use interface to customers and data analysts, including a graphical workflow designer. This requires the design of a workflow notation that can both be used in a web-based workflow designer and apply the required concepts of Big Data analysis in a cloud environment. An important challenge here is to provide a notation that is both useful and flexible. Several workflow systems exist that describe workflows and allow to execute them. Moreover, there is an abundance of workflow systems [3, 24].

Designing a new workflow language in a crowded field would therefore seem illogical. However, the usability and flexibility targets combined with the application of an NRC-based query language pose new challenges for a workflow language. Applying the NRC formalism will result in an algebraic approach to workflow design. This allows for interesting optimisations as well as formal verification of the feasibility of a workflow. Furthermore, applying this formalism in an environment that aims at non-technical users requires simplifications to make it an intuitive system to use.

The challenges in this design research are therefore interesting enough to develop a new workflow notation. However, aspects of the designed workflow notation are inspired on existing solutions. Furthermore, the design aims to provide a notation that can be applied in different environments, which requires a good fit with existing workflow solutions as well.

1.2 Approach

The approach of this design research is based on the guidelines for design science described by Hevner et al. [10]. This section will describe this approach and relate aspects of the approach to the guidelines discussed by Hevner et al. The main design process is described in Hevner's Guideline 6, which will be elaborated on in Subsection 1.2.1. Furthermore, evaluation of the results of the design research as discussed in Guideline 3 will be clarified in Subsection 1.2.2.

The other guidelines will only be summed up briefly here. The design should produce an artefact, show relevance, show contributions and clearly communicate the design [10, Guidelines 1,2,4&7]. The artefacts of this design are the actual results, which are presented in Chapter 4, Chapter 5, and Chapter 6. Relevance of the design research is shown in Section 1.1 and Chapter 2. The contributions of this work are discussed in Section 1.3 and Chapter 9. The goal of this thesis is to clearly communicate the design, its artefacts and contributions. Furthermore, the design applies rigorous methods and relies upon scientific results, as described in Guideline 5 [10].

1.2.1 Design Process

The design process, as described by Hevner et al., is an iterative process. This process is regarded a cycle between generating new design alternatives and testing these against the requirements and constraints. In this design research this method is applied, as it is analogous to the agile development method applied at ORTEC. This development method revolves around an iterative process, where each iteration provides useful results and can be improved on in the following iteration.

The design process is described as a search process in Guideline 6 [10]. This search process is a process where each step leads to both a better design as well as a better understanding of the problem itself. The search starts at defining small parts of the problem and expanding with each iteration towards solving the entire design problem.

For this thesis this approach proves to be very useful, as there are many challenging aspects to the design, which can be dealt with more easily when split up into sub-problems. The resulting design is compared to the goals and matched with results of other sub-problems. In this way a balance is found between scientific support for the design, a useful artefact and a fit in the ORTEC environment.

1.2.2 Evaluation

A good design requires rigorous evaluation. Each iteration in the design process requires to evaluate the results and compare these to the design goals. The final evaluation of the design presented here is given in Chapter 7. For this design research a descriptive evaluation method was chosen. Hevner et al. discuss a number of evaluation methods in their work and emphasise the importance of selecting the correct method [10].

The descriptive method is used here as the designed artefacts are of a conceptual nature. The final goal to allow users to visually edit the workflow notation does not lie within the scope of this work. Therefore, user testing is not the best approach to evaluate the results. An informed argument can be made to show the utility of the design artefacts, by relating to other research work and applying formal analysis to the design artefacts. Chapter 7 applies this method to show that the design fits the goals set in Chapter 3.

1.3 Contributions

The design research in this thesis results in a number of artefacts: the workflow model, compiler, notation, and its validation schema. These artefacts are based on existing work in the field of workflow systems, nested relational query languages, and data analysis. Additionally, this work presents several contributions to these research fields. These contributions are presented throughout this thesis and are listed here for completeness.

1.3.1 NRC-based Algebraic Workflow Model

An important contribution presented in this thesis is the model for data analysis workflows, as presented in Chapter 4. The model is based on both the application domain and NRC. This model defines the workflow definition as ORTEC wishes to present this to data analysts. It presents this in a user friendly manner, whilst showing that it can still be mapped to NRC. The simplifications needed for this presentation are a novel contribution to the NRC research field.

1.3.2 Application of NRC-like Rewrite Rules to The Workflow Notation

As shown in Subsection 7.3.1, rewrite rules known for NRC can be applied to DAWN. This application enables a number of interesting optimisations to be applied to DAWN workflows. Although more research is needed, this is an interesting contribution.

1.3.3 Flexible Workflow Notation

The workflow model designed is presented as a JSON notation as well. This notation is shown to be applicable in a web-based environment and provides a definition that can be visually edited. An important aspect for this notation is that it represents flexibility. It allows for different execution engines to run the workflow as well as a number of different visual designer interfaces. Additionally, the notation allows to be extended with new components. This flexibility enables users to add new features without the need to replace the workflow notation.

1.3.4 Named Typing System for Semi-structured Data

The workflow notation presents a named type system in Subsection 4.2.2 to allow validation of the data flow. This type system is specifically designed to describe complex, nested, semi-structured data. This type system enables a more user friendly workflow design process, by verifying validity of the designed workflows. It provides the users more insight into the data being processed by the workflow.

1.4 Structure

This thesis will present the design research in the following structure. Chapter 2 presents related work and important concepts. The purpose is to give an impression of the field and position this work in the correct context. Chapter 3 discusses the design goals to which the design must adhere. These goals are used throughout the design process and are verified in the evaluation. Chapter 4 presents the graphical workflow notation. A formal definition is given for the DAWN workflow syntax and semantics, and a comparison is made to sNRC. Chapter 5 presents a JSON serialisation of the workflow notation, which is used for storing and editing workflows in the BDP. Chapter 6 describes the compiler that was built, along with the challenges the workflow notation and execution engine presented for this compiler. Chapter 7 evaluates the results of the design by verifying whether the goals have been achieved. Chapter 8 discusses future improvements to the designed notation and compiler. This chapter shows which follow-up research poses interesting challenges, based on this thesis. Chapter 9 concludes the thesis and reflects on the work presented.

Chapter 2

Background

This chapter gives an overview of concepts and technologies which are relevant to this thesis. The purpose of this chapter is twofold: on one hand it gives an introduction of the topics discussed in the following chapters, on the other hand it illustrates the position of this work in relevant research fields as well as within the business of ORTEC. Furthermore, the concepts explained in this chapter provide terminology which will be used throughout this work.

2.1 ORTEC

This thesis work was performed at a company called the ORTEC Consulting Group. The Consulting Group is a business unit within the whole ORTEC organisation. To give an insight in this business and its implications for the design, this section will discuss some key facts about ORTEC. ORTEC was founded in 1981 and has developed to be one of the largest providers of optimisation and planning solutions. ORTEC products perform optimisations or calculations to help solve business challenges. Example solutions developed by ORTEC are: planning, scheduling, routing and forecasting solutions.

From the perspective of this work there are several aspects of development at ORTEC worth considering. ORTEC has a Microsoft-preferred development environment, which translates to standards within the company. .NET Framework is the standard development framework that is applied in the majority of the projects at ORTEC. Microsoft Azure is the preferred cloud platform, with several Microsoft solutions readily available as services. Web development is done in a model-view-controller approach, using the .NET Framework web standards. Furthermore, popular frameworks and libraries are used for web projects, among which *AngularJS* [1] and *Bootstrap* [2]. A combination of these techniques and technologies is defined as the ORTEC web stack, which is the standard approach to developing web applications at ORTEC.

[1] `https://angularjs.org/`
[2] `http://getbootstrap.com/`

2.2 Big Data Portal

ORTEC has a research and development project called the *Big Data Portal* (BDP), which the design presented in this thesis is a part of. The goal of this project is to provide a comprehensive, cloud-based, Big Data analytics solution. Business analysts, or data analysts, are the target users of the BDP. These analysts can be either employees at ORTEC that help clients analyse their data, or analysts employed at one of ORTEC's clients. The following two key characteristics define the ideas behind the BDP.

First, ORTEC aims for the BDP to be a comprehensive, all-in-one solution for Big Data analysis in the cloud. A variety of data analysis techniques which are available to data analysts should be presented to the user through the portal. Moreover, the BDP should provide the user with the possibility to create workflows that operate on a large number of different data types. This means that it should allow for different data types and data analysis solutions to be integrated in the portal.

Second, ORTEC wants to provide these options to the user in a simple and easy to understand interface. The interface should allow the user to design their workflow with minimal effort, whilst at the same time providing the user with information on the workflow execution and the data analysis results. In essence, the portal should be an easy to use tool for data analysts. The BDP will be presented to the user through a web interface.

Summarising, the BDP will be a easy to use portal, provided to data analysts to harness the potential of Big Data analytics in the cloud, while requiring minimal knowledge of cloud technology.

2.3 Big Data Analysis

Big Data is an increasingly popular topic in both science and business at this moment. Many applications are being developed that utilise Big Data to provide scientific insights or business value. The BDP is an example of such an application. However, the definition of Big Data is not straightforward and multiple interpretations exist. This section will briefly introduce the definition of Big Data, as it will be applied in this thesis. Furthermore, this section describes the challenges posed by Big Data, that are relevant to this design research.

2.3.1 Big Data

Big Data is a term to describe datasets of such large size, that they can not be managed by classical database software. Moreover, it considers datasets which are often too large to be read into memory in regular machines. Big Data is generally characterised in dimensions of a number of "Vs" [12, 2], this work will consider the following five Vs.

Volume of the datasets. Big Data comes in large sizes, often datasets have sizes which exceed terabytes. The larger the dataset, the more time, computational power and storage space is required to process and analyse the data. Therefore the complexity of the analysis increases as volume increases.

Velocity at which the data is generated. This is the speed at which new data is added to a dataset. In some applications data is gathered real-time or streaming, for instance in case it is sensory data. Higher speed adds complexity, as the data processing needs to keep up with the speed at which the data is collected.

Variety of the data source and structure. Big Data can come from a large variety of data sources and in many different forms. The data can be structured, semi-structured, or unstructured. Less structure in the data indicates an increase in complexity for analysis.

Veracity of the data. Veracity indicates trustworthiness of the data and the reliability of the data source. As the source is less reliable, it is harder to gain valuable insights from data analysis.

Value that the data represents. Value is given by Big Data in the form of scientific or business value that can be found in the data. Creating value is considered a goal of the Big Data analysis process.

There are researchers who consider additional dimensions for Big Data. However, for this thesis only the above mentioned dimensions will be considered, as these are most relevant to this work.

2.3.2 Analysis

The Big Data dimensions indicate that analysing Big Data poses complex challenges for data analysts. Successful data analysis can, however, provide the analysts with interesting business value. The data analytics system designed in this thesis should consider the challenges in the implementation of data transformation steps and optimisations of the workflow. However, for this work the focus is on designing the workflow specification on a higher level, rather than designing the lower level data processing steps.

The main technical challenge in analysing Big Data lies in its size. Classic data storage and analysis techniques are often not capable or not powerful enough to process Big Data in a timely manner. Therefore Big Data analysis requires a different approach than regular data analysis. Several solutions have been developed recently. These often perform computations which read and write data directly from disk, as reading all data into memory is not feasible at this size. One of the most well known Big Data analysis solutions is the MapReduce programming model [6], developed by Google.

This programming model splits up the data in partitions. These partitions are distributed over workers which then perform the user-defined Map function on the input data and produce intermediate key/value pairs. The intermediate data is grouped by key and passed to the user-defined Reduce function, again distributed over a number of workers. The Reduce function merges the intermediate results, reducing the size of the dataset, and produces the final result.

By splitting the dataset into small partitions and spreading the work over a number of workers, each worker uses only a small amount of memory, whilst still producing the correct result. This programming model is applied by many Big Data analysis systems,

most of which are based on the popular open source MapReduce implementation: Hadoop [25].

ORTEC will not attempt to create such a system itself. Rather, the BDP will provide an interface to harness the power of such Big Data analysis tools in the form of cloud services. For example, Microsoft Azure provides HDInsight[3], a Hadoop-based cloud service.

2.4 Workflow Systems

This thesis discusses the design of a workflow notation to enable Big Data analysis. Data analysis is a comprehensive field and as Barker and Hemert [3] discuss, workflow systems are abundant. To emphasise the large number of workflow languages available, van der Aalst and ter Hofstede introduced *Yet Another Workflow Language* (YAWL) [24]. The work on YAWL mostly represents an analysis of different workflow patterns. This section will give an introduction to a number of workflow systems, while distinguishing between two main classes of workflow structures.

A workflow is defined as a sequence of tasks or operations needed to manage a computational activity [22]. This means that a workflow consists of computational parts, which need to be executed in a certain order to achieve the desired outcome or outcomes. These computational parts are referred to with a number of different terms, such as tasks, activities, operations, or components. To define the order of these components, workflows specify either a sequence or relations between components.

2.4.1 Classes of Workflow Structures

In the *Workflows for e-Science* book [22, Chapter 11], Shields distinguishes between two classes of workflow structures: control-driven workflows and data-driven workflows. The author also discusses the hybrid option, in which a system incorporates elements of both classes. In the YAWL publication, the authors describe these structures as perspectives [24]. Multiple perspectives are discussed, but the control flow perspective and data flow perspective are noted as the main two perspectives. Moreover, van der Aalst and ter Hofstede mention that the data flow perspective is in fact an approach that rests on the control flow perspective. Here we will discuss the differences between the two classes and give some examples of each class.

Control-driven workflows

Control-driven workflows, or control flows, specify the transfer of control between components through relations. This class of workflows typically consists of a sequence of activities, where control is passed from each activity to its successor. The activities read their input from and write their output to a common data store, the activity that is active has control over the data store. Optional movement of data is handled by a specific activity tasked only with data transferral. Petri nets [16] are example tools which can model both control and data flows [22].

[3] `http://azure.microsoft.com/en-us/services/hdinsight/`

Data-driven workflows

Data-driven workflows, or data flows, specify the data dependencies between components through relations. This approach allows to disregard any sequence; each component can be run concurrently, only halting to await its data dependency. The components take data as input and produce data as output, which is then provided to any dependent components. Data flow representations are regarded to be simple, as they only specify the components and the dependencies between them. The user does not have to consider the sequence of the components in this class of workflows. There are a large number of data flow languages.

Kepler [1] and Triana [21] are some well-known scientific data flow systems. Kepler is a tool that supports scientists to design and execute scientific workflows, as well as streamlines the execution process. Triana provides scientists with a comprehensive workflow environment that supports multiple services, mainly focussed on peer-to-peer and grid technologies. Another well-known example of a data-driven workflow language is Pig Latin [18], with its workflow system Pig [9]. Pig is a data flow system that aims to provide a simplification over Map-Reduce workflows, by implementing elements of an SQL-like query language.

2.5 Nested Relational Calculus

For the workflow notation designed in this thesis, the *Nested Relational Calculus* (NRC) formalism is used as a basis. NRC is a calculus allowing to reason and iterate over collections of values. To give some insight in this formalism, its origin will be briefly discussed first.

Codd introduced a model for representation of relational data in large databases [5]. This model allows to give a generic representation of relational data, independent of the internal data representation in a machine. Codd's work was extended by Roth, Korth and Silberschatz, who considered relaxing the first normal form constraint to allow for nested relations [19]. Roth et al. introduced an algebra and a calculus for nested relational data by extending the classical flat relational algebra. Formalised definitions of this algebra and calculus were presented in Wong's PhD. thesis [26].

NRC provides a query language that can reason over collections and complex objects. Hidders et al. have shown that NRC can be used as a base to develop a workflow language [11]. This workflow language can also be edited in a graphic designer, as was shown in later work [20]. Another interesting aspect of NRC as a workflow language is that it potentially provides useful optimisations for Big Data analysis. Dias et al. [7] argue that algebraic, data-centric workflows allow for interesting optimisations both from a scientific and a business perspective. Moreover, Fegaras and Maier [8] discuss the use of an effective calculus to optimise object queries.

As NRC allows to reason over complex objects, this work will consider a specific class of data structures, semi-structured data. This is defined in a dialect of NRC called *NRC for semi-structured data* (sNRC), which will be discussed in the following section.

2.6 NRC for semi-structured data

sNRC is an NRC dialect developed by Hidders in work which is at this time not yet published. Preliminary material of this work can be found in Appendix A, where a complete discussion of sNRC, an accompanying algebra, and a graphical workflow notation are presented by Hidders. This thesis builds upon this work, specifically on the application of sNRC as a graphical workflow notation. This section gives a brief summary of the important concepts of sNRC.

As discussed in Section 2.3, one of the challenging aspects of Big Data analysis is the variety of data and the structure of this data. Semi-structured data represents a commonly encountered challenge in Big Data analysis. Therefore sNRC is chosen as a basis for the workflow notation designed in this thesis.

2.6.1 Semi-structured Data

Semi-structured data is data which, in contrary to relational databases, lacks a strict data model. Often this form of structured data is self-describing, examples of such data structures are *Extensible Markup Language* [4] (XML) and *JavaScript Object Notation* [5] (JSON). An important concept here is that elements in semi-structured data are identifiable through descriptive tags, although there is no guarantee that a specific tag is present in the data structure. Data tables on the other hand, provide a number of columns and each entry in the table has that exact same structure.

