MASTER THESIS

DATA PROCESSING NETWORKS MADE EASY

IMPROVING DEVELOPMENT POSSIBILITIES FOR PEOPLE WITH LIMITED COMPUTER SCIENCE KNOWLEDGE

Christiaan Alexander Nouta

SOFTWARE ENGINEERING GROUP
FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE

GRADUATION COMMITTEE
Dr. Andreas Wombacher
Dr. Hasan Sözer
Rezwan Huq, M.Sc.

UNIVERSITY OF TWENTE.

JANUARY 2011

Data Processing Networks Made Easy

Improving development possibilities for people
with limited computer science knowledge

by

Christiaan Alexander Nouta

born on the 1st of August 1985
in Sneek, The Netherlands

MASTER THESIS

Graduation committee:

Dr. Andreas Wombacher
Dr. Hasan Sözer
Rezwan Huq, M.Sc.

University of Twente

Software Engineering Group
Faculty of EEMCS
PO Box 217
7500 AE Enschede
The Netherlands

January 2011

"The function of good software is to make the complex appear to be simple."
- Grady Booch

Acknowledgements

The writing of this thesis marks the end of my Computer Science Master at the University of Twente. It was a wonderful time with many great moments which I will carry with me for the rest of my life. This project was a great challenge and I would like to thank everybody who made this possible.

I would not have been able to successfully complete this project without the support of my supervisors Andreas Wombacher, Hasan Sözer and Rezwan Huq. I learned a lot from them, and their critical questions and professional comments have put this project, and thesis, to a higher level. I would also like to thank the members of the CCES project RECORD for their support. In particular Paolo Perona for providing me the required photo material. I am inspired by their dedication and enjoyed the trip to their field site.

Due to setbacks in the project and opportunities beside my study, it was sometimes hard to keep focused and motivated. I would like to thank Sharon Vonk for her support during these moments.

I thank Sabine Padberg-Heskamp and Barbara Spikker-Sieverink for their administrative support.

I would like to thank all my college friends for making these years in Enschede unforgettable. In particular, Joost and Matthias, for the endless laughs we had and the nice projects we did. Even though our planning was often far from optimal, I enjoyed every minute. Furthermore, I would like to thank my roommates Maarten, Tim and Tri for the nice time we had together. With them I discovered the fun of cooking and I will never forget some of their recipes.

The past few years where quite busy and therefore I would like to thank my girlfriend, Sybrigje Pietsje, for her endless amount of patience and for being there for me, no matter what. I have had endless love and support from my family throughout my life. I would like to thank my parents, Klaas and Ankie, and my sisters, Ella and Nanda, for the close bond we have.

Abstract

Data processing networks are not only used by computer science people, nowadays researchers in all kinds of research topics are processing large amounts of data. Considering the required knowledge and amount of time, the development of a distributed and/or (de)centralized data processing network is often not an option for them. Sometimes they create scripts for the various processing steps, which are manually executed step-by-step. It is clear that such a procedure is far from optimal, especially in a streaming data environment. Due to the popularity of the internet, distributed data or even decentralized processing is used more often. By introducing decentralization or distributed data, the complexity of the overall processing network is increased dramatically. Both aspects require some overhead which lowers the understandability of the data processing network in general.

We consider the following problems in supporting the development of data processing networks by people with limited computer science knowledge. First, a large group of users does not have a good overall understanding of the key concepts and reusable components of a data processing networks. Second, existing architectural styles are not well-suited for documenting a data processing view. Third, the current tool support is insufficient for people with limited computer science knowledge.

To tackle the first problem, we introduce a basic model for data processing networks. Implementing a data processing network is a costly-process, it is important that the users have a good understanding of the general structure. This basic model can be used to overcome a potential knowledge gap between the users and developers. To create a more detailed (technical) view of a data processing network, we introduce a specialized model.

As a new architectural style, we introduce the data processing style for documenting a data processing view. For documenting a more detailed (technical) view of a data processing network, we introduce the DatProNet style. By creating a specialized data processing view, the understandability of the general structure is increased.

As a solution for the third problem, we propose the DatProNet framework that supports the development of data processing networks by people with limited computer science knowledge. The complete development life cycle of data processing networks is covered by the DatProNet framework, which reduces the development and maintenance effort and increases the ease of use. XML-editors can be used for the creation, and validation, of architectural descriptions. Based

on a valid architectural description, the framework can generate a concrete implementation. The DatProNet framework provides a set of reusable components, which can be used to extend the framework if needed. Communication with external systems is possible through the use of serializers. The framework is completely implemented in JAVA, which implies a high portability.

Contents

Chapter 1

Introduction

Data processing can be defined as the concept of applying a series of operations on data in order to fulfil a certain task [52]. A *network* can be defined as a group of computers or (software) components connected through communication channels [45]. Based on these two definitions, a *data processing network* can be defined as a network of computers or (software) components which apply a series of operations on data in order to fulfil a certain task. Data processing networks can be *centralized* or *decentralized*. Moreover, the data to process can be distributed among various computers. The development of a distributed and/or decentralized data processing network can be a complex task and requires a considerable amount of time. The concept of distributed data and/or decentralization introduces some overhead to the processing algorithm which increases the complexity and lowers the overall *understandability*.

Data processing networks are not only used by computer science people, nowadays researchers in all kinds of research topics are processing large amounts of data. Considering the required knowledge and amount of time, the development of a distributed and/or (de)centralized data processing network is often not an option for them. Sometimes they create scripts for the various processing steps, which are manually executed step-by-step. It is clear that such a procedure is far from optimal, especially in a streaming data environment. In this thesis, various methods and techniques are introduced to support the development of data processing networks by *people with limited computer science knowledge*.

1.1 Thesis Scope

The work presented in this thesis has been carried out to support the CCES project RECORD [10]. The main objective of this project is to investigate the effects of river restoration on the surrounding environment in hydrological, biogeochemical and ecological terms [10]. The activities of the CCES project RECORD [10] are focused on the Swiss river Thur. The river is monitored by various devices, including two digital cameras. The researchers are now interested to see how computer vision algorithms can be used to support the calibra-

tion of their environmental models. For this calibration they need a segmented version of the pictures taken by the cameras. A picture needs to be segmented using two classes: water and non-water. Unfortunately the researchers do not have the required knowledge to design and develop such kind of data processing network. Currently, they are using various scripts to process their scientific data. These steps are manually executed, one at a time, which can be a time consuming and error-prone process.

Nowadays distributed and/or decentralized data processing networks become more and more popular. Both aspects introduce some advantages, but it also increases the complexity of the data processing network. Therefore it is harder to understand or develop such kind of data processing network, especially for people with limited computer science knowledge. The development of a data processing network is a costly-process, it is important that the users have a good understanding of the general structure to prevent expensive redesigns. This increasing complexity brings us to the main question of this thesis:

How can we support the development of a data processing network by people with limited computer science knowledge?

To find an answer to the question above, answers to the following sub questions are needed:

1. What are the main concepts and reusable components of a data processing network?
2. How can we document a data processing network design?
3. How can we support the realization of data processing networks by people with limited computer science knowledge?

The first sub question is essential to the other two sub questions. With a clear model of data processing networks it becomes easier to document and implement data processing networks. The main question can then be answered by combining the answers from the three sub questions.

1.2 Motivation

Researchers are processing more and more data nowadays. This trend is supported by the ever-increasing processing capacity and increasing complexity of research. Sometimes this is done using various independent scripts, which are manually executed one after another. It is clear that this can be a time consuming and error-prone process. Data processing networks can be used to support the processing of these streams of data. Due to the popularity of the internet, distributed data or even decentralized processing is used more often. By introducing decentralization or distributed data, the complexity of the overall processing network is increased dramatically. Both aspects require some overhead which lowers the understandability of the data processing network in general. Because not everyone has adequate knowledge of distributed data or decentralized processing, more and more people have to rely on other people for the implementation of their data processing networks. Due to this knowledge gap, the risk on costly-redesigns and delays is increased. By introducing

a model-driven development approach, where unnecessary implementation details are abstracted away, these risks can be minimalized. The realization of a data processing network can be automated by creating a model describing its software architecture as a composition of reusable architectural elements.

First of all, we need to perform a domain analysis on data processing networks. By selecting the general concepts and reusable components, a basic model can be created. This model can be used to get a better understanding of a data processing network which will improve the communication between the various stakeholders. After we created a basic model, a specialized model can be build which supports a higher level of implementation details.

After we performed a domain analysis, the possibilities of documenting a data processing network design are analyzed. A data processing network has a software architecture and can therefore be documented by a software architecture description which commonly uses more than one architectural views. Each view supports one or more concerns of the stakeholders involved. Capturing all these concerns inside a single architectural view is often not possible due to the complexity of a software system. An analysis of the current practise of representing architectural views reveals that new dedicated views are needed to document a data processing view.

The implementation of a data processing network is not always trivial and requires a substantial development and maintenance effort, especially for people with limited computer science knowledge. Developers need to be supported for the implementation of a data processing network.

Accordingly, this thesis provides modelling, documenting and realization techniques to improve the development opportunities of data processing networks by people with limited computer science knowledge.

1.3 The Approach

In the following subsections, the followed approaches for supporting the documentation and development of data processing networks are summarized. The overall goal is to employ documentation and implementation techniques to improve the understandability and development of data processing networks.

1.3.1 Data Processing Network Analysis

After performing a domain analysis, a basic, and easy understandable, model of data processing networks should be created. Since implementing a data processing network is a costly-process, it is important that the users have a good understanding of the general structure. The *basic model* can be used to overcome a potential knowledge gap between the users and developers. To provide a more detailed (technical) representation of a data processing network, a more *specialized model* should be created. This model can be used if a more detailed (technical) view on the design is needed, but requires additional knowledge. After this step, we should be able to answer the first sub question.

1.3.2 Architectural Style for Data Processing

Once a basic and a more specialized model is created, a special *data processing view* can be added to an *architectural description* of a data processing network. A data processing view will increase the understandability of the structure and can be used to communicate architectural design decisions with respect to data processing. A practical and easy-to-use method is needed to create such a data processing view. For this purpose, the *data processing style* is introduced. This style defines a notation based on the concepts identified in the basic data processing network model. As a further specialization of the data processing style, the *DatProNet style* is introduced. The DatProNet style defines a notation based on the concepts mentioned in the specialized data processing network model and can be used for documenting a more detailed view of a data processing network. Using the DatProNet style, a data processing view for the CCES project RECORD [10] case study is created. After this step, we should be able to answer the second sub question.

1.3.3 Framework for the Realization of Data Processing Networks

After the software architecture of a data processing network is designed and documented, it can be implemented accordingly. To reduce the implementation and maintenance effort, an *architecture description language (ADL)* is introduced which can be used to generate a concrete implementation through the DatProNet framework. This ADL is based on the same model as the DatProNet style, so it should be easy to map a data processing view to such an architectural description. Using this ADL and the DatProNet framework, a data processing network is created for the CCES project RECORD [10] case study. After this step, we should be able to answer the third sub question.

1.4 Thesis Overview

This thesis is organized as follows.

Chapter 2 provides background information on our case study, CCES project RECORD [10], which is used throughout this thesis. To support the calibration of the environmental models used by those researchers, an algorithm for water segmentation on aerial photos using a water and non-water class is proposed.

Chapter 3 provides a domain analysis on data processing networks. It defines a basic model for data processing networks. A further specialized model is created for a more detailed (technical) representation of a data processing network.

Chapter 4 introduces a new architectural style, called *data processing style* for modelling the structure of a data processing network. This style can be used to communicate and analyse architectural design decisions with respect to data processing. A more specialized style is introduced, the *DatProNet style*, to support a more detailed data processing view. Both styles are based on

the models presented in chapter 3. The usage of these styles is illustrated by defining a data processing view for the CCES project RECORD [10].

Chapter 5 presents an architecture description language, called the *DatProNet language*, which can be used for the automated realization of a data processing network by the *DatProNet framework*. The usage of this language and framework is illustrated by defining a data processing network for the CCES project RECORD [10] case study.

Chapter 6 presents an evaluation of the DatProNet framework. The performance of the segmentation algorithm is evaluated using a small and a larger set of aerial photos from our CCES project RECORD [10] case study.

Chapter 7 provides our conclusions. The discussions, possible extensions and related work for the various sub questions are provided in the corresponding chapters.

An overview of the main chapters can be found in the figure below. The rectangles present the chapters of this thesis, the solid arrows indicate the recommended reading flow. The reader can start at Chapter 2 or 3, both are self-contained. Chapter 4 should be read before Chapter 5. Chapter 6 is divided into two parts. The first part, about the evaluation of the DatProNet framework, depends on Chapter 5. The second part, about the evaluation of the segmentation algorithm, only depends on Chapter 2.

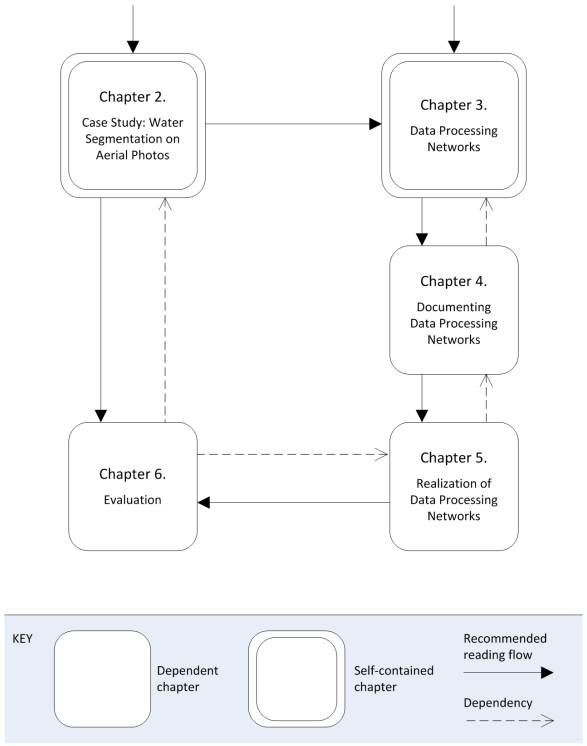

Figure 1.1: The main chapters of this thesis and the recommended reading flow

Chapter 2

Case Study: Water Segmentation on Aerial Photos

The work presented in this thesis has been carried out to support the CCES project RECORD [10]. The main objective of this project is to investigate the effects of river restoration on the surrounding environment in hydrological, biogeochemical and ecological terms [10]. The CCES project RECORD [10] is focused on a small part of the Swiss river Thur. At their project site the river is monitored by different devices, including two digital cameras. The researchers are interested to see how computer vision algorithms can be used to support the calibration of their environmental models. For this calibration they need a segmented version of the photos taken by the cameras. In this chapter an algorithm is proposed which segments an aerial photo using a water and non-water class.

This chapter is organized as follows. Section 2.1 provides a more detailed description of the actual problem. In section 2.2 an algorithm for the segmentation of aerial photos using a water and non-water class is presented. A conclusion is given after discussing the alternatives and related work in sections 2.3 and 2.4 respectively.

2.1 Problem Description

The CCES project RECORD [10] is aimed at investigating the effects of river restoration on the surrounding environment in hydrological, biogeochemical and ecological terms [10]. Their activities are focused on the river Thur, the largest river in Switzerland without a natural or artificial reservoir [5]. Their field site is located at the restored section at Niederneunforn and Altikon [5]. The river, and the surrounding environment, are analyzed, monitored and observed by different instruments, among them are two digital colour cameras looking up-

or downwards to the river. Pictures are taken every thirty minutes and stored on a computer located at the project site. The researchers are curious to see how computer vision techniques can be used to support their research. As a start, they are interested in the possibilities of using the pictures taken by the cameras for the calibration of their scientific models. For a sample photo taken by the upstream camera, see picture 2.1.

Figure 2.1: Sample photo taken by the upstream camera

The main goal is to develop an algorithm which automatically segments aerial photos using a water and non-water class. The algorithm must be as generic as possible, in other words, it must be easy to apply the algorithm on aerial photos from another natural environment.

2.2 Algorithm for Water Segmenting on Aerial Photos

Image segmentation is a fundamental problem in computer vision. Over the past few years various methods and techniques about image segmentation are proposed in the literature. Interesting for reading are [34, 55, 38, 43]. Inspired by this literature, we developed an algorithm which segments pictures using a water and non-water class. The algorithm uses two different computer vision techniques, *colour-based pixel classification* and *pixel correlation*. Both steps are discussed in more detail below. The proposed algorithm is used throughout this thesis as an example processing task.

2.2.1 Colour-Based Pixel Classification

Pixel classification can be done using various techniques which can be categorized into the following two groups: *method-based* or *learning-from-example* classification. Using the first technique all the conditional and prior probabilities are derived from general knowledge and mathematical models. In case of the learning-from-example approach, the various probabilities are derived from sample objects. Different measurements are done on the sample objects and the gathered data is used to train a classifier. Because the appearance of water is influenced by many factors, which are hard or even impossible to model, the second approach is used.

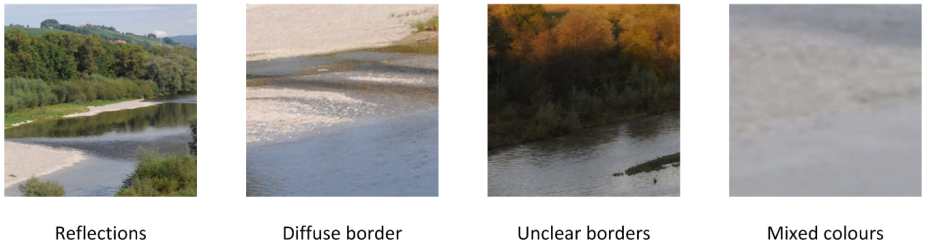

Figure 2.2: Possible segmentation problems

A perfect aerial photo would contain water with a high contrast and clear boundary to its surroundings. Unfortunately this is not always the case, as you can see in picture 2.2. The colour of the water is influenced by many factors. Among them are the depth of the water, the amount of sediment and of course the weather conditions. Pixel classification implies that each pixel is classified individually by a so called *classifier*. A classifier can be trained in two different ways: *supervised* and *unsupervised*. During supervised learning the related class is known for each sample measurement. This is not the case for unsupervised learning. In our case, the first approach is more suitable because there is no clear distinguishing between water and non-water.