2.6.2 Data Model

The sNRC language distinguishes between three different data constructs: bags, tuples and constants. Bags are unordered collections of items that allow for duplicate items to be present, similar to multisets. Tuples are ordered pairs of bags. Constants can be any basic value, such as strings, booleans, and integers. A singleton bag is defined as a bag with a single constant in it. A complete and formal description of the data model can be found in Appendix A.

2.6.3 Semantics

sNRC presumes every value to be a bag, which can contain constants and tuples. Because every value is considered a bag, tuples can only contain bags and constants are defined in the form of a singleton bag. However, a bag of bags is not a valid construct. This ensures definedness of the result of each sNRC expression, similar to the approach of XQuery. It also allows the query language to apply any function on any input. Operators which do not expect a bag as input, are mapped over all elements of a bag. Furthermore, any operator also produces a bag as result. The operator that is mapped over a bag produces a bag with results for each element in the original bag.

[4] `http://www.w3.org/XML/`
[5] `http://json.org/`

2.6.4 DAWN

Hidders introduces a graphical notation for n-ary sNRC functions called DAWN in Section A.7. An n-ary function is a function with n input parameters. Similarly, a function with a single input parameter is called a unary function. As sNRC functions can have more than one input parameter, these functions are called n-ary functions. A more detailed discussion of n-ary functions can be found in Section A.6. This graphical notation allows to represent the functions of sNRC in a workflow graph structure.

This graphical notation is an interesting tool to base the workflow design for the BDP on. The notation designed in this thesis will therefore be called DAWN. However, the notation designed in this thesis will make some alterations to the specification given by Hidders. These differences will be discussed in more detail in Section 4.3. For clarity, the work by Hidders will be referred to as sNRC, while the notation presented in this thesis will be referred to as DAWN.

Chapter 3

Design Goals

The main goal of the BDP project is to provide a user friendly portal for data analysts to help them design and use their data analysis workflows. Moreover, customers will need to interact with the portal and its workflows as well. It is important that the portal is easy to understand for customers with less experience in data science than the ORTEC analysts. Additionally, ORTEC wants to provide users of the BDP with the latest in Big Data analysis technologies, which requires the workflow system to be flexible enough to implement this. Furthermore, this design research aims to add scientific design rigour to the workflow system by implementing the sNRC formalism as described in Appendix A. This formalism potentially provides useful improvements such as optimisation of workflows.

This chapter defines the goals for the design research conducted in this thesis. The goals are derived from the above stated main goals and provide a clear set of requirements. These goals are used throughout the design process to evaluate design alternatives, as described in Section 1.2. Moreover, the evaluation of this work presented in Chapter 7 will verify whether the design artefacts adhere to the goals. For clarity, the goals are divided in three categories: user friendly, flexible, and optimisable.

3.1 User Friendly

As the main goal of the BDP is to provide a user friendly interface for designing workflows, user friendliness is an important goal. To design a user friendly workflow notation, the users and their preferences should be known. For this, both literature as well as discussions at ORTEC were taken as a guideline. The users of the workflow notation will, in the first place, be business analysts at ORTEC. In later stages business analysts outside of the company might use the BDP as well, although their requirements for the workflow notation are expected to be similar.

Business analysts are skilled data analysts that use several data analysis techniques to achieve business value. Techniques which are often applied at ORTEC are data querying languages such as SQL, and scripting languages such as R[1]. Visualisation of data is also an important tool to help provide value and information to a customer,

[1] http://www.r-project.org/

a commonly used tool at ORTEC is Spotfire[2]. Goals formulated here are aimed at designing a workflow notation that is easy to use for these data analysts.

Goal 1. *Design a notation that matches the data oriented view of the users.*

As the intended users of the BDP are data analysts, the approach to the workflow notation should match their perspective. Data analysts in general have a data-centric view on their analysis work, as they are accustomed to working with querying languages and manipulate data. Moreover, dataflow representations are often simple in nature, as they do not provide control constructs such as loops [22]. Therefore, the workflow notation should match the conceptual view of the data analysts and implement a data-driven approach.

Goal 2. *Provide users with an easy-to-use, graphical interface to design their workflows.*

A graphical interface for editing workflows is considered the most user friendly. Although the data analysts do have knowledge of scripting languages and querying data, they are not experienced programmers. Therefore, designing a graphical workflow editor is more suitable for these users than designing a programming language. Moreover, editing a dataflow notation in a visual manner is an intuitive approach to workflow editing [14, 21, 22] and helps give customers better insight in the data analysis process.

3.2 Flexible

To be able to provide users with the data analysis tools they need, and be able to upgrade to the latest technology, the designed workflow notation needs to be flexible. ORTEC wants to provide a comprehensive set of data analysis technologies through the BDP. However, customers might require specific data analysis tools that need to be added to the workflow system. This requires the notation to be flexible enough to support easy addition of new components. Furthermore, if new workflow technologies arise, the notation should be portable to a new system. This enables ORTEC to provide the latest in data analysis and workflow technology, whilst keeping the workflow notation the same. Therefore, all existing workflow specifications can be used in a new environment.

Goal 3. *Support the addition of User Defined Functions to the workflow system.*

Customers might require specific data analysis solutions. The workflow notation needs to take into account that new components can be added in the form of a *User Defined Function* (UDF), to support this customer need. These UDFs can implement new cloud services, specific scripting languages or other data analysis tools. Adding such extensions should be a simple task, such that ORTEC has to make minimal changes to the notation to implement these functions.

Goal 4. *Design a notation that is portable to different workflow environments.*

[2] `http://spotfire.tibco.com/`

ORTEC wants to keep improving the BDP, future versions might therefore require the use of new workflow technologies. A new technology might enable ORTEC to use a better execution engine, which gives better performance, or is more cost-effective. In any of these cases it should be possible for ORTEC to implement a new execution engine, without requiring the users to redesign all their workflows. The notation should therefore be generic enough to enable the use of different execution engines.

A similar improvement might be made on the user interface side. ORTEC wants to provide a modern interface to the user, which requires regular updates. The workflow notation is part of this interface. However, it should be possible to redesign the interface without the need for changes in the workflow notation.

3.3 Optimisable

To keep costs low and processing time as short as possible, the workflow execution should be as efficient as possible. Being able to improve the performance of the workflow execution engine is therefore an interesting challenge. As Olston et al. discuss in their study of automatic optimisations in Pig [17], there are several interesting optimisations which can be applied to dataflow systems.

Olston et al. distinguish between logical and physical optimisations. Logical optimisations are changes to the dataflow graph structure which produce a semantically equal, but more efficient workflow. Physical optimisations are improvements in the compiler that produce a more efficient execution of the workflow. The two goals described here address both types of optimisations.

Goal 5. *Apply logical optimisations in the form of rewrite rules for NRC-like languages.*

The sNRC formalism makes it possible to implement several interesting cost-based optimisations, similar to those discussed by Wong [26, Chapter 6]. Wong presents a number of rewrite rules for NRC. Wong shows that these rules can be applied to rewrite any query to a normalised form in a finite number of steps. He argues that reordering queries can reduce costs in a sense that the amount of data that is processed in expensive components can be reduced. For example, reordering a query to filter datasets early on in the workflow, reduces the amount of intermediate data and therefore limits unnecessary data processing.

By applying concepts from NRC-like languages, rewrite rules can also apply to the designed workflow notation. An optimiser can be implemented that applies the rewrite rules to achieve a cost reduction in the workflows. Users will be able to design their own workflows, without having to consider the performance of their specification. The optimiser will transform their workflow to a more optimal form.

Goal 6. *Apply physical optimisations in the compiler or execution engine.*

Physical optimisations involve the design of an execution plan for a workflow. This type of optimisation is therefore applied at the level of the compiler or the execution engine. These optimisations often consider the order in an execution plan or involve more in-depth knowledge of the components to be able to execute these in a specific way. An example of such an optimisation is parallelisation. Parallelising parts of the

execution of a workflow helps improve the processing speed, as multiple machines perform part of the work.

Chapter 4

Data Analytics Workflow Notation

This chapter presents the workflow notation that is designed in this thesis, called DAWN. It is an adaptation of the graphical notation for sNRC as introduced by Hidders in Section A.7. The graphical notation introduced by Hidders is altered to improve its usability in the BDP. The differences between both notations and the implications of these differences will be discussed in more detail in Section 4.3. The design considers both the goals stated in Chapter 3 and the formal definition of sNRC, as provided in Appendix A. The resulting design artefacts presented here are a formal definition of the DAWN syntax and semantics, and a DAWN type system. Finally, a discussion on the implications of adding UDFs to DAWN will be given in Section 4.4.

4.1 Syntax

This section will give a formal definition of the DAWN syntax. The syntax is the set of rules that defines the notation of DAWN. To give this definition, the concept of a workflow will be defined first. This concept is used to describe a graph representation of workflows and is a special type of graph, which will be related to a regular graph, called the associated graph. The workflow and associated graph concepts will be used to give the formal definition of the DAWN syntax. To clarify the DAWN syntax, several properties will be discussed in more detail. These properties describe key elements of the DAWN syntax.

4.1.1 Workflow

DAWN is a graphical representation of a dataflow graph structure, a workflow. This is considered an intuitive method to describe dataflows [22] and it fits the design goals. In this graph structure, data operations are depicted as vertices and data dependencies as directed edges. Data dependencies are dependencies between components, where the target component requires input data from the source component. Edges can therefore be seen as a description of data flow, as their direction shows the flow of data through the workflow.

The workflow graph serves as a graphical representation, but has a formal description as well. However, a workflow is not a regular graph as defined in graph theory. A special property of a workflow is that the components, or vertices, are not directly connected, but connected through ports. These ports represent input or output for a

component or the whole workflow. Definition 1 gives a definition of a workflow, as it will be used throughout this thesis. The notation chosen here is comparable to that of graph theory, to emphasise the relation between the concepts.

Definition 1. A workflow $G_{WF} = (C, P, D)$, consists of a set of components C, a set of ports P, and a set of edges D, such that:

- The sets C, P, and D are mutually disjoint.
- $P = P^{\text{i}} \cup P^{\text{o}} \cup P^{\text{i}}_C \cup P^{\text{o}}_C$, where P^{i}, P^{o}, P^{i}_C, and P^{o}_C are mutually disjoint, and:
  - P^{i} is a set of workflow input ports.
  - P^{o} is a set of workflow output ports.
  - P^{i}_C is a set of component input ports.
  - P^{o}_C is a set of component output ports .
- Each component $c \in C$ has a set of input ports $P^{\text{i}}_c \subseteq P^{\text{i}}_C$, and a set of output ports $P^{\text{o}}_c \subseteq P^{\text{o}}_C$.
- Each edge $(p,q) \in D$ is an ordered pair of ports p and q, such that $p \in P^{\text{i}} \cup P^{\text{o}}_C$ and $q \in P^{\text{o}} \cup P^{\text{i}}_C$.

Summarising, a workflow graph consists of components, edges, and ports. Components represent data operations and have their own input and output ports. Edges connect workflow ports and component ports to describe the flow of data. This graph definition has a graphical representation similar to that of sNRC used by Hidders in Section A.7. An example of a DAWN workflow graph can be seen in Figure 4.1.

All DAWN workflows in this thesis are visualised in this same style. The workflow in Figure 4.1 has four components, labelled: f,g,h and i. The input and output ports of a workflow are displayed at the left and right edges of the workflow respectively. The example workflow has two input ports on the left side of the workflow and two output ports on the right side. Components are visualised similarly, with input ports on the left and output ports on the right side of the component. The directed edges are shown by arrows pointing from their source to their destination.

4.1.2 Associated Graph

As discussed in Subsection 4.1.1, a workflow is not a regular graph as defined in graph theory. The ports in a workflow make it a special type of graph. However, a workflow can be represented as a directed graph. The directed graph related to a workflow will

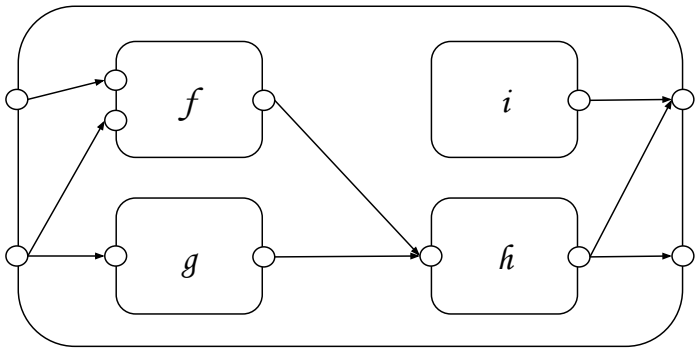

Figure 4.1: **Example of a DAWN workflow graph.**

be referred to as its *associated graph*. The formal definition of the associated graph for a workflow is given in Definition 2.

Definition 2. For each workflow $G_{WF} = (C, P, D)$, with $P = P^{\mathrm{i}} \cup P^{\mathrm{o}} \cup P^{\mathrm{i}}_C \cup P^{\mathrm{o}}_C$, there is an *associated graph* $G_A = (V, E)$, such that:

- $V = C \cup P^{\mathrm{i}} \cup P^{\mathrm{o}}$
- For each $(p,q) \in D$, where $p \in P^{\mathrm{i}}$ and $q \in P^{\mathrm{o}}$, there is an $(p,q) \in E$.
- For each $(p,q) \in D$, where $p \in P^{\mathrm{o}}_r$ and $q \in P^{\mathrm{i}}_s$, there is an $(r,s) \in E$, with $r,s \in C$.
- For each $(p,q) \in D$, where $p \in P^{\mathrm{i}}$ and $q \in P^{\mathrm{i}}_s$, there is an $(p,s) \in E$, with $s \in C$.
- For each $(p,q) \in D$, where $p \in P^{\mathrm{o}}_r$ and $q \in P^{\mathrm{o}}$, there is an $(r,q) \in E$, with $r \in C$.

The associated graph represents each component, workflow input port, and workflow output port as a vertex, while omitting the component ports. All edges connected to component ports in the workflow connect to the component vertex directly in the associated graph. Figure 4.2 shows the associated graph for the DAWN workflow shown in Figure 4.1. As the associated graph defines a directed graph, rules from graph theory apply to it, this feature will be used to give the definition of the DAWN syntax.

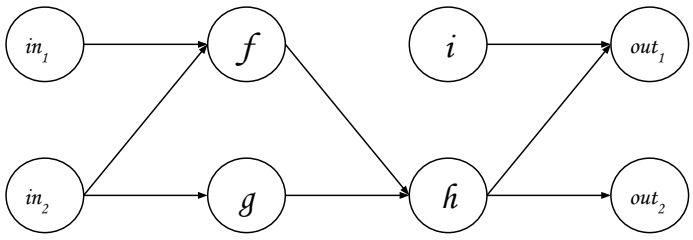

Figure 4.2: **Example associated graph for the workflow graph in Figure 4.1.**

4.1.3 DAWN Syntax

Using the definitions given in Definition 1 and 2, this section will present the formal definition of the DAWN syntax. DAWN is a type of workflow that a poses several restrictions on the design of the graph that improve usability or help users design valid workflows. These properties will be discussed in more detail in Subsection 4.1.4. Definition 3 gives the formal definition of a DAWN workflow in terms of a workflow and its associated graph.

Definition 3. DAWN is a workflow $G_{WF} = (C, P, D)$, with an associated graph $G_A = (V, E)$, if and only if all of the following statements hold:

- G_A is a directed acyclic graph.
- G_A is a weakly connected graph.
- G_A has a directed path from each $v \in V$ to a $p \in P^{\text{o}}$, where $P^{\text{o}} \subseteq V$.
- $P^{\text{o}} \neq \emptyset$.
- For each $c \in C : P_c^{\text{o}} \neq \emptyset$.
- For each $p \in P_C^{\text{i}} \cup P^{\text{o}}$, there exists an ordered pair $(r, p) \in D$.

4.1.4 Properties

To clarify the definition of the DAWN syntax and the design choices behind it, this section will discuss several main properties of DAWN workflows. These properties are derived from the definitions of a workflow, the associated graph and the dawn syntax as presented in the previous sections.

Property 1. The associated graph is a directed acyclic graph.

A directed acyclic graph is a graph for which it holds that all edges are directed and there are no directed cycles in the graph. This means that there is no vertex v in the graph for which a directed path can be found that returns to vertex v. If the associated graph is acyclic, the DAWN workflow cannot have any cycles either. Cycles in a DAWN workflow are unwanted, as the edges of the workflow represent the flow of data. Data should not flow back to a component from which it originated. Doing so would potentially cause an infinite loop in the workflow execution. Iterations over datasets should rather be specified inside a component than in the workflow.

Property 2. The associated graph is a weakly connected graph.

A directed graph is weakly connected if removing the direction of the edges results in a strongly connected graph. A strongly connected graph is a graph for which all vertices can be reached by traversing the graph from any vertex. Practically, this means all vertices are in a single graph.