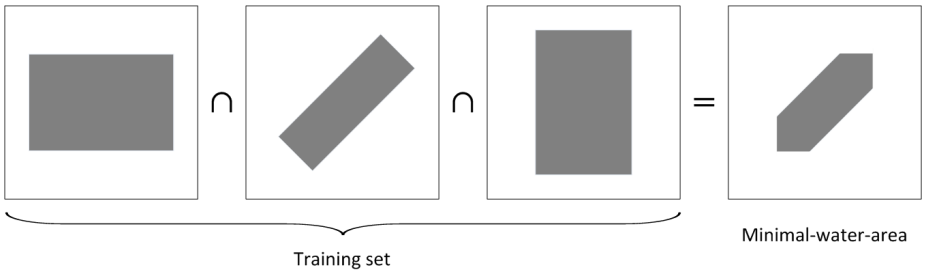

Figure 2.3: Calculation of the minimal-water-area

Following the learning-by-example approach, the classifier needs to be trained with some reference data. This data can be extracted during a so called *training phase*. Because the appearance of the water can change overtime, reference data is collected from the current photo. Training of the non-water class is hard because almost everything can appear in the surroundings of water. Therefore

a *single-class classifier* is used. In order to extract meaningful training information from the current picture, some references are needed. To determine useful reference points, a training phase is introduced. During this phase, the algorithm is provided with a set of manual segmented photos. From this collection, the *minimal-water-area* is determined. The minimal-water-area is defined as the intersection of all the water areas inside the trainings photos, see figure 2.3 for an example. To (partially) overcome the problem of human segmentation errors, a configurable margin is used around the borders of the water areas. For the classification, a configurable amount of reference points are used to gather training data. The colour of the water can be quite different inside a single picture, therefore local reference points are used if possible. The gathered colour information is then used by the classifier to classify each pixel, in other words it calculates the probability that a specific pixel belongs to the water class.

2.2.2 Correlation-Based Classification

The second step is aimed at the correlation between pixels inside a single picture. This idea is derived from a filled bowl. If water is detected at the edges of a bowl, we know for sure that there will be water in centre (due to its shape). For aerial pictures, this idea is more complicated since the shape of the river is probably not so regular. Even though this information can be quite useful in combination with the results of the first step. The correlation between the pixels is calculated during the training phase.

The results from the pixel classification step are used to determine reference points. Pixels with a probability above a certain threshold are selected, and their correlated pixels are determined. The correlated pixels are ranked by its *correlation factor*, which is defined by the number of times it is correlated by another pixel. Pixels below a configurable lower limit are regarded as non-water and pixels above the configurable upper limit are regarded as water. All pixels ranked between these boundaries, are submitted for a reclassification. This reclassification is done by a two-class classifier. The classifier is trained with colour information gathered from the already known water and non-water pixels.

After the reclassification, the final result can be determined. Pixels are classified as water if they have a correlation rank above the upper limit or if they are classified as water by the two-class classifier.

2.3 Discussion

Determination of the training set

During the training phase of the algorithm, a set of pictures is needed to determine the minimal-water-area and the correlation between the pixels. The overall performance of the algorithm is strongly related to the usefulness of the training set. The usefulness of a training set is influenced by various aspects. For the determination of the minimal-water-area, it is important to have at

least one picture with a water level at a certain minimum. Secondly, a picture with a water level at a certain maximum is needed to enlarge the set of possible water pixels. The algorithm only segments those pixels which have bin at least once classified as water during the training phase. By discarding pixels which are definitely non-water, for example the sky, the processing time is decreased. The lower and upper boundaries of the algorithm are defined by the minimal-water-area and union of all the segmented training pictures.

To ensure the usefulness of the correlation step, a diverse training set is needed. Add as much as possible pictures with various water levels. Be careful with mixing older and more recently taken pictures. The usefulness of the correlation step can decrease dramatically if the shape (or flow) of the river is changing.

Determination of the thresholds

In both steps various configurable thresholds are used. The overall result can be strongly influenced by these values. The (sub)optimal values can be different for each set of aerial photos, and are determined by a trail-and-error technique. Automated determination of optimal thresholds is considered as a possible extension.

2.4 Related Work

A two-stage algorithm for shoreline detection is proposed in [53]. This algorithm starts with classifying the image to one of the following two types: reflection-unidentifiable and reflection-identifiable. For reflection-unidentifiable images, a thresholding method based on the grey level intensity is used. A line-fitting technique is then used to eliminate outliers. For reflection-identifiable images, a two-class region classifier is used. The image is segmented into small areas based on their colour homogeneity. The areas are then classified by a classifier using the symmetry and brightness characteristics of the areas. In our case, both reflection-unidentifiable and reflection-identifiable images are possible. Unfortunately, the appearance of the water (and especially the colour) can have a high overlap with the surrounding nature. A thresholding method based on the grey level intensity is therefore almost useless. Furthermore, the reflections appearing inside the CCES project RECORD [10] pictures are not always symmetric and thus makes their reflection based classification less useful.

Algorithms for coastal boundary detection on satellite images can be found in [26, 9]. These algorithms often assume a clear boundary, caused by high contrast in colour. Unfortunately this is not the case for the pictures taken by the two cameras of the CCES project RECORD [10]. Furthermore, the difference with respect to the camera viewpoint makes the methods and techniques proposed in this research topic less useful.

A water detection algorithm for autonomous off-road navigation is presented in [35]. This algorithm uses a multi-cue approach. Colour, texture and reflections characteristics are used to detect the water inside a picture. Instead of using a supervised learning method, they try to model the characteristics of the water by

analyzing the appearance of water on a large set of aerial photos. The reflection-cue uses stereo-imaging techniques. The CCES project RECORD [10] does not have access to a stereo based camera set-up and because of the strong weight of this reflection-cue on the final classification the algorithm is less useful at this moment, but it is considered as a possible extension.

2.5 Conclusions

In this chapter a two-step algorithm for segmenting aerial pictures using a water and non-water classes is proposed. A training phase is required to determine reference points and the correlation between pixels. Colour-based pixel classification is used to extend the set of reference points. Pixel correlation is used to perform the final segmentation of the aerial photo.

Chapter 3

Data Processing Networks

Recall our definition of a data processing network: a network of computers or (software) components which apply a series of operations on data in order to fulfil a certain task. To support the development of a data processing network by people with limited computer science knowledge, a domain analysis is needed. In this chapter the main concepts and reusable elements of a data processing network are identified, see also sub question number one as defined in section 1.1. These concepts are used to create a basic model and a more specialized model, with respect to implementation and platform alternatives. A clear model will increase the understandability of a data processing network and prevents expensive redesigns.

This chapter is organized as follows. Section 3.1 provides a basic model of data processing networks. In section 3.2 a more specialized model is presented which uses a more detailed representation with respect to implementation and platform alternatives. This specialized model will be used throughout this thesis. A conclusion is given after discussing the alternatives and related work in sections 3.3 and 3.4 respectively.

3.1 Basic Model

For a good understanding of a data processing network, a basic model is needed. This model identifies the basis concepts and the relations among them. Using our definition of a data processing network, various general concepts are identified which can be found in figure 3.1. Three types of relations are possible: *associations*, *aggregations* and *generalizations*. An aggregation indicates a part-whole relationship. Associations and aggregations consists of a *role* and an optional *cardinality*. For example, the diagram illustrates that a data processing network *fulfils* (the role) *one or more* (the cardinality) tasks, a transformation is part of a filter, and a data source is a specialization of an input port.

There are a few important concepts which need some explanation. A *filter*, which is deployed on a *node*, consists of one or more *input ports*, a *transformation* and one or more *output ports*. An input port can be a *data source*, which

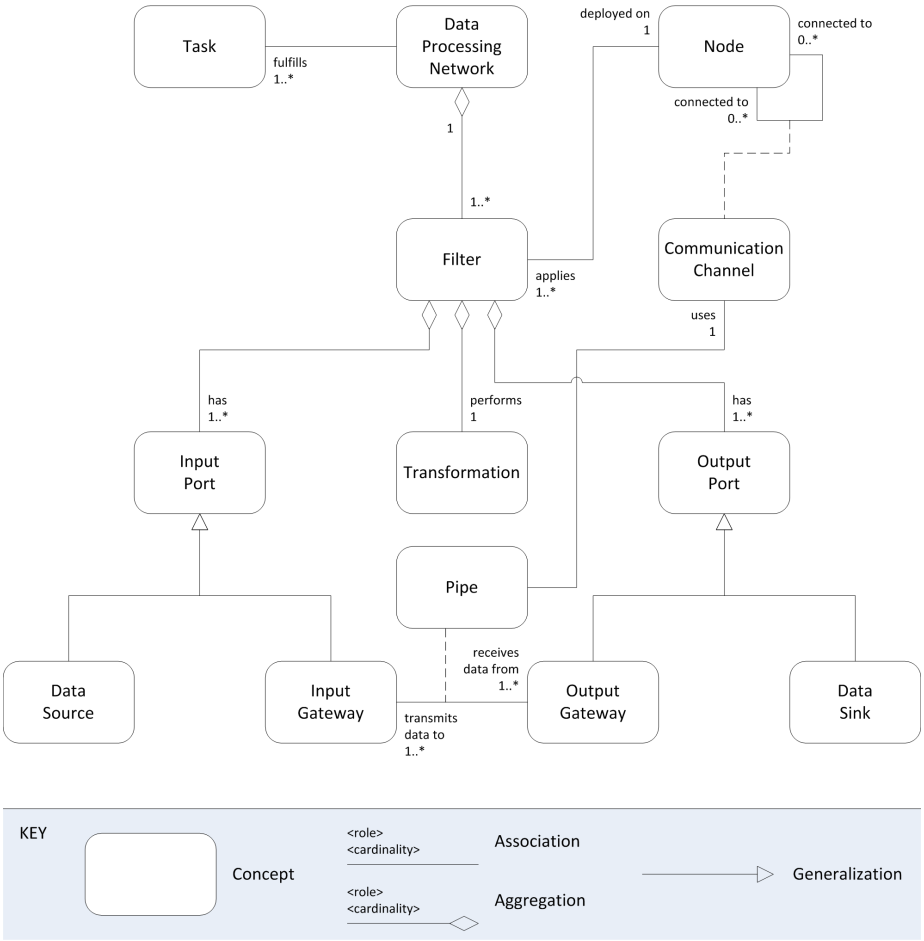

Figure 3.1: Data processing networks, a basic model

originates and provides data, or an *input gateway*, which provides an entrance to a filter. An output port can be a *data sink*, which stores data for further use, or an *output gateway* which provides an entrance to a pipe. Nodes can be connected to other nodes by communication channels. A *pipe* transmits data from an output gateway to an input gateway using a communication channel.

3.2 Specialized Model

The basic model provides a high level representation of a data processing network. In this section a more specialized model is presented which can be used to provide a more detailed representation, with respect to implementation and platform alternatives. This specialized model is created using the information from the CCES project RECORD [10] case study (see chapter 2) and various other data processing networks [40, 17, 51].

Based on our case study, as presented in chapter 2, three extensions of the basic model can be derived. The first extension is a specialization of a data source, called a *file system listener*, which monitors the file system for changes. A second specialization of a data source is added, called a *random value generator*, which can be used for debugging purposes or as a trigger for a specific filter. Thirdly, a specialization of a data sink is created, called file system storage, which stores data packages on the file system.

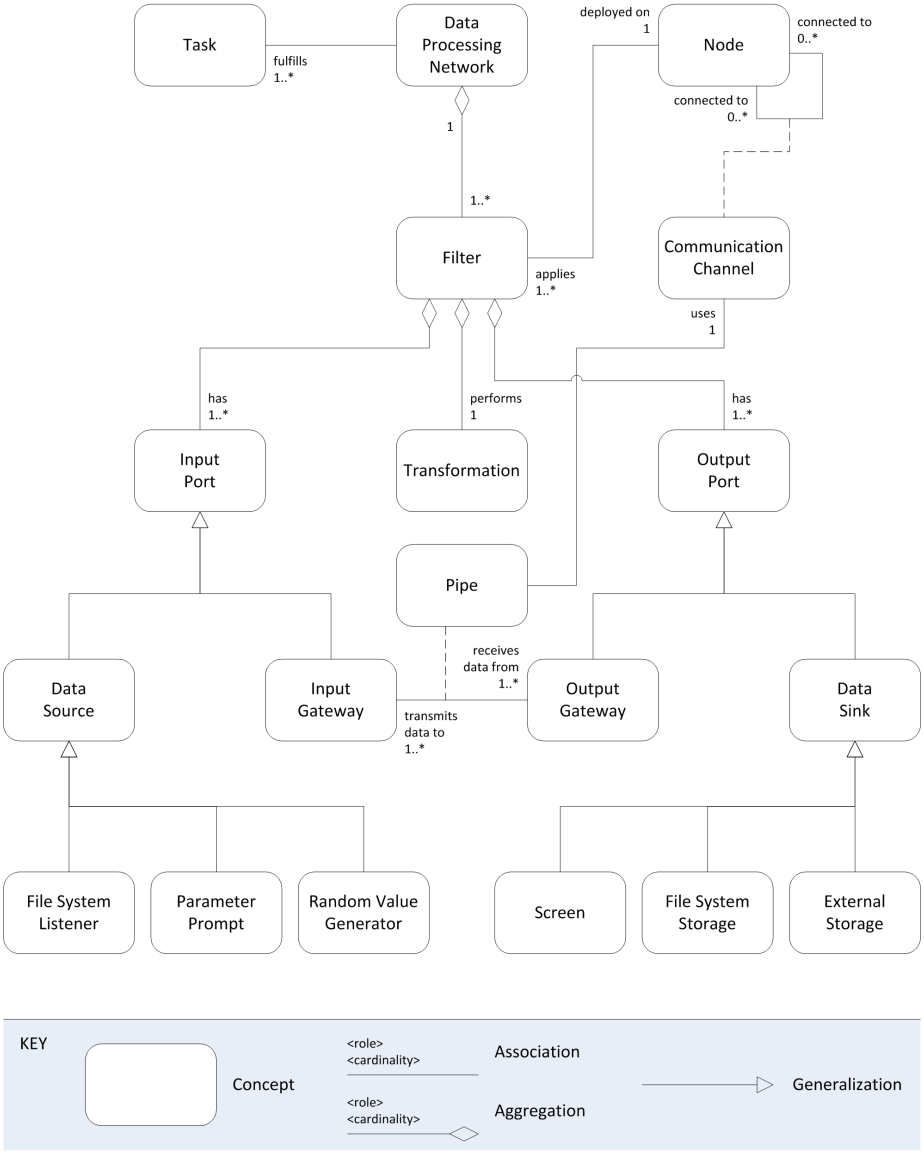

Figure 3.2: Data processing networks, a specialized model

From the data processing networks presented in [40, 17, 51] various other specialisations of a data source and data sink can be derived, see figure 3.2. These three data processing networks need some static input parameters for their pro-

cessing tasks. Parameters can be put into a configuration file, but sometimes it is easier, and nicer, to ask the user for these constants. A *parameter prompt* asks the user to fill-in some constants and put them into a single data package.

These data processing networks are all printing there results onto the screen, therefore the concept of a *screen* is introduced as a specialization of a data sink. Another specialization of a data sink can be derived from the current trend of storing data inside a database using a *DataBase Management System (DBMS)* like Oracle database [33] or SQL Server [28]. A DBMS controls not only the storage but also the organization, retrieval, security and integrity of data inside a database. An *external storage* data sink is created to represent the logic needed to transmit the data to an external storage system like a DBMS.

3.3 Discussion

Decomposition of transformation

In the basic model, and even the specialized model, the transformation is not further specialized. Because the transformation represents the actual processing operation, there are countless specializations possible. With adding specializations of a transformation, the model will be harder to read. In our case the possible benefits of a further decomposed transformation do not outweigh the disadvantages with respect to the understandability of the model.

Data source and data sink specializations

The specialized model contains three data sources and three data sinks. Even though it is possible that another data source or data sink is needed. The specialized model contains commonly used specializations of these concepts, but can be extended if needed.

3.4 Related Work

Data processing networks are a special case of Kahn process networks [46], a model based on deterministic processes which are communicating through unbounded FIFO channels. There are a couple important differences between data processing networks and Kahn process networks. First, Kahn process networks are using unbounded FIFO channels for the communication between processes. The basic, and also the specialized, model do not specify any constrains on the boundedness of channels. Secondly, determinism is not required for data processing networks. Kahn process networks are deterministic, which means that it must produce always the same output given a particular input (history). Another difference can be found in the way the channels are used. In a Kahn process network, reading from a channel is blocking while writing to a channel is non-blocking. Our data processing network models do not contain such a restriction.

A model for dataflow process systems is proposed in [23]. Dataflow process systems tend to be a special case of Kahn process networks. The model introduces the concept of a dataflow actor and firing rules. When a dataflow actor fires, it will map input to output tokens. The moment of firing is specified by a set of firing rules. These rules specify the input tokens needed for an actor to fire. Once it fires, input tokens are consumed and output tokens are produced. Unlike Kahn processes, dataflow processes can be interleaved by a so called scheduler. This scheduler determines the order in which the various actors can fire. An advantage of these networks is that there is no context switch overhead for the suspension or resumption of a process.

Conventional query processing methods assume a relatively static and predictable computation environment [6]. Unfortunately, these techniques are not sufficient in an constantly changing environment with large streams of continues queries on data streams [6]. The TelegraphCQ [6] query processing system is introduced to overcome these problems. It is focused on its adaptability inside a frequently changing environment.

The past few years a lot of research has been done on the subject of data stream processing. Many of these have led to the development of a so called data stream management system, like Aurora [4]. These systems often use a graphical user-interface to build queries using a set of operators. An overview of the models and issues related to data stream (management) systems can be found in [2, 14, 30].

A software architecture cannot be described in a simple one-dimensional fashion [7]. An architectural description is therefore organized by a set of views. Each view illustrates a specific aspect of the system and supports one or more concerns. In [7] a model is presented for documenting a component-and-connector view. One of the specializations of the component-and-connector view, called the pipe-and-filter style, has many similarities with the models presented in the previous sections. The pipe-and-filter style contains definitions of a pipe and filter, but are more high level with respect to their functionality.

A workflow is an automated business process in which documents, information or tasks are processed according a set of rules [8]. A data processing networks is in fact a workflow, but a workflow is not always a data processing network. Workflow systems like Kepler [19] or Taverna [41] are using a very detailed model with respect to the actual transformation, which requires more technical knowledge and increases the complexity of the overall model.

3.5 Conclusions

In this chapter a basic model for data processing networks is presented. A filter which is deployed on a node, consists of one or more input ports, a transformation and one or more output ports. An input port can be a data source, which originates and provides data, or an input gateway, which provides an entrance to a filter. An output port can be a data sink, which stores data for further use, or an output gateway which provides an entrance to a pipe. Nodes can be connected to other nodes by communication channels. A pipe transmits data

packages from an output gateway to an input gateway using a communication channel.

A specialized model is presented to provide a more detailed representation, with respect to implementation and platform alternatives. This specialized model contains specializations of a data source and data sink. A data source can be a file system listener, which monitors the file system for changes, a random value generator, which can be used for debugging purposes or as a trigger for a specific filter or a parameter prompt, which asks the user to fill-in some constants and put them into a single data package. A data sink can be a screen, which prints the contents onto the screen, a file system storage, which stores the data packages on the file system, or an external storage, which can be used for transmitting data packages to an external storage system like a DBMS.

Chapter 4

Documenting Data Processing Networks

Each data processing network has an architecture. The *software architecture* of a data processing network consists of the structure or structures of that system, which comprise software elements, the externally visible properties of those, and the relationships among them [3].