DAWN requires that the workflow is represented by a single, connected graph. This means two thing: there cannot be any disconnected components, i.e. components that are not connected to the workflow, and there cannot be multiple graphs defined in

one workflow. Figure 4.3 shows an example of two disconnected graphs in one workflow. The workflow is similar to that in Figure 4.1, however there is an edge missing at the location of the red cross. Although both the workflow containing component i and the workflow containing components f, g and h are considered valid workflows, the definition of both in one workflow is not.

The motivation behind this design choice is that a data analyst working in the BDP will work on something which is conceptually called a "workflow". The analyst would therefore not expect this workflow to contain multiple workflows. If the user wants to create multiple workflows, they can create another workflow object in a new specification. This makes it easier to keep an overview of all separate workflows as well as allow for separate execution of these workflows.

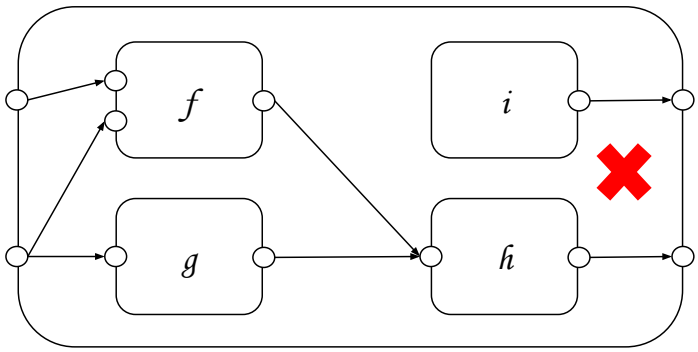

Figure 4.3: **An example of a disconnected workflow. The cross marks the missing arrow w.r.t. Figure 4.1.**

Property 3. A DAWN workflow has at least one output port.

A workflow is expected to produce data, for the simple reason that this is its purpose. An output port provides a result of the workflow. Therefore, a workflow should always have at least one output port. This is formally stated in Definition 3 as: $P^{\mathrm{o}} \neq \emptyset$. The notation allows for multiple outputs, as data analysts might require multiple different datasets to be generated by a workflow. For example, data analysis might produce both a dataset that is used to generate a report as well as a dataset with the complete analysis results. By outputting these results to different ports, the notation makes it explicit that there are multiple results produced by the workflow. This allows to design DAWN workflows which are side-effect free.

On the other hand, a workflow can have zero or more input ports as it is possible that the workflow does not require input data. Specifically, it is possible to have a component that does not require input, but does produce output. A workflow containing only this component would be considered a valid workflow. A more elaborate discussion of components without input will be given in Property 4. Furthermore, a workflow requiring multiple datasets as input is also possible. An input port is required for each input dataset. Figure 4.1 shows an example workflow with multiple input ports, multiple output ports and a component (i) that does not require any input.

Property 4. Each component has at least one output port.

Analogous to the workflow, a component requires at least one output port, as a component always produces output data. Definition 3 states: $P_c^{\text{o}} \neq \emptyset$ for each $c \in C$. In general, a component represents an n-ary function, such as those specified in sNRC in Section A.6. Thus, a component would generally have only a single output port. However, the workflow notation allows users to nest an existing workflow as a component inside another workflow. Workflows can have multiple output ports, as discussed in Property 3. Therefore, a component can also have multiple output ports.

As components represent n-ary functions, they reason over n input parameters. This requires the components to have zero or more input ports, with each port representing one of the input parameters. An example of a component that does not require input is a random number generator. Such a generator might be useful in a specific workflow. This requires the random number generator to be defined in this notation as a component with no input ports and a single output port. An example of this component is shown in Figure 4.4.

Figure 4.4: **A Random Number Generator component showing no input ports and a single output port.**

Property 5. An edge connects a workflow input port or a component output port to a workflow output port or a component input port.

The formal definition of this property was given in Definition 1 as: each edge $(p,q) \in D$ is an ordered pair of ports p and q, such that $p \in P^{\text{i}} \cup P_C^{\text{o}}$ and $q \in P^{\text{o}} \cup P_C^{\text{i}}$. This statement prescribes that each edge should point in the correct direction. In a workflow, the data flows from the workflow input ports, possibly through components, to the workflow output ports. Flowing through components means the data enters the component through a component input port and exits through a component output port.

This description of dataflow implies the definition of directed edges in a workflow. Any other edges would result in an infeasible workflow. For example, an edge from a component input port to a component output port has no logical meaning, as data is not returned by an input port and cannot be consumed by an output port. Figure 4.5 illustrates this by means of a small workflow consisting of two components. The input port of component h is connected to the output port of component g. This incorrect workflow makes it clear that there is no correct flow of data due to the marked edge.

Property 6. All component input ports and workflow output ports have at least one incoming edge.

To ensure functionality of a workflow, all output ports of the workflow should return some results. This is only possible if there is an incoming edge on that port. However, the result returned can be empty, as that depends on the functionality of the workflow. Moreover, components cannot perform their data operation if not all input is received. This means that each input port on a component should have an incoming

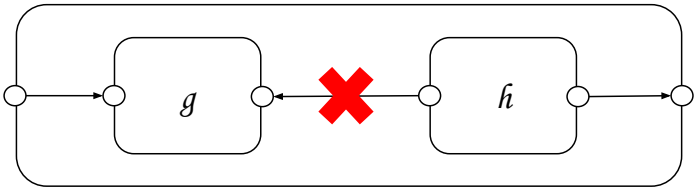

Figure 4.5: **A workflow containing an incorrect edge.**

edge as well. This holds that for each workflow output port and component input port, there should be an edge that connects to it. Definition 3 specifies this as follows: for each $p \in P_C^{\text{i}} \cup P^{\text{o}}$, there exists an ordered pair $(r, p) \in D$. P_C^{i} is the set of all component input ports and P^{o} is the set of all workflow output ports.

Property 7. All workflow input ports and components contribute to the workflow output.

All data input and data operations in the workflow should contribute to a result of the workflow. Inputs or components that do not have a directed path to a workflow output port do not contribute to the workflow result. Therefore, Definition 3 states: G_A has a directed path from each $v \in V$ to a $p \in P^{\text{o}}$, where $P^{\text{o}} \subseteq V$. A directed path to a workflow output port in the associated graph G_A for each vertex v ensures that all workflow inputs and components contribute to a result of the workflow.

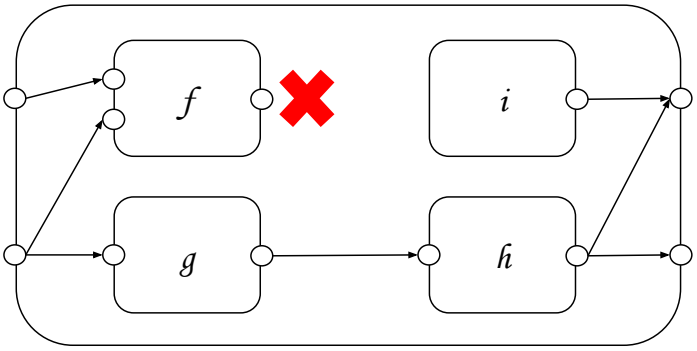

Figure 4.6: **A workflow containing unconnected component f.**

An example of an incorrect workflow is given in Figure 4.6. In this figure component f has no outgoing edge from its output port. In the associated graph Figure 4.7 for this workflow, it can be seen that there is no directed path from in_1 or f to any of the output ports. The result of this component is therefore not used for any output of the workflow, which means the component can be removed from the workflow without effecting the result.

Similarly, Figure 4.8 has an output port at component f which is not connected. However, this is a valid workflow, as all components and inputs have a directed path to a workflow output port. The associated graph for this workflow is identical to that given in Figure 4.2.

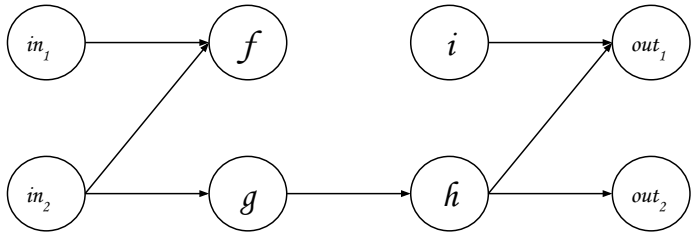

Figure 4.7: **The associated graph for the workflow in Figure 4.6.**

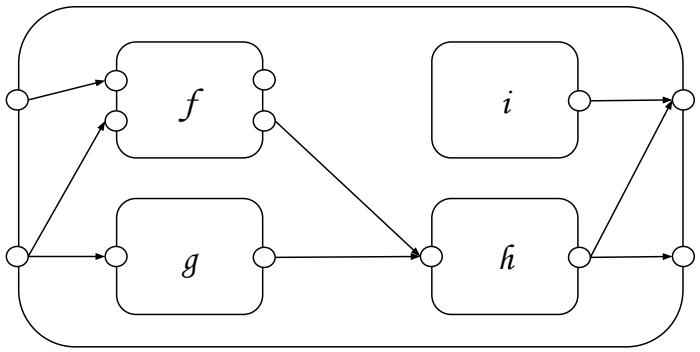

Figure 4.8: **A workflow containing an unconnected output port at component f.**

4.2 Semantics

Having defined the syntax, this section will discuss the definition of the DAWN semantics. The semantics defines the meaning of DAWN in the sense that it defines how the specification should be interpreted. DAWN implements several solutions to help users design workflows. Examples of these solutions are a type system and adding implicit components. This section will give definitions for the DAWN type system, implicit components and discuss the semantics of DAWN workflows.

4.2.1 DAWN Values

To be able to define the DAWN semantics, it is necessary to define the values that are used in DAWN first. The values represent the data that is input and output to the workflow and the components in the workflow. Definition 4 specifies the set W, which is the set of all DAWN values. This definition shows that DAWN values can consist of all real numbers $\mathbb{R}$, boolean values $TRUE$ and $FALSE$, strings in Σ^*, sets of DAWN values, and records R. The implications of these value types will be discussed in the following section.

As the model for semi-structured data depends on labelled data elements, field names are an important aspect for the DAWN data model. For the following definition, the set A is therefore defined as the set of all possible names, which are used to label a data element.

Definition 4. DAWN values are recursively defined as the smallest set W, for which the following holds:

- $\mathbb{R} \subseteq W$
- $\{TRUE, FALSE\} \subseteq W$
- $\Sigma^* \subseteq W$, where Σ represents an alphabet of characters.
- $\{w \mid w \subseteq W\} \subseteq W$
- $(a_1 : w_1, ..., a_n : w_n) \in W$, if $w_1, ..., w_n \in W$, and $a_1, ..., a_n \in A$ are distinct.

An important data structure used in DAWN is the record. A record is a named collection of data elements, described in Definition 4 as: $(a_1 : w_1, ..., a_n : w_n)$. The subset of all records in W is defined as R in Definition 5.

Definition 5. $R = \{(a_1 : w_1, ..., a_n : w_n) \mid (a_1 : w_1, ..., a_n : w_n) \in W\}$

4.2.2 DAWN Type System

To help data analysts design valid and meaningful workflows, a typing feature is added to DAWN. Typing in this notation is done by specifying the data structure of a dataset at a port. This means that a component defines specific data structures for each input port. Moreover, a component specifies the data structure it returns on each output port. This allows a user to easily notice whether or not the workflow is processing the expected data structure. Additionally, typing helps the user discover the data structure of intermediate data.

The type system consists of a named data structure description. As discussed in Subsection 2.6.1, semi-structured data often consists of a nested structure with named elements. The type system designed for DAWN specifies the structure of the data as well as named elements in the semi-structured data. This approach was chosen as it allows to describe the data structures most common to the BDP in which DAWN will be applied. These structures include XML and JSON, as well as structured data sources such as data tables.

Using named elements allows to ensure that the correct data is being compared or addressed. As the order of elements in semi-structured data is uncertain, it is necessary to label the values with names. Furthermore, this typing system considers names to help users design meaningful workflows. Rather than just regarding data types to verify correctness, names are used for this as well. For example, a port that provides a string “user name” and integer “age” should not be input for a port that expects a string “product name” and integer “quantity”. Although these ports have the same data types, this data flow is considered incorrect based on the names. Logically, such a data flow would not represent anything meaningful, as both ports represent completely different concepts.

The definition of the type system is given in Definition 6 and 7. This definition considers three different forms of types: primitives, records and collections. Primitives are value types common to most typed programming languages: bool, float, int, and

string. A bool is a boolean value, an int is an integer, a float is a floating point value, and a string is a sequence of characters. This is defined in the semantic definition of the DAWN type, in Definition 7. Primitives are denoted by P in Definition 6. Records represent sets of named values, for instance the example mentioned above would be a record with a string "user name" and an integer "age". Records are denoted as $(a_1 : T_1, ..., a_n : T_n)$ in Definition 6. As records can be used to map an name on a value, it is clear that all names in a record must be unique. Collections represent repetition and are denoted as $\{T\}$ in Definition 6. Repetition means that the type defined in the collection is repeated. This is useful for collections such as lists and tables.

Definition 6. A DAWN type T is given by the following syntactical rules:
$T \rightarrow R \mid C \mid P$
$R \rightarrow (a_1 : T_1, ..., a_n : T_n)$, where $a_1, ..., a_n$ are mutually distinct.
$C \rightarrow \{T\}$
$P \rightarrow bool \mid float \mid int \mid string$

Definition 7. The semantics of a DAWN type is given as follows:

- $[\![(a_1 : T_1, ..., a_n : T_n)]\!] = \{r \in R \mid r.a_1 \in [\![T_1]\!], ..., r.a_n \in [\![T_n]\!]\}$, where R is the set of records as defined in Definition 5.
- $[\![\{T\}]\!] = \{t \mid t \subseteq [\![T]\!]\}$
- $[\![int]\!] = \mathbb{Z}$
- $[\![float]\!] = \mathbb{R}$
- $[\![bool]\!] = \{TRUE, FALSE\}$
- $[\![string]\!] = \Sigma^*$

These definitions specify type T as an object which can consists of a record of nested named types R, a collection C of a type T, or a primitive P. The lower-case character a denotes the name of a named type. The notation used in Definition 6 is inspired by the NRC object description given by Van den Bussche et al. [23] and applied in a production rule fashion. This definition allows to describe the semi-structured and structured data types common to the BDP in the form of key/value pairs. To illustrate this, examples will be presented here for both structured data and semi-structured data.

Examples

An example of structured data is the data table shown in Table 4.1. A definition in the DAWN type system of this data table is given by Equation 4.1. In words, this type definition states: T_a is a collection of records, containing three primitives: "`name`" as a string, "`surname`" as a string, and "`age`" as an integer.

$$T_a = \{(\texttt{name} : string, \texttt{surname} : string, \texttt{age} : int)\} \tag{4.1}$$

An example of semi-structured data is given in JSON in Code Fragment 4.1. The DAWN type of this JSON object is given by Equation 4.2. In words, this type definition

name	**surname**	**age**
John	Doe	45
Jane	Doe	33
...	...	...

Table 4.1: Example data of type T_a.

name	**age**
John	45
Jane	33
...	...

Table 4.2: Example data of type T_b.

Code Fragment 4.1: Example data of type T_c.

```
{
  "type": "Sales",
  "products": [
    {
      "name": "Product A",
      "amount": 12.50
    },
    {
      "name": "Product D",
      "amount": 13.60
    },
    ...
  ]
}
```

states: T_c is a record of nested types "`type`" as a string and "`products`" as a collection. This collection contains a record, containing two primitives: "`name`" as a string, and "`amount`" as a float.

$$T_c = (\texttt{type} : string, \texttt{products} : \{(\texttt{name} : string, \texttt{amount} : float)\}) \tag{4.2}$$

Subtype

An important aspect of the use of typing in DAWN is the subtype relation. This relation specifies whether or not a type is a subtype of another type. The subtype relation is used to determine if the input for a component port complies with the required data structure. Specifically, if a type T_a is a subtype of type T_b, then T_a is considered valid input for a port requiring type T_b.

An illustration of this relation is given by the types of the data tables given in Table 4.1 and Table 4.2. T_a was given in Equation 4.1, T_b is given in Equation 4.3. Type T_a as presented in Equation 4.1 is a collection containing a record of three named primitives. Type T_b as presented in Equation 4.3 is a collection containing a record of two named primitives.

$$T_b = \{(\texttt{name} : string, \texttt{age} : int)\} \tag{4.3}$$

Type T_a is considered to be a subtype of T_b, denoted by $T_a \leq T_b$, as T_a contains at least all named types in T_b, in the same structure. It can easily be seen that the struc-

tures of T_b and T_a are the same: both are a collection containing a record. Furthermore, all named primitives in the record of T_b can be found in the record of T_a. This leads to the conclusion that $T_a \leq T_b$. However, the inverse is not true: $T_b \not\leq T_a$. The record in T_a has an extra string "`surname`", which is not present in T_b. The formal definition of the subtype relation is given in Definition 8.