A software architecture represents a common abstraction of a system that can be used by people, interested in the construction of the software system, for mutal understanding and communication among them [3]. Between these so called *stakeholders*, there can be a quite large knowledge gap. In this chapter the *data processing style* and its specialization the *DatProNet style* are introduced, which can be used for documenting a data processing network design, see also sub question two as defined in section 1.1.

This chapter is organized as follows. Section 4.1 provides a short introduction about documenting software architectures. In section 4.3 the data processing style is introduced and its specialization, the DatProNet style, is described in section 4.4. Section 4.5 illustrates the usage of the DatProNet style for the CCES project RECORD [10] case study (see chapter 2). A conclusion is given after discussing the alternatives and related work in sections 4.6 and 4.7 respectively.

4.1 Documenting Software Architectures

The software architecture of a system embodies various important design decisions, which impacts all the next steps inside the software development life cycle (e.g. development, maintenance, etcetera). It is clear that these decisions must be carefully documented. Software architectures improve the reusability of architectural models between systems with similar functional requirements and/or quality attributes [3].

In the following subsections some basic concepts and techniques are introduced about describing software architectures.

4.1.1 Software Architecture Descriptions

The software architecture of a system is described by a collection of documents called the *architectural description*. The IEEE 1471 standard, *Recommended practice for architectural description of software-intensive systems*, [25] is a useful guideline for the creation of such an architectural description. This standard introduces a conceptual framework which can be found in the diagram below.

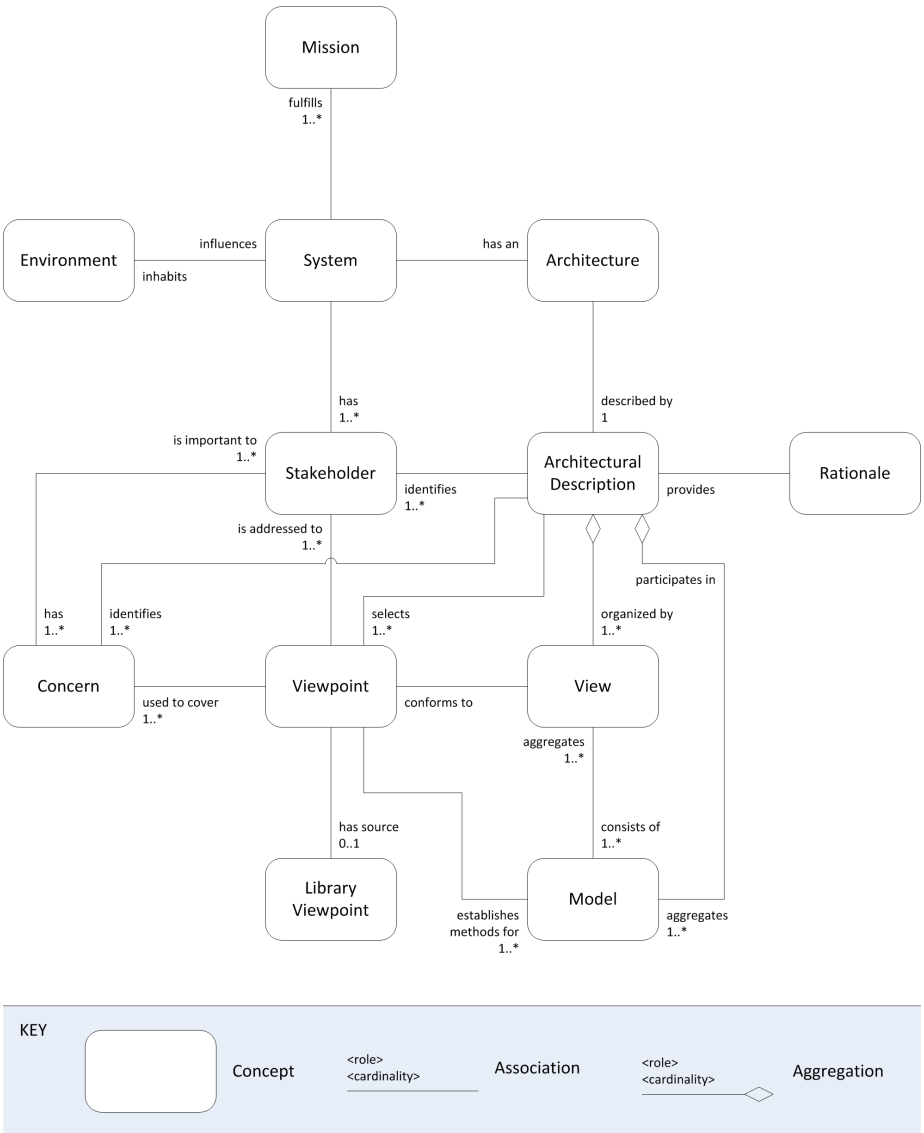

Figure 4.1: The conceptual framework as proposed in IEEE 1471 [25]

The diagram shows the different concepts and the relations among them. Two types of relations are possible: *associations* and *aggregations*. An aggregation indicates a part-whole relationship. Both types consists of a *role* and an op-

tional *cardinality*. For example, the diagram illustrates that a stakeholder *has* (the role) *one or more* (the cardinality) concerns and that a view *is part of* an architectural description. There are a few important concepts which need some explanation. A *stakeholder* is a person or organisation that is interested in the construction of the system. Some typical stakeholders are the end-user(s), software architect, system engineer, developer, designer, etcetera. Each stakeholder has some interests, called *concerns*, in the development, operation or any other critical aspect of the software system. In some cases these concerns are conflicting to each other. A *view* consists of a collection of models illustrating a particular aspect of the system. A *viewpoint* is a specification for creating and using a particular view.

IEEE 1471 [25] provides a practical guideline for describing software architectures. It does not standardize the process of designing an architecture nor the format or notation used for the documentation. In the literature various architectural design methods or software design processes are proposed, such as, Attribute-Driven Design [3], Model-Driven Design [20] and Rational Unified Process [22]. In practise, UML [37] is often used for describing architectures, but there are also several specific *architecture description languages* (ADLs) proposed in the literature, such as, Aesop [13], Darwin [24] and Weaves [15].

4.1.2 Software Architecture Views

A software architecture cannot be described in a simple one-dimensional fashion [7]. An architectural description is therefore organized by a set of views. Each view illustrates a specific aspect of the system and supports one or more concerns. In the literature some authors have proposed fixed sets of views to document a software architecture. For example, Kruchten's 4+1 view approach [21] introduces a logical view, a development view, a process view and a physical view. On the other hand the current trend is more like taking different sets of views for different systems instead of a fixed set for all software systems. The IEEE 1471 standard has taken this trend into account by just proposing a multi-view approach and not a particular set of views.

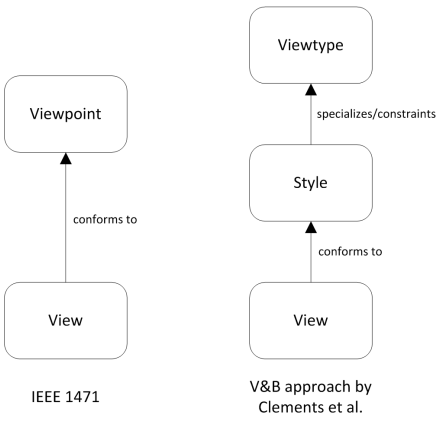

Figure 4.2: Viewpoint, viewtype, style and view

In the literature many different views can be found. Clements et al. proposed their Views and Beyond (V&B) approach [7] to bring some order to all these views. They basically split the viewpoint concept into a *viewtype* and a *style* (see figure 4.2). A viewtype defines the elements and their possible relationships which can be used to describe the system from a particular perspective [7]. A style specializes the elements and relationships defined by a viewtype and adds some (additional) constrains on their usage [7]. The V&B approach introduces three different viewtypes: the *module viewtype*, the *component-and-connector viewtype* and the *allocation viewtype*. Furthermore they introduce different specializations, for example, the *pipe-and-filter style* is a specialization of the component-and-connector viewtype. The concept of viewtypes and styles makes the V&B approach easy adaptable since an architect can define any style needed.

4.2 The Need for a Domain-Specific View

Data processing networks are used in many different domains. The scientific research domain is just one of them. During the development of a data processing network, there can be a quite large knowledge gap between the developers and the researchers. They are mainly interested in how the data flows and less about how this is achieved. Therefore it is important to have a specialized *data processing view* which can be easily understood by all interested stakeholders.

The IEEE 1471 standard proposed a multi-view approach but it does not introduces any specific views. The V&B approach introduces various different styles which can be used for documenting a software architecture. One of the styles they propose is the pipe-and-filter style which is often used for documenting data transformation systems.

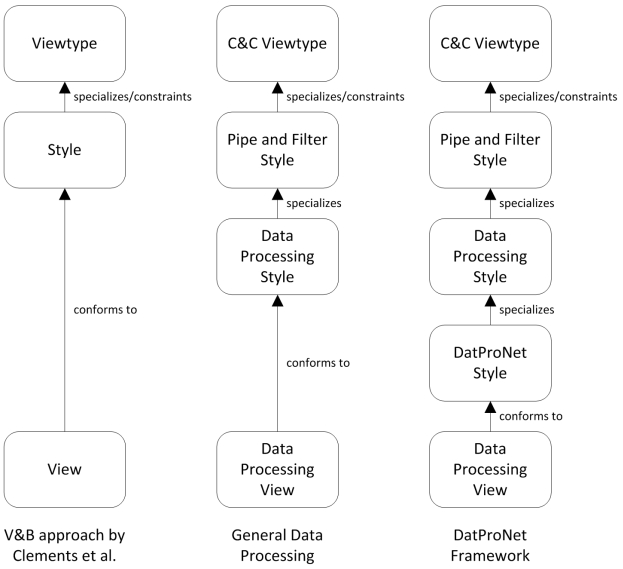

Figure 4.3: Relation between the data processing style and the DatProNet style

In the next section, a data processing style is proposed which specializes the elements and relations of the pipe-and-filter style. To document data processing networks represented by the specialized model from section 3.2, another style is created, called the DatProNet style, in which the elements and relations of the data processing style are further specialized (see figure 4.3).

4.3 Data Processing Style

The data processing style can be used for documenting a *data processing view*. The notation for the key concepts from the basic model (see section 3.1) can be found in figure 4.4. For notation purposes only, the concept of a binding is added.

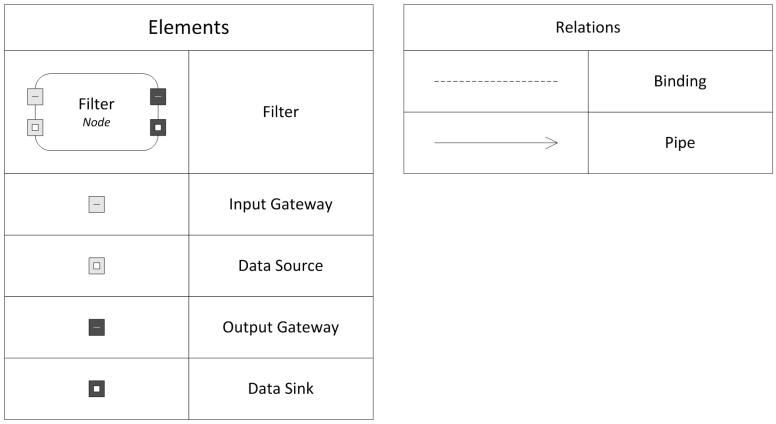

Figure 4.4: Data processing style notation

- *Elements*: Filter. Filter ports must be either input (light coloured) or output (dark coloured) ports. An input port can be an input gateway or a data source. An output port can be an output gateway or a data sink.
- *Relations*: Binding, Pipe.
- *Properties of elements*: Properties of a Filter: a name which suggest its functionality, a description indicating its place inside the data processing network, one or more input ports and one or more output ports.
- *Properties of relations*: Properties of a Pipe: type of communication (for example synchronous or a-synchronous).
- *Topology*: A pipe connects an output gateway to an input gateway. A binding connects an input port to an input gateway or connects an output gateway to an output port. Bindings can only be used inside a filter. Every input and output gateway should be connected using these rules. Filters can be placed inside other filters using the binding relation and the rules above. Each set of connected filters should contain at least one filter with a data source input port and one filter with a data sink output port (can be combined in a single filter).

The DatProNet style introduces the filter element with different kinds of input and output ports. Data packages are arriving through the input ports. Inside the filter there will be some data transformation applied onto the packages. Data packages are leaving the filter through its output ports. Input gateways are just passing data packages from a binding or pipe to the filter. Data sources are originating and providing data to the filter. Output gateways are passing data packages from the filter to a binding or pipe. Data sinks store data packages for further use. A binding can only be used to connect two filters placed inside of each other. A pipe is used for the communication between two filters. Each pipe uses its own type of communications and has its own quality properties (in the sense of recoverability, performance, etcetera).

4.4 DatProNet Style

In this section, the DatProNet style is presented in which the elements and relations of the data processing style are further specialized according to the specialized model as introduced in section 3.2. The key elements and relations can be found in figure 4.5.

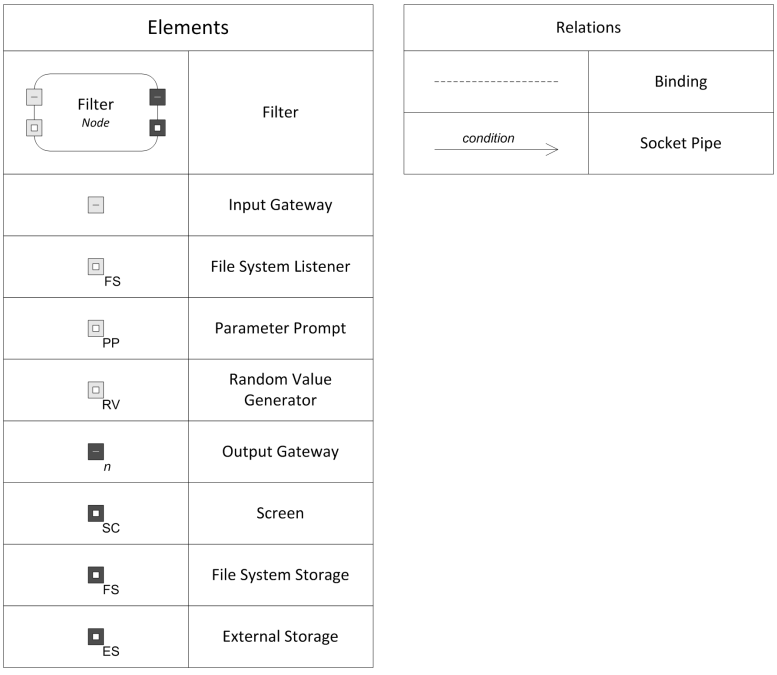

Figure 4.5: DatProNet style notation

- *Elements*: Filter. Filter ports must be either input (light coloured) or output (dark coloured) ports. An input port can be an input gateway, a file system listener, a parameter prompt or a random value generator. An output port can be an output gateway, a screen, a file system storage or an external storage.

- *Relations*: Binding, Socket Pipe.

- *Properties of elements*: Properties of a Filter: a name which suggest its functionality, a description which suggest its place inside the data processing network, one or more input ports and one or more output ports. Properties of a File System Listener: the element (directory or file) it listens to and some performance based properties (e.g. delay between two polling sessions, minimal delay between the generation of two data packages, etcetera). Properties of a Parameter Prompt: a set of parameters which has to be provided (manually) by the user and the type of communication used (for example command-line). Properties of a Random Value Generator: a range specifying the possible values it can generate and some performance based properties (same as File System Listener). Properties of an Output Gateway: a number indicating the amount of pipes which will be selected (randomly) to pass an incoming data package on. Properties of a File System Storage: a directory indicating the location where the packages will be stored. Properties of an External Storage: a description of the external application used for storing the incoming data packages.

- *Properties of relations*: Properties of a Socket Pipe: the port number used for the socket communication and an optional condition indicating in which cases the pipe accepts data packages.

- *Topology*: A pipe connects an output gateway to an input gateway. A binding connects an input port to an input gateway or connects an output gateway to an output port. Bindings can only be used inside a filter. Every input and output gateway should be connected using these rules. Filters can be placed inside each other using the binding relation and the rules above. Each set of connected filters should contain at least one filter with a data source input port and one filter with a data sink output port (can be combined in a single filter).

4.5 Using the DatProNet Style

Figure 4.6 illustrates the usage of the DatProNet style for documenting a data processing view for the CCES project RECORD [10] case study. A description of the CCES project RECORD case study can be found in chapter 2.

The figure illustrates just one possible data processing view. Many alternatives are possible. One of them would be to remove the specializations of the training and analyze filter. Some computer vision details are lost but depending on the interested stakeholders, it could be a valid choice.

Another design alternative would be to move (and copy) the file system listener input port of the settings filter to the two merge filters. In that case the communication overhead between the settings filter and the two merge filters is removed, but two file system listeners are created instead (with their own polling interval). From a performance point of view, its is better to have just one settings filter since the configuration file will not change frequently.

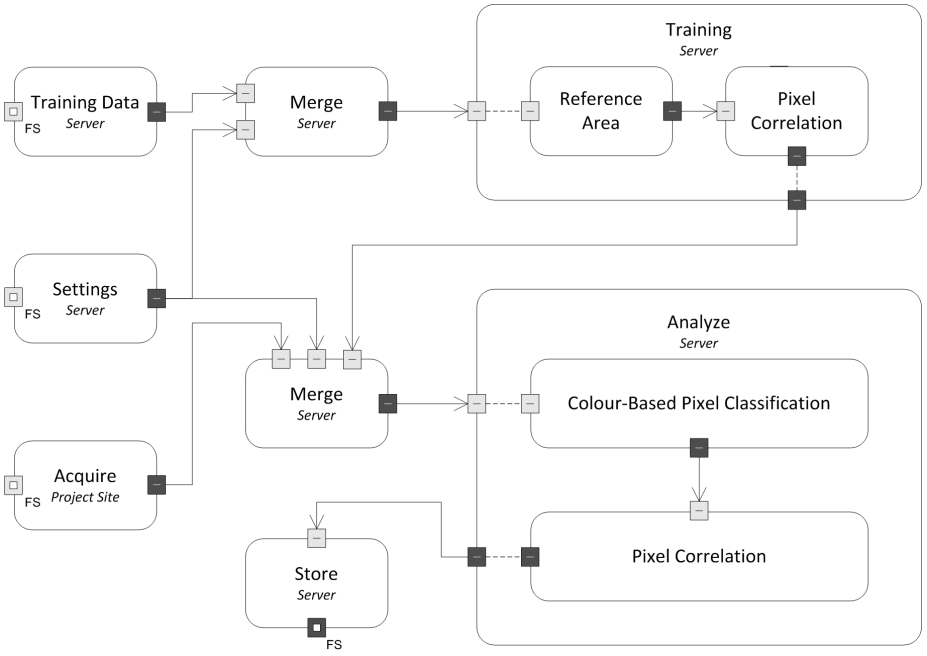

Figure 4.6: A data processing view for the CCES project RECORD [10] case study (for key see figure 4.5)

4.6 Discussion

Style or Viewpoint?