Definition 8. The subtype relation, denoted by $T_a \leq T_b$, is given by the smallest relation for which the following holds:

- $int \leq float$
- $P_a \leq P_a$ for all a
- $\{T_a\} \leq \{T_b\}$ if $T_a \leq T_b$
- $(a_1 : T_1, ..., a_n : T_n) \leq (b_1 : T'_1, ..., b_m : T'_m)$ if for each $b_y : T'_y \in (b_1 : T'_1, ..., b_m : T'_m)$, there exists an $a_x : T_x \in (a_1 : T_1, ..., a_n : T_n)$, such that $a_x = b_y$ and $T_x \leq T'_y$.

Note that this definition is equivalent to stating $[\![T_a]\!] \subseteq [\![T_b]\!]$ as defined in Definition 7. If a type semantically defines a subset of another type, these types have the subset relation. From the previous definitions it can be seen that:

- $[\![P_a]\!] \subseteq [\![P_a]\!]$ for all a
- $[\![\{T_a\}]\!] \subseteq [\![\{T_b\}]\!]$ if $[\![T_a]\!] \subseteq [\![T_b]\!]$
- $[\![(a_1 : T_1, ..., a_n : T_n)]\!] \subseteq [\![(b_1 : T'_1, ..., b_m : T'_m)]\!]$ if for each $b_y : [\![T'_y]\!] \in (b_1 : [\![T'_1]\!], ..., b_m : [\![T'_m]\!])$, there exists an $a_x : [\![T_x]\!] \in (a_1 : [\![T_1]\!], ..., a_n : [\![T_n]\!])$, such that $a_x = b_y$ and $[\![T_x]\!] \subseteq [\![T'_y]\!]$.

Moreover, it can be seen that $T_a \leq T_b$ for any $[\![T_a]\!] \subseteq [\![T_b]\!]$ and vice versa. This also explains the first rule in Definition 8, as from the subset relation $\mathbb{Z} \subseteq \mathbb{R}$, it follows that $int \leq float$.

4.2.3 Implicit Components

To help users design their DAWN workflow, some concepts are not present in the notation: implicit components. Implicit components are components which are implied by the notation, but are not specified in the notation. These components often perform a trivial task in manipulating data, for correct processing. An example of this is an additive union component, this component takes multiple inputs of the same type, performs an additive union on the input, and returns a single output. An additive union is an operation where all elements of each input are placed in a single collection, returning a combined set of elements.

As will become clear from Definition 9, DAWN does not support multiple, different inputs on a single component input port or a workflow output port. However, the syntax allows for multiple edges to connect to these ports. This is solved by the use of implicit components, by inserting the additive union component at each port with multiple incoming edges. For each incoming edge, a port is added to the additive union component and the edge is connected to it. The output of the additive union component is connected to the port which had multiple incoming edges.

4.2.4 DAWN Semantics

Components are semantically assumed to be binary relations that map functions onto functions. These functions map a port onto a DAWN value. Specifically, a component represents a binary relation that maps functions for its input ports onto functions for its output ports. This means that without knowing the exact operation of the component, we can give the domain for the binary relation it represents. This domain helps formally define the DAWN workflow. Furthermore, the semantics of edges in a DAWN workflow is that both ports connected to the edge map to the same value. Therefore, applying the rules for edges from the DAWN syntax given in Definition 3, the semantics of a DAWN workflow must also be a binary relation. The formal definition of the DAWN semantics is given in Definition 9. The following section will elaborate on this definition in a number of properties.

Definition 9. The semantics of a DAWN workflow $G_{WF} = (C, P, D)$ is defined as follows:

- The semantics of a component $c \in C$ is assumed to be a binary relation
  $B_c \subseteq I \times O$, where $I = \{b \mid b : P_c^{\mathrm{i}} \to W\}$ and $O = \{b \mid b : P_c^{\mathrm{o}} \to W\}$
- The semantics of a DAWN workflow $G_{WF} = (C, P, D)$ is therefore given by the following binary relation:
  $G = \{(b[P^{\mathrm{i}}], b[P^{\mathrm{o}}]) \mid b : P \to W, \forall (q,r) \in D : b(q) = b(r), \forall c \in C : (b[P_c^{\mathrm{i}}], b[P_c^{\mathrm{o}}]) \in B_c\}$

  Here b denotes a function that maps ports P onto DAWN values W and $b[X]$ denotes a restriction of function b to the domain of X. The semantic definition of an edge can be seen in this relation as $\forall (q,r) \in D : b(q) = b(r)$, meaning that both ports connected to an edge map to the same value. Binary relation G respects the semantic definition of components, the semantic definition of edges, and the syntax as defined in Definition 3.

4.2.5 Properties

Property 8. There can only be an edge $(a,b) \in D$ for ports a and b, with types T_a and T_b respectively, if $T_a \leq T_b$.

Given Definition 9, and the fact that ports that are connected by an edge bind to the same value, it is only possible to connect ports that have a subtype relation. A subtype relation ensures that the value bound to port a contains all elements required by port b. However, as the definition of the semantics indicate, the values are equal in both bindings. To this end the subtype is altered to be the exact same as the supertype. It can be seen that if $T_a \leq T_b$, removing elements from the value with T_a ensures it to be identical to T_b.

Property 9. Data is duplicated over all outgoing edges of a port.

A port can have multiple outgoing edges. This holds for workflow input ports and component output ports. As Definition 9 states, both endpoints of an edge bind to the same value. This means that a port that has multiple outgoing edges, will provide the same data to all endpoints of its outgoing edges.

Property 10. A port with multiple incoming edges is shorthand for an additive union component.

A port that has multiple incoming edges cannot exist due to Definition 9. Seen as both endpoints of an edge bind to the same value, this is not feasible for multiple edges to end in one port. In this case the implicit additive union component is inserted before this port. All incoming edges are connected to their own input port on the additive union component and the output port of the additive union component connects to the port that had multiple incoming edges. This ensures that the binding of values is correct for all ports in the workflow.

4.3 Relation to sNRC

The DAWN definition presented here is similar to the notation introduced by Hidders in Appendix A. However, some design decisions in this work have led to differences between the two. These decisions are mostly for practical application of the notation in the BDP, keeping the customers as users in mind. In this section the relation between the two languages will be discussed. This is done by analysing the differences and explaining how DAWN can be mapped to sNRC. Thereby showing that DAWN has a solid theoretical base as well as practical application. This theoretical base in sNRC proves useful in optimisations and helps achieve Goal 5.

4.3.1 Iterating Input Ports

The specification presented by Hidders in Appendix A defines iterating input ports in components. These specially marked input ports indicate that the dataset that is input to that port will be iterated over. This fits the idea of sNRC that everything is a bag, which can be iterated over. To simplify the notation, iterating ports are not present in the definition of DAWN. Moreover, DAWN does not consider everything to be a bag.

Data can still be iterated over, however this shall be defined in the implementation of a specific component. The DAWN notation does not consider the specific implementation of a component, it focusses on the workflow as a whole and the interaction between components. Although this does cause a decrease in expressiveness in DAWN compared to sNRC.

4.3.2 Connectedness of the Graph

The definition of the DAWN syntax, Definition 3, states that the associated graph of a DAWN workflow is a weakly connected graph. This holds that all components in a workflow should be in a single connected graph. Both the definition of DAWN and the definition of sNRC do not allow for disconnected components. Disconnected components are not part of the workflow graph and therefore do not contribute to the workflow result.

Technically, the graphical notation introduced by Hidders in Appendix A does allow for multiple graphs in one workflow. This is not allowed in the DAWN definition presented here. However, each of the graphs in sNRC would need to be a complete workflow by itself. The result of having multiple workflows specified in a single graph

is that these independent workflows are executed simultaneously. As there are no edges between the workflows, there cannot be any dependencies between the workflows.

It can be intuitively seen that any number of workflow graphs can be represented in DAWN using multiple separate workflow specifications. The reason that this design choice was made, is that it is considered more user friendly to have a concept of a workflow in the BDP, which contains only a single workflow. The possibility of simultaneously running workflows is also available in the BDP, by scheduling them at the same time or interval.

4.3.3 Multiple Output Ports

DAWN allows for multiple output ports to be defined in workflows, as discussed in Property 3. This is done to allow users to have multiple forms of output for a workflow and clearly specify them in the notation. As a result of the possibility to nest workflows, this results in having components with multiple output ports as well. Because sNRC is a query language that uses n-ary functions it does expect multiple inputs, but always produces one output. However, DAWN allows workflows do have multiple outputs, an example was shown in Figure 4.1.

A special component could be defined and added to such a workflow, to allow a mapping to sNRC with a single output port. This component has an input port for each workflow output port. The result is a record, as defined in the DAWN type system in Subsection 4.2.2, containing each input dataset as an element, labelled with the name of the original workflow output port. The workflow output ports are replaced by a single workflow output port returning this record. A reversed version of this component can be implemented to allow the different outputs to be split up again. This allows the workflow to be nested as a component in another workflow.

4.3.4 Typing

DAWN provides typing as an aid to the user to design meaningful and valid workflows. Subtyping is applied to confirm the validity of a connection between ports, as discussed in Property 8. This typing consists of defining a named data structure as discussed in Definition 6 and 7. The sNRC query language does consider both typed and untyped data, while assuming everything to be a bag. This construct allows to ensure definedness of expression input and results.

As DAWN does not consider all data to be bags, this definedness needs to be ensured differently. This is achieved by applying the type system. The type system defines input and output types for each component, while the editor of the workflow indicates incorrect edges in a workflow. This ensures that each component is presented with the correct, expected, data structure.

4.4 User Defined Functions

An important aspect of the BDP is that data analysis tools and services can be easily integrated into the portal. This is done through UDFs. UDFs are components which can be added to the BDP workflow editor. Users can then add these to their workflows.

These components are identical to existing components, as defined earlier in this chapter. They have input ports, output ports and they have types associated with their ports. Moreover, UDFs have a semantics that is compatible with the specified types. UDF components are in fact also binary relations in the sense of Definition 9.

Although UDFs do not necessarily depend on external services, this is the most likely use case for the BDP, as described in Section 2.2. External services are in this case data analysis tools which are available as cloud services, such as the aforementioned Microsoft Azure HDInsight. One of the main goals for the BDP is to make such external services available for use in workflows.

Several challenges arise from UDFs depending on external services. For example, if an external service processes the data in a workflow, there is no guarantee that there is no result stored at this service. This would constitute an output, which is not defined in the workflow. Therefore, it is hard to decide whether or not a workflow is side-effect free, as discussed in Property 3. Moreover, deciding termination for a workflow with UDFs is also more complex. An external service might not be able to communicate its status, leaving the workflow waiting for output which might not arrive. These challenges have to be dealt with in the implementation of the external services and the component itself.

Chapter 5

DAWN in JSON

DAWN is a graphical workflow graph representation. A graphical representation is useful for the user to design workflows. However, for efficient reading and storing DAWN, a machine-readable format is required. For this purpose a serialisation of the notation has been designed. This serialisation is a textual representation of the DAWN workflow in JSON. This chapter will discuss the serialisation of DAWN, as designed for the BDP. This includes the format and structure which are applied in the notation. The structure and format are important, as they define the practicality of the notation. Furthermore, the elements in the workflow notation are discussed, to show how a workflow can be defined using the notation.

5.1 JSON Format

For the workflow notation, the JSON format was chosen, as this is a well-known and widely used format. Furthermore, it fits within the ORTEC web stack as mentioned in Section 2.1. It can be easily implemented in the JavaScript based web interface that the BDP will use.

As alternative format XML was also considered, which has similar capabilities. XML is considered less suitable, as the interface has native support for JSON and XML is more verbose. An important advantage of XML is that it has a standardised schema definition, which allows for validation of XML documents. Similar validation is also available for JSON, although it is not yet considered a standard, as will be discussed further in Section 5.3.

5.2 Elements

This section will show the structure of a JSON formatted workflow document and discuss the elements defined in the notation. The notation contains a number of elements that combine to describe the workflow graph and additional information. A full example workflow specification can be found in Appendix B. Note that this section omits some non-crucial elements in the notation, for legibility.

Code Fragment 5.1: JSON root element

```
{
  "dawn": {
    "version": 0.9,
    "resources": [ ... ],
    "workflow": { ... }
  }
}
```

5.2.1 Root

The root of the JSON specification is an object containing a `dawn` element. Code Fragment 5.1 gives an example of this element. This element contains the workflow specification in the `workflow` element and two other elements. There is a `version` element, which states to which version of the workflow notation this specification adheres. This version number is mostly useful for development, to keep track of the notation version used in the given specification. There is also a `resources` list, which will be discussed next. All these elements in the `dawn` element are required for a valid specification.

5.2.2 Resources

The DAWN notation distinguishes between two types of workflows: nested workflows and standalone workflows. The difference between the two is that nested workflows do not specify resources. These workflows are used as components, nested in another workflow. Standalone workflows do specify resources, which are linked to input and output ports of the workflow. These resources represent data sources and they are used to point to the location of the data storage. The BDP allows users to specify a number of resources, which can be used in several workflows. The reason that this element is outside of the `workflow` element is that it is optional, as it is not a part of nested workflows. In standalone workflows, resources are referenced from the input and output ports of a workflow.

In the example Code Fragment 5.2 a *Comma Separated Values* (CSV) resource is specified. Two fields are used to identify this resource: `id` and `name`. The type of the resource is given by `typeId`, types such as CSV files, SQL databases, HDFS storage, and document stores can be though of. These fields match the data stored in the BDP database regarding resources. The `elements` field defines resource type-specific properties. In the example, settings specific to CSV file processing are given, such as: location, the presence of a header line and the delimiter character.

5.2.3 Workflow

The `workflow` element is the main component of the notation, as this gives the specification of the workflow itself. In Code Fragment 5.3 all parts of the workflow can be seen. There are three elements that help identify the workflow: `id`, `name`, and `description`. The other elements define the workflow graph.

Code Fragment 5.2: Resources example with CSV resource

```
"resources": [
  {
    "id": 1,
    "name": "Uploaded CSV file",
    "typeId": "csv",
    "elements": {
      "csvLocation": "..\\input\\",
      "csvFilename": "data.csv",
      "csvHeader": "true",
      "csvDelimiter": ";"
  }
  },
  ...
]
```

Code Fragment 5.3: Workflow element

```
"workflow": {
  "id": 1,
  "name": "Ranking with R script",
  "description": "Executing a ranking R Script on CSV data.",
  "inputs": [ ... ],
  "outputs": [ ... ],
  "nodes": [ ... ],
  "edges": [ ... ]
}
```

The graph structure of the workflow can be clearly seen in the specification of `nodes` (vertices) and `edges`. The `inputs` and `outputs` lists given here are the workflow input and output ports that provide access to data sources. These sources can be resources, or, in the case of a nested workflow, point to data sources provided by the nesting workflow. The `input` list can be empty, the `output` list on the other hand should have at least one element, as defined in Definition 3.

5.2.4 Nodes

The `nodes` list contains all nodes in the workflow graph. A node element contains information to identify the node: `id` and `name`. Furthermore, a node contains a `component`. This `component` element specifies the actual function of the node. The `type` specifies the task of the component, in example Code Fragment 5.4 this is an R script, which will be executed on the input data. The component could also reference another workflow, if it were to be nested here. The `properties` field contains a set of component type-specific settings, which are required to perform the task.

Similar to a workflow, a node contains `input` and `output` lists. These lists specify

Code Fragment 5.4: Nodes element

```
"nodes": [
  {
    "id": 8,
    "name": "Run an R Script",
    "component": {
      "type": "rScript",
      "properties": {
        "scriptName": "hello.r"
      }
    },
    "inputs": [ ... ],
    "outputs": [ ... ]
  }
]
```

the input and output ports of a node, respectively. There must be at least one output port in a node, the input port list can be empty, as defined in Definition 3. All fields in the `node` element are required fields.

User Defined Functions

UDFs, as discussed in Section 4.4, should be added to the JSON notation as well. The design of this notation is aimed at flexibility in the `component` element. This allows different types of components to be defined, without any need for changes to the notation. As stated before, `type` defines a name for the component type which is executed. A UDF would be given a unique name, which can be referenced here. This does require some implementation work in the compiler, which will be discussed in Section 6.5. Any extra settings for the UDF component can be placed in the `properties` field. This field explicitly allows for key-value pairs where the value can be any object. This allows for lists or other objects to be added to the properties as well.

5.2.5 Edges

The `edges` list is required to connect the different ports in the workflow. Edges are defined with an `id` element, a `start` element specifying the starting port and an `end` element specifying the ending port for the edge. Therefore, all ports have unique identifiers.

5.2.6 Ports

Input and output ports are defined using an `id`, `portName` and `type`. All ports in the workflow should have a unique `id`, as these values are referenced from the edges. All ports in a component should have a unique `portName`, as this is used in the component to address the input or output. The `type` specifies the data structure that a port

Code Fragment 5.5: Edges element

```
"edges": [
  {
    "id": 18,
    "start": 3,
    "end": 9
  },
  {
    "id": 19,
    "start": 10,
    "end": 4
  }
]
```

Code Fragment 5.6: Input or Output element

```
{
  "id": 3,
  "portName": "",
  "type": { ... },
  "resource": 1
}
```

expects to receive or return. This typing system was discussed in Subsection 4.2.2, a specification of the `type` element is given in the following section.

Ports are similar for workflows and nodes, with as only difference that the workflow ports can reference a resource, whereas the node ports do not. In Code Fragment 5.6, an example workflow input port is shown.

5.2.7 Type

Type specification is used in the notation to help the user design a valid workflow, which is also discussed in Subsection 4.2.2. It gives a definition of the data structure that is expected in a port. For input ports this is the structure of the data it expects as input. For output ports this is the structure of the data it returns.

This type system defines three elements: `object`, `collection` and `typeName`. The first two refer to the record and collection respectively, defined as $(a_1 : T_1, ..., a_n : T_n)$ and $\{T\}$ in Definition 6. The latter describes a primitive type, such as a boolean, float, integer, or string. An `object` is a list of nested type elements, which can be any of the provided elements. Similarly a `collection` is a nested type, however this represents a repeated construct of a single type. Each type element has a name, as defined in Definition 6.

The example given in Code Fragment 5.7 helps illustrate this element. In this example a data structure is described that contains a collection of records with two fields. Specifically, there is a `collection` element, containing an `object` element,

Code Fragment 5.7: Example JSON type specification

```
"type": {
  "collection": {
        "object": [
      {
        "name": "Name",
        "typeName": "string"
      },
      {
        "name": "Amount",
        "typeName": "float"
      }
    ]
  }
}
```

which contains two primitives. These sub elements define a *"Name"* as string and an *"Amount"* as float. This data structure is therefore a description of a data table with two columns: *"Name"* and *"Amount"*, with their respective data types.

5.3 JSON Schema

The notation that is presented here is formalised in a *JSON Schema* [1] (draft v4). JSON Schema is a language that allows to describe contracts for the structure of a JSON document. It must be noted that JSON Schema is officially still in a draft version and it is not yet accepted as a standard by any organisation. However, it is widely used and there is a large number of implementations available that validate JSON according to JSON Schema. [2]

The schema can be used to verify if a given workflow definition is a valid JSON structure. JSON Schema is not powerful enough to verify the complete validity of a workflow specification, such as the graph structure. However, it can be used to verify that all required fields are specified and all given values are of the correct type. This validation helps prevent parsing errors while trying to read a workflow specification. The complete JSON Schema for the workflow notation is given in Appendix C.

[1] http://json-schema.org/latest/json-schema-core.html
[2] http://json-schema.org/implementations.html

Chapter 6

Compiler

As a final artefact of this design work, a compiler was built that enables the execution of the workflow notation on a workflow engine chosen by ORTEC. The engine chosen is *Windows Workflow Foundation* [1] (WF), which will be elaborated on in Section 6.1. An important challenge for the use of WF is that it is control flow oriented, a class of workflow systems discussed in Subsection 2.4.1. This as opposed to the data flow oriented approach used in DAWN. Section 6.2 will discuss how this was dealt with. Furthermore, as a control flow does not consider the location of the data, addressing is a challenge, which is discussed in Section 6.3. The compiler is designed in several parts, considering the possibility that another workflow execution engine might need to be implemented in the future.

As Goal 4 states, the notation should be portable. The compiler therefore splits up its work into several steps. The first part of the compiler only considers parsing the given JSON notation, translating this to a graph object and verifying the validity of the workflow. This part can be considered generic and relies only on the definition of DAWN. Once a valid workflow has been read from the specification, the next steps will transform the specification into an engine-specific notation. These steps and the engine-specific notation will be discussed in this section.

6.1 Windows Workflow Foundation

Windows Workflow Foundation, or WF, is a part of the .NET Framework. .NET Framework is the preferred technology used at ORTEC, which is why ORTEC chose to work with this execution engine. Furthermore, WF comes with many workflow engine features which are important to ORTEC as well. One of these features is that it supports persistence of long-running processes. This enables the engine to run workflows that might require long running data analytics processes and maintain their work in a cloud environment.

WF provides a workflow designer as well, workflows can be designed within the Visual Studio development environment. ORTEC considers this not an appropriate environment for data analysts to edit their workflows. This is the reason that another workflow design approach is required. The visual editor for WF can, however, be used by developers implementing new components in the execution environment.

[1] `https://msdn.microsoft.com/en-us/library/dd489441.aspx`

6.1.1 Extensible Application Markup Language

WF specifies its workflow in an *Extensible Application Markup Language* [2] (XAML). This notation is a XML based notation that allows to describe applications, objects, and relations between these object. For WF, XAML describes the whole workflow in activities. Activities are the computational steps in the workflow, which can also specify input variables.

A flow of activities is nested in a sequence, meaning that these activities are executed one after the other. Parallel activity execution is possible as well. Furthermore, the WF language allows to check conditions and execute loops. These last features are not present in DAWN, but can be implemented in a component build in WF.

6.2 Data Flow versus Control Flow

As mentioned, one of the key challenges of using WF as an execution engine for DAWN is that both workflow systems have different approaches to workflows. DAWN is a data flow language, whereas WF is a control flow system. This requires the compiler to translate the data flow into a control flow, or sequence of activities, for it to be executed on this engine.

The sequence of activities should respect the data dependencies defined in DAWN. For this, an activity should be executed after all activities on which it depends are finished. Translating this to the dawn notation: a component can be executed after all components that have a directed path to it have been executed. The preliminary version of the compiler that was implemented for this thesis supports only a simple approach to this translation.

6.2.1 Component Order

A concept of graph theory is applied to get the components in the workflow in an executable order: topological sorting. This is an operation that can be performed on a directed acyclic graph, as will become clear from its definition. Topological sorting places all vertices of a graph in an ordered list, in such a way that for all vertices a and b, where an edge from a to b exists, b will come after a in the ordered list. This does not require b to follow directly after a in the list, rather require it to be at a later point in the list. To illustrate this, Figure 6.1 shows the components from Figure 4.1 in a topological order. Note that switching, for instance, components f and g in this ordering would also constitute a valid topological order.

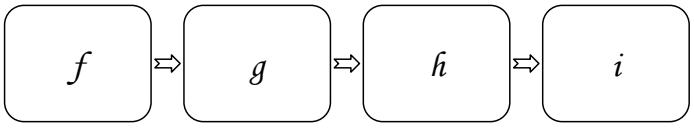

Figure 6.1: **A topological order of the components in Figure 4.1.**

[2] https://msdn.microsoft.com/en-us/library/hh700354.aspx

The resulting list contains all components in an order where all data dependencies have been met. This means that all input data for a component is available before processing that component. The compiler follows this ordering and adds the corresponding WF implementation of a component to a sequence, which is then described in XAML. Parallel execution of components is not taken into consideration by the compiler at this moment. Although a different approach to this sorting algorithm could allow to verify whether or not a number of components could be executed in parallel.

6.3 Data Addressing

Another challenge that arises from the difference in approach to the data is that of data addressing. In the control flow approach applied by WF, there is no concept of data being passed from one activity to another, data and variables only exist inside an activity. However, WF does allow for input and output arguments to be defined for each activity, this feature is used for data addressing. Data addressing is the way each component points to its data sources. To elaborate on this process, it is split in to three parts: the resource API, locating data, and using the data. The following subsections will discuss each of these parts.

6.3.1 Resource API

The resource API is an Application Programming Interface, which allows remote connections to request data or post data. Because the BDP offers the use of multiple external cloud services as part of a workflow, it is important that all these cloud services are able to access the required data. Providing an API, which is used both internally as well as externally, allows for easy integration with different platforms. The only requirement is that the implemented external platform is capable of making HTTP requests.

The resource API handles requests for a URL, specifying the resource to be used and the credentials needed to access the resource. Referring to the correct resource is done through the use of the resource id, as it is present in both the DAWN specification and the BDP database. The resource API then handles the correct connection to possible external resources or internal storage. The reason both are used will be discussed in the following section.

6.3.2 Locating Data

The main idea of the applied approach is that activities are responsible for both retrieving and storing their own data. This means that activities are provided with a set of parameters that help them address the data related to a given port name. For instance, a component defines an input port named “input1”. In the activity implementation, this input port name is known, but the location of the data it represents is not. The parameters given to the activity therefore consist of a mapping of port names to resource ids that can be used to contact the resource API.

To be able to provide the correct mapping to an activity, the compiler analyses the workflow graph as it was read from the DAWN notation. For each input port, the edges are traced back to their source to discover the input data. For each output port,

the edges are followed to find the target for the output data. There are two different possibilities to be distinguished: resources, and intermediary data stores.

Resources

Resources are a simple case for the compiler, as they can be dealt with in an easy manner. Resources are external data storages, connected to the workflow input and output ports, as discussed in Chapter 5. In case an edge from a port of a component is connected to a workflow port, the compiler adds the id for this specific resource to the mapping.

Intermediary Data Stores

Intermediary data stores are a bit more complicated, as these require some extra work from the compiler. Intermediary data stores are locations to store data that is being transferred between components. The compiler recognises the need for such stores by finding edges that connect to component ports on both ends. For each such edge, the compiler generates a new unique id in the resource API, to be able to reference this specific data store. However, this data store does not exist yet at that point in time.

As a consequence of this approach, these stores need to be provisioned and cleaned up as well. For this, the compiler adds an extra activity at the start and an extra activity at the end of a workflow. These activities are provided with the information needed to respectively provision and clean up any intermediary data stores.

Intermediary data stores only have a lifespan as long as the runtime of a workflow, to prevent any conflicts in the data stores. Moreover, if multiple instances of the same workflow are running, each instance uses separate intermediary data stores. This is only possible if the compiler, which generated the unique addresses, is run before each instantiation of a workflow. This makes the compiler a just-in-time compiler, which is only run before execution of the code it produces. This also fits the approach for the BDP, where only DAWN notation will be stored, XAML will be generated only for execution.

6.3.3 Using the Data

With the two parts in place as described here, the least amount of effort is required to address data in the component itself. An API is provided that can deliver and receive data, and the mapping of internal port names to unique resources ids is generated by the compiler. For components that operate in the BDP, i.e. some basic data manipulations, an implementation of the API is provided. However, as the BDP focusses strongly on providing external data analysis services in the cloud, these need to be able to connect to the resource API as well. This requires some implementation that is platform-specific and is able to connect to the API through HTTP requests.

Many platforms and programming languages have support for this communication method, especially amongst cloud services. However, there is still a need for such platform-specific implementations for each UDF. The approach of using a single resource API helps limit the amount of work required. If each external service would require an implementation for each supported data storage option supported by the BDP, this process would become very labour intensive. New data storage options can

be added as well, while only requiring extra implementation at the resource API. Moreover, the resource API can convert data to a standardised format to ease processing in the component.

It must be noted that adding this API as a middle man results in slower data transferring. The data need to pass through another process before the actual analysis can take place. However, given the benefit of limited platform-specific implementations in accordance with the goal of the BDP to enable the use of external cloud services in a workflow, this is considered a trade-off worth making.

6.4 Implicit Components

As discussed in Section 4.2, in some cases the DAWN specification requires components which are not explicitly defined by the user. These components are called implicit components. An example given in Property 10 is the additive union. This component is added at a port which has multiple incoming edges and combines all incoming data into a single dataset. The compiler deals with these implicit components through a set of rules. These rules are applied to the workflow to discover the need for any implicit components and insert them where needed. Therefore, the additive union is added wherever there are multiple incoming edges to a port.

Another example is a component that moves data. As discussed in Section 6.3, a component is in charge of retrieving and storing data. An example of a valid DAWN workflow is a workflow consisting of an input port and an output port, with an edge between them. Although the theoretical value of this workflow might be debatable, it is a valid workflow. And a practical application can be found in a scheduled data transfer between resources. As activities are in charge of the data transferral, this workflow needs an activity to actually move the data. This is done by the implicit "move data" component. It is inserted in a workflow where there is a direct edge between a workflow input port and a workflow output port. The sole purpose of this component is to retrieve the data from its input and store it in its output.

The use of implicit components enables the user to design simple workflows, while trivial tasks can be dealt with by these implicit component.

6.5 User Defined Functions

As discussed in Section 5.2.4, UDFs can be added to the notation without requiring any changes to it. The same goes for the compiler. The compiler has been set up in such a way that it retrieves the correct WF activity using dependency injection based on the name of the component. This means that there is a class with the exact name for each component, that returns the correct WF activity implementation to the compiler. The compiler will give an error if a non-existent component is referenced from the notation. The above stated is not only true for UDFs, but for all components provided in the workflow. This decouples the compiler from the workflow components, allowing changes to the components, while leaving the compiler unchanged. Moreover, the actual WF activity implementation can be replaced without the component type requiring changes. This allows for replacing an external cloud service, without changing any workflows that use that component.

However, as the activities do need to be implemented in the BDP, not all code can remain unchanged while adding a new component. There are several steps that need to be taken to add new components to the BDP. Firstly, a WF activity needs to be implemented that performs the data processing action for the new component. These activities could do data manipulations, or handle communication with an external service. Secondly, the activity needs to be referenced from a class that implements the interface for component definitions. This class is the link between the component name and the activity implementation. It is provided to the compiler through means of dependency injection. Thirdly, the component, its parameters, and its input ports and output ports need to be defined in the component database, to allow the workflow editor to generate the correct component visualisation and definition. Although this last step is of no relevance to the compiler.

Chapter 7

Evaluation

This chapter will evaluate the results of the design research performed for this thesis. The aim of this evaluation is to show whether the designed artefacts accomplish the goals set in Chapter 3. Evaluation is a crucial part of the design process, as it steers the design process and confirms the appropriateness of the resulting artefacts [10]. As the design approach used here was an iterative process, a number of intermediate results have been tested to the goals and the design was changed accordingly. The evaluation discussed in this chapter will consider the final result as presented in the previous chapters.

Hevner et al. [10] discuss several design evaluation methods. This evaluation applies a descriptive method, because this method has the best fit with the design work presented in this thesis. Other evaluation methods are not as suitable for this work, as the most important artefacts of this design are the conceptual model for the notation and the notation itself. It is for instance not possible to perform user tests. User tests would require a graphical workflow designer, which is not yet implement due to time constraints. The informed argument method is used, which applies research knowledge to validate the design. This knowledge can be utilised as arguments to show whether or not the design complies with the goals. For clarity, the structure of this chapter resembles that of Chapter 3.

7.1 User Friendly

One of the main goals for this design research is to provide a user friendly interface for data analysts to design their workflows. The work presented in the previous chapters aims to do this through a number of goals. Although it was not feasible to produce a functional graphical interface, in some ways the goals have been met. These goals and their evaluation are discussed here.

7.1.1 Goal 1: Design a notation that matches the data oriented view of the users.

This goal was set to enable data analysts to design workflows in an intuitive way. As data analysts prefer a data oriented view on data processing, the dataflow approach was chosen. A typical data-driven workflow is defined by Shields [22] as a graph of

operators with data dependencies between these operators. All operators in a dataflow can be executed simultaneously, only halting to wait for their dependencies.

DAWN uses the dataflow approach to define workflows, as specified in Section 4.1. Its components represent data operators and the edges in the workflow graph represent data dependencies between the components. This specification of a dataflow is analogous to that in a large number of workflow system publications [24, 7, 11, 22]. Furthermore, this work has shown that DAWN can be compiled to a control flow oriented execution engine in Section 6.2. This section emphasises the differences in approach for dataflow and control flow.

7.1.2 Goal 2: Provide users with an easy-to-use, graphical interface to design their workflows.

Graphically editing the workflow is an important aspect in making the workflow designer more user friendly. A data analyst can intuitively edit a graphical representation of the workflow graph. Due to time constraints this work does not deliver the graphical interface needed to design workflows. Designing a good user interface is a timely process, which must be done carefully and correctly to provide the best user experience. This is therefore left for future work. However, DAWN is designed with a visual representation in mind. The notation specified in Chapter 4 is shown to be visualisable, allowing for it to be used in a graphical workflow editor.

As DAWN is an adaptation of the work presented by Hidders in Section A.7, it is an inherently visual notation. Therefore, it is clear that this can be represented in visual graph. Moreover, workflow systems with graphical representations often have in common that this representation is in the form of a graph [20, 16, 21, 1]. These workflow languages apply similar concepts to the notation presented here. Vertices represent data operations and edges represent data dependencies. This model for a workflow is therefore considered visualisable. Moreover, the BDP has a preliminary visualisation of the workflow graph in its interface. However, this is not editable and, at this time, does not consider ports in the visualisation.

This evaluation aims to show that the JSON notation defined in Chapter 5 can be used to generate a complete visual representation, including ports. For the JSON notation to be visualisable, it requires that the elements defined in the notation can be mapped to graphical objects. In the BDP, JavaScript and HTML are used as web technologies, which require an implementation which is compatible with these technologies.