The data processing style is a specialization of the pipe-and-filter style introduced by the V&B approach [7]. The V&B approach does not describe the concept of a viewpoint, as defined by the IEEE 1471 standard [25], however it introduces the viewtypes and styles. A viewpoint describes the language and notation used to create a particular view. Therefore the data processing style, and also the DatProNet style, can be considered as a viewpoint as well since both styles define a notation to describe a data processing view. Because both styles are strongly related to the component-and-connector viewtype and the pipe-and-filter style, the terminology of the V&B approach is selected.

Data Flow Views

A data processing view is not the same as a data flow view. The relations defined by the data processing style have a different meaning than the ones used in a data flow view. A pipe, as defined by the data processing style, represents a connector with a clear computational meaning. The relations drawn in a data flow view have little computation meaning, there simply flows some data between the two elements (which can be realized by a connector).

Static and dynamic analysis

Static analysis can support the user with the development of a faultless data processing network. Static analysis contains at least a verification of the topology requirements as stated in the previous sections. But even after performing this verification, it is still possible to develop a data processing network containing one or more faults. For example, if a data processing view contains a loop, it is impossible to verify (with static analysis) that a specific data package eventually will arrive at a data sink output port. With dynamic analysis more verification can be done. For example, a verification of the performance properties of the various input ports (together with a suggestion of their optimal values). Other possibilities of applying static and dynamic analysis on a data processing view is considered as feature work.

4.7 Related Work

In [18] UML 2.0 is discussed as a language/notation for documenting software architectures and in particular component-and-connector views. With respect to earlier versions of UML there is some improvement but still its not possible to represent connectors (the pipes) in an intuitive way.

A comparison of architectural styles for network-based software architectures can be found in [11]. After a classification and comparison of the existing styles, the *Representational State Transfer (REST) style* for distributed hypermedia systems is introduced. In comparison with the data processing style, and also the DatProNet style, the REST style is more focused on the network properties instead of the data processing aspects. Depending on the interest of the stakeholders, both styles can be used inside an architectural description.

Workflows can be created with a Workflow Management Systems like Kepler [19] or Taverna [41] using an intuitive drag-and-drop interface. A workflow is defined as the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules [8]. A data processing network is in fact a workflow, but a workflow is not always a data processing network. The workflows created by these systems support some decentralization aspects, but there is always a central system needed to execute the workflow, so it is not completely decentralized. In comparison with the data processing style, the workflows created by Kepler [19] or Taverna [41] offer a more detailed view on the processing task, especially on how the transformation is done. De data processing style, and even the DatProNet style, are more high level and thus require less knowledge about the application.

4.8 Conclusions

In this chapter a data processing style is introduced for documenting a data processing network. The data processing style is a specialization of the pipe-

and-filter style as defined in the V&B approach [7]. The DatProNet style is introduced in which the elements and relations of the data processing style are further specialized. The usage of these styles has been illustrated by defining a data processing view for the CCES project RECORD [10] case study.

Chapter 5

Realization of Data Processing Networks

In the previous chapter the data processing style and its specialization, the DatProNet style, where introduced. After designing, documenting and analyzing the architecture of a data processing network, the various architectural elements and relations should be implemented. In this chapter the DatProNet framework is introduced to reduce the development and maintenance effort of a data processing network, see also sub question number three as defined in section 1.1.

This chapter is organized as follows. Section 5.1 provides an outline of the basic requirements. In section 5.2, the DatProNet framework is introduced. Section 5.3 discusses the extensibility of the DatProNet framework. Section 5.4 illustrates the usage of the DatProNet framework for the CCES project RECORD [10] case study (see chapter 2). A conclusion is given after discussing the alternatives and related work in sections 5.5 and 5.6 respectively.

5.1 Basic Requirements

The development of a data processing network imposes certain requirements to its architecture. Based on our models from chapter 3 and the CCES project RECORD [10] case study (see chapter 2), the following requirements are identified (sorted by importance):

- *Reliability*: A data package flowing through a data processing network may never be lost, not even during any kind of system failure.
- *Ease of use*: People with limited computer science knowledge should be able to develop, and maintain, a data processing network through the framework.
- *Interoperability*: A typical data processing network uses various external applications to perform the processing steps. The framework should

provide a way to communicate (bi-directional) with these applications preserving the requirements and quality aspects denoted.

- *Portability*: The framework should support various system environments. It should be possible to develop data processing networks which can be deployed on a Windows [29], Linux (e.g. RedHat Linux [36]) or Unix (e.g. FreeBSD [12]) based platform. Furthermore it should support the possibility of having various system environments inside one specific data processing network.

- *Recoverability*: In case of a system failure (hardware or software) the data processing network should be able to recover from this event and return to a consistent state. The framework must be captured to deal with unavailability of nodes, for example, by sending retrying messages or by generating exceptions.

- *Extensibility*: It should be possible to extend the framework with customized input or output ports, transformations and/or pipes. In this way each user can customize the framework to its own special needs without having to develop a complete data processing network from scratch.

5.2 DatProNet: A Framework for the Realization of Data Processing Networks

To reduce the development and maintenance effort of a data processing network, the DatProNet framework is introduced. The framework has been implemented in the JAVA language [32]. Through the concept of *write-once-run-anywhere* [50], a variety of platforms are supported. The DatProNet framework comprises a collection of reusable components and an architecture description language, both items are discussed in the following subsections.

The development life cycle of a data processing network is illustrated in figure 5.1. Once a data processing view is created, it can be mapped to an architectural description using the DatProNet language. This architectural description can be put into the DatProNet framework which creates the required deployment packages for the various machines. Each deployment package can only be used on the machine it is created for.

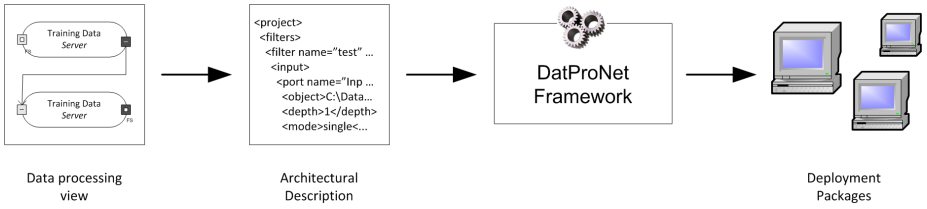

Figure 5.1: The development life cycle of a data processing network

5.2.1 Reusable Components

The frameworks includes a set of reusable abstractions for data processing networks. Figure 5.2 shows a conceptual view of the DatProNet framework as an UML class diagram [44]. The framework comprises eight main components: *Filter*, *InputPort*, *OutputPort*, *Transformation*, *Condition*, *PipeEnd*, *DataPackage* and *Serializer*. Using these reusable components, the DatProNet framework can be easily extended. Section 5.3 provides more information about extending the DatProNet framework with user-defined components.

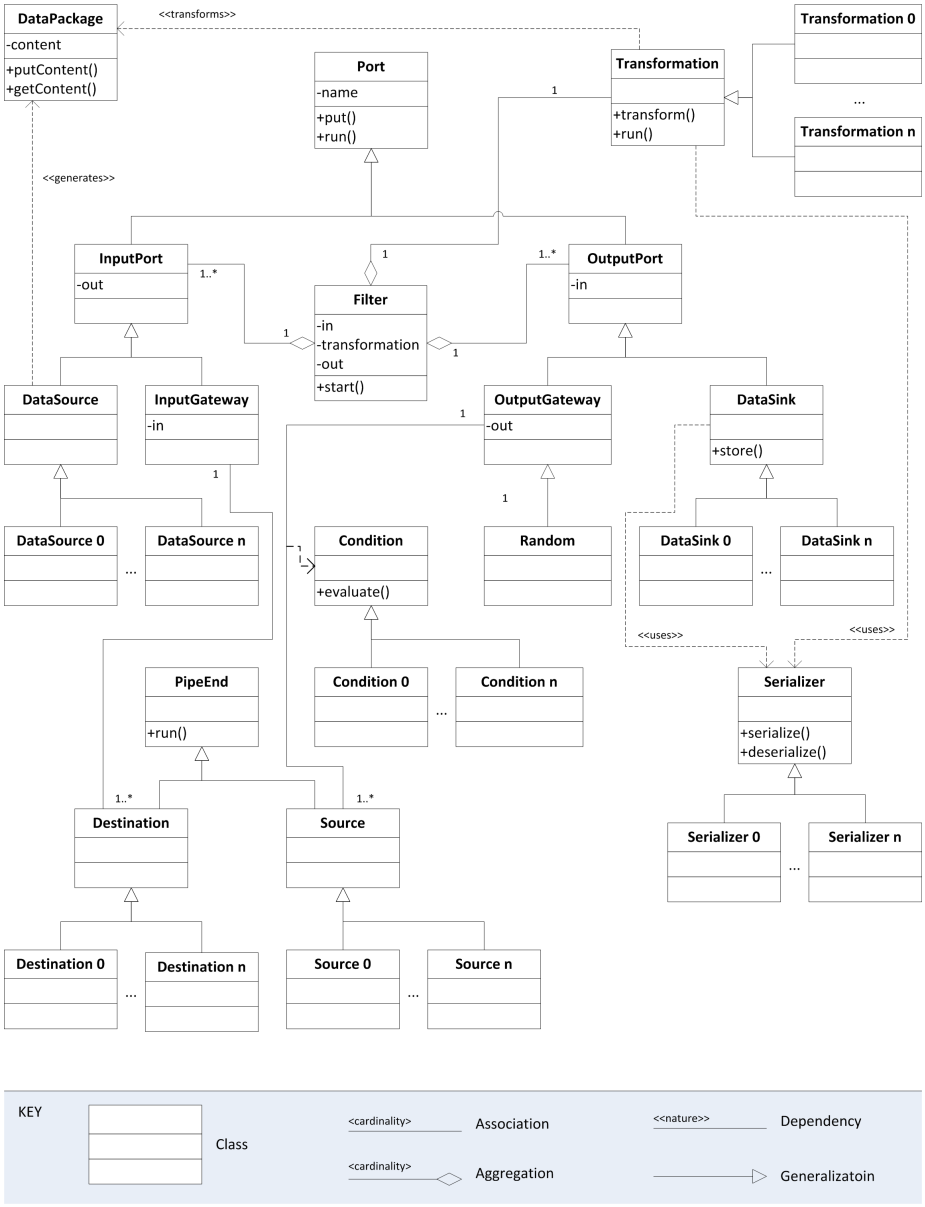

Figure 5.2: A Conceptual View of the DatProNet framework

Each node inside a data processing network is mapped onto a *Filter*. Each *Filter* connects the related *InputPorts* to the *Transformation*, and the *Transformation* to the related *OutputPorts*. A data package flowing through the data processing network is encapsulated by a *DataPackage* which contains a list of data items and a list of related resources. A pipe is divided into two parts (so called *PipeEnds*): a *Source* and a *Destination*.

DataPackages are generated by *DataSources*. Each data processing network must contain at least one DataSource. The framework contains three predefined DataSources: one that monitors the file system for changes, one that generates DataPackages with a random value and one that asks the user for some additional input. Other DataSources can be developed by implementing DataSource. *InputGateways* are passing DataPackages from a Destination PipeEnd to the related Transformation.

Once the DataPackages are arrived at the Transformation, various operations are possible. The framework contains four predefined transformations. The first transformation applies a Cross-Join operation on the incoming DataPackages. The second one provides the possibility of adding and removing data items inside a DataPackage. The third transformation can be used for (bi-directional) communication with an external application. The last one is an empty transformation, it just passes the DataPackages without any transformation. Other transformations can be developed by implementing Transformation.

OutputGateways are passing DataPackages from the Transformation to the connected Source PipeEnds. By default, incoming DataPackages are put on all the connected, and activated through Condition, Source PipeEnds. The *Random* specialization selects randomly a specific number of connected, and activated, Source PipeEnds. DataPackages can be stored by *DataSinks* using a specific *Serializer*. The framework contains three predefined DataSink implementations: one for storing the DataPackages onto the file system, one for external storage and one for showing incoming DataPackages onto the screen. Other DataSinks can be developed by implementing DataSink.

A Serializer can be used for converting DataPackages to any other format. The framework contains a *MATSerializer* which can be used for serializing DataPackages to a MatLab-readable format [42]. Serializers for other applications or formats can be developed by implementing Serializer.

The InputPort, Transformation, OutputPort and PipeEnd components use an acknowledge based method to communicate to each other. In this way a reliable data processing network can be created. Furthermore, The PipeEnd components are using a polling technique to recover from unavailable nodes inside the data processing network.

5.2.2 Architecture Description Language

The DatProNet language can be used to create an architectural description. The language is XML-based and defined by its own XML-schema which can be found in Appendix A. Because it is XML-based, the architectural descriptions can be created and validated by XML-editors like XMLBlueprint [39] or

XML Notepad [27]. With this architecture description language all the features of the DatProNet framework can be configured. An architectural description consists of various components, and all these components are defined by XML-tags. When we refer to an XML-tag including its contents, we call it an XML-element. An XML-element thus includes all enclosed XML-tags and XML-elements. An XML-tag consist of a keyword enclosed by the brackets '<' and '>'. The DatProNet language is strongly structured and consists of a root element called `<project>`, which contains a `<filters>` element. The `<filters>` element consists of a set of `<filter>` elements. A `<filter>` element has three sub elements: an `<input>` element, a `<transformation>` element and an `<output>` element.

```
1  <project>
2    <filters>
3      <filter name="Sample Filter" runat="127.0.0.1"
          buffer="10">
4        <input>
5          ...
6        </input>
7        <transformation>
8          ...
9        </transformation>
10       <output>
11         ...
12       </output>
13     </filter>
14   </filters>
15 </project>
```

Listing 5.1: The basic structure of an architectural description

The `<filter>` element has three attributes: `name`, `runat` and `buffer` (optional). The first attribute depicts its name and must be unique. The second attribute shows the ip-address of the machine where the filter will be deployed on. The last attribute indicates the maximum number of DataPackages inside the internal buffer. In the following subsubsections the three sub elements of the `<filter>` element are discussed.

The `<input>` element

The `<input>` element consists of a set of `<port>` elements. Each `<port>` element has two attributes: `name` and `type`. The name must be unique inside the parent `<input>` element. Four types of InputPorts are defined: `gateway`, `filesystemlistener`, `parameterprompt` and `randomvaluegenerator`. The first type has no aditional sub elements and just passes the incoming DataPackages to the related Transformation. The other three types have their own set of sub elements.

A `filesystemlistener` InputPort monitors a specific file or directory on the file system, and when it detects a change (e.g. a file is added or removed), the

modified files are put into a DataPackage. The file or directory can be specified by the `<object>` element. In the case of a directory, no subdirectories are taken into account by default, if needed they can be set by the `<depth>` element. Three operational modes are possible in case of a directory: `all`, `collection` and `single`, which can be specified by the `<mode>` element. When `single` is selected, only the modified file is put into a DataPackage. In case of multiple modified files, each file is put into a separate DataPackage. When `collection` is selected, all the modified files are put into one single DataPackage. When the option `all` is selected, the complete contents of the directory is put into one single DataPackage. The `<key>` element is used to define the name for the entries inside the DataPackage. With the `<initialization>` element, the start-up procedure can be defined. Two types of initialization are possible: `clear` (default) or `current`. When `clear` is selected, the file or all the files inside the directory are said to be modified at start-up. When `current` is selected, the modification time of the file or the contents of the directory is stored at start-up and used for the next check. With the optional `<delay>` and `<lifetime>` elements, the delay between two checks and the lifetime of the generated DataPackages can be set (milliseconds).

```
1 <port name="Input" type="filesystemlistener">
2   <object>C:\images\</object>
3   <depth>1</depth>
4   <key>configuration</key>
5   <initialization>clear</initialization>
6   <mode>single</mode>
7   <delay>700</delay>
8   <lifetime>700</lifetime>
9 </port>
```

Listing 5.2: An InputPort of type `filesystemlistener`

A `parameterprompt` InputPort asks the user to provide some additional input. These parameters can be defined through the `<parameters>` tag. The `<parameters>` tag consists of one or more `<parameter>` tags. A `<parameter>` tag consists of the following tags: a `<key>` tag indicating its name inside a DataPackage, a `<title>` tag indicating the title of the parameter (will be shown to the user) and a `<type>` tag indicating the parameter type. Possible types are `boolean`, `file`, `float`, `integer`, and `string`.

```
1 <port name="Input" type="parameterprompt">
2   <parameters>
3     <parameter>
4       <key>Count</key>
5       <title>Number of loops</title>
6       <type>integer</type>
7     </parameter>
8   </parameters>
9 </port>
```

Listing 5.3: An InputPort of type `parameterprompt`

A `randomvaluegenerator` InputPort generates DataPackages with a pseudo-random integer value inside. This can be useful for debugging purposes or to trigger another filter. The range can be defined using the `<min>` (inclusive) and `<max>` (exclusive) elements. The `<key>` element is used to define the name for the entry inside the DataPackage. With the optional `<delay>` and `<lifetime>` elements, the delay between the generation of two DataPackages and their lifetime can be set.

```
1 <port name="Input" type="randomvaluegenerator">
2   <min>0</min>
3   <max>100</max>
4   <key>trigger</key>
5   <delay>700</delay>
6   <lifetime>700</lifetime>
7 </port>
```

Listing 5.4: An InputPort of type `randomvaluegenerator`

The `<transformation>` element

A `<transformation>` element has only one attribute indicating its type. There are four types of Transformations possible: `empty`, `external`, `join` and `reform`. The `empty` Transformation does not apply any transformation onto the incoming DataPackages. The other three types can be configured by their own set of sub elements.