To show that the notation fits the user interface designed for the BDP, a brief discussion of graph visualisation in such a web interface will be given. There are a large number of JavaScript libraries available that visualise graphs, charts, and diagrams, some examples are: vis.js[1], Cytoscape.js[2], jsPlumb[3] and sigmajs [4]. A brief search on GitHub[5] provides even more (open source) libraries with similar functionality. All these libraries have in common that they explicitly specify a list of nodes and a list

[1] http://visjs.org/
[2] http://js.cytoscape.org/
[3] https://jsplumbtoolkit.com
[4] http://sigmajs.org/
[5] https://github.com/

of edges in their data model. As DAWN has these lists of nodes and edges, it can be seen that this can be visualised with minimal effort. An addition could be made in the notation to specify the visual position of a node in the workflow, i.e. the x and y coordinates on a canvas-like display.

The ports in both the components and the workflow represent a challenge for this visualisation. They can, however, be modelled as extra nodes in the graph. A modification should be made to represent them as smaller nodes, displayed at the edges of workflows or components. This would require some specific implementation and limit the usability of a predefined library as mentioned above. However, this implementation is trivial and can be easily added to the user interface.

7.2 Flexible

Flexibility is another main goal for the design of DAWN. The goal is to be able to use the workflow notation in different environments and allow for the addition of new functionality. This requires a notation that is independent of the workflow designer as well as the execution engine. Furthermore, it requires the workflow to provide a simple method to add new functionality. These goals will be evaluated here.

7.2.1 Goal 3: Support the addition of User Defined Functions to the workflow system.

One of the main goals for the BDP is to offer the user a large selection of the latest data analysis tools in the form of components. As the data analysis field keeps developing, it is important to be able to add new components, or UDFs, to the BDP. This requires flexibility of both the notation as well as the compiler. This work has shown that both the notation and the compiler are designed in a flexible manner, requiring no changes to either implementation to add UDFs.

As discussed in Section 4.4, it is possible to add UDFs to DAWN. Although there are some risks in adding UDFs for execution, they behave the same as all other components in the notation. Moreover, in the implementation of the BDP there is no difference between UDFs and existing components. Both are implemented using the same approach. Section 5.2.4 discussed the JSON notation for DAWN and showed that the specification remains unchanged, while allowing for addition of new components. The notation also takes into account that components can have multiple different parameters which need to be defined in the workflow specification for the component to operate. This allows the notation to be used for a large number of different component types.

The compiler is decoupled from the component implementation, as discussed in Section 6.5. This approach allows components to be added without requiring changes in the compiler. However, components do need to have a WF activity implementation, which will be used to execute the operation of the components. This does require making changes to the implementation of the BDP for the addition of UDFs.

7.2.2 Goal 4: Design a notation that is portable to different workflow environments.

This goal is important, as advancements in workflow technology might motivate ORTEC to implement a new workflow execution engine. Preferably, ORTEC would want to maintain its existing workflows and move these to a new system with minimal adaptation. This requires the workflow notation to be portable to the new engine. To achieve this, the notation should be independent of the workflow execution engine.

As Chapter 6 shows, the workflow notation has a different workflow perspective than the workflow execution engine. Nevertheless, the notation allows to be compiled to the workflow execution engine. Several workflow systems implement similar strategies to execute dataflows in a control flow engine [9, 1]. The discussion on the compiler has also shown that parsing the workflow notation and compiling to execution engine-specific code is decoupled in the implementation. Therefore, DAWN can be applied to different execution engines, only requiring part of the compiler to be redesigned.

The workflows designed in DAWN do not require modification depending on the execution engine. Given that all information needed to perform the data operations is specified in DAWN, settings which are execution engine-specific would not have to be added to the notation. Rather, the compiler can handle such specifics, sparing the user this effort.

As shown in Subsection 7.1.2, the workflow notation allows for multiple visualisation methods as well. This shows that a change in user interface could also be feasible with minimal changes to the notation. An aspect that needs to be taken into account is that additions might be required in the notation to specify visual positions of component.

7.3 Optimisable

The final part of the goals discusses optimisations. To keep costs low and make data processing fast, optimisations can be applied at several levels in a workflow system. These optimisations improve the performance by lowering cost or improving processing speed. A distinction can be made between logical optimisations and physical optimisations. Logical optimisations pertain to the workflow specification and the ordering of its components. Physical optimisations pertain to the execution plan, or optimising the implementation of the execution engine. These two goals will be evaluated here.

7.3.1 Goal 5: Apply logical optimisations in the form of rewrite rules for NRC-like languages.

Rewrite rules applied to the workflow language help improve the performance of the workflow system by reducing the size of intermediate data and therefore the cost of processing this data. The goal to apply rewrite rules was not achieved in this implementation, due to time constraints it was not possible to implement and test an optimiser for the workflows. However, this evaluation will show that the workflow notation designed here does allow for these rules to be applied.

To be able to apply known rewrite rules to the notation, it must be shown that a mapping can be made between the notation and NRC. As shown in Chapter 4, DAWN is an adaptation of sNRC and can, in general terms, be mapped to sNRC. As sNRC is a dialect of NRC Hidders argues in Appendix A that rewrite rules can be applied to sNRC as well.

Section 4.3 discussed the differences between DAWN and sNRC. In this section it was also discussed how these differences can be translated to allow DAWN to be mapped on sNRC. This can be used to show that it is possible to apply rewrite rules for NRC on the workflow notation. DAWN workflows can be mapped to the notation presented by Hidders. This enables the DAWN workflows to be represented as sNRC expressions. These expressions are a dialect of NRC and therefore, some of the rewrite rules applicable to NRC can apply to this dialect. Therefore these rewrite rules can apply to DAWN. As an example the equivalence in Equation 7.1 will be applied to a DAWN workflow.

$$set(e_1 \uplus e_2) \equiv set(set(e_1) \uplus set(e_2)) \tag{7.1}$$

This rule states that performing a set operation on the union of two expressions is equivalent to performing set operations on the expressions separately and then performing the set operation again on the union of both resulting sets. As the *set* operator removes duplicates from a collection, it can be easily seen that these expressions are equivalent.

In this case, a cost reduction could be achieved if performing the set operation before the union of the expressions would reduce the size of the intermediate data. However, this can only be the case if the cost of performing the set operation is less than the cost of processing excess data. For this example, the assumption will be made that this is the case

The equivalence presented in Equation 7.1 can be applied to DAWN as well, as is shown in the following figures. Figure 7.1 and Figure 7.2 show both sides of the equivalence respectively. Recall that two incoming edges on an input port constitute an additive union. This indicates that this rewrite rule can be applied to a DAWN workflow. Further research is needed to confirm that more rewrite rules will be applicable.

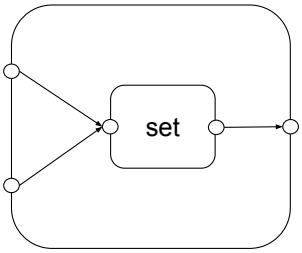

Figure 7.1: **The left side of Equation 7.1.**

≡

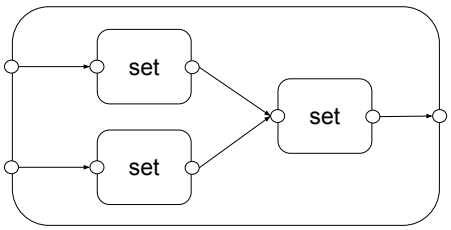

Figure 7.2: **The right side of Equation 7.1.**

7.3.2 Goal 6: Apply physical optimisations in the compiler or execution engine.

Optimisations in the workflow execution can help improve the performance of a workflow without the need for changes to the workflow itself. These optimisations are therefore related to the interpretation of the workflow notation, which in this work takes place in the compiler, as discussed in Chapter 6. This work has implemented a preliminary compiler that translates the specification in DAWN to a specification that the workflow execution engine can interpret. This compiler does not perform any physical optimisations. However these can be added in a simple manner, as will be shown in this evaluation.

Parallelisation

An important step that can be taken to improve performance is parallelisation. Parallelisation can be applied at many levels in the workflow execution: on a data processing level by splitting the data into parts, as well as at a workflow level by parallelising the execution of components. The first can be done in the workflow execution engine, which in this case is a given and will therefore not be considered. The latter is of interest for the compiler, as will be discussed here. From the graph notation used in DAWN, it is clear that components that have no data dependencies between them can be executed in parallel.

Formally this can be defined as follows: if there is no directed path in either direction between two components, they can be executed in parallel. The current preliminary implementation of the compiler only supports sequential processing of the workflow, it should be altered to support parallelisation. Several algorithms exist that can recover this information from a graph structure, requiring only minor changes to the compiler. Modifications to the notation are not required to achieve this. Therefore, it is possible to parallelise work in the workflow with the information given by the DAWN notation.

Data Locality

Another example of optimisation that can be achieved in the compiler is data locality. Data locality in this context describes the location of data in the BDP with respect to the computational work performed in a component. In Big Data applications, data locality is an important concept. Recalling Subsection 2.3.1, there is a large volume of data in these applications. Moving large volumes of data between machines is therefore an expensive operation. As DAWN allows to add external services in the workflow, data must be transferred to and from these services. As discussed in Subsection 6.3.1, the applied approach in the BDP can be expensive if a large number of data transfers are required.

The following example illustrates this. An external service is used as a UDF. This service needs to receive input data and return output data. In a workflow there are multiple components that perform work on this external system. The first component processes part of the workflow input and the second component process the result of the first, plus another dataset from the workflow. In this case, moving the result of the first component back to the BDP only to transfer it back would be an expensive and

unnecessary operation. The compiler can recognise this and alter the data addressing information, as specified in Section 6.3, to have the first results remain on the storage of the external service. Only the second input then needs to be transferred, reducing the cost of data transfers.

For this, the compiler needs to know which components require data transfers to be able to correctly perform the data addressing. The workflow notation supplies the information on the components and the data dependencies between them. As the execution location of a component is known, the compiler only needs to combine this information with the information on the location of the next component. Concluding, if the components are directly connected, the compiler can point the second component directly to the data on the storage of the external service. Thus, reducing the cost of data transfers. However, this implementation would require several changes to the resource API as discussed in Subsection 6.3.1.

Chapter 8

Future Work

The design research presented in this work delivers a graphical workflow notation, its serialisation and compiler for workflows, to be used in the BDP. As shown in the evaluation in Chapter 7, the design goals were partially met. However, there is room for improvement and further research to provide a fully functional portal for data analysis workflows. This chapter will describe future work in both development as well as research. Most of these topics have been discussed in the previous sections, they are reiterated here for completeness.

8.1 Graphical Editor

As discussed in Goal 2, the workflow notation is designed with a graphical editor in mind. In Subsection 7.1.2 it was shown that the notation presented here enables the use of a graphical workflow designer. However, this designer has not been implemented at this moment. To further support the use of this workflow notation to design workflows, this graphical designer should be developed.

Designing a good graphical interface for editing workflow graphs is not only a development challenge. It poses a number of new design challenges on how to display the information related to the workflow. Therefore this addition to the workflow notation is an interesting extension to the work presented in this thesis and provides ground for new work to be added.

8.2 Data Provenance

Another interesting feature that can be added to the workflow designer and the BDP is provenance. Provenance describes the origin of a data element. In data analysis it is applied to show how a specific result was achieved. This feature can help data analysts design their workflows and analyse the performance of the workflow itself. Moreover, enabling users to view the intermediate data as it is being processed in a workflow can greatly improve the debugging capabilities of the portal. The analyst would then be able to more easily spot inconsistencies in the workflow and correct them using the provenance information.

Adding this feature to the BDP poses interesting new challenges for research and development. These challenges include the design of a workflow debugging interface,

and a provenance storage solution. Provenance information adds large amounts of metadata to a dataset and therefore requires an efficient storage approach.

8.3 Different Execution Engines

Goal 4 indicates that the notation should be independent of its execution engine. This was shown in both the implementation of the compiler, in Chapter 6, and the evaluation in Subsection 7.2.2. Although these discussions show that the notation has the potential to be used in combination with different execution engines, there is at this time no implementation of different execution engines for DAWN.

Interesting challenges would be to integrate with workflow systems that have a data oriented approach or apply other dataflow concepts. For example, running DAWN workflows on a MapReduce implementation would show both the flexibility of the notation as well as an interesting new platform to run the workflows on. Such an implementation poses several interesting challenges, such as the implementation of UDFs. The workflow execution engine employed at this moment focusses strongly on implementing such custom functions. Other platforms might make it more difficult to support this.

8.4 Optimisations

Optimisations are an important part of improving the workflow execution performance. Several optimisations have been suggested in Section 7.3. Moreover, it was shown that such optimisations would be feasible in the current workflow notation and compiler. However, the implementation of the compiler does not incorporate these optimisation at this moment. The parallelisation of component execution could be added to the workflow compiler in a simple manner, resulting in an immediate improvement in workflow performance. Similarly, data locality as described in Section 7.3.2 can be added to the compiler for this same purpose.

Further research into rewrite rules could result in a workflow optimiser. Such an optimiser can then be applied in the compiler, optimising the order of the components before compiling them. An interesting challenge here is to see how such an optimiser could handle UDFs.

8.5 Workflow and Component Store

The current implementation of the BDP does consider the storage of components in a database. However, this is currently not provided in a user friendly manner to the users of the BDP. Moreover, the storage of template workflows is not considered in the current implementation. Adding such features would improve the usability of the portal tremendously, as users can select predefined workflows to get started designing their own workflows. This feature would not only require a storage solution, a good search feature is important as well. The user could search for key words related to their proposed task and get an overview of related workflows or components. A good search system and storage system would therefore be interesting future improvements to the usability of the portal.

8.6 Expressiveness of the DAWN Type System

A final challenge lies in elaborating on the DAWN type system. This type system, as discussed in Subsection 4.2.2, was shown to be suitable for application in the BDP and useful for verifying the validity of a DAWN workflow. However, this work considers only a small set of possible data structures to be added. It is uncertain if the DAWN type system is able to describe all possible data structures that might need to be considered. Moreover, the DAWN type system is not designed for recursively nested data types. An example of such a type would be a type T_a that consists of some elements of type T_a. This type recursively nests itself and its depth is therefore unknown. The expressiveness of the DAWN type system as defined in this work, might therefore proof limited for application on certain data structures. Further research is needed to fully grasp the expressiveness of the DAWN type system and show the data structures that can be dealt with and, more importantly, those that cannot.

Chapter 9

Conclusions

The work set out in this thesis is a design research to develop a user friendly workflow system to analyse Big Data. The BDP that ORTEC wishes to develop will implement this workflow system in an interface to edit the workflows and in an execution engine to process the data analytical work. The main goal for this thesis is therefore to develop a workflow interface to allow users to design workflows and provide these users with the tools needed to perform their data analysis.

The artefacts presented in this work contribute to this goal. A workflow model is designed, based on NRC, with usability for data analysts in mind. This model is defined in a notation that enables the design of workflows. This notation was shown to be easily visualisable, allowing for a graphical editor to be developed for it. A graphical editor is not designed yet and remains as future work. Another artefact delivered in this design is a compiler that allows to execute a workflow definition on an execution engine. Although its implementation is elementary, it is a useful base for improvements towards a better compiler.

Concluding, the design presented here has shown to fit the goals set by the BDP and the scientific theory. However, not all elements have been implemented at this time due to time constraints. These elements and improvements to the designed artefacts remain as future work. DAWN has been shown to deliver a user friendly, flexible, and useful workflow notation in the context of Big Data analysis in the cloud.

Glossary

API : Application Programming Interface

BDP : Big Data Portal

BPEL : Business Process Execution Language

CSV : Comma Separated Values

JSON : JavaScript Object Notation

NRA : Nested Relational Algebra

NRC : Nested Relational Calculus

SaaS : Software as a Service

sNRC : NRC for semi-structured data

UDF : User Defined Function

WF : Windows Workflow Foundation

XML : Extensible Markup Language

XAML : Extensible Application Markup Language

YAWL : Yet Another Workflow Language

Bibliography

[1] Ilkay Altintas, Chad Berkley, and Efrat Jaeger. Kepler: an extensible system for design and execution of scientific workflows. *Proceedings. 16th Int. Conf. Sci. Stat. Database Manag.*, 2004.

[2] Marcos D. Assunção, Rodrigo N. Calheiros, Silvia Bianchi, Marco a.S. Netto, and Rajkumar Buyya. Big Data computing and clouds: Trends and future directions. *J. Parallel Distrib. Comput.*, 79-80:3–15, 2014.

[3] Adam Barker and Jano Van Hemert. Scientific Workflow: A Survey and Research Directions. *Proc. 7th Int. Conf. Parallel Process. Appl. Math.*, pages 746–753, 2008.

[4] Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34(4):1–48, 2009.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):1–13, 2008.

[7] Jonas Dias, Eduardo Ogasawara, Daniel De Oliveira, Fabio Porto, Patrick Valduriez, and Marta Mattoso. Algebraic dataflows for big data analysis. *Proc. - 2013 IEEE Int. Conf. Big Data, Big Data 2013*, pages 150–155, 2013.

[8] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[9] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: The Pig Experience. *Proc. VLDB Endow.*, 2(2):1414–1425, August 2009.

[10] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Q.*, 28(1):75–105, 2004.

[11] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. DFL: A dataflow language based on Petri nets and nested relational calculus. *Inf. Syst.*, 33(3):261–284, May 2008.