An `external` Transformation passes DataPackages to an external application. The external application can be defined by the `<command>` element. Parameters can be passed through the `<parameters>` element which consists of a set of `<parameter>` elements. The value of the `<parameter>` element may contain `%f` and `%d` markers which are substituted for the temporary DataPackage filename and directory respectively. With the `<serializer>` element, the required Serializer can be selected. The DatProNet framework contains only one predefined serializer. This `matlab` serializer can be used to convert a DataPackage to a Matlab-readable format [42]. To prevent endless waiting, a timeout can be set with the optional `<timeout>` element. The default value is set to 60 seconds. In case the timeout expires, an exception will be thrown. The exit value can be defined by the optional `<exitvalue>` element (default 0). When the current exit value differs from the configured one, an exception will be thrown.

```
1 <transformation type="external">
2   <command>matlab.exe</command>
3   <parameters>
4     <parameter>-nodesktop</parameter>
5     <parameter>-wait</parameter>
6     <parameter>-r</parameter>
7     <parameter>load('%f'); Node1_input_test =
        Node1_input_test + 1; save -v6 '%f'
        Node1_input_test; exit;</parameter>
```

```
8    </parameters>
9    <serializer>matlab</serializer>
10   <timeout>3600000</timeout>
11   <exitvalue>0</exitvalue>
12 </transformation>
```

Listing 5.5: A Transformation of type `external`

A `join` Transformation applies a cross-join operation on the incoming DataPackages. For each connected InputPort, a separate buffer is created. If a DataPackage is arriving through an InputPort, it is joined with all the DataPackages inside the other buffers. DataPackages stay in their buffer until they expire (by their lifetime) or when the buffer reach its capacity, which can be defined by the `<capacity>` element. When the maximum capacity is reached, DataPackages are removed using the first in - first out principle.

```
1  <transformation type="join">
2    <capacity>100</min>
3  </transformation>
```

Listing 5.6: A Transformation of type `join`

A `reform` Transformation can be used for adding or removing data entries to or from a DataPackage. The `reform` Transformation can be used for example to create a looping mechanism inside a data processing network. The required `<packages>` element consists of a set of `<package>` elements. A `<package>` element contains a set of `<addfield>` and `<removefield>` elements. The first element, `<addfield>`, can have two attributes: `key` and `override` (optional). The `key` attribute indicates the name of the (new) entry, and `override` indicates if an existing key may be overwritten by its new value. The `<removefield>` has only one attribute: the `key` of the entry that must be removed. If `n` `<package>` elements are defined, each incoming DataPackage will result in `n` outgoing DataPackages.

```
1  <transformation type="reform">
2    <packages>
3      <addfield key="x" override="false">0</addfield>
4      <removefield key="y" />
5    </packages>
6  </transformation>
```

Listing 5.7: A Transformation of type `reform`

The `<output>` element

The `<output>` element consists of a set of `<port>` elements. Each `<port>` element has two attributes: `name` and `type`. The `name` must be unique inside the parent `<output>` element. Five types of OutputPorts are defined: `gateway`, `external`, `file system`, `random` and `screen`. The first four types have their

own set of sub elements, the `screen` type just prints the contents of a DataPackage onto the screen and requires no further configuration.

A `gateway` OutputPort passes incoming DataPackages onto the connected, and activated, pipes. The related pipes can be defined through the `<pipes>` element. The optional activation rules can be defined through the `<conditions>` element. The `<conditions>` element consists of a set of `<condition>` elements. The `<condition>` element has one attribute indicating its name which must by unique inside its parent `<conditions>` element. The `<condition>` element contains a boolean value tag, like `<true>` or `<false>`, or a boolean determination tag, like `<match>` or `<time>`, or a boolean operator tag, like `<and>`, `<or>` or `<not>`, or a reference to another `<condition>` tag using the `<reference>` tag. The `<and>`, `<or>` and `<not>` elements behave like the boolean operations `and`, `or` and `not` and consist of at least one sub tag (boolean value, determinator, operator or reference).

The `<match>` element can be used to match a DataPackage by its contents. It can be configured by a `<key>` and/or `<value>` element. If only the `<key>` element is defined, the `<match>` element returns `true` when an entry with the value of the `<key>` element is defined inside the DataPackage. If only the `<value>` element is defined, the `<match>` element returns `true` when the DataPackage contains an entry with a value equal to the value of the `<value>` element. If both elements, `<key>` and `<value>`, are defined, the `<match>` element returns `true` if this combination exists inside the DataPackage.

Character	Description	Example
d	Day of the month, 2 digits with leading zeros	01 to 31
j	Day of the month without leading zeros	1 to 31
N	ISO-8601 numeric representation of the day of the week	1 (for Monday) through 7 (for Sunday)
z	The day of the year (starting from 0)	0 through 365
m	Numeric representation of a month, with leading zeros	01 through 12
M	Numeric representation of a month, without leading zeros	1 through 12
W	ISO-8601 week number of year, weeks starting on Monday	42 (the 42nd week in the year)
y	A two digit representation of a year	99 or 10
Y	A full numeric representation of a year, 4 digits	1999 or 2010
h	12-hour format of an hour with leading zeros	01 through 12
H	24-hour format of an hour with leading zeros	00 through 23
i	Minutes with leading zeros	00 through 59
s	Seconds with leading zeros	00 through 59

Table 5.1: Possible characters for the `<format>` element. Based on [16]

The `<time>` element has one attribute indicating its type and has two sub elements: `<format>` and `<value>`. The value of the `<format>` element can be formatted using the characters found in Table 5.1. The type is used to compare the current date, formatted by the `<format>` element, against a limit, defined by the `<value>` element. The type of comparison can be defined by the `type` attribute of the `<time>` element. There are six types possible: `less`, `lessequal`, `equal`, `greater`, `greaterequal` and `notequal`. For example, when the `less` type is selected, the `<time>` element will return `true` if the formatted date is less then the specified limit.

```
1  <port name="output" type="gateway">
2    <conditions>
3      <condition name="weekdays">
4        <time type="lessequal">
5          <format>M</format>
6          <value>5</value>
7        </time>
8      </condition>
9      <condition name="weekend">
10       <not>
11         <reference name="weekdays" />
12       </not>
13     </condition>
14   </conditions>
15   <pipes>
16     <pipe destination="Node1.input"
           condition="weekdays" type="local" />
17     <pipe destination="Node2.input"
           condition="weekend" type="tcp">
18       <portnumber>1444</portnumber>
19     </pipe>
20   </pipes>
21 </port>
```

Listing 5.8: An OutputPort of type `gateway`

The `<pipes>` element consists of a set of `<pipe>` elements. A `<pipe>` element can have three attributes: `destination`, `condition` (optional) and `type`. The `destination` attribute indicates the destination InputPort of the pipe. A destination consists of the destination filter name, followed by a dot and then the name of the InputPort. The `condition` attribute contains the name of the condition which decides if the pipe accepts DataPackages. The DatProNet framework provides two predefined pipes: `local` and `tcp`. The `local` pipe requires no further configuration but can only be used between filters on the same host. This type of communication is the most simple one, and provides the highest performance. The `tcp` pipe can be used to connect Filters on different hosts. Communication will be provided through the *TCP protocol* [48]. The port number used for this type of communication can be defined through the `<portnumber>` element.

A `random` OutputPort passes incoming DataPackages onto a number of pseudorandom selected and activated pipes. This type of OutputPort can be used for example to apply a simple way of load balancing. The configuration is completely the same as a `gateway` OutputPort except that an additional element is added: the `<select>` element, which defines the number of pipes to select.

```
1 <port name="output" type="random">
2   <pipes>
3     <pipe destination="Node1.input" type="local" />
4     <pipe destination="Node2.input" type="local" />
5     <pipe destination="Node3.input" type="local" />
6     <pipe destination="Node4.input" type="local" />
7   </pipes>
8   <select>2</select>
9 </port>
```

Listing 5.9: An OutputPort of type `random`

An `external` OutputPort passes incoming DataPackages to an external application for storage. The external application can be defined by the `<command>` element. Parameters can be passed through the `<parameters>` element, which consists of a set of `<parameter>` elements. The value of the `<parameter>` element may contain `%f` and `%d` markers which are substituted for the temporary DataPackage filename and directory respectively. With the `<serializer>` element, the required Serializer can be selected. The DatProNet framework contains only one predefined serializer. This `matlab` serializer can be used to convert a DataPackage to a Matlab-readable format [42]. To prevent endless waiting, a timeout can be set with the optional `<timeout>` element. The default value is set to 60 seconds. In case the timeout expires, an exception will be thrown. The exit value can be defined by the optional `<exitvalue>` element (default 0). When the current exit value differs from the configured one, an exception will be thrown.

```
1  <port name="output" type="external">
2    <command>matlab.exe</command>
3    <parameters>
4      <parameter>-wait</parameter>
5      <parameter>-r</parameter>
6      <parameter>load('%f');
           storeCurrentWorkspace(); exit;</parameter>
7    </parameters>
8    <serializer>matlab</serializer>
9    <timeout>3600000</timeout>
10   <exitvalue>0</exitvalue>
11 </port>
```

Listing 5.10: An OutputPort of type `external`

A `filesystem` OutputPort stores incoming DataPackages onto the file system. The `<directory>` element specifies the directory to store the packages into. For

each DataPackage a separate directory will be created. The files are stored using a Serializer, defined by the `<serializer>` element.

```
<port name="output" type="filesystem">
  <directory>C:\Storage\</directory>
  <serializer>matlab</serializer>
</port>
```

Listing 5.11: An OutputPort of type `filesystem`

5.3 Extending the DatProNet Framework

The DatProNet framework can easily be extended using XML and JAVA. Inside the installation directory of the DatProNet framework, a folder called `extensions` can be found. For each extension, a new subfolder needs to be created. The name of this folder is not important to the system. Each extension can contain multiple user-defined components. Inside this new folder, a file has to be created, called `config.xml`. This extension configuration file uses XML, and is defined by its own XML-schema which can be found in Appendix B. The configuration file consists of a root element called `<extension>`, which contains a set of `<component>` tags. Each `<component>` tag defines a single user-defined component. A `<component>` tag has only one required attribute, called `type`, with four possible values: `inputport`, `transformation`, `outputport` and `pipe`.

```
<extension>
  <component type="inputport">
    <key>temperaturesensor</key>
    <xmlschema>temperaturesensor.xml</xmlschema>
    <class>datpronet.extensions.temperaturesensor.
       TemperatureSensor</class>
  </component>
  <component type="transformation">
    <key>fahrenheittocelcius</key>
    <class>datpronet.extensions.temperaturesensor.
       FahrenheitToCelcius</class>
  </component>
  <component type="outputport">
    <key>simplemysqlinterface</key>
    <xmlschema>simplemysqlinterface.xml</xmlschema>
    <class>datpronet.extensions.temperaturesensor.
       SimpleMySQLInterface</class>
  </component>
</extension>
```

Listing 5.12: A sample extension configuration file

A `<component>` tag consists of the following sub elements: `<key>`, `<class>` and

`<schema>` (optional). The `<key>` tag is used to indicate the identifier inside an architecture description. For example, when `inputport` is selected as the component `type`, and `temperaturesensor` is used as a key, then it is possible to define an InputPort of type `temperaturesensor` inside an architectural description.

The `<class>` tag indicates the full qualified java class name, including the package name, of the user-defined component. In case of a `<component>` tag of type `pipe` this `<class>` tag consists of two sub elements: `<source>` and `<destination>`. Both elements contain the full qualified java class name of the user-defined *PipeEnd* component. User-defined classes can be created by extending one of the reusable components as described in the previous sections. The optional `<schema>` tag can be used to refer to an XML-schema for defining additional parameters (and their properties). These XML-schemas can be merged with the original DatProNet language XML-schema to validate an architectural description with user-defined components.

5.4 Using the DatProNet Framework

The data processing view of the CCES project RECORD [10] case study, as created in section 4.5, can be mapped to an architectural description using the DatProNet language. Every Filter from the data processing view is mapped onto a corresponding `<filter>` element inside the architectural description. The complete architectural description can be found in Appendix C.

The Training Data filter

The *Training Data* filter is mapped to a `<filter>` element with a single InputPort of type `filesystemlistener`. When a file is added or removes from the directory, the InputPort has to create a DataPackage with the complete training set. Furthermore, it has no transformation to perform, so an `empty` transformation is used. A single OutputPort of type `gateway` is added to pass the DataPackages to the *Merge* filter. Because the Merge filter will be deployed on the same host, a pipe of type `local` is used.

```
1  <filter name="TrainingData" runat="127.0.0.1">
2    <input>
3      <port name="input" type="filesystemlistener">
4        <key>trainingpictures</key>
5        <mode>all</mode>
6        <object>C:\TrainingData\</object>
7        <depth>1</depth>
8        <delay>700</delay>
9        <initialization>clear</initialization>
10     </port>
11   </input>
12   <transformation type="empty" />
```

```
13    <output>
14      <port name="output" type="gateway">
15        <pipes>
16          <pipe destination="Merge1.input1"
                type="local" />
17        </pipes>
18      </port>
19    </output>
20  </filter>
```

Listing 5.13: The configuration of the Training Data filter

The Settings filter

The *Settings* filter is mapped onto a `<filter>` element with a single InputPort of type `filelistener`. The InputPort has to create a DataPackage every time the file has changed. It performs no further transformation, so an `empty` transformation can be used. A single OutputPort of type `gateway` is added to pass the DataPackages to the *Merge* filter. Two pipes are added to pass the DataPackages to the two *Merge* filters, and because they will be deployed on the same host, pipes of type `local` are used.

```
1   <filter name="Settings" runat="127.0.0.1">
2     <input>
3       <port name="input" type="filesystemlistener">
4         <key>configuration</key>
5         <object>C:\configuration.m</object>
6         <delay>700</delay>
7         <initialization>clear</initialization>
8       </port>
9     </input>
10    <transformation type="empty" />
11    <output>
12      <port name="output" type="gateway">
13        <pipes>
14          <pipe destination="Merge1.input2"
                type="local" />
15          <pipe destination="Merge2.input1"
                type="local" />
16        </pipes>
17      </port>
18    </output>
19  </filter>
```

Listing 5.14: The configuration of the Settings filter

The first Merge filter

The first *Merge* filter is mapped to a `<filter>` element with two separate InputPorts of type `gateway`. Because the filter has to merge the incoming DataPackages, a transformation of type `join` is used with a capacity set to one. A single OutputPort of type `gateway` is added to pass the DataPackages to the *Training* filter. Because the Training filter will be deployed on the same host, a pipe of type `local` is used.

```
1  <filter name="Merge1" runat="127.0.0.1">
2    <input>
3      <port name="input1" type="gateway" />
4      <port name="input2" type="gateway" />
5    </input>
6    <transformation type="join">
7      <capacity>1</capacity>
8    </transformation>
9    <output>
10     <port name="output" type="gateway">
11       <pipes>
12         <pipe destination="Training.input"
               type="local"/>
13       </pipes>
14     </port>
15   </output>
16 </filter>
```

Listing 5.15: The configuration of the first Merge filter

The Training filter

The *Training* filter is mapped to a `<filter>` element with a `gateway` InputPort. Once the DataPackages are arrived, they are passed to an `external` Transformation for further processing. The internal filters, called *Reference Area* and *Pixel Correlation*, are deployed inside the Matlab application. Once a DataPackage is processed by Matlab, the trainings data is passed to an OutputPort of type `gateway`. This OutputPort passes the DataPackages to the second *Merge* filter using a pipe of type `local` (both on the same host).

```
1  <filter name="Training" runat="127.0.0.1">
2    <input>
3      <port name="input" type="gateway" />
4    </input>
5    <transformation type="external">
6      <command>matlab.exe</command>
7      <parameters>
8        <parameter>-nodesktop</parameter>
9        <parameter>-wait</parameter>
10       <parameter>-r</parameter>
```

```
11        <parameter>load('%f'); startTraining()l save
              -v6 '%f'; exit;</parameter>
12      </parameters>
13      <serializer>matlab</serializer>
14      <timeout>3600000</timeout>
15    </transformation>
16    <output>
17      <port name="output" type="gateway">
18        <pipes>
19          <pipe destination="Merge2.input2"
                type="local" />
20        </pipes>
21      </port>
22    </output>
23  </filter>
```

Listing 5.16: The configuration of the Training filter

The Acquire filter

The *Acquire* filter is mapped to a `<filter>` element with an InputPort of type `directorylistener`. Every time the camera has taken a picture, it will be put into a specific directory. The InputPort monitors this directory and when a new picture is added, a DataPackage (with the new picture) is created. There is no further transformation needed, so an `empty` transformation is used. A single OutputPort of type `gateway` is added to pass the DataPackages to the second *Merge* filter. Because the second Merge filter will be deployed on another host, a pipe of type `tcp` is used.

```
1  <filter name="Acquire" runat="10.0.0.138">
2    <input>
3      <port name="input" type="filesystemlistener">
4        <key>trainingpictures</key>
5        <mode>single</mode>
6        <object>C:\Camera\</object>
7        <depth>1</depth>
8        <delay>700</delay>
9        <initialization>clear</initialization>
10     </port>
11   </input>
12   <transformation type="empty" />
13   <output>
14     <port name="output" type="gateway">
15       <pipes>
16         <pipe destination="Merge2.input3"
               type="tcp">
17           <portnumber>1444</portnumber>
18         </pipe>
19       </pipes>
```

```
20      </port>
21    </output>
22  </filter>
```

Listing 5.17: The configuration of the Acquire filter

The second Merge filter

The second *Merge* filter is mapped to a `<filter>` element with three separate InputPorts of type `gateway`. Because the incoming DataPackages have to be merged, a `join` transformation with a capacity of one is used. A single OutputPort of type `gateway` is used to pass the DataPackages to the *Analyze* filter. Because the Analyze filter will be deployed on the same host, a pipe of type `local` is used.

```
1  <filter name="Merge2" runat="127.0.0.1">
2    <input>
3      <port name="input1" type="gateway" />
4      <port name="input2" type="gateway" />
5      <port name="input3" type="gateway" />
6    </input>
7    <transformation type="join">
8      <capacity>1</capacity>
9    </transformation>
10   <output>
11     <port name="output" type="gateway">
12       <pipes>
13         <pipe destination="Analyze.input"
               type="local"/>
14       </pipes>
15     </port>
16   </output>
17 </filter>
```

Listing 5.18: The configuration of the second Merge filter

The Analyze filter

The *Analyze* filter is mapped to a `<filter>` element with a single InputPort of type `gateway`. Once the DataPackages are arrived through the InputPort, they are put to the Matlab application for further processing. The resulting DataPackages are passed to an OutputPort of type `gateway`. This OutputPort passes the DataPackages to the *Store* filter. The Store filter will be deployed on the same host, thus a pipe of type `local` is used.

```
1  <filter name="Analyze" runat="127.0.0.1">
2    <input>
3      <port name="input" type="gateway" />
```

```
4   </input>
5   <transformation type="external">
6     <command>matlab.exe</command>
7     <parameters>
8       <parameter>-nodesktop</parameter>
9       <parameter>-wait</parameter>
10      <parameter>-r</parameter>
11      <parameter>load('%f'); analyzePicture();
            save -v6 '%f'; exit;</parameter>
12    </parameters>
13    <serializer>matlab</serializer>
14    <timeout>3600000</timeout>
15  </transformation>
16  <output>
17    <port name="output" type="gateway">
18      <pipes>
19        <pipe destination="Store.input"
              type="local" />
20      </pipes>
21    </port>
22  </output>
23 </filter>
```

Listing 5.19: The configuration of the Analyze filter

The Store filter

The *Store* filter is mapped to a `<filter>` element with a single InputPort of type `gateway`. No transformation is needed, thus an `empty` transformation is used. A single OutputPort of type `filesystem` is used to store the DataPackages on the file system.

```
1  <filter name="Store" runat="127.0.0.1">
2    <input>
3      <port name="input" type="gateway" />
4    </input>
5    <transformation type="empty" />
6    <output>
7      <port name="output" type="filesystem">
8        <directory>C:\Storage\</directory>
9        <serializer>matlab</serializer>
10     </port>
11   </output>
12 </filter>
```

Listing 5.20: The configuration of the Store filter

5.5 Discussion

The definition of pipes

During the design of the architecture description language there was discussion about where to put the definition of pipes. The alternative would be to bundle the definition of pipes, which can be an advantage, within a `<pipes>` tag beneath the `<filters>` tag. In this case, both ends of the pipe should be specified and it is harder to see which pipes are connected through a specific output gateway. Another disadvantage would be that the definition of a pipe is not close to its starting point, and therefore less intuitive.

5.6 Related Work

In [31] a highly-extensible architecture description language for software and systems is proposed, called xADL, which is defined by a set of XML schemas. Although xADL provides a large and extensive framework, it is not used within this project because it would require to much specific knowledge from the user in order to define an architectural description. The possibility of converting an architectural description based upon the DatProNet language to xADL is considered as future work.

A data stream management system called Aurora [4] has been developed for monitoring, filtering and/or processing of data from numerous streams. Through a graphical user-interface, queries can be build using a small set of operators. Medusa [54] can be used to add distribution functionality. A new distributed stream processing engine, called Borealis [1], is developed which uses the Aurora system in conjunction with Medusa. Borealis address the problem of dynamic revision of query results and dynamic query modification.

5.7 Conclusions

In chapter 5 the DatProNet framework is introduced together with an architecture description language. A data processing view can be mapped to an architectural description using the DatProNet language. Based on a valid architectural description, the framework can generate a concrete implementation.

The DatProNet framework offers various ways to interoperate with external systems. Serializers can be used to convert data packages to almost any format needed. Throughout the framework an acknowledge technique is used to provide reliable data processing networks. Input ports, transformations and output ports are all equipped with a simple recovery technique. If for example a node is (temporarily) unavailable, the other (connected) nodes will send retrying message within a predefined interval. After a certain number of attempts, the nodes will stop trying and inform the user about the current problem. The DatProNet framework is completely implemented in JAVA, which implies a high portability. The DatProNet framework requires no difficult or complex installation, just

copy the files and compile your architectural description. Even though the extension module is not yet available inside the prototype, the user can develop their own input ports, transformations or output ports by implementing the selected component.

Chapter 6

Evaluation

In the previous chapter we proposed the DatProNet framework to support the development of data processing networks by people with limited computer science knowledge. In this chapter an evaluation of the DatProNet framework is presented, based on the requirements as stated in section 5.1. In chapter 2, we proposed an algorithm for the segmentation of aerial photos using a water and non-water class. Using the DatProNet framework, this algorithm can be implemented into a data processing network. The performance of the segmentation algorithm is evaluated by a small and a larger set of aerial photos from our CCES project RECORD [10] case study.

This chapter is organized as follows. Section 6.1 presents an evaluation of the DatProNet framework. An evaluation of the segmenetation algoritm is presented in section 6.2. A conclusion is given after discussing the alternatives in section 6.3.

6.1 DatProNet framework

Empirical research is often an important aspect of the evaluation of a framework. Unfortunately, such material is not yet available for the DatProNet framework, therefore the basis requirements from section 5.1 are used as a guideline for the evaluation.

Reliability

Throughout the framework an acknowledge technique is used to provide a reliable data processing network. Local buffers are used to ensure that data packages are not lost within a single filter. Data packages are only removed from these buffers if an acknowledgement is received from the connected party. These acknowledgements are only send if the data packages are stored successfully in the local buffer of the connected party. A data processing network created with the DatProNet framework can interoperate with other external systems, but the reliability is only guaranteed for the standard (internal) components. Develop-

ers of customized input ports, transformation or output ports should take there own precautions to provide a reliable behaviour.

Ease of use

The complete life cycle of the development of a data processing network is provided by the DatProNet framework. A basic model is provided to increase the overall understandability of data processing networks by people with limited computer science knowledge. Depending on the required technical details, the specialized model can be used.

The design of a data processing network can be documented by a data processing view using the data processing style or the more specialized DatProNet style. Both styles use the same terminology as the models presented in chapter 3. A data processing view can be easily mapped onto a architectural description, using the DatProNet language. This architecture description language is XML-based and thus strongly structured. Using the provided XML-schema, architectural descriptions can be easily validated using the XML-editor of your choice. For the creating of an architectural description, no specific computer science knowledge is required, only the format and requirements of the DatProNet language.

Once a valid architectural description is created, the framework can create fully automatically the concrete implementation of the data processing network. During the validation of the architectural description, all the requirements are checked by the compiler. In case of an error, the user is presented with a helpful error message. The framework requires no complex installation, it only needs the JAVA environment to be available. The deployment packages generated by the framework can directly, and only, be used on the nodes specified inside the architectural description. Deployment mistakes are not possible because a deployment package can only be deployed on the node it is compiled for.

Interoperability

The DatProNet framework offers various ways to interoperate with external systems. The communication can be directional (e.g. external data sink), or bi-directional (e.g. external transformation). Serializers can be used to convert data packages to almost any format needed. The DatProNet framework contains just one predefined serialized, which can be used to convert data packages to a Matlab-readable format, more serializers can be developed by implementing the Serializer class. The predefined external transformation and external data sink components are using a timeout and exit code validator to monitor the communication with an external system. If case of any problems, the user gets warned through warnings or exceptions.

Portability

The DatProNet framework is completely implemented in JAVA, which implies a high portability. The DatProNet framework requires no difficult or complex

installation, just copy the files and compile your architectural description. To prevent errors, deployment packages generated by the DatProNet framework can only be deployed on the machines they are specified for.

Recoverability

Input ports, transformations and output ports are all equipped with a simple recovery technique. If a filter or node goes down, or if one of the components suffers from a software error, the node will be automatically recovered without loosing a single data package. If a node is (temporarily) unavailable, the other (connected) nodes will send retrying message within a predefined interval. After a certain number of attempts, the nodes will stop trying and inform the user about the current problem.

Extensibility

Even though the extension module is not yet available inside the prototype, the user can develop their own input ports, transformations or output ports by implementing the related component. Without using the extension module, as described in section 5.3, the user should implement the desired class, modify the DatProNet language and recompile the complete framework. With the extension module, it is much easier to extend the various components. By placing the user defined extensions inside a separate directory, extensions can be easily added and removed without recompiling the complete framework.

6.2 Segmentation Algorithm

In this section the segmentation algorithm, as presented in chapter 2, is evaluated by a small and large set of aerial photos from the CCES project RECORD [10] case study. In order to evaluate these results, a clear model for measuring the segmentation quality is needed.

6.2.1 Performance Measurement

For the evaluation of the segmentation algorithm, as presented in chapter 2, we need to define a model for performance measuring. Inspired by the information retrieval field, two statistical measurements properties are selected: precision and recall [47]. In the field of information retrieval, precision is defined as the fraction of retrieved documents which are relevant to the search [47]. Recall is defined as the fraction of all the relevant documents which are successfully received [47]. Below their formal definitions are given (taken from [47]).

$$Precision = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|retrieved\ documents|}$$

$$Recall = \frac{|\{relevant\ documents\} \cap \{retrieved\ documents\}|}{|relevant\ documents|}$$

Unfortunately, these definitions cannot be used directly because we need to measure the performance in a classification context. We can modify these definitions by using the type I and type II errors from statistical hypothesis testing: false positive and false negative [49]. Below the modified definitions of precision and recall using type I and type II errors are given (taken from [49]).

$$Precision = \frac{tp}{tp + fp} \qquad Recall = \frac{tp}{tp + fn}$$

$$tp = water\ is\ classified\ as\ water$$

$$fn = water\ is\ classified\ as\ non\text{-}water$$

$$fp = non\text{-}water\ is\ classified\ as\ water$$

$$tn = non\text{-}water\ is\ classified\ as\ non\text{-}water$$

In the context of our case study, a precision score of 1 means that all the classified water is indeed water (exactness). On the other hand, a recall score of 1 means that all the water is classified correctly (completeness). For the CCES project RECORD [10] case study precision and recall are equally important so for the evaluation the weighted harmonic mean of precision and recall is used (also known as the F-measurement [47]).

$$Performance = 2 \times \frac{precision \times recall}{precision + recall}$$

The performance of the classification is of course the most important part of the evaluation. On the other hand we are interested to see how much time is saved by using our segmentation algorithm. The processing time of our algorithm is highly depending on the hard- and software it is running on, but it is still interesting to see if automatically segmentation is faster and if so, how much. We use the definition below for measuring the amount of time saved by using our segmentation algorithm.

$$Time\ saved = 1 - \frac{Ax + By + C}{A(x + y)}$$

$$x = size\ of\ training\ set$$

$$y = size\ of\ test\ set$$

$$A = average\ time\ needed\ for\ manual\ segmenation\ of\ a\ picture$$

$$B = average\ time\ needed\ by\ the\ algoritm\ for\ the\ segmenation\ of\ a\ picture$$

$$C = time\ needed\ for\ the\ traning\ of\ the\ algorithm$$

A negative result means that the use of our algorithm has increased the time needed for the segmentation of all the photos. A positive result means that we saved a fraction of our time by using the segmentation algorithm.

6.2.2 Qualitative Evaluation

For a qualitative evaluation of our segmentation algorithm, a set of 75 pictures is selected from the upstream camera. All these pictures, with a resolution of 2144 (width) by 1424 (height) pixels, are manually segmented, which took approximately 15 minutes per picture. Depending on the desired precision this time can be lowered. A training set of 15 pictures is created. The selection of these pictures is based on a maximum diversity with respect to the water level.

The architectural description, as created in chapter 5, is now used to generate the required deployment packages. In this evaluation, all the filters are deployed on a single workstation with an Intel Pentium Core2Duo E8400 processor and 3.5GB of internal memory running Microsoft Windows XP Home with service pack 3. After installing the segmentation algorithm, the data processing network is started. For the segmentation algorithm, Matlab R2010b is used. The training, based on these 15 pictures, took less then 10 minutes.

After the training phase, the actual segmentation can be done. The pictures are now one-by-one segmented using a water and non-water class. The average processing time for a picture turns out to be approximately 30 seconds. Once all the pictures are segmented, the results are compared with our manual segmented versions. The resulting performance can be found in figure 6.1.

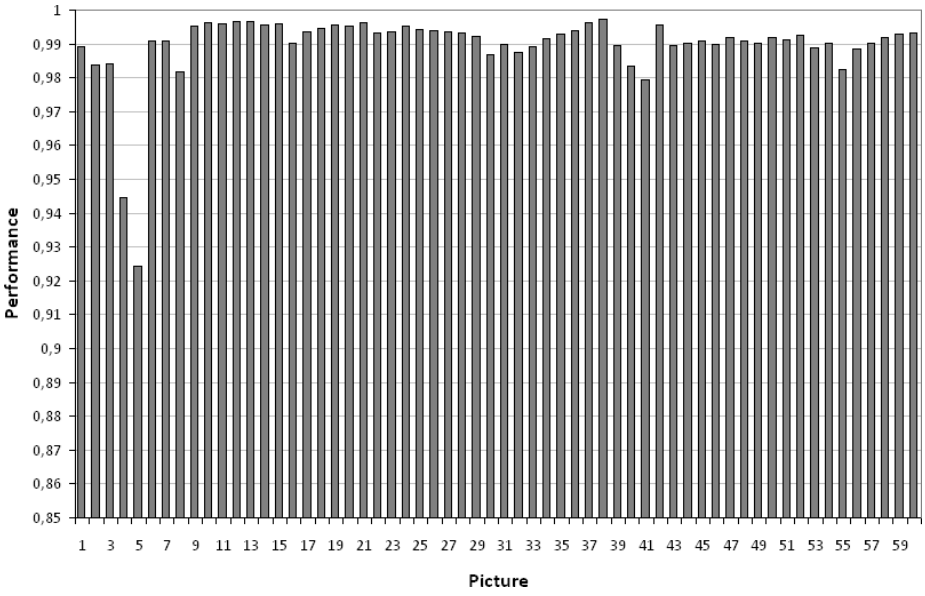

Figure 6.1: Performance of the segmentationg algorithm on a test set of 60 pictures using a training set of 15 pictures

The results are quite promising, and will now be evaluated by discussing the worst case, an average case and of course the best case. For each case we show the results as an overlay on the picture. Blue indicates the water detected by our segmentation algorithm, red indicates the classifications errors (false negatives and false positives) . The overlay is made transparent, so it is possible to see what is behind.

Figure 6.2: Worst case: picture #5

The picture with the lowest performance can be found in figure 6.2. The precision could not be better, but the recall score leaves much to be desired. The photo suffers from a relatively large colour overlap between the sandbank and the surrounding water. Furthermore, classification errors can be found around the trees and bushes. Classification of these pixels is hard because there is no clear boundary between the water and for example the trees. These errors can also be caused by human classification faults inside the reference picture.

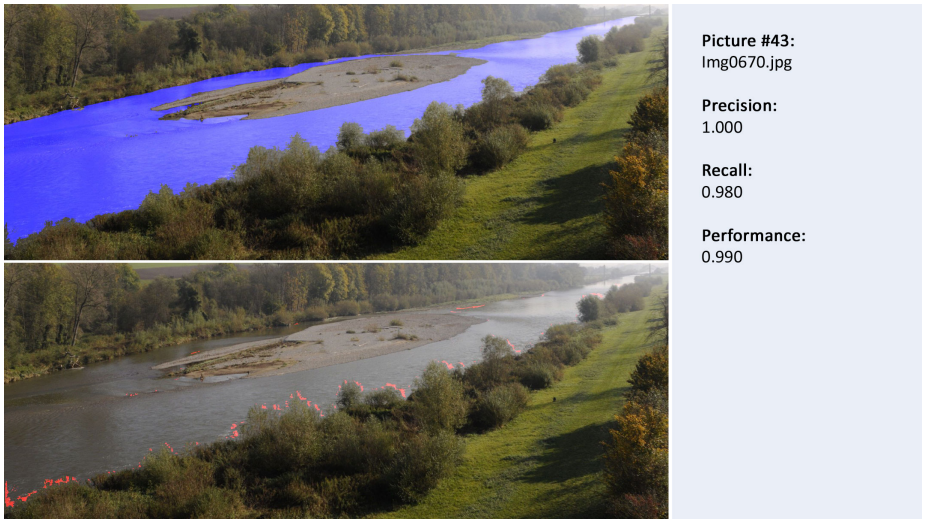

Figure 6.3: Average case: picture #43

A picture with an average performance can be found in figure 6.3 . The result is quite nice, especially if we look to the colour overlap between the sandbank

and the surrounding water. The areas around the trees and bushes are also this time a problem. The surrounding nature is constantly changing, which causes the pixel correlation step to be less useful in the areas around trees and bushes.

Figure 6.4: Best case: picture #38

The picture with the highest performance can be found in figure 6.4. The segmentation result is pretty good, even at the problem areas like the sandbank. The high performance is a result of the high contrast between the water and the surrounding nature. Furthermore, the sandbank does not contain a colour overlap with the surrounding water, which makes the classification more easier.

The manual segmentation of the complete set of pictures, 75 in total, took almost 20 hours. By using our segmentation algorithm, we completed the task in less than 4½ hours, which means a saving of more than 75 percent!

6.2.3 Evaluation by Example

After a qualitative evaluation, we are interested to see how the segmentation algorithm performs on a set of more then 1500 pictures taken by the upstream camera. For the training set, 35 pictures are selected. Because the pictures where taken during a period of more than a year, the training set contained a lot of different scenarios. Once these pictures where manually segmented, the training of the algorithm was started.

Due to the large test set, it was not possible to create a reference set of these pictures. Therefore these results are not evaluated by exact performance, but on a more high level scale. Just 6 pictures, with mixed results, are selected to get a good overview of the overall performance. Figure 6.5 shows some results of the segmentation process as an overlay on the original pictures. Blue indicates the water detected by our segmentation algorithm. The overlay is made transparent, so it is possible to see what is behind.

Figure 6.5: Sample results from an evaluation based on more than 1500 pictures

It is clear that these results are not as good as the ones from the qualitative evaluation. All pictures suffer from a low recall. The precession on the other hand is relatively high, except for the pictures with a flood. The problems are highly related to the fact that the pictures are not stabilized before they are processed. Due to camera movements, there can be a huge difference between two consecutive pictures. This problem causes the minimal-water-area to be relatively small. Therefore it will be harder for the algorithm to collect sufficient reference information during the classification. Another problem is that the pixel correlation step is less useful with such a large and diverse training set. There is just to much difference between the training pictures. To improve these results, the collection can be divided into a number of subsets which will be processed separately (together with their own training set).

6.3 Discussion

Manual segmentation errors

The results of the segmentation algorithm evaluation are highly depending on the manually segmented reference pictures. A manual classification fault, can result in a lower recall and precision score during the evaluation. During the segmentation of the training pictures, users are facing the same segmentation problems, as described in chapter 2. Take for example a picture with a large black spot around the bushes, caused by a shadow. Inside this shadow, it is hard to see which pixels belongs to the water and which not. To partially overcome the problem of human segmentation errors, a don't care border is used around the classified water. The results within this area are ignored during the evaluation of the pictures. During the qualitative evaluation, a don't care border of 8 pixels is used.

6.4 Conclusions

In this chapter, we have performed an evaluation of the DatProNet framework, based on the basic requirements from section 5.1. A performance measurement model is introduced for the evaluation of the segmentation algorithm. A qualitative evaluation of the segmentation algorithm is performed using a set of 75 photos from our CCES project RECORD [10] case study. Furthermore, an evaluation by example is performed using a set of 1500 pictures, covering a period of more than a year. These results of the qualitative evaluation where promising, nevertheless the results of the second evaluation where a bit disappointing. In a rapidly changing environment it is better to divided the collection of pictures into smaller subsets (with their on training set) otherwise the minimal-water-area, and the pixel correlation step will be less useful. A stabilization step can be introduced for further optimization.

Chapter 7

Conclusions

Data processing networks are not only used by computer science people, nowadays researchers in all kinds of research topics are processing large amounts of data. Considering the required knowledge and amount of time, the development of a distributed and/or (de)centralized data processing network is often not an option for them. Sometimes they create scripts for the various processing steps, which are manually executed step-by-step. It is clear that such a procedure is far from optimal, especially in a streaming data environment. Due to the popularity of the internet, distributed data or even decentralized processing is used more often. By introducing decentralization or distributed data, the complexity of the overall processing network is increased dramatically. Both aspects require some overhead which lowers the understandability of the data processing network in general. To solve these problems, methods and techniques are introduced to support the development of a data processing network by people with limited computer science knowledge.

7.1 Problems

In this thesis, we have addressed the following problems related to the development of data processing network by people with limited computer science knowledge.

- *Analyzing data processing networks*: Since implementing a data processing network is a costly-process, it is important that the users have a good understanding of the general structure. A basic, and easy understandable, model should be created to overcome a potential knowledge gap between the users and developers. A more detailed model is needed to provided a more detailed view, with respect to implementation and platform alternatives, on the design of a data processing network.
- *Documenting a data processing network design*: Once a basic and a more specialized model is created, a special data processing view can be added to an architectural description of a data processing network. A data pro-

cessing view will increase the understandability of the structure and can be used to communicate architectural design decisions with respect to data processing. A practical and easy-to-use method is needed to create such a data processing view.