[12] P Hitzler and K Janowicz. Linked Data, Big Data, and the 4th Paradigm. *Semant. Web J.*, 0(0):233–235, 2013.

[13] Christoph Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. V, 2005.

[14] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward a. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurr. Comput. Pract. Exp.*, 18(10):1039–1065, August 2006.

[15] McKinsey & Company. Big data: The next frontier for innovation, competition, and productivity. *McKinsey Glob. Inst.*, (June):156, 2011.

[16] Tadao Murata. Petri Nets : Properties , Analysis and Applications. *Proc. IEEE*, 77(4):541–580, 1989.

[17] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic Optimization of Parallel Dataflow Programs. *Int. J. Parallel Program.*, 31(6):429–449, December 2003.

[18] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. *Proc. 2008 ACM SIGMOD Int. Conf. Manag. data - SIGMOD '08*, pages 1099–1110, 2008.

[19] Mark a. Roth, Herry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases, 1988.

[20] Jacek Sroka, Piotr Wlodarczyk, Lukasz Krupa, and Jan Hidders. DFL designer. In *Proc. 1st Int. Work. Work. Approaches to New Data-centric Sci. - Wands '10*, pages 1–6, New York, New York, USA, 2010. ACM Press.

[21] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. The Triana Workflow Environment : Architecture and Applications. In *Work. e-Science*, pages 320–339. Springer London, 2007.

[22] Ian J. Taylor, Ewa Deelman, Dennis Gannon, and Matthew S. Shields. Workflows for e-Science: Scientific Workflows for Grids. *Work. e-Science Sci. Work. Grids*, pages 1–523, 2007.

[23] Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. A crash course on database queries. In *Proc. twenty-sixth ACM SIGMOD-SIGACT-SIGART Symp. Princ. database Syst. - Pod. '07*, pages 143 – 154, New York, New York, USA, 2007. ACM Press.

[24] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, June 2005.

[25] Tom White. *Hadoop : The Definitive Guide*. O'Reilly, first edition, 2009.

[26] Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, 1994.

Appendix A

NRC for semi-structured data

NB: *The work presented in this appendix was performed by Jan Hidders. It presents several concepts and definitions applied in this thesis. However, this work is not published at this moment. Therefore, part of it is included in this appendix.*

A.1 Goal of document

This is a working document that describes our investigations into the design of a data analytics workflow language. The languages are based on well-known formalisms such as NRC and formal versions of XQuery. The intent is to design a language that (1) is well-understood in terms of semantics and expressive power, (2) can be efficiently implemented specifically on back-ends offering large scale parallelised processing and (3) allows the application of well-known optimisation techniques such as cost-based query rewriting. Moreover, the language comes in different textual and graphical notations that can be used for both theoretical analysis and workflow-like graphical presentations of the analytical workflows.

A.2 The Underlying Data Model: Nested Values

To define the data model on which we will operate, we postulate the following sets and basic concepts:

- C denotes the set of basic value constant denotations (like booleans, strings, integers, etc.), $\mathcal{B}$ denotes the set of basic values, we distinguish one special basic value constant denoted as $\langle\rangle$
- bags / multisets are denoted as $\{\{1,1,2\}\}$, the additive bag union is denoted as $\uplus$, we use $\{\{f(\bar{x}) \mid \varphi(\bar{x})\}\}$ to denote bag comprehension. The empty bag is denoted as $\emptyset$.
- ordered pairs containing values x and y are denoted as $\langle x, y\rangle$
- $\mathcal{V}$ denotes the set of *nested values*, which are bags of items, where *items* are either (1) a basic value in $\mathcal{B}$, (2) an ordered pair $\langle v_1, v_2\rangle$ where $v_1, v_2 \in \mathcal{V}$. Note that we do not allow bags of bags, and that tuples can contain only bags. We

explicitly allow heterogeneous bags, i.e., bags that contain elements of different types. We will allow only finitely nested values, i.e., we assume $\mathcal{V}$ is the smallest set that satisfies this definition.

- The ordered pair $\langle x, y\rangle$ can also be understood as a key-value pair, rather then a small tuple. So JSON objects can be represented as bags of such pairs. That captures the idea that field names are first class citizens in the language.
- It may seem odd that we allow only bags in ordered pairs, but it is a consequence of the "all expressions return a bag" principle that is followed in this language and simplifies its semantics and allows us to ignore typing.
- As a shorthand we will let $\langle\rangle$ denote the pair $\langle\emptyset,\emptyset\rangle$, and $\langle v\rangle$ the value $\langle\emptyset, v\rangle$.

A.3 NRC for semistructured data: sNRC

We give here a formal definition of the dialect of NRC we will study here.

A.3.1 Preliminary notions

We postulate the following sets and basic concepts:

- X denotes the set of variable names
- B denoting the set of basic (user-defined) functions, with each $b \in B$ we associated a binary relation $[\![b]\!]$ that associates nested values with nested values

A.3.2 The syntax of sNRC

The syntax for the calculus over bags we intend to use:

$$\begin{aligned} E ::= {} & \mathbf{in} \mid X \mid C \mid \langle E, E\rangle \mid E.1 \mid E.2 \mid \\ & \emptyset \mid E \uplus E \mid \{\![E \mid X \in E, \ldots, X \in E]\!\} \mid \\ & B(E) \mid \dot{\mathbf{set}}(E) \mid E \doteq E. \end{aligned}$$

Here **in** denotes the input value, X denotes variables, C basic value constants, $\{\![e \mid \Delta]\!\}$ denotes the flattening bag comprehension (i.e., it is a comprehension which additionally flattens the result to avoid bags of bags), B the basic user-defined functions. The function $\dot{\mathbf{set}}()$ eliminates duplicates and non-basic values. The expression $e_1 \doteq e_2$ compares basic values and returns a bag containing as many occurrences of $\langle\rangle$ as there are pairs of occurrences in e_1 and e_2, respectively, that represent the same basic value. E.g., comparing the value $\{\!\{1,2,2,3,3,4\}\!\}$ with $\{\!\{2,2,3\}\!\}$ using the $\doteq$ operator results in $\{\!\{\langle\rangle,\langle\rangle,\langle\rangle,\langle\rangle,\langle\rangle,\langle\rangle\}\!\}$.

We allow for denotations of values the same short-hands as for the values themselves: $\langle\rangle$ denotes the value $\langle\rangle = \langle\emptyset,\emptyset\rangle$ and $\langle e\rangle$ denotes $\langle e\rangle = \langle\emptyset, e\rangle$. We let $\{\![e \mid]\!\}$ also be simply denoted as $\{\![e]\!\}$. We will use in the right-hand side of comprehensions the short-hand $(e_1 \doteq e_2) \triangleq z \in (e_1 \doteq e_2)$ with z some fresh variable.

The $\dot{\mathbf{set}}()$ operator may seem weak, but does allow us to express duplicate elimination as it happens in the relational model. For example, if R_1 contains a bag of

pairs containing singleton basic values then the corresponding set can be expressed as $\{[\langle x,y\rangle \mid x \in \dot{\mathbf{set}}(r.1), y \in \dot{\mathbf{set}}(\{[z.2 \mid z \in r, z.1 \doteq x]\})]\}$. Note that these expressions can also be used to simulate reasoning in settings where the input contains no duplicates and only singleton fields, by replacing the input relation R_1 with this expression. If this expression is called r' we can determine if $f(r) \equiv g(r)$ for such R_1 by determining if $f(r') \equiv f(r')$.

A.3.3 The semantics of sNRC

To ensure the definedness of the result of each expression in a semi-structured and possibly untyped setting we will assume that all values, both inputs and outputs, are bags. This is similar to the approach taken in XQuery. In fact the language is similar to *XQuery core* as studied in terms of expressive power and evaluation complexity in [13] and [4]. Note that this means that the expression 12 in fact denotes the bag $\{\{12\}\}$ rather then the number 12. The rule of thumb for operators that normally do not expect a bag is that they are mapped over the elements of the bag. So e.1 in fact constructs a bag by iterating over each element from the result of e and for each pair returning the first element. Also as in XQuery, the comprehension automatically flattens the result to avoid the construction of bags directly nested inside bags. So, for example $\{[\{[5]\} \mid x \in e]\}$ is equivalent to $\{[5 \mid x \in e]\}$ which returns a bag containing only the number 5 and is of the size of the result of e. Indeed $\{\{e\}\}$ is always equivalent to e. Consequently the expressions $\{[1]\}$ and 1 both denote the value $\{\{1\}\}$, and the expressions $1 \uplus 2$ and $\{[1]\} \uplus \{[2]\}$ both denote $\{\{1,2\}\}$.

The semantics is defined in terms of propositions of the form $\Gamma \vdash e \Rightarrow v$ where Γ is variable binding, i.e., a function that maps variable names to *items*, e an NRC expression and v a bag of nested values that represents the result of the evaluation of e under Γ. Note that Γ maps variable names to items, rather then nested values, i.e., variables are bound to basic values and ordered pairs but not to bags. This is done for the sake of simplicity as in this research we mostly use variables to iterate over the elements of a bag. Moreover, assigning a bag b to a variable x can be simulated by assigning the item $\langle \emptyset, b\rangle$ and everywhere that x occurs freely in the expression replacing it with $x.2$.

$$\frac{}{\Gamma \vdash x \Rightarrow \{\{\Gamma(x)\}\}} \qquad \frac{}{\Gamma \vdash c \Rightarrow \{\{c\}\}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow v_2 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \langle e_1, e_2 \rangle \Rightarrow \{\{\langle v_1, v_2 \rangle\}\}}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash e.i \Rightarrow \{\{u \mid \langle w_1, w_2 \rangle \in v, u \in w_i\}\}} \qquad \frac{}{\Gamma \vdash \emptyset \Rightarrow \emptyset} \qquad \frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow w}{\Gamma \vdash e_1 \uplus e_2 \Rightarrow v \uplus w}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash \{[e \mid]\} \Rightarrow v} \qquad \frac{\Gamma \vdash e_2 \Rightarrow \{\{v_1, \ldots, v_m\}\} \qquad \forall_{i=1}^{m} (\Gamma_{[x \mapsto v_i]} \vdash \{[e_1 \mid \Delta]\} \Rightarrow w_i)}{\Gamma \vdash \{[e_1 \mid x \in e_2, \Delta]\} \Rightarrow \uplus_{i=1}^{m} w_i}$$

$$\frac{\Gamma \vdash e \Rightarrow v \qquad (v, w) \in [\![b]\!]}{\Gamma \vdash b(e) \Rightarrow w} \qquad \frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash \dot{\mathbf{set}}(e) \Rightarrow \{x \mid x \in v, x \in \mathcal{B}\}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow w}{\Gamma \vdash e_1 \doteq e_2 \Rightarrow \{\{\langle\rangle \mid c_1 \in v, c_2 \in w, c_1 = c_2, c_1 \in \mathcal{B}\}\}}$$

Note that the iterators in the comprehension iterate over all the elements of a bag. The semantics of an sNRC expression can be interpreted as a total function that maps variable bindings to a nested values, presuming that all user-defined functions are also total functions that map nested values to nested values.

A.4 NRA for semi-structured data: sNRA

We give here a formal definition of the dialect of NRA that is the algebraic counterpart of sNRC.

A.4.1 The syntax of sNRA

As an algebraic counterpart of sNRC we present sNRA. It has the following syntax:

$$F ::= \mathbf{id} \mid {}^{\lambda}C \mid F \circ F \mid {}^{\lambda}\langle F, F \rangle \mid \pi_1 \mid \pi_2 \mid$$
$${}^{\lambda}\emptyset \mid {}^{\lambda}\uplus \mid \mathbf{fmap}(F) \mid {}^{\lambda}\times \mid B \mid \dot{\mathbf{set}} \mid {}^{\lambda}\doteq.$$

Note that we annotate some constructs with λ to indicate they denote functions rather then values.

A.4.2 The semantics of sNRA

The semantics of the algebra is defined by the following rules. They define the proposition $(x, y) \in [\![e]\!]$ which denotes that the function associated with e maps the value x to the value y.

$$\frac{x \in \mathcal{V}}{(x,x) \in [\![\mathbf{id}]\!]} \qquad \frac{x \in \mathcal{V}}{(x,c) \in [\![{}^{\lambda}c]\!]} \qquad \frac{(x,y) \in [\![f]\!] \qquad (y,x) \in [\![g]\!]}{(x,z) \in [\![g \circ f]\!]}$$

$$\frac{(x,y) \in [\![f]\!] \qquad (x,z) \in [\![g]\!]}{(x,\{\!\{\langle y,z\rangle\}\!\}) \in [\![{}^{\lambda}\langle f,g\rangle]\!]} \qquad \frac{x \in \mathcal{V} \qquad y = \{\!\{v_i \mid \langle v_1,v_2\rangle \in x\}\!\}}{(x,y) \in [\![\pi_i]\!]} \qquad \frac{x \in \mathcal{V}}{(x,\emptyset) \in [\![{}^{\lambda}\emptyset]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{z \mid \langle u,v\rangle \in x, z \in (u \uplus v)\}\!\}}{(x,y) \in [\![{}^{\lambda}\uplus]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{v \mid z \in x, (\{\!\{z\}\!\},u) \in [\![f]\!], v \in u\}\!\}}{(x,y) \in [\![\mathbf{fmap}(f)]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{\langle \{\!\{s\}\!\}, \{\!\{t\}\!\}\rangle \mid \langle u,v\rangle \in x, s \in u, t \in v\}\!\}}{(x,y) \in [\![{}^{\lambda}\times]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{z \mid z \in x, z \in \mathcal{B}\}}{(x,y) \in [\![\dot{\mathbf{set}}]\!]}$$

$$\frac{x \in \mathcal{V} \qquad y = \{\!\{\langle\rangle \mid \langle u,v\rangle \in x, s \in u, t \in v, s = t, s \in \mathcal{B}\}\!\}}{(x,y) \in [\![{}^{\lambda}\doteq]\!]}$$

Note that no rule is specified for basic functions in B since their semantics was already postulated.

A.5 The relationship between sNRA and sNRC

There is a direct relationship between NRA and NRC in expressive power. To illustrate this we show that they can be mapped to each other.

A.5.1 Mapping sNRA to sNRC

Each sNRA expression can be represented by an sNRC expression with a single special free variable **in** that represents the input value, and vice versa. We let $e_{[x/e']}$ denote the expression e with all free occurrences of x replaced with e'.

$M(\mathbf{id}) = \mathbf{in}$
$M({}^{\lambda}c) = c$
$M(g \circ f) = \{[M(g)_{[\mathbf{in}/x]} \mid x \in M(f)]\}$
$M({}^{\lambda}\langle f,g\rangle) = \langle M(f), M(g)\rangle$
$M(\pi_i) = \mathbf{in}.i$
$M({}^{\lambda}\emptyset) = \emptyset$
$M({}^{\lambda}\uplus) = \mathbf{in}.1 \uplus \mathbf{in}.2$
$M(\mathbf{fmap}(f)) = \{[M(f)_{[\mathbf{in}/x]} \mid x \in \mathbf{in}]\}$
$M({}^{\lambda}\times) = \{[\langle y,z\rangle \mid x \in \mathbf{in}, y \in x.1, z \in x.2]\}$
$M(b) = b(\mathbf{in})$

$M(\dot{\mathbf{set}}) = \dot{\mathbf{set}}(\mathbf{in})$
$M(^{\lambda}\doteq) = (\mathbf{in}.1 \doteq \mathbf{in}.2)$

A.5.2 Mapping sNRC to sNRA

We show that each sNRC expression with a single free variable **in** can be represented by an sNRA expression:

$M'(\mathbf{in}) = \mathbf{id}$
$M'(c) = {}^{\lambda}c$
$M'(\langle e_1, e_2\rangle) = {}^{\lambda}\langle M'(e_1), M'(e_2)\rangle$
$M'(e.i) = \pi_i \circ M'(e)$
$M'(\emptyset) = {}^{\lambda}\emptyset$
$M'(e_1 \uplus e_2) = {}^{\lambda}\uplus \circ {}^{\lambda}\langle M'(e_1), M'(e_2)\rangle$
$M'(\{[e\,|]\}) = M'(e)$
$M'(\{[e \mid x \in e']\}) = \mathbf{fmap}(M'(e_{[\mathbf{in}/\mathbf{in}.1, x/\mathbf{in}.2]})) \circ {}^{\lambda}\times \circ {}^{\lambda}\langle \mathbf{id}, M'(e')\rangle$
$M'(\{[e \mid x \in e', \Delta]\}) = M'(\{[\{[e \mid \Delta]\} \mid x \in e']\})$
$M'(b(e)) = b \circ M'(e)$
$M'(\dot{\mathbf{set}}(e)) = \dot{\mathbf{set}} \circ M'(e)$
$M'(e_1 \doteq e_2) = {}^{\lambda}\doteq \circ {}^{\lambda}\langle M'(e_1), M'(e_2)\rangle$

A.6 A note on *n*-ary functions

In the previous setting we used sNRA and sNRC to define unary functions, i.e., functions with one input parameter, but we easily interpret these functions also as n-ary functions for some n as follows: if we interpret $f : \mathcal{V} \to \mathcal{V}$ as n-ary then we get the function $f^{[n]} : \mathcal{V}^n \to \mathcal{V}$ such that $f^{[n]}(v_1, \ldots, v_n) = f(\{\{\langle v_1, \{\{\langle v_2, \ldots \{\{\langle v_n, \emptyset\rangle\}\} \ldots\rangle\}\}\rangle\}\})$. And, vice versa, if we take an n-ary function $f^{[n]}$ then we can interpret it as a unary function f which is defined such that $f(v) = f^{[n]}(v.1, v.2.1, ..., v(.2)^{n-1}.1)$.