- *Realization of data processing networks*: The implementation of a data processing network is not always trivial and requires a substantial development and maintenance effort, especially for people with limited computer science knowledge. Developers need to be supported for the implementation of a data processing network.

7.2 Solutions

In this section, an outline of our solutions to the addressed problems can be found. Each solution addresses a specific step inside the development life cycle of a data processing network, from domain analysis to the actual implementation.

7.2.1 Analyzing Data Processing Networks

A basic model for data processing networks is presented in chapter 3. A filter which is deployed on a node, consists of one or more input ports, a transformation and one or more output ports. An input port can be a data source, which originates and provides data, or an input gateway, which provides an entrance to a filter. An output port can be a data sink, which stores data for further use, or an output gateway which provides an entrance to a pipe. Nodes can be connected to other nodes by communication channels. A pipe transmits data packages from an output gateway to an input gateway using a communication channel.

A specialized model is presented to provide a more detailed representation, with respect to implementation and platform alternatives. This specialized model contains specializations of a data source and data sink. A data source can be a file system listener, which monitors the file system for changes, a random value generator, which can be used for debugging purposes or as a trigger for a specific filter or a parameter prompt, which asks the user to fill-in some constants and put them into a single data package. A data sink can be a screen, which prints the contents onto the screen, a file system storage, which stores the data packages on the file system, or an external storage, which can be used for transmitting data packages to an external storage system like a DBMS.

7.2.2 Documenting Data Processing Networks

In chapter 4 a data processing style is introduced for documenting a data processing network. The data processing style is a specialization of the pipe-and-filter style as defined in the V&B approach [7]. The DatProNet style is introduced in which the elements and relations of the data processing style are

further specialized. The usage of these styles has been illustrated by defining a data processing view for the CCES project RECORD [10] case study.

7.2.3 Realization of Data Processing Networks

In chapter 5 the DatProNet framework is introduced along with an architecture description language. A data processing view can be mapped to an architecture description using the DatProNet language. This architecture description can be put into the DatProNet framework which then creates the various deployment packages.

Because the complete development life cycle of data processing networks is covered by the DatProNet framework, the development and maintenance effort is further reduced and the ease of use increased. XML-editors can be used for the creation, and validation, of architectural descriptions. Based on a valid architectural description, the framework can generate a concrete implementation. During the validation of the architectural description, all the requirements are checked by the compiler. In case of an error, the user is presented with a helpful error message.

The DatProNet framework offers various ways to interoperate with external systems. Serializers can be used to convert data packages to almost any format needed. Throughout the framework an acknowledge technique is used to provide reliable data processing networks. Input ports, transformations and output ports are all equipped with a simple recovery technique. If for example a node is (temporarily) unavailable, the other (connected) nodes will send retrying message within a predefined interval. After a certain number of attempts, the nodes will stop trying and inform the user about the current problem. The DatProNet framework is completely implemented in JAVA, which implies a high portability. The DatProNet framework requires no difficult or complex installation, just copy the files and compile your architectural description. Even though the extension module is not yet available inside the prototype, the user can develop their own input ports, transformations or output ports by implementing the selected component.

7.3 Future Work

In the following, we provide several future directions regarding to the methods and techniques proposed in this thesis.

The DatProNet framework requires an architectural description language for creating deployment packages. An improvement would be to develop some-kind of graphical user-interface which can be used to create architectural descriptions though a drag-and-drop interface. With such kind of application, people with limited computer science knowledge are further supported during the development of a data processing network.

A centralized extension repository can be developed to support the exchange of extensions and to reduce the implementation effort. Through an central

system, for example a website, information about the various extensions could be distributed. By sharing information or extensions, errors can be prevented and more users could be attracted

Static analysis can support the user by the development of a faultless data processing network. Static analysis contains at least a verification of the topology requirements as stated in chapter 3 and 4. But even after performing this verification, it is still possible to develop a data processing network containing one or more faults. For example, if a data processing view contains a loop, it is impossible to verify (with static analysis) that a specific data package will arrive at a data sink eventually. With dynamic analysis more verification can be done. For example, a verification of the performance properties of input ports (together with a suggestion of their optimal values).

xADL [31] is a highly-extensible architecture description language for software systems, defined by a set of XML schemas. Because xADL provides a large and extensive framework, it would be nice to have the possibility to convert an architecture description, created with the DatProNet language, to xADL.

Appendix A

XML-Schema of the DatProNet Language

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/
       XMLSchema">
3
4  <xs:simpleType name="nametype">
5    <xs:restriction base="xs:string">
6      <xs:pattern value="[a-zA-Z0-9_-]+"/>
7    </xs:restriction>
8  </xs:simpleType>
9
10 <xs:simpleType name="dateexpressiontype">
11   <xs:restriction base="xs:string">
12     <xs:pattern value="[djNzmMWyYhHiIsS]+"/>
13   </xs:restriction>
14 </xs:simpleType>
15
16 <xs:simpleType name="booleantype">
17   <xs:restriction base="xs:string">
18     <xs:pattern value="[true|false]"/>
19   </xs:restriction>
20 </xs:simpleType>
21
22 <xs:simpleType name="emailaddresstype">
23   <xs:restriction base="xs:string">
24     <xs:pattern value="[A-Za-z0-9_]+([-+.'][A-Za-z0
         -9_]+)*@[A-Za-z0-9_]+([-.][A-Za-z0-9_]+)*
         \.[A-Za-z0-9_]+([-.][A-Za-z0-9_]+)*"/>
25   </xs:restriction>
26 </xs:simpleType>
27
28 <xs:simpleType name="ipaddresstype">
```

```
29    <xs:restriction base="xs:string">
30       <xs:pattern value
            ="((1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5]).){3}
31       (1?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])"/>
32    </xs:restriction>
33  </xs:simpleType>
34
35  <xs:element name="exceptionHandlerCriteria">
36    <xs:sequence>
37      <xs:element name="period" type="xs:
           positiveInteger" minOccurs="1" maxOccurs
           ="1"/>
38      <xs:element name="max" type="xs:positiveInteger
           " minOccurs="1" maxOccurs="1"/>
39      <xs:element name="timeout" type="xs:
           positiveInteger" minOccurs="1" maxOccurs
           ="1"/>
40    </xs:sequence>
41  </xs:element>
42
43  <xs:group name="emailExceptionHandlerParameters">
44    <xs:sequence>
45      <xs:element name="from" type="emailaddresstype"
            minOccurs="1" maxOccurs="1"/>
46      <xs:element name="to" type="emailaddresstype"
           minOccurs="1" maxOccurs="1"/>
47      <xs:element name="smtp" type="ipaddresstype"
           minOccurs="1" maxOccurs="1"/>
48      <xs:element name="subject" type="xs:string"
           minOccurs="0" maxOccurs="1"/>
49      <xs:element ref="exceptionHandlerCriteria"
           minOccurs="0" maxOccurs="1"/>
50    </xs:sequence>
51  </xs:group>
52
53  <xs:group name="logfileExceptionHandlerParameters">
54    <xs:sequence>
55      <xs:element name="file" type="xs:string"
           minOccurs="1" maxOccurs="1"/>
56      <xs:element ref="exceptionHandlerCriteria"
           minOccurs="0" maxOccurs="1"/>
57    </xs:sequence>
58  </xs:group>
59
60  <xs:group name="screenExceptionHandlerParameters">
61    <xs:element ref="exceptionHandlerCriteria"/>
62  </xs:group>
63
64  <xs:complexType name="exceptionhandler">
65    <xs:choice>
```

```
66      <xs:group ref="emailExceptionHandlerParameters
           "/>
67      <xs:group ref="
           logfileExceptionHandlerParameters"/>
68      <xs:group ref="screenExceptionHandlerParameters
           "/>
69    </xs:choice>
70    <xs:attribute name="name" type="nametype" use="
         required"/>
71    <xs:attribute name="type" use="required">
72      <xs:simpleType>
73        <xs:restriction base="xs:string">
74          <xs:enumeration value="email"/>
75          <xs:enumeration value="logfile"/>
76          <xs:enumeration value="screen"/>
77        </xs:restriction>
78      </xs:simpleType>
79    </xs:attribute>
80  </xs:complexType>
81
82  <xs:group name="fstInputPortParameters">
83    <xs:sequence>
84      <xs:element name="object" type="xs:string"
           minOccurs="1" maxOccurs="1"/>
85      <xs:element name="key" type="nametype"
           minOccurs="1" maxOccurs="1"/>
86      <xs:element name="depth" type="xs:
           positiveInteger" minOccurs="0" maxOccurs
           ="1"/>
87      <xs:element name="initialization" minOccurs="0"
            maxOccurs="1">
88        <xs:simpleType>
89          <xs:restriction base="xs:string">
90            <xs:enumeration value="clear"/>
91            <xs:enumeration value="current"/>
92          </xs:restriction>
93        </xs:simpleType>
94      </xs:element>
95      <xs:element name="mode" minOccurs="1" maxOccurs
           ="1">
96        <xs:simpleType>
97          <xs:restriction base="xs:string">
98            <xs:enumeration value="all"/>
99            <xs:enumeration value="collection"/>
100           <xs:enumeration value="single"/>
101         </xs:restriction>
102       </xs:simpleType>
103     </xs:element>
104     <xs:element name="delay" type="xs:
           positiveInteger" minOccurs="0" maxOccurs
```

```
        ="1"/>
105     <xs:element name="lifetime" type="xs:
           positiveInteger" minOccurs="0" maxOccurs
           ="1"/>
106   </xs:sequence>
107 </xs:group>
108
109 <xs:group name="ppInputPortParameters">
110   <xs:sequence>
111     <xs:element name="parameters" minOccurs="1"
           maxOccurs="1">
112       <xs:complexType>
113         <xs:element name="parameter" minOccurs="1"
              maxOccurs="unbounded">
114           <xs:complexType>
115             <xs:sequence>
116               <xs:element name="key" type="nametype
                     " minOccurs="1" maxOccurs="1"/>
117               <xs:element name="title" type="xs:
                     string" minOccurs="1" maxOccurs
                     ="1"/>
118               <xs:element name="type" minOccurs="1"
                      maxOccurs="1">
119                 <xs:simpleType>
120                   <xs:restriction base="xs:string">
121                     <xs:enumeration value="boolean
                           "/>
122                     <xs:enumeration value="file"/>
123                     <xs:enumeration value="float"/>
124                     <xs:enumeration value="integer
                           "/>
125                     <xs:enumeration value="string
                           "/>
126                   </xs:restriction>
127                 </xs:simpleType>
128               </xs:element>
129             </xs:sequence>
130           </xs:complexType>
131         </xs:element>
132       </xs:complexType>
133     </xs:element>
134   </xs:sequence>
135 </xs:group>
136
137 <xs:group name="rvgInputPortParameters">
138   <xs:sequence>
139     <xs:element name="min" type="xs:positiveInteger
           " minOccurs="1" maxOccurs="1"/>
140     <xs:element name="max" type="xs:positiveInteger
           " minOccurs="1" maxOccurs="1"/>
```

```
141     <xs:element name="key" type="nametype"
            minOccurs="1" maxOccurs="1"/>
142     <xs:element name="delay" type="xs:
            positiveInteger" minOccurs="1" maxOccurs
            ="1"/>
143     <xs:element name="lifetime" type="xs:
            positiveInteger" minOccurs="0" maxOccurs
            ="1"/>
144   </xs:sequence>
145 </xs:group>
146
147 <xs:group name="externalTransformationParameters">
148   <xs:sequence>
149     <xs:element name="command" type="xs:string"
            minOccurs="1" maxOccurs="1"/>
150     <xs:element name="serializer" minOccurs="1"
            maxOccurs="1">
151       <xs:simpleType>
152         <xs:restriction base="xs:string">
153           <xs:enumeration value="matlab"/>
154         </xs:restriction>
155       </xs:simpleType>
156     </xs:element>
157     <xs:element name="parameters" minOccurs="0"
            maxOccurs="1">
158       <xs:complexType>
159         <xs:element name="parameter" type="xs:
                string" minOccurs="0" maxOccurs="
                unbounded"/>
160       </xs:complexType>
161     </xs:element>
162     <xs:element name="timeout" type="xs:
            positiveInteger" minOccurs="0" maxOccurs
            ="1"/>
163     <xs:element name="exitvalue" type="xs:
            positiveInteger" minOccurs="0" maxOccurs
            ="1"/>
164   </xs:sequence>
165 </xs:group>
166
167 <xs:group name="joinTransformationParameters">
168   <xs:sequence>
169     <xs:element name="capacity" type="xs:
            positiveInteger" minOccurs="1" maxOccurs
            ="1"/>
170   </xs:sequence>
171 </xs:group>
172
173 <xs:group name="reformTransformationParameters">
174   <xs:sequence>
```

```
175     <xs:element name="packages" minOccurs="1"
           maxOccurs="1">
176       <xs:complexType>
177         <xs:element name="package" minOccurs="1"
              maxOccurs="unbounded">
178           <xs:complexType>
179             <xs:all>
180               <xs:element name="addfield" minOccurs
                    ="0" maxOccurs="unbounded">
181                 <xs:complexType>
182                   <xs:simpleContent>
183                     <xs:extension base="xs:string">
184                       <xs:attribute name="key" type
                            ="nametype" use="required
                            "/>
185                       <xs:attribute name="override"
                             type="booleantype"/>
186                     </xs:extension>
187                   </xs:simpleContent>
188                 </xs:complexType>
189               </xs:element>
190               <xs:element name="removefield"
                    minOccurs="0" maxOccurs="unbounded
                    ">
191                 <xs:complexType>
192                   <xs:attribute name="key" type="
                        nametype" use="required"/>
193                 </xs:complexType>
194               </xs:element>
195             </xs:all>
196           </xs:complexType>
197         </xs:element>
198       </xs:complexType>
199     </xs:element>
200   </xs:sequence>
201 </xs:group>
202
203 <xs:group name="expressions">
204   <xs:choice>
205     <xs:element name="and" type="
           multipleExpressionType"/>
206     <xs:element name="condition">
207       <xs:complexType>
208         <xs:attribute name="name" type="nametype"/>
209       </xs:complexType>
210     </xs:element>
211     <xs:element name="false"/>
212     <xs:element name="match">
213       <xs:complexType>
214         <xs:choice minOccurs="1" maxOccurs="2">
```

```
215           <xs:element name="key" type="nametype"
                 minOccurs="0" maxOccurs="1"/>
216           <xs:element name="value" type="xs:string"
                  minOccurs="0" maxOccurs="1"/>
217         </xs:sequence>
218       </xs:complexType>
219     </xs:element>
220     <xs:element name="not" type="
           singleExpressionType"/>
221     <xs:element name="or" type="
           multipleExpressionType"/>
222     <xs:element name="time">
223       <xs:complexType>
224         <xs:choice minOccurs="1" maxOccurs="2">
225           <xs:element name="format" type="
                 dateexpressiontype"/>
226           <xs:element name="value" type="xs:
                 positiveInteger"/>
227         </xs:sequence>
228         <xs:attribute name="type" use="required">
229           <xs:simpleType>
230             <xs:restriction base="xs:string">
231               <xs:enumeration value="less"/>
232               <xs:enumeration value="lessequal"/>
233               <xs:enumeration value="equal"/>
234               <xs:enumeration value="greater"/>
235               <xs:enumeration value="greaterequal
                     "/>
236               <xs:enumeration value="notequal"/>
237             </xs:restriction>
238           </xs:simpleType>
239         </xs:attribute>
240       </xs:complexType>
241     </xs:element>
242   </xs:choice>
243 </xs:group>
244
245 <xs:complexType name="multipleExpressionType">
246   <xs:sequence>
247     <xs:group ref="expressions" minOccurs="2"
           maxOccurs="unbounded"/>
248   </xs:sequence>
249 </xs:complexType>
250
251 <xs:complexType name="singleExpressionType">
252   <xs:choice>
253     <xs:group ref="expressions"/>
254   </xs:choice>
255 </xs:complexType>
256
```

```
257 <xs:complexType name="conditions">
258   <xs:sequence>
259     <xs:element name="condition" minOccurs="0"
          maxOccurs="unbounded">
260       <xs:complexType>
261         <xs:choice>
262           <xs:group ref="expressions"/>
263         </xs:choice>
264         <xs:attribute name="name" type="nametype"
            use="required"/>
265       </xs:complexType>
266     </xs:element>
267   </xs:sequence>
268 </xs:complexType>
269
270 <xs:complexType name="pipes">
271   <xs:sequence>
272     <xs:element name="pipe" minOccurs="1" maxOccurs
          ="unbounded">
273       <xs:complexType>
274         <xs:attribute name="destination" type="xs:
            string" use="required"/>
275         <xs:attribute name="condition" type="xs:
            string"/>
276         <xs:attribute name="type" use="required">
277           <xs:simpleType>
278             <xs:restriction base="xs:string">
279               <xs:enumeration value="local"/>
280               <xs:enumeration value="tcp"/>
281             </xs:restriction>
282           </xs:simpleType>
283         </xs:attribute>
284       </xs:complexType>
285     </xs:element>
286   </xs:sequence>
287 </xs:complexType>
288
289 <xs:group name="basicOutputPortParameters">
290   <xs:sequence>
291     <xs:element ref="conditions" minOccurs="0"
          maxOccurs="1"/>
292     <xs:element ref="pipes" minOccurs="1" maxOccurs
          ="1"/>
293   </xs:sequence>
294 </xs:group>
295
296 <xs:group name="externalOutputPortParameters">
297   <xs:sequence>
298     <xs:element name="command" type="xs:string"
          minOccurs="1" maxOccurs="1"/>
```

```
299       <xs:element name="serializer" minOccurs="1"
              maxOccurs="1">
300         <xs:simpleType>
301           <xs:restriction base="xs:string">
302             <xs:enumeration value="matlab"/>
303           </xs:restriction>
304         </xs:simpleType>
305       </xs:element>
306       <xs:element name="parameters" minOccurs="0"
              maxOccurs="1">
307         <xs:complexType>
308           <xs:element name="parameter" type="xs:
                  string" minOccurs="0" maxOccurs="
                  unbounded"/>
309         </xs:complexType>
310       </xs:element>
311       <xs:element name="timeout" type="xs:
              positiveInteger" minOccurs="0" maxOccurs
              ="1"/>
312       <xs:element name="exitvalue" type="xs:
              positiveInteger" minOccurs="0" maxOccurs
              ="1"/>
313     </xs:sequence>
314   </xs:group>
315
316   <xs:group name="fileSystemOutputPortParameters">
317     <xs:sequence>
318       <xs:element name="directory" type="xs:string"
              minOccurs="1" maxOccurs="1"/>
319       <xs:element name="serializer" minOccurs="1"
              maxOccurs="1">
320         <xs:simpleType>
321           <xs:restriction base="xs:string">
322             <xs:enumeration value="matlab"/>
323           </xs:restriction>
324         </xs:simpleType>
325       </xs:element>
326     </xs:sequence>
327   </xs:group>
328
329   <xs:group name="randomOutputPortParameters">
330     <xs:sequence>
331       <xs:element ref="conditions" minOccurs="0"
              maxOccurs="1"/>
332       <xs:element ref="pipes" minOccurs="1" maxOccurs
              ="1"/>
333       <xs:element name="select" type="xs:
              positiveInteger" minOccurs="1" maxOccurs
              ="1"/>
334     </xs:sequence>
```

```
335 </xs:group>
336
337 <xs:complexType name="filter">
338   <xs:sequence>
339     <xs:element name="input" minOccurs="1"
          maxOccurs="1">
340       <xs:complexType>
341         <xs:sequence>
342           <xs:element name="port" minOccurs="1"
                maxOccurs="unbounded">
343             <xs:complexType>
344               <xs:choice minOccuers="0">
345                 <xs:group ref="
                      fstInputPortParameters"/>
346                 <xs:group ref="
                      ppInputPortParameters"/>
347                 <xs:group ref="
                      rvgInputPortParameters"/>
348               </xs:choice>
349               <xs:attribute name="name" type="
                    nametype" use="required"/>
350               <xs:attribute name="type" use="
                    required">
351                 <xs:simpleType>
352                   <xs:restriction base="xs:string">
353                     <xs:enumeration value="gateway
                          "/>
354                     <xs:enumeration value="
                          filesystemlistener"/>
355                     <xs:enumeration value="
                          parameterprompt"/>
356                     <xs:enumeration value="
                          randomvaluegenerator"/>
357                   </xs:restriction>
358                 </xs:simpleType>
359               </xs:attribute>
360             </xs:complexType>
361           </xs:element>
362         </xs:sequence>
363       </xs:complexType>
364     </xs:element>
365     <xs:element name="transformation" minOccurs="1"
          maxOccurs="1">
366       <xs:complexType>
367         <xs:choice minOccuers="0">
368           <xs:group ref="
                externalTransformationParameters"/>
369           <xs:group ref="
                joinTransformationParameters"/>
370           <xs:group ref="
```

```
            reformTransformationParameters"/>
371       </xs:choice>
372       <xs:attribute name="type" use="required">
373         <xs:simpleType>
374           <xs:restriction base="xs:string">
375             <xs:enumeration value="empty"/>
376             <xs:enumeration value="external"/>
377             <xs:enumeration value="join"/>
378             <xs:enumeration value="reform"/>
379           </xs:restriction>
380         </xs:simpleType>
381       </xs:attribute>
382     </xs:complexType>
383   </xs:element>
384   <xs:element name="output" minOccurs="1"
        maxOccurs="1">
385     <xs:complexType>
386       <xs:sequence>
387         <xs:element name="port" minOccurs="1"
              maxOccurs="unbounded">
388           <xs:complexType>
389             <xs:choice minOccuers="0">
390               <xs:group ref="
                    basicOutputPortParameters"/>
391               <xs:group ref="
                    externalOutputPortParameters"/>
392               <xs:group ref="
                    fileSystemOutputPortParameters
                    "/>
393               <xs:group ref="
                    randomOutputPortParameters"/>
394             </xs:choice>
395             <xs:attribute name="name" type="
                  nametype" use="required"/>
396             <xs:attribute name="type" use="
                  required">
397               <xs:simpleType>
398                 <xs:restriction base="xs:string">
399                   <xs:enumeration value="gateway
                        "/>
400                   <xs:enumeration value="external
                        "/>
401                   <xs:enumeration value="
                        filesystem"/>
402                   <xs:enumeration value="random
                        "/>
403                   <xs:enumeration value="screen
                        "/>
404                 </xs:restriction>
405               </xs:simpleType>
```

```
406                </xs:attribute>
407              </xs:complexType>
408            </xs:element>
409          </xs:sequence>
410        </xs:complexType>
411      </xs:element>
412    </xs:sequence>
413    <xs:attribute name="name" type="nametype" use="
         required"/>
414    <xs:attribute name="runat" type="ipaddresstype"
         use="required"/>
415    <xs:attribute name="buffer" type="xs:
         positiveInteger"/>
416    <xs:attribute name="exceptionhandler" type="
         nametype"/>
417    <xs:key name="PK_inputport">
418      <xs:selector xpath=".//input/port" />
419      <xs:field xpath="@name" />
420    </xs:key>
421    <xs:key name="PK_outputport">
422      <xs:selector xpath=".//output/port" />
423      <xs:field xpath="@name" />
424    </xs:key>
425  </xs:complexType>
426
427  <xs:element name="project">
428    <xs:complexType>
429        <xs:sequence>
430          <xs:element name="exceptionhandlers"
               minOccurs="0">
431          <xs:complexType>
432              <xs:element ref="exceptionhandler"
                   minOccurs="0" maxOccurs="unbounded
                   "/>
433          </xs:complexType>
434          <xs:key name="PK_exceptionhandlers">
435            <xs:selector xpath=".//exceptionhandler"
                 />
436            <xs:field xpath="@name" />
437          </xs:key>
438          </xs:element>
439          <xs:element name="filters" minOccurs="1"
               maxOccurs="1">
440          <xs:complexType>
441              <xs:element ref="filter" minOccurs="1"
                   maxOccurs="unbounded"/>
442          </xs:complexType>
443          <xs:key name="PK_filter">
444            <xs:selector xpath=".//filter" />
445            <xs:field xpath="@name" />
```

```
446             </xs:key>
447             </xs:element>
448         </xs:sequence>
449       </xs:complexType>
450   </xs:element>
451
452   </xs:schema>
```

Appendix B

XML-Schema for DatProNet Extensions

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/
       XMLSchema">
3
4  <xs:simpleType name="nametype">
5    <xs:restriction base="xs:string">
6      <xs:pattern value="[a-zA-Z0-9_-]+"/>
7    </xs:restriction>
8  </xs:simpleType>
9
10 <xs:group name="pipeConfiguration">
11   <xs:sequence>
12     <xs:element name="key" type="nametype"
           minOccurs="1" maxOccurs="1"/>
13     <xs:element name="xmlschema" type="xs:string"
           minOccurs="0" maxOccurs="1"/>
14     <xs:element name="class">
15       <xs:complexType>
16         <xs:sequence>
17           <xs:element name="source" type="xs:string
                " minOccurs="1" maxOccurs="1"/>
18           <xs:element name="destination" type="xs:
                string" minOccurs="1" maxOccurs="1"/>
19         </xs:sequence>
20       </xs:complexType>
21     </xs:element>
22   </xs:sequence>
23 </xs:group>
24
25 <xs:group name="simpleConfiguration">
26   <xs:sequence>
```

```
27      <xs:element name="key" type="nametype"
           minOccurs="1" maxOccurs="1"/>
28      <xs:element name="xmlschema" type="xs:string"
           minOccurs="0" maxOccurs="1"/>
29      <xs:element name="class" type="xs:string"
           minOccurs="1" maxOccurs="1"/>
30    </xs:sequence>
31  </xs:group>
32
33  <xs:complexType name="component">
34    <xs:sequence>
35      <xs:element name="port" minOccurs="1" maxOccurs
           ="unbounded">
36        <xs:complexType>
37          <xs:choice minOccuers="0">
38            <xs:group ref="simpleConfiguration"/>
39            <xs:group ref="pipeConfiguration"/>
40          </xs:choice>
41          <xs:attribute name="name" type="nametype"
              use="required"/>
42          <xs:attribute name="type" use="required">
43            <xs:simpleType>
44              <xs:restriction base="xs:string">
45                <xs:enumeration value="inputport"/>
46                <xs:enumeration value="transformation
                     "/>
47                <xs:enumeration value="outputport"/>
48                <xs:enumeration value="pipe"/>
49              </xs:restriction>
50            </xs:simpleType>
51          </xs:attribute>
52        </xs:complexType>
53      </xs:element>
54    </xs:sequence>
55  </xs:complexType>
56
57  <xs:element name="extension">
58    <xs:complexType>
59        <xs:sequence>
60          <xs:element name="component" minOccurs="1"
               maxOccurs="1">
61          <xs:complexType>
62              <xs:element ref="component" minOccurs
                   ="1" maxOccurs="unbounded"/>
63          </xs:complexType>
64          <xs:key name="PK_component">
65            <xs:selector xpath=".//component" />
66            <xs:field xpath="/key" />
67          </xs:key>
68          </xs:element>
```

```
69      </xs:sequence>
70    </xs:complexType>
71  </xs:element>
72
73  </xs:schema>
```

Appendix C

Architectural Description of Case Study

```
<project>
  <filters>

    <filter name="TrainingData" runat="127.0.0.1">
      <input>
        <port name="input" type="filesystemlistener
           ">
          <key>trainingpictures</key>
          <mode>all</mode>
          <object>C:\TrainingData\</object>
          <depth>1</depth>
          <delay>700</delay>
          <initialization>clear</initialization>
        </port>
      </input>
      <transformation type="empty" />
      <output>
        <port name="output" type="gateway">
          <pipes>
            <pipe destination="Merge1.input1" type
               ="local" />
          </pipes>
        </port>
      </output>
    </filter>

    <filter name="Settings" runat="127.0.0.1">
      <input>
        <port name="input" type="filesystemlistener
           ">
          <key>configuration</key>
```

```
29          <object>C:\configuration.m</object>
30          <delay>700</delay>
31          <initialization>clear</initialization>
32        </port>
33      </input>
34      <transformation type="empty" />
35      <output>
36        <port name="output" type="gateway">
37          <pipes>
38            <pipe destination="Merge1.input2" type
                 ="local" />
39            <pipe destination="Merge2.input1" type
                 ="local" />
40          </pipes>
41        </port>
42      </output>
43    </filter>
44
45    <filter name="Merge1" runat="127.0.0.1">
46      <input>
47        <port name="input1" type="gateway" />
48        <port name="input2" type="gateway" />
49      </input>
50      <transformation type="join">
51        <capacity>1</capacity>
52      </transformation>
53      <output>
54        <port name="output" type="gateway">
55          <pipes>
56            <pipe destination="Training.input" type
                 ="local"/>
57          </pipes>
58        </port>
59      </output>
60    </filter>
61
62    <filter name="Training" runat="127.0.0.1">
63      <input>
64        <port name="input" type="gateway" />
65      </input>
66      <transformation type="external">
67        <command>matlab.exe</command>
68        <parameters>
69          <parameter>-nodesktop</parameter>
70          <parameter>-wait</parameter>
71          <parameter>-r</parameter>
72          <parameter>load('%f'); startTraining()l
               save -v6 '%f'; exit;</parameter>
73        </parameters>
74        <serializer>matlab</serializer>
```

```
75          <timeout>3600000</timeout>
76        </transformation>
77        <output>
78          <port name="output" type="gateway">
79            <pipes>
80              <pipe destination="Merge2.input2" type
                    ="local" />
81            </pipes>
82          </port>
83        </output>
84      </filter>
85
86      <filter name="Acquire" runat="10.0.0.138">
87        <input>
88          <port name="input" type="filesystemlistener
                ">
89            <key>trainingpictures</key>
90            <mode>single</mode>
91            <object>C:\Camera\</object>
92            <depth>1</depth>
93            <delay>700</delay>
94            <initialization>clear</initialization>
95          </port>
96        </input>
97        <transformation type="empty" />
98        <output>
99          <port name="output" type="gateway">
100           <pipes>
101             <pipe destination="Merge2.input3" type
                    ="tcp">
102               <portnumber>1444</portnumber>
103             </pipe>
104           </pipes>
105         </port>
106       </output>
107     </filter>
108
109     <filter name="Merge2" runat="127.0.0.1">
110       <input>
111         <port name="input1" type="gateway" />
112         <port name="input2" type="gateway" />
113         <port name="input3" type="gateway" />
114       </input>
115       <transformation type="join">
116         <capacity>1</capacity>
117       </transformation>
118       <output>
119         <port name="output" type="gateway">
120           <pipes>
121             <pipe destination="Analyze.input" type
```

```
                ="local"/>
122           </pipes>
123         </port>
124       </output>
125     </filter>
126
127     <filter name="Analyze" runat="127.0.0.1">
128       <input>
129         <port name="input" type="gateway" />
130       </input>
131       <transformation type="external">
132         <command>matlab.exe</command>
133         <parameters>
134           <parameter>-nodesktop</parameter>
135           <parameter>-wait</parameter>
136           <parameter>-r</parameter>
137           <parameter>load('%f'); analyzePicture();
                 save -v6 '%f'; exit;</parameter>
138         </parameters>
139         <serializer>matlab</serializer>
140         <timeout>3600000</timeout>
141       </transformation>
142       <output>
143         <port name="output" type="gateway">
144           <pipes>
145             <pipe destination="Store.input" type="
                    local" />
146           </pipes>
147         </port>
148       </output>
149     </filter>
150
151     <filter name="Store" runat="127.0.0.1">
152       <input>
153         <port name="input" type="gateway" />
154       </input>
155       <transformation type="empty" />
156       <output>
157         <port name="output" type="filesystem">
158           <directory>C:\Storage\</directory>
159           <serializer>matlab</serializer>
160         </port>
161       </output>
162     </filter>
163
164   </filters>
165 </project>
```

Bibliography

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S.B. Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2nd Confention on Innovative Data Systems Research*, pages 277–289, January 2005.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGACTSIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, June 2002.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Boston, 2003.

[4] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - a new class of data management applications. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 215–226, August 2002.

[5] CCES. Cces - record. `http://www.cces.ethz.ch/projects/nature/Record`, January 2011.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proceedings of the 1st Confention on Innovative Data Systems Research*, pages 269–280, January 2003.

[7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, 2003.

[8] Workflow Management Coalition. The workflow reference model. Document Number TC00-1003, Issue 1.1, January 1995.

[9] K. Di, R. Ma, and R. Li. Automatic shoreline extraction from highresolution ikonos satellite imagery. In *Proceedings of ASPRS 2003 Conference*, pages 5–9, Anchorage, May 2003.

[10] Swiss Experiment. Record:home - swissexperiment. `http://www.swiss-experiment.ch/index.php/Record:Home`, January 2011.

[11] R.T. Fielding. Architectural styles and the design of network-based software architecture. Phd dissertation, Dept. of Information and Computer Science, University of California, Irvine, 2000.

[12] The FreeBSD Foundation. The freebsd project. `http://www.freebsd.org`, January 2011.

[13] D. Garlan. An introduction to the aesop system. `http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps`, July 1995.

[14] L. Golab and M. T. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.

[15] M. M. Gorlick and R. R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13)*, pages 23–34, Austin, May 1991.

[16] The PHP Group. Php: date - manual. `http://nl3.php.net/manual/en/function.date.php`, January 2011.

[17] D. Hull. Ebi interproscan. `http://www.myexperiment.org/workflows/4.html`, January 2011.

[18] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J.R.O. Silva. Documenting component and connector views with uml 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute, Carnegie Mellon University, 2004.

[19] Kepler. The kepler project. `http://kepler-project.org`, January 2011.

[20] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained, The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, 2003.

[21] P. Kruchten. The "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, 1995.

[22] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, Boston, second edition, 2000.

[23] E.A. Lee and T. Parks. Dataflow process networks. In *Proceedings of the IEEE*, volume 83, pages 1–63, May 1995.

[24] J. Magee, N. Dulay abd S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, Barcelona, September 1995.

[25] M.W. Maier, D. Emery, and R. Hilliard. Software architecture: Introducing IEEE standard 1471. *IEEE Computer*, 34(4):107–109, 2001.

[26] D.C. Mason and I.J. Davenport. Accurate and efficient determination of the shoreline in ers-1 sar images. *IEEE Transactions on Geoscience and Remote Sensing*, 34:1243–1253, 1996.

[27] Microsoft. Download details: Xml notepad 2007. `http://www.microsoft.com/downloads/en/details.aspx?FamilyID=72d6aa49-787d-4118-ba5f-4f30fe913628`, January 2011.

[28] Microsoft. Sql server. `http://www.microsoft.com/sqlserver/2008/en/us/`, January 2011.

[29] Microsoft. Windows: Products. `http://www.microsoft.com/windows/`, January 2011.

[30] S. Muthukrishnan. Data streams: Algorithms and applications. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms.*, January 2003.

[31] University of California. xadl 2.0 - a highly extensible architecture description language for software and systems. `http://www.isr.uci.edu/projects/xarchuci/index.html`, January 2011.

[32] Oracle. For java developers. `http://www.oracle.com/technetwork/java/index.html`, January 2011.

[33] Oracle. Oracle database. `http://www.oracle.com/us/products/database/index.html`, January 2011.

[34] N.R. Pal and S.K. Pal. A review on image segmentation techniques. *Pattern Recognition*, 26:1277–1294, 1993.

[35] A. Rankin, L. Matthies, and A. Huertas. Daytime water detection by fusing multiple cues for autonomous off-road navigation. In *Proceedings of the 24th Army Science Conference*, Orlando, November 2004.

[36] Inc. Red Hat. The world's open source leader. `http://www.redhat.com`, January 2011.

[37] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Boston, 1998.

[38] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[39] Monkfish XML Software. Xmlblueprint. `http://www.xmlblueprint.com/`, January 2011.

[40] F. Tanoh. Example of a conditional execution workflow. `http://www.myexperiment.org/workflows/52.html`, January 2011.

[41] Taverna. Taverna - open source and domain independent workflow management system. `http://www.taverna.org.uk`, January 2011.

[42] Inc. The MathWorks. Matlab - the language of technical computing. `http://www.mathworks.com/products/matlab/`, January 2011.

[43] F. van der Heijden, R.P.W. Duin, D. de Ridder, and D.M.J. Tax. *Classification, Parameter Estimation and State Estimation: An Engineering Approach using MATLAB*. John Wiley & Sons Ltd, Chichester, 2004.

[44] Wikipedia. Class diagram. `http://en.wikipedia.org/wiki/Class_diagram`, January 2011.

[45] Wikipedia. Computer network. http://en.wikipedia.org/wiki/Computer_network, January 2011.

[46] Wikipedia. Kahn process networks. http://en.wikipedia.org/wiki/Kahn_process_networks, January 2011.

[47] Wikipedia. Precision and recall. http://en.wikipedia.org/wiki/Precision_and_recall, January 2011.

[48] Wikipedia. Transmission control protocol. http://en.wikipedia.org/wiki/Transmission_Control_Protocol, January 2011.

[49] Wikipedia. Type i and type ii errors. http://en.wikipedia.org/wiki/Type_I_and_type_II_errors, January 2011.

[50] Wikipedia. Write once, run anywhere. http://en.wikipedia.org/wiki/Write_once,_run_anywhere, January 2011.

[51] A. Williams. Pipelined list iteration. http://www.myexperiment.org/workflows/1372.html, January 2011.

[52] WordNet. Data processing. http://wordnetweb.princeton.edu/perl/webwn?s=data+processing, January 2011.

[53] A. Subramanian X. Gong and C.L. Wyatt. A two-stage algorithm for shoreline detection. In *IEEE Workshop on Applications of Computer Vision (WACV'07)*, 2007.

[54] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26(1), March 2003.

[55] S.C. Zhu and A. Yuille. Region comperition: Unifying snakes, region growing, and bayes/mdl for multiband image segmentation. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18(9):884–900, 1996.