It follows that we can view sNRC and sNRA also as definition languages for n-ary functions. In that case the notion of semantical equivalence changes somewhat since the n-ary interpretation only considers some parts of the input. However, we can define it in terms of the original notion of semantical equivalence. For that we introduce a special short-hands in sNRC denoted as $\mathbf{in}^{[n]}$, which is defined by induction on n such that (1) $\mathbf{in}^{[0]} = \emptyset$ and (2) $\mathbf{in}^{[n+1]} = \langle \mathbf{in}.1, (\mathbf{in}^{[n]})_{[\mathbf{in}/\mathbf{in}.2]}\rangle$. For example, $\mathbf{in}^{[3]} = \langle \mathbf{in}.1, \langle \mathbf{in}.2.1, \langle \mathbf{in}.2.2.1, \emptyset\rangle\rangle\rangle$. Informally this function interprets the input as a tuple of n elements and projects on those. We now say that an sNRC expression e is an *n-ary expressions* if it holds that $e_{[\mathbf{in}/\mathbf{in}^{[n]}]} \equiv e$.

Likewise For sNRA we can define the corresponding projection function $\mathbf{id}^{[n]}$ which is defined by induction on n such that (1) $\mathbf{id}^{[0]} = {}^{\lambda}\emptyset$ and (2) $\mathbf{id}^{[n+1]} = {}^{\lambda}\langle \pi_1, \mathbf{id}^{[n]} \circ \pi_2\rangle$. For example, $\mathbf{id}^{[3]} = {}^{\lambda}\langle \pi_1, {}^{\lambda}\langle \pi_1, {}^{\lambda}\langle \pi_1, {}^{\lambda}\emptyset\rangle \circ \pi_2\rangle \circ \pi_2\rangle$. Then, for an sNRA expression f we say it is an *n-ary function* if it holds that $f \circ \mathbf{id}^{[n]} \equiv f$.

When defining n-ary functions in sNRC we will adopt the convention to let $\mathbf{in}_i$ with $i \leq n$ denote the i'th input value. This is essentially a syntactic short-hand for $\mathbf{in}(.2)^{i-1}.1$. We will from now on assume that all user-define functions are n-ary, and the notation $b(e_1, \ldots, e_n)$ will be used to denote the passing of n parameters to function b.

Theorem A.6.1 (*n*-ary functions in sNRC). For every *n*-ary function in sNRC there is an equivalent sNRC function that uses no **in** but only $\mathbf{in}_i$ for $i \leq n$.

A.7 A graphical notation for sNRC/sNRA

We introduce a graphical notation called DAWN to represent the *n*-ary functions that can be defined by sNRC and sNRA. The general notation is the usual workflow style as is shown for example in Figure A.1. Every workflow has zero or more input ports, and exactly one output port. If a port has multiple outgoing edges it means the output is copied for each output edge. If an input port has multiple incoming edges, it means all the inputs are combined with an additive bag union. So the input of h is the bag union of the output of f and g. Every output port needs to have at least one outgoing edge, i.e., the result of every component must be used somewhere. The input ports of the whole workflow have zero or more outgoing edges. So not all inputs must be necessarily used. The output port of the whole workflow must have at least one incoming edge, or it will not have a defined value. The workflow must be acyclic, i.e., if we consider each component as a single node, then the connecting edges do not form a directed cycle.

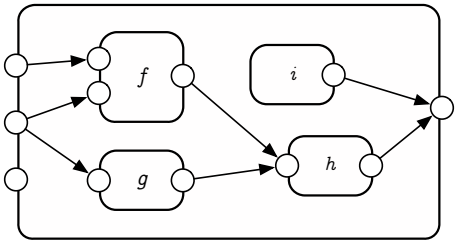

Figure A.1: **General workflow notation**

We will define the semantics of workflows in terms of sNRC expressions denoting *n*-ary functions, and so do not use **in** but do use $\mathbf{in}_i$. Consider the example in Figure A.1. Assuming that we have sNRC expressions e_f, e_g, e_h and e_i for the corresponding components, then the semantics of the whole workflow is given by $\{[(x_i.2 \uplus x_h.2) \mid x_f \in \langle \emptyset, e_f(\mathbf{in}_1, \mathbf{in}_2)\rangle, x_g \in \langle \emptyset, e_g(\mathbf{in}_2)\rangle, x_i \in \langle \emptyset, \langle \emptyset, e_i()\rangle, x_h \in \langle \emptyset, e_h(x_f.2 \uplus x_g.2)\rangle\rangle]\}$. Here expressions of the form $e(e_1, \ldots, e_2)$ are a short-hand for $e_{[\mathbf{in}_1/e_1, \ldots, \mathbf{in}_n/e_n]}$. Note that if we do not wrap the results of the components in a pair, they will be iterated over rather then assigned as a whole to the iterating variable.

We of course allow the workflows to be recursively nested, so a component in the workflow can itself be again a complex workflows.

A special feature is that input ports of a component can be marked as *iterating*, which is indicated by a star, as is shown in Figure A.2. The meaning is that for these ports the incoming bags are iterated over, i.e., the function is applied to each element of the bag (wrapped as a singleton bag). If multiple ports are marked, then all combinations of the elements are taken. So in the figure the computed function is $\{[e_f(x, y, \mathbf{in}_3) \mid x \in \mathbf{in}_1, y \in \mathbf{in}_2]\}$.

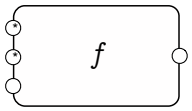

Figure A.2: **Iterating input ports**

The primitive components we need to represent the functions in sNRC and sNRA are shown in Figure A.3. They consist of components for: (1) constants c, (2) the pair constructor, (3) the projection operators, (4) the empty bag, (5) the basic value equality operator, (6) user-defined functions and (7) the set operator that returns a set of basic values.

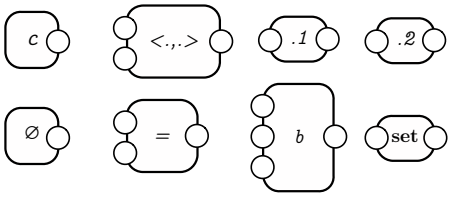

Figure A.3: **Primitive workflow components**

From the previous the following will be clear.

Theorem A.7.1. For every DAWN workflow with n input ports there is an equivalent n-ary sNRC expression.

The converse also holds.

Theorem A.7.2. For every n-ary sNRC expression there is an equivalent DAWN workflow with n input ports.

The mapping is illustrated in Figure A.4.

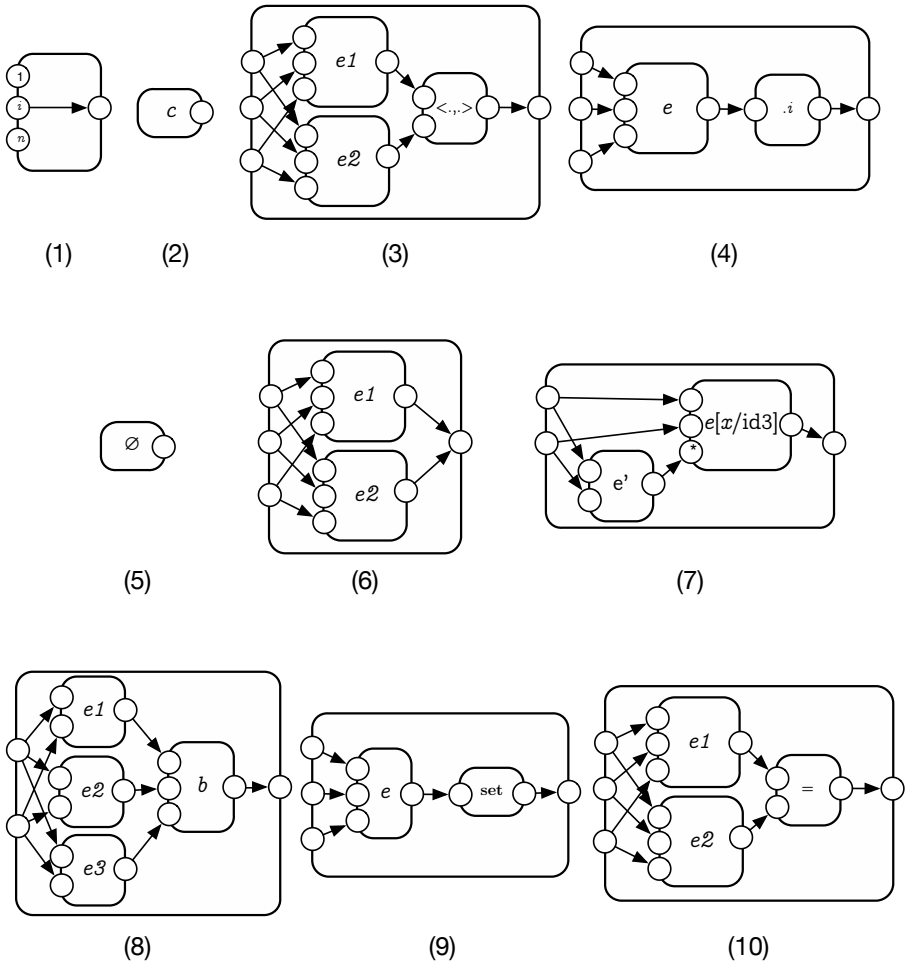

Figure A.4: **Mapping *n*-ary sNRC expressions to DAWN workflows**

Appendix B

Example JSON workflow notation

Code Fragment B.1: Example workflow definition in JSON

```
{
  "dawn": {
    "version": 0.9,
    "resources": [
      {
        "id": 1,
        "name": "csv 1",
        "typeId": "csv",
        "modifiedDate": "20150608",
        "modifiedBy": "Mick",
        "elements": {
          "csvLocation": "..\\input\\",
          "csvFilename": "data.csv",
          "csvHeader": "true",
          "csvDelimiter": ";"
        }
      },
      {
        "id": 2,
        "name": "csv 2",
        "typeId": "csv",
        "modifiedDate": "20150608",
        "modifiedBy": "Mick",
        "elements": {
          "csvLocation": "..\\output\\",
          "csvFilename": "result.csv",
          "csvHeader": "true",
          "csvDelimiter": ";"
        }
      }
    ],
    "workflow": {
```

```
"id": 1,
"name": "R Script test WF",
"description": "Executing an R Script on CSV data.",
"inputs": [
  {
    "id": 3,
    "type": {
      "name": "records",
      "collection": [
        {
          "name": "Name",
          "typeName": "string"
        },
        {
          "name": "Amount",
          "typeName": "float"
        }
      ]
    },
    "resource": 1
  }
],
"outputs": [
  {
    "id": 4,
    "type": {
      "name": "records",
      "collection": [
        {
          "name": "Name",
          "typeName": "string"
        },
        {
          "name": "Amount",
          "typeName": "float"
        }
      ]
    },
    "resource": 2
  }
],
"nodes": [
  {
    "id": 8,
    "name": "run R Script",
    "component": {
      "type": "rScript",
```

```
        "properties": {
          "scriptName": "hello.r"
        }
      },
      "inputs": [
        {
          "type": {
            "name": "records",
            "collection": [
              {
                "name": "Name",
                "typeName": "string"
              },
              {
                "name": "Amount",
                "typeName": "float"
              }
            ]
          },
          "id": 9
        }
      ],
      "outputs": [
        {
          "type": {
            "name": "records",
            "collection": [
              {
                "name": "Name",
                "typeName": "string"
              },
              {
                "name": "Amount",
                "typeName": "float"
              }
            ]
          },
          "id": 10
        }
      ]
    }
  ],
  "edges": [
    {
      "id": 18,
      "start": 3,
      "end": 9
```

```
            },
            {
              "id": 19,
              "start": 10,
              "end": 4
            }
          ]
        }
      }
    }
```

Appendix C

DAWN JSON Schema

Code Fragment C.1: JSON Schema specification for DAWN

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://dawn/schema.json",
  "definitions": {
    "object": {
      "type": "array",
      "minSize": 1,
      "items": {
        "$ref": "#/definitions/type"
      },
      "additionalProperties": false
    },
    "type": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string"
        },
        "typeName": {
          "type": "string",
          "enum": [
            "string",
            "int",
            "bool",
            "float"
          ]
        },
        "collection": {
          "$ref": "#/definitions/type"
        },
        "object": {
          "$ref": "#/definitions/object"
```

```
        }
      },
      "required": [
        "name"
      ],
      "oneOf": [
        {
          "required": [
            "typeName"
          ]
        },
        {
          "required": [
            "object"
          ]
        },
        {
          "required": [
            "collection"
          ]
        }
      ],
      "additionalProperties": false
    }
  },
  "type": "object",
  "properties": {
    "dawn": {
      "type": "object",
      "properties": {
        "version": {
          "type": "number",
          "minimum": 0.9
        },
        "resources": {
          "type": "array",
          "items": {
            "properties": {
              "id": {
                "type": "integer",
                "minimum": 1
              },
              "name": {
                "type": "string"
              },
              "typeId": {
                "type": "string",
```

```
            "enum": [
              "csv",
              "xml",
              "sql"
            ]
          },
          "modifiedDate": {
            "type": "string"
          },
          "modifiedBy": {
            "type": "string"
          },
          "type": {
            "$ref": "#/definitions/type"
          },
          "elements": {
            "type": "object"
          }
        },
        "additionalProperties": false,
        "required": [
          "id",
          "name",
          "typeId",
          "modifiedDate",
          "modifiedBy",
          "elements"
        ]
      }
    },
    "workflow": {
      "type": "object",
      "properties": {
        "id": {
          "type": "integer",
          "minimum": 1
        },
        "name": {
          "type": "string"
        },
        "description": {
          "type": "string"
        },
        "inputs": {
          "type": "array",
          "minItems": 0,
          "items": {
```

```
        "type": "object",
        "properties": {
          "id": {
            "type": "integer",
            "minimum": 1
          },
          "portName": {
            "type": "string"
          },
          "type": {
            "$ref": "#/definitions/type"
          },
          "resource": {
            "type": "integer",
            "minimum": 1
          }
        },
        "additionalProperties": false,
        "required": [
          "id",
          "type"
        ]
      }
    },
    "outputs": {
      "type": "array",
      "minItems": 1,
      "items": {
        "type": "object",
        "properties": {
          "id": {
            "type": "integer",
            "minimum": 1
          },
          "portName": {
            "type": "string"
          },
          "type": {
            "$ref": "#/definitions/type"
          },
          "resource": {
            "type": "integer",
            "minimum": 1
          }
        },
        "additionalProperties": false,
        "required": [
```

```
        "id",
        "type"
      ]
    }
  },
  "nodes": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "id": {
          "type": "integer",
          "minimum": 1
        },
        "name": {
          "type": "string"
        },
        "component": {
          "type": "object",
          "properties": {
            "type": {
              "type": "string"
            },
            "properties": {
              "type": "object"
            }
          },
          "additionalProperties": false,
          "required": [
            "type",
            "properties"
          ]
        },
        "inputs": {
          "type": "array",
          "minItems": 0,
          "items": {
            "type": "object",
            "properties": {
              "id": {
                "type": "integer",
                "minimum": 1
              },
              "type": {
                "$ref": "#/definitions/type"
              }
            },
```

```
                "additionalProperties": false,
                "required": [
                  "id",
                  "type"
                ]
              }
            },
            "outputs": {
              "type": "array",
              "minItems": 1,
              "items": {
                "type": "object",
                "properties": {
                  "id": {
                    "type": "integer",
                    "minimum": 1
                  },
                  "type": {
                    "$ref": "#/definitions/type"
                  }
                },
                "additionalProperties": false,
                "required": [
                  "id",
                  "type"
                ]
              }
            }
          },
          "additionalProperties": false,
          "required": [
            "id",
            "name",
            "component",
            "inputs",
            "outputs"
          ]
        }
      },
      "edges": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "id": {
              "type": "integer",
              "minimum": 1
```

```
                    },
                    "start": {
                      "type": "integer",
                      "minimum": 1
                    },
                    "end": {
                      "type": "integer",
                      "minimum": 1
                    }
                  },
                  "additionalProperties": false,
                  "required": [
                    "id",
                    "start",
                    "end"
                  ]
                }
              }
            },
            "additionalProperties": false,
            "required": [
              "id",
              "name",
              "description",
              "inputs",
              "outputs",
              "nodes",
              "edges"
            ]
          }
        },
        "additionalProperties": false,
        "required": [
          "version",
          "resources",
          "workflow"
        ]
      }
    },
    "additionalProperties": false,
    "required": [
      "dawn"
    ]
}
